The Journal of Open Source Software

Hypothesis: A new approach to property-based testing

David R. MacIver[1], Zac Hatfield-Dodds[2], and many other contributors[3]

1 Imperial College London **2** Australian National University **3** Various

DOI: 10.21105/joss.01891

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @luizirber
- @djmitche

Submitted: 12 November 2019
Published: 21 November 2019

Summary

Property-based testing is a style of testing popularised by the QuickCheck family of libraries, first in Haskell (Claessen & Hughes, 2000) and later in Erlang (Arts, Hughes, Johansson, & Wiger, 2006), which integrates generated test cases into existing software testing workflows: Instead of tests that provide examples of a single concrete behaviour, tests specify properties that hold for a wide range of inputs, which the testing library then attempts to generate test cases to refute. For a general introduction to property-based testing, see (MacIver, 2019).

Hypothesis is a mature and widely used property-based testing library for Python. It has over 100,000 downloads per week[1], thousands of open source projects use it[2], and in 2018 more than 4% of Python users surveyed by the PSF reported using it[3]. It will be of interest both to researchers using Python for developing scientific software, and to software testing researchers as a platform for research in its own right.

Hypothesis for Testing Scientific Software

Python has a rich and thriving ecosystem of scientific software, and Hypothesis is helpful for ensuring its correctness. Any researcher who tests their software in Python can benefit from these facilities, but it is particularly useful for improving the correctness foundational libraries on which the scientific software ecosystem is built. For example, it has found bugs in astropy (Price-Whelan et al., 2018)[4] and numpy (Walt, Colbert, & Varoquaux, 2011)[5].

Additionally, Hypothesis is easily extensible, and has a number of third-party extensions for specific research applications. For example, hypothesis-networkx[6] generates graph data structures, and hypothesis-bio[7] generates formats suitable for bioinformatics. As it is used by more researchers, the number of research applications will only increase.

By lowering the barrier to effective testing, Hypothesis makes testing of research software written in Python much more compelling, and has the potential to significantly improve the quality of the associated scientific research as a result.

[1] https://pypistats.org/packages/hypothesis
[2] https://github.com/HypothesisWorks/hypothesis/network/dependents
[3] https://www.jetbrains.com/research/python-developers-survey-2018/
[4] e.g. https://github.com/astropy/astropy/pull/9328, https://github.com/astropy/astropy/pull/9532
[5] e.g. https://github.com/numpy/numpy/issues/10930, https://github.com/numpy/numpy/issues/13089, https://github.com/numpy/numpy/issues/14239
[6] https://pypi.org/project/hypothesis-networkx/
[7] https://pypi.org/project/hypothesis-bio/

The Journal of Open Source Software

Hypothesis for Software Testing Research

Hypothesis is a powerful platform for software testing research, both because of the wide array of software that can be easily tested with it, and because it has a novel implementation that solves a major difficulty faced by prior software testing research.

Much of software testing research boils down to variants on the following problem: Given some interestingness condition (e.g., that it triggers a bug in some software), how do we generate a "good" test case that satisfies that condition?

Particular sub-problems of this are:

1. How do we generate test cases that satisfy difficult interestingness conditions?
2. How do we ensure we generate only valid test cases? (the *test-case validity problem* - see Regehr et al. (2012))
3. How do we generate human readable test cases?

Traditionally property-based testing has adopted random test-case generation to find interesting test cases, followed by test-case reduction (see Regehr et al. (2012), Zeller & Hildebrandt (2002)) to turn them into more human readable ones, requiring the users to manually specify a *validity oracle* (a predicate that identifies if an arbitrary test case is valid) to avoid invalid test cases.

The chief limitations of this from a user's point of view are:

- Writing correct validity oracles is difficult and annoying.
- Random generation, while often much better than hand-written examples, is not especially good at satisfying difficult properties.
- Writing test-case reducers that work well for your problem domain is a specialised skill that few people have or want to acquire.

The chief limitation from a researcher's point of view is that trying to improve on random generation's ability to find bugs will typically require modification of existing tests to support new ways of generating data, and typically these modifications are significantly more complex than writing the random generator would have been. Users are rarely going to be willing to undertake the work themselves, which leaves researchers in the unfortunate position of having to put in a significant amount of work per project to understand how to test it.

Hypothesis avoids both of these problems by using a single universal representation for test cases. Ensuring that test cases produced from this format are valid is relatively easy, no more difficult than ensuring that randomly generated tests cases are valid, and improvements to the generation process can operate solely on this universal representation rather than requiring adapting to each test.

Currently Hypothesis uses this format to support two major use cases:

1. It is the basis of its approach to test-case reduction, allowing it to support more powerful test-case reduction than is found in most property-based testing libraries with no user intervention.
2. It supports Targeted Property-Based Testing (Löscher & Sagonas, 2017), which uses a score to guide testing towards a particular goal (e.g., maximising an error term). In the original implementation this would require custom mutation operators per test, but in Hypothesis this mutation is transparent to the user and they need only specify the goal.

The internal format is flexible and contains rich information about the structure of generated test cases, so it is likely future versions of the software will see other features built on top of it, and we hope researchers will use it as a vehicle to explore other interesting possibilities for test-case generation.

References

Arts, T., Hughes, J., Johansson, J., & Wiger, U. T. (2006). Testing telecoms software with quviq QuickCheck. In M. Feeley & P. W. Trinder (Eds.), *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang* (pp. 2–10). ACM. doi:10.1145/1159789.1159792

Claessen, K., & Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In M. Odersky & P. Wadler (Eds.), *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)* (pp. 268–279). ACM. doi:10.1145/351240.351266

Löscher, A., & Sagonas, K. (2017). Targeted property-based testing. In T. Bultan & K. Sen (Eds.), *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 46–56). ACM. doi:10.1145/3092703.3092711

MacIver, D. R. (2019). In praise of property-bsed testing. https://increment.com/testing/in-praise-of-property-based-testing/.

Price-Whelan, A. M., Sipőcz, B. M., Günther, H. M., Lim, P. L., Crawford, S. M., Conseil, S., Shupe, D. L., et al. (2018). The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package, *156*, 123. doi:10.3847/1538-3881/aabc4f

Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., & Yang, X. (2012). Test-case reduction for C compiler bugs. In J. Vitek, H. Lin, & F. Tip (Eds.), *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI '12)* (pp. 335–346). ACM. doi:10.1145/2254064.2254104

Walt, S. van der, Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, *13*(2), 22–30. doi:10.1109/MCSE.2011.37

Zeller, A., & Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, *28*(2), 183–200. doi:10.1109/32.988498