Introduction to Property-Based Testing

Zidi Chen, Yuehao Sui

April 2021

1 Introduction

Software testing plays an important role to ensure the software works as expected [2]. It usually needs a lot of human effort. To make testing efficient, test automation was introduced [6]. Property-based testing is one of the ways to make tests automated. It has useful features, including shrinking and generators, which help to develop better tests.

Property-based testing should be a good topic for DevOps developers. The basic idea behind DevOps is to increase the efficiency of software life-cycle [10, 1]. Property-based testing could contribute to DevOps culture, since it allows to write better tests with less effort in coding in a long term [8].

This essay aims to introduce property-based testing, explain how it works and how it compares to traditional tests, as well as give suggestions of when to apply it.

2 Property-Based testing

Property-based testing is to generate tests based on properties [7], which will be explained by an example in section 2.1. It can detect bugs in many categories by helping developers narrow down the problematic area. There was an example that nearly 60,000 lines of code were tested by around 500 lines of code using a tool of property-based testing [8]. The bugs came from:

- Issues of software performance: timing, handling faults, race condition, and system restrictions [8]
- Issues of coding: usage of API, programming logic, typing [8]
- Other issues: documentation and hardware [8]

In this section, an example is used to introduce what is the basic idea of property-based testing. Then, two features: generator and shrinking are introduced.

2.1 Basic idea of Property-based testing

Figure 1 shows a function which needs to be tested. This function multiplies two input values.

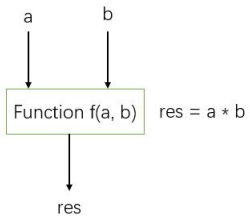

Figure 1: Multiplication function

To test this function, instead of thinking about concrete test cases, the most critical thing is to find the properties. Multiplication has two important properties:

- Associative property: (a * b) * c = a * (b * c)
- Commutative property: a * b = b * a

According to these properties, the tests could be:

- Associative property: verify f(f(a, b), c) = f(a, f(b, c))
- Commutative property: verify f(a, b) = f(b, a)

The data of a, b and c should be automatically generated to verify property rules. To improve the tests, property-based testing provides generators to create better test cases.

2.2 Generator

Generators are responsible for creating input data randomly [3], which is a critical component of property-based testing. High randomness in test cases implies high coverage, which indicates that the tests are highly reliable [8].

The property-based testing tools provides predefined generators for basic data types [3]. For example, PropEr, which is a property-based testing tool for Erlang has an generator *rangeMin, Max* [8]. This generator offers an integer in the range of *Min* and *Max*, including the boundaries [8]. Programmers can also utilize basic generators to form more complex generators. [3]. For example, *list(integer())* is to create a list of integers each iteration, using *list()* and *integer()*.

Sometimes, even the combination of the basic generators can not meet the needs. For more complicated demands, property-based testing allows custom

generators [8]. Custom generators can do many things. They can collect statistical analysis of the test, such as a report of distribution of generated input [8]. It is also possible to clarify the size of the generator, use a self-written function as a generator, reformat the generated data, limit the generated data by demand, and enable the probabilistic distribution of complex data types [8]. In addition, generators in a recursive manner are supported by some tools [8]. Property-based testing also provides symbolic calls as a notation to make data structures more explicit and understandable [8]. Custom generators enable more possibilities for developers to create better tests.

Developers should use default generators and customize specific generators according to demands. Different property-based tools might have variation in generators, which should be paid attention.

2.3 Shrinking

Shrinking is an important feature of property-based testing. It can narrow down and regenerate where the error happens, which helps developers to figure out problems [11].

Here is an example which is adapted from a paper of Fang-Yi Lo et al.. Figure 2 shows a function which removes the elements in a list. The removed elements should be equal to the first input parameter [11]. In this example, the fault in implementation is that it only deletes the first equivalent element in the list [11].

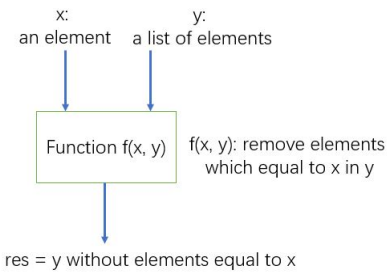

Figure 2: Remove function

Property-based testing could provide a result that **[3, 4, 7, 10, 4, 16, 5, 9]** is not able to pass the test [11]. Based on that, shrinking reduces the size of this counterexample to **[4, 4]** [11]. Then, developers can guess that the possible mistake in the function [11].

Shrinking can narrow the counterexample, which helps developers to locate mistakes. This is one of the best features of property-based testing.

3 Comparison with traditional tests

Fred Hebert gave an intuitive example in the first section of his book [8]. The tested function merges many sorted lists to a single sorted list [8]. Figure 3 presents the traditional test, which is written in Erlang. Developers need to generate test cases to cover all possible situations, which is an annoying process. In addition, there are still more possible test cases besides the ones listed, which means that it is always hard for developers to write a comprehensive test [8].

```
merge_test() ->
    [] = merge([]),
    [] = merge([[]]),
    [] = merge([[],[]]),
    [] = merge([[],[],[]]),
    [1] = merge([[1]]),
    [1,1,2,2] = merge([[1,2],[1,2]]),
    [1] = merge([[1],[],[]]),
    [1] = merge([[],[1],[]]),
    [1] = merge([[],[],[1]]),
    [1,2] = merge([[1],[2],[]]),
    [1,2] = merge([[1],[],[2]]),
    [1,2] = merge([[],[1],[2]]),
    [1,2,3,4,5,6] = merge([[1,2],[],[5,6],[],[3,4],[]]),
    [1,2,3,4] = merge([[4],[3],[2],[1]]),
    [1,2,3,4,5] = merge([[1],[2],[3],[4],[5]]),
    [1,2,3,4,5,6] = merge([[1],[2],[3],[4],[5],[6]]),
    [1,2,3,4,5,6,7,8,9] = merge([[1],[2],[3],[4],[5],[6],[7],[8],[9]]),
    Seq = seq(1,100),
    true = Seq == merge(map(fun(E) -> [E] end, Seq)),
    ok.
```

Figure 3: Traditional way of testing [8]

Figure 4 presents how the function can be tested with PropEr. It basically generates many sorted lists with different length and elements as inputs. After that, it verifies the result of the function with these inputs equals to applying *append* and *sort* function.

```
sorted_list(N) -> ?LET(L, list(N), sort(L)).

prop_merge() ->
    ?FORALL(List, list(sorted_list(pos_integer())),
        merge(List) == sort(append(List))).
```

Figure 4: Property-based way of testing [8]

These code have fewer lines than traditional code [8]. Test developers need to find out the properties to verify, and to code the generator, which would be not as tedious as thinking about all possible inputs.

In addition, this example tells that there is difference between traditional testing and property-based testing in the aspect of coverage metric. Coverage metric it important, because it evaluates how much the test cases cover the possibilities [4]. Property-based testing allows powerful coverage metric because

of the randomly generated inputs [7]. As in the example, the code in figure 4 covers far more possibilities than the traditional way [8].

Moreover, as the previous sections introduced, property-based testing provides more useful features than traditional testing, such as shrinking and generators, which helps test developers to create better tests conveniently.

To summarize, property-based testing performs distinctly compared with traditional testing, in the aspect of coverage and decreasing human effort to do the time-consuming work. It also offers developers convenient features to develop tests.

4 Tools for property-based testing

One of the firstly-created tools is QuickCheck, which was created for Haskell by Koen Claessen and John Hughes in 1999 [9]. It can generate a large amount of test inputs according to the specified specifications [5]. Then, the inventors of the QuickCheck build QuviQ, a company to supply the services of QuickCheck for business customers, and they build another version of QuickCheck for Erlang to enlarge the market. Later, based on this tool, the Erlang community released two open source derivatives, PropEr and Triq. [8]

After the release of QuickCheck, lots of programming language communities released re-implementations in other languages. For example, ScalaCheck for Scala, CppQuickCheck for C++, QuickTheories for Java, Hypothesis for Python and so on.

Compared with QuickCheck, the later tools have various changes. Take Hypothesis as an example, Hypothesis provides a stronger way to reduce test-cases automatically, and it offers targeted property-based testing [13].

Tools for property-based testing would make test development easier. Developers should choose tools based on their own situations, because tools are different in functionalities and prices. For instance, Erlang developers might have to choose between QuickCheck and PropEr. For the developers who need free tools, PropEr could be a better choice. But for developers who place emphasis on finding concurrency bugs, QuickCheck could be worthy.

5 Evaluation and discussion

Property-based testing would be useful in some situations. They are often suitable for situations where the inputs cannot be identified, especially if the objects to test are associated with real-world situations. If there are a lot of possible combinations of inputs, it could be hard for developers in traditional unit testing to cover.

However, property-based testing is not irreplaceable in all situations. The cost of setting up a property-based test would be higher than creating a traditional test. The biggest difference with unit testing is that the property-based testing are generalized but unit tests can only be used for one possible scenario.

Due to this, property-based testing is often much slower than unit testing. This is taken for granted, according to the design of property-based testing, each test is performed multiple times for a particular property, not just a particular case. In addition, developing tests to be generic rather than specific is not a simple task [8]. Finding good properties would be relatively more difficult than finding specific test cases.

Thus, property-based testing is not yet a complete replacement for traditional testing, such as unit testing. Developer teams need to decide whether or not to apply property-based testing based on multiple factors carefully.

For future research, there are many sub-fields which can be investigated. For example, targeted property-based testing is to create inputs by certain strategy instead of randomly generation [12]. There are also studies on how to test stateful properties. In addition, improving the coverage of property-based test could be a topic.

6 Conclusion

To summarize, property-based testing is to test by automatically generated test cases based on given properties, instead of testing by specified inputs from developers. It provides powerful features, such as shrinking and custom generators. There are many tools for property-based testing available such as QuickCheck, PropEr and Hypothesis. Developers could decide whether to apply property-based testing based on cost and concrete conditions. While there are many advantages with property-based testing, it cannot replace traditional testing completely now.

References

[1] Prashant Agrawal and Neelam Rawat. Devops, a new approach to cloud development & testing. In *2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, volume 1, pages 1–4. IEEE, 2019.

[2] S.S.Riaz Ahamed. Studying the feasibility and importance of software testing: An analysis. *International Journal of Engineering Science and Technology*, 1(3):119–128, 2009.

[3] Bernhard K Aichernig, Silvio Marcovic, and Richard Schumi. Property-based testing with external test-case generators. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 337–346. IEEE, 2017.

[4] Hana Chockler, Orna Kupferman, and Moshe Y Vardi. Coverage metrics for formal verification. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 111–125. Springer, 2003.

[5] Koen Claessen. QuickCheck: Automatic testing of Haskell programs. `https://hackage.haskell.org/package/QuickCheck`, 2000. [Online; accessed 22-April-2021].

[6] Mark Fewster and Dorothy Graham. *Software test automation*. Addison-Wesley Reading, 1999.

[7] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[8] Fred Hebert. *Property-Based Testing with PropEr, Erlang, and Elixir*. The Pragmatic Programmers, LLC, 2019.

[9] John Hughes. Quickcheck testing for fun and profit. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–32. Springer, 2007.

[10] Muhammad Owais Khan, Awais Khan Jumani, Farhan, Waqas Ahmed Siddique, and Asad Ali Shaikh. Fast delivery, continuously build, testing and deployment with devops pipeline techniques on cloud. *Indian Journal of Science and Technology*, 13(05):552–575, 2020.

[11] Fang-Yi Lo, Chao-Hong Chen, and Ying-ping Chen. Shrinking counterexamples in property-based testing with genetic algorithms. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[12] Andreas Löscher and Konstantinos Sagonas. Automating targeted property-based testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 70–80. IEEE, 2018.

[13] David R Maclver, Zac Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.