ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/nilesh-jagnik/358

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science
https://urfpublishers.com/journal/artificial-intelligence

Vol: 2 & Iss: 1 *Research Article*

Property Based and Fuzz Testing in C++

Nilesh Jagnik*

Citation: Jagnik N. Property Based and Fuzz Testing in C++. J Artif Intell *Mach Learn & Data Sci 2024*, 2(1), 1598-1600. DOI: doi.org/10.51219/JAIMLD/nilesh-jagnik/358

Received: 03 January, 2024; **Accepted:** 28 January, 2024; **Published:** 30 January, 2024

***Corresponding author:** Nilesh Jagnik, Los Angeles, USA, E-mail: nileshjagnik@gmail.com

ABSTRACT

Testing code is an essential practice for reliability and correctness of software systems. Traditionally testing, also known as example-based testing requires developers to think of edge cases where system may produce erroneous results and side effects. This technique relies on the expertise of the developer and is prone to missed edge cases, thereby reducing the efficacy of tests. In this paper, we discuss generative techniques that test code against a large number of generated test cases. These techniques allow for comprehensive testing by detecting bugs, vulnerabilities and performance regressions in code, thus improving the quality of software. We also discuss the FuzzTest library that provides support for these techniques in C++.

Keywords: software testing, example-based testing, property-based testing, fuzzing, performance testing

1. Introduction

Traditional software development involves writing code and tests that assert that code works correctly. The developer has to choose test cases that cover all edge cases. This is also called example-based testing. The problem with this approach is that, depending on the complexity of the code being tested, it might be easy to miss some edge cases. Some APIs may accept a wide range of inputs. For such APIs, creating test cases for every possible input would not even be possible.

Property-based testing can be used to solve this problem. These testing techniques implicitly create test cases based on specifications provided when writing tests. In this setup, tests are written in a generic manner and must specify input parameters. These parameters should be used by test logic to create the input to the test code and also perform assertions. The input parameters are then varied by a testing framework. How to vary the test inputs is specified to the framework by the test developer.

Property-based testing results in more thorough testing of application code. This is due to the fact that testing is more rigorous and captures many more test cases in comparison to traditional software development. Not only this, test developers can specify properties that a test must check for. This enables tests to be smarter and detect bugs which are hard to detect otherwise.

In this paper we discuss the benefits and limitations of property-based testing. We also cover how to implement these testing techniques in C++.

2. Problems with Example Based Testing

Example-based testing refers to the traditional testing paradigm which involves identifying a set of test cases that represent the normal and edge case inputs which the code under test must process. The tests assert that the right output and side effects are generated by the code while processing inputs. There are several shortcomings of example-based testing.

A. Hard to Identify Edge Cases: The main issue with example-based testing is the developer has to think of all edge cases where the code under test is likely to fail. Often times, the developer may not know the full domain of inputs that may be passed into the code at runtime. Even if the developer knows the domain, it may be hard to identify edge cases since they may not be obvious. This leads to less-than-optimal testing and bugs and vulnerabilities at runtime.

B. Too Many Test Cases: For through testing of applications may require testing against many test inputs. This is done to improve correctness and ensure that there are no vulnerabilities in the system. Using example-based testing in such cases may require writing too many tests. This would make tests tedious to write and cumbersome to maintain. A large number of tests would also make the test code hard to read. All of these are undesirable characteristics of test code.

3. Property Based, Table Driven and Fuzz Testing

In contrast to testing for specific test cases as discussed in example-based testing, property-based testing emphasizes testing for properties being satisfied by test inputs. Property-based testing requires writing test assertions in a form that tests that certain properties are satisfied regardless of input that is passed to a test. These tests must be written in a generic way and test inputs are parameterized. Then, parameterized testing is used to pass in inputs to these generic property tests. The inputs can be passed in a few ways:

A. Table Driven Testing: In cases where the set of interesting or edge case input is already known, these inputs can be passed in to the generic tests written for property-based testing. Table-driven tests are parameterized tests where the input can be read from a table of values.

B. Fuzz Testing: Fuzz testing (aka Fuzzing) creates test input by generating data points that satisfy the input domain. Fuzzing can be used to create a lot of data points to even cover the full set of inputs that can actually be passed into a test. Historically, this technique has been used extensively to detect bugs and vulnerabilities in code. As expected, fuzzing tools may take a long time to generate the input set and execute tests with them. Fuzz testing is quite exhaustive and therefore detects bugs which are very difficult to detect otherwise.

Note that both table-driven testing and fuzz testing are input generation techniques and can be used independently of property-based testing. That being said, it is quite common to use them together as that yields the most benefit.

4. Benefits Of Property Based and Fuzz Testing

Property-based and fuzz testing when used together can solve most of the issues present in example-based testing.

a. **Better Test Quality:** In property-based testing, the properties that the code under test are clearly specified in code. This leads to more robust tests. Property-based testing also increases code re-use leading to fewer lines to code that needs to be maintained.

b. **Exhastive Coverage of Input Domains:** Fuzz testing can automatically generate all possible inputs within the domain of inputs. This helps test exhaustively for errors and vulnerability in the code under test. Doing this results in identifying all inputs for which the system behaves erroneously.

c. **Performance Testing:** Apart from errors and vulnerabilities, fuzz testing tests can also be used to detect efficiency and performance bugs in code. By running a parameterized test with fuzzed inputs, performance reports can be generated for measuring performance for processing different inputs.

5. C++ FuzzTest

FuzzTest is a testing framework in C++ for writing property-based tests which are executed using coverage guided fuzzing. In coverage guided fuzzing, test inputs are considered interesting if they cover more of the code under test. FuzzTest can be used with GoogleTest or other unit testing frameworks. The examples presented in this paper use GoogleTest.

a. Overview: (Figure 1) shows a typical example-based unit test written using GoogleTest. This test only covers one input. However, there may be some tricky edge cases which are hard to detect.

```
TEST(DetectOddsTest, DetectsOddsCorrectly) {
 int input = 31;
 bool output = DetectsOdds(input);
 EXPECT_TRUE(output);
}
```

Figure 1: Unit test that detects odd numbers.

FuzzTest can be used here to write a test which isn't specific to a single input. The fuzzy test in **(Figure 2)** specifies to the framework that the test must be run over domain of integer inputs. This ensures that the test isn't tied to a specific input and that the framework can generate inputs to test it thoroughly.

```
void DetectsOddsCorrectly(int number) {
 int input = number*2 + 1;
 const bool output = DetectsOdds(input);
 EXPECT_TRUE(output);
}
FUZZ_TEST(DetectOddsTest, DetectsOddsCorrectly)
 .WithDomains(/*number=*/Arbitrary<int>());
```

Figure 2: Fuzzy Unit test that detects odd numbers.

b. Property Function: FuzzTest requires writing parameterized tests. The main function that is run over all inputs in the property function.

This function should contain all necessary assertions and should return void. In the example in Figure 2, the DetectsOddsCorrectly method is the property function.

c. Input Domains: Contrary to parameterized tests, parameters in FuzzTest are specified in the form of input domains. This is the domain from which test inputs must be generated by the framework. The example in Figure 2 uses arbitrary integer as the domain, but many types of domains can be specified for inputs including but not limited to strings, chars, ints, enums, structs and protos.

d. Initial Seeds: Although not necessary, seed values can be provided for specifying the initial parameter values to the property function. The framework generates new parameters using seed values as a base. Seed values can be provided using the withSeeds() clause.

6. Limitations of Property Based and Fuzz Testing

a. **Difficult to Write:** Writing good property tests is difficult as it requires parameterizing tests and writing good property functions. As such, it has a learning curve associated with it.

b. **Time Consuming and Resource Intensive:** Since tests are run through a lot of generated inputs, this can get quite expensive especially for APIs that do complex operations.

c. **Difficult to Setup:** For code that makes external RPCs, it may not be feasible to setup fakes/mocks that handle all generated inputs. Using the real external endpoint may add too much traffic on it and mocking it efficiently may not be possible.

d. **Complex Input Domain:** Testing APIs which have complex input may not be feasible, either due to framework limitations or simply due to difficulties associated with resources required for generating complex input.

7. Conclusion

Traditional example-based testing is fine for basic testing in scenarios where the domain of possible inputs is small. However, for productionizing services at large scale, thorough testing is needed. Property-based and fuzz testing should be used to ensure that software does not have any bugs or vulnerabilities. Not only this, these techniques can also be used to detect performance regressions. Using these techniques can ensure that software systems run efficiently and reliably.

8. References

1. https://increment.com/testing/in-praise-of-property-based-testing/#
2. https://dave.cheney.net/2019/05/07/prefer-table-driven-tests
3. https://hypothesis.works/articles/what-is-property-based-testing/
4. https://www.mayhem.security/blog/what-is-property-based-testing
5. https://github.com/google/fuzztest
6. https://www.code-intelligence.com/blog/caroline-le-mieux-expanding-fuzzing
7. https://www.covertswarm.com/post/fuzzing-hacking