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]. Developing and designing automated tests is expensive, but it would be cheaper than traditional test as time goes by [6]. Property-based testing is one of the ways to make tests automated.

Property-based testing should be noticed by DevOps developers. The basic idea behind DevOps is to increase the efficiency of software planning, integration, deployment, testing, feedback and monitoring [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 introduces what is property-based testing, what are differences compared with traditional tests, how to do property-based testing, and tries to give suggestions through discussion and evaluation about when to apply property-based testing.

2 What is property-based testing?

Property-based testing is to generate tests based on property requirements and features [7]. It can detect bugs in many categories. 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 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.

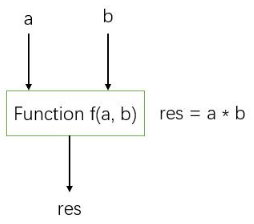

Figure 1: Multiplication function

To test this function, instead of thinking about concrete test cases, the most critical thing is to find the properties of the function. As known to all, pure 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 by property-based testing tools. In this way, testing data are automatically created to verify property rules. To improve the tests, property-based testing provides generators to generate better test cases, and it owns the feature of shrinking to reproduce the error.

2.2 Generator

Generators are responsible for creating input data randomly [3], which is a critical element 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 default 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 can offer an integer in the range of *Min* and *Max*, including the value of *Min* and *Max* [8]. Programmers can also utilize the combination of predefined generators for data with higher complexity [3].

Sometimes, even the combination of the basic generators can not meet the needs. For data which are not in basic data types and more complicated demands, property-based testing allows custom generators [8]. Custom generators can do many things. It can collect the statistical analysis of the test, such as giving 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 also supported by some tools, for suitable situations, such as counting the words [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 and situations. Different property-based tools might have variation in generators, which should be paid attention by developers.

2.3 Shrinking

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

Here is an example which is adapted from a a paper from Fang-Yi Lo et al.. Figure 2 shows a function which removes the elements in a list, and the removed elements should be equal to the first input parameter [11].

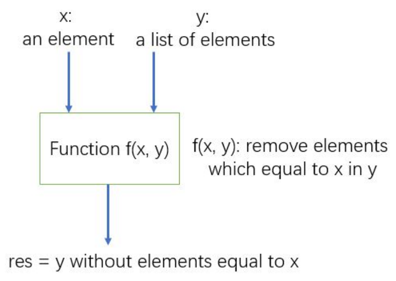
x:
an element
y:
a list of elements
Function f(x, y)
f(x, y): remove elements which equal to x in y
res = y without elements equal to x

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 could reduce the size of this counterexample to [4, 4] [11]. From this reduced result, developers can guess that the possible mistake in the function [11]. In this example, the fault in implementation of the function only deletes the first equivalent element in the list [11].

Shrinking can narrow the counterexample, which helps developers to find where the mistake is. 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]. In this example, the tested function merges many sorted function to one single sorted function [8]. Figure 3 presents the traditional way of testing, which is written in Erlang. As presented, the developers need to generate test cases to cover all possible situations, which is a very annoying process. In addition, there are still more possible test cases besides the ones in figure 3, 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 the help of PropEr, which is a property-based testing tool for Erlang. It basically generates many sorted list 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 and they need less effort from test developers [8]. Test developers only need to find out the property to verify, and to code the generator, which would be not as boring as thinking about all possible test cases.

In addition, this example tells that there is difference between traditional testing and property-based testing in the aspect of coverage metric. Since it is not feasible to test all possible inputs, coverage metric it important, because it evaluates how much the test cases cover the possibilities [4]. Coverage metric can indicate if the more test cases are needed, or the test cases are enough to have valuable conclusion [4]. Property-based testing allows powerful coverage metric because of the automatically generated inputs [7]. As in the example, the code in figure 4 covers far more possibilities than the traditional way of testing in figure 3 [8].

Moreover, property-based testing provides many useful features, such as shrinking and generators, which helps test developers to locate the errors and create tests in a convenient way, as the previous sections introduces.

To summarize, property-based testing performs distinctly compared with traditional testing, in the aspect of coverage and decreasing human effort to find all possible test cases. It also offers developers convenient features to develop property-based tests.

4 Ways to property-based testing

In this section, some tools for property-based testing are introduced.

One of the most popular tool 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 authers of the QuickCheck build QuviQ, a company to supply the services of QuickCheck for business customers, and they build Quickcheck for Erlang to gain a larger market. Later, based on this tool, the Erlang community released two other open source derivatives, PropEr and Triq [8].

After the release of QuickCheck, lots of programming language communities released re-implementations in their languages [eg. CC++, JAVA, PHP, etc]. However, compared to the original QuickCheck, i.e. the version implemented by Haskell, Erlang or Scala, the other implementations have more or less lack of functionality or supports.

It is probably for this reason that python fans have developed a set of testing tools called Hypothesis based on the idea of property-based testing.

All in all, QuickCheck is definitely the best choice if you are developing in a functional language. If you are using Erlang, PropEr, powered by the community, is also a good choice. For python developers, using the native framework hypothesis is an easy decision. For developers of other languages, you should read the documentation of the corresponding version of QuickCheck to understand the features that have been implemented.

5 Evaluation and discussion

The first thing we can all agree on is that not all code is worth testing using property-based methods. For example, you don't want to wait for more than hundreds of rendering when testing an HTML page.

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.

Thus, property-based testing is not yet a complete replacement for unit testing, especially in the early days of TDD development, where sample-based testing will perform better.

Property-based tests are often suitable for the kind of situations where the input cannot be identified, especially if the object you are testing is associated with a real-world situation. However, considering tests to be generic rather than specific is not a simple task[8], especially writing good properties. It's relatively easy to write unit tests for just one situation, if you know what the specific inputs and outputs are.

In any case, this is still a fresh and unfamiliar approach for most testers. Perhaps for some teams, switching to property-based testing would be a good opportunity when you're refactoring a project.

6 Conclusion

refer to other sub fields: targeted property based testing [the input generation is guided by a search strategy instead of being random ¡Automating Targeted Property-Based Testing¿]

stateful property based testing: https://propertesting.com/book$_{s}tateful_{p}roperties.html$

Coverage guided, property based testing

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.