CS239: Review and Comparison

Zhaowei Tan
504777867

May 25, 2018

Abstract

This report reviews and compares three famous papers under the area of test generation [5, 2, 4]. We first review what are the techniques developed by the papers, and then compare them for a broader discussion in this area.

1 Introduction

Manually generating sufficient tests cases is often a time-consuming and painful procedure. Therefore, automatic test input generation, as a remedy to this situation, has been an active research area among the past few decades. Much literature is produced focusing on different approaches and solutions for better code coverage and other properties.

In this report, we discuss three significant papers in this domain. We first start with Quickcheck [2], which describes a random test generation of Haskell programs (§2.1). We then move on to Java Pathfinder [5], which incorporates symbolic execution and model checking ideas for test inputs generation (§2.2). Last but not least, we describe Randoop [4], a test generator based on the feedback (§2.3).

Next, we briefly compare the differences among the papers described above in §3. We describe the differences among those approaches, and the pros and cons for each of those.

2 Review of Three Papers

2.1 QuickCheck

Motivation. The authors intent to provide a tool which is able to help the Haskell programmers debug the program. As long as the programmer specifies the properties that the functions should satisfy, the tool will generate test data either automatically or follow user-defined generator. The authors continue to discuss QuickCheck: how this tool enables the property definition by the user, and how it generates the random test cases. After defining the property and generator, QuickCheck could generate random inputs and test whether the property holds.

An important feature for this tool is *lightweight*, which results in the fact that QuickCheck will need some user input to generate the inputs and perform testing.

Technique. For defining a property, the authors design a domain-specific language of testable specifications for programmers to specify expected properties. More specifically, this language supports:

- Simple properties with variables.

  For this category, QuickCheck simply represents the property as a Haskell function. Intuitively, if the function returns True for every single test case, this property holds. The paper makes an example that

  $$\text{prop\_RevApp xs ys} = \text{reverse(xs ++ ys)} == \text{reverse ys++reverse xs}$$

  One thing to mention is that due to the overloading, the user has to specify what exactly the type is. Following the same example

  $$\text{prop\_RevApp} :: [\text{Int}] \to [\text{Int}] \to \text{Bool}$$

- Properties with quantify over functions.

  To formulate this type of property, the authors define an extensional equality (===) as (f === g) x = fx == gx. Therefore, as an example, to test whether the association property holds, we could write as

  $$\text{prop\_CompAssoc f g h} = \text{f . (g . h)} === \text{(f . g) . h}$$
  (also need a type signature for specification)

- Laws hold conditionally. We also want to test whether some properties hold under certain conditions in QuickCheck. To represent this, QuickCheck provides an implication combinator to represent this, for example

  $$\text{prop\_MaxLe} :: \text{Int} \to \text{Int} \to \text{Property}$$
  $$\text{prop\_MaxLe x y} = \text{x} \leq \text{y} \to \text{max x y} == \text{y}$$

  By testing a few cases that satisfy the condition (in this case, $x \leq y$), we get the result of this property.

- Monitoring the test data. QuickCheck is also able to generate the statistics of the test data generated.

- Infinite structure. QuickCheck could as well deal with the infinite structures. This is used to handle some properties such as cycle xs == cycle (xs ++ xs). QuickCheck rewrites this as take n (cycle xs) == take n (cycle (xs ++ xs)) to avoid the infinite structure.

For defining generator, QuickCheck adopts random testing under a test data generation language defined by user. Concretely speaking, this language supports:

- Arbitrary type class.

  The QuickCheck introduces a type class called Arbitrary. A type is an instance of it if we want to generate arbitrary element of this type. Inside this class we have an abstract type Gen a, which is a generator either built in or specified by user. A simple example generator for integers is

  ```
  instance Arbitrary Int where
  arbitrary = choose (-20, 20)
  ```

  Here choose (-20, 20) means choose a random integer between -20 and 20.

- Generating user-defined type.

  Generating user-defined types rely on the user input to keep the tool lightweight. The programmer is able to utilize the gadgets of QuickCheck to construct the generator. One simple example is

  ```
  oneof [return Red, return Blue, return Green]
  ```

  To generate an arbitrary list, we could use liftM and frequency keywords to realize this. For even finer control, QuickCheck allows the programmer to define the function that limits the size of the test input, so that avoid infinite loop without terminating.

- Generating functions.

  Some properties involve function as well. This QuickCheck is able to generate arbitrary functions as well. This is realized by transforming a function to a generator which generates data type depending on the argument value. For this purpose, we define a class called Coarbitrary, which generates arbitrary from its first argument. To define a concrete instance, we need a function called variant, which can constructs a generator based on the random number seed past to it. We use this function to construct instances of coarbitrary.

Case Study. The paper introduces several case studies. Here we demonstrate one of them to show the usage of QuickCheck. Lava [1] is a tool to describe, simulate and formally verify hardware. It could be used to describe the hardware circuits as a function from input signals to output signals. Using QuickCheck, not only the programmers are able to check the property of Lava that is also verifiable inside Lava, we can also check some properties that are very complex to verify inside the Lava. This extends the scale of testing for Lava. Indeed, the authors found some errors inside this tool.

2.2 Java PathFinder

Motivation. This paper tries to utilize model checking and symbolic execution to generate test inputs. The author starts with introducing their previous work: Java PathFinder (JPF) model checker and its extension [3]. After that, we use Red-Black tree as example to see how we could program properties for a red-black tree. Three different methods for generating

test inputs are introduced. The goal of using model checking and symbolic execution is to increase the *code coverage* especially for complex data structures.

Technique. As the background, the paper introduces the following techniques/knowledge

- Symbolic execution and framework in Java PathFinder.

  Symbolic execution uses symbolic values as input instead of real data. It outputs an execution tree. It updates path condition (PC) at each branch of the tree (e.g. PC is $x \leq y$ at this node). If one input condition can never be satisfied (e.g. PC is $x > y$ and $x < y$ at this node), the symbolic execution will not continue for this path.

  In the authors' framework, the normal program is first translated to a special form which includes nondeterminism and path condition support. I was not sure what exactly they are talking about until I see the example in §2.2.4: the paper illustrates how the code is transformed and how the symbolic execution tree is generated based on an example of node implementation. To transform the function, types are replaced with symbolic types; field reads and writes are replaced by get and set methods; etc. We can also see that the symbolic execution generates some branches that we do not need to explore. Note that this could be eliminated if we use precondition (e.g. we first check whether the input does not have a loop).

- Red-black tree and its properties.

  The paper introduces four properties of a red-black tree. They are all expressible in Java language. The paper selects the property that "If a node is red, then both its children are black" as an illustration. By coding this property into the "repOk" function, we prepare ourselves for the test input generation.

The paper then moves on and talks about three types of test input generation.

- Model checking as testing.

  This approach directly applies model checking as a testing method. This method works because model checking, when underapproximating the environment, automatically downgrades to a type of testing. For example, we could use model checker to analyze all the sequences of put and remove calls in the TreeMap library.

- Black-box testing.

  We discuss before (in the example of red-black tree) of how to generate a precondition for a data structure. Here, we generate test inputs that satisfy a precondition for black-box testing. The way we do this is to apply symbolic execution to the code of the preconditions. We then get the constraints in path condition and then use constraint solver to obtain the test input. We can then feed these inputs to the methods we want to test.

  Note that, because we are using black-box testing here, the inputs we obtain do not necessarily cover all the code. On the other hand, this approach works better than brute force, mainly because the symbolic execution will eliminate the branches that cannot be satisfied.

- White-box testing.

  For white-box testing, we symbolically execute the method we want to test. After getting the output of this, we use model checking techniques to compare it with the properties of the testing criterion. There are three special notes here. Firstly, because that the structure could be partially uninitialized, we will be conservative when testing the constraint, i.e.we only say it violates the rule when we are sure that the already initialized fields are breaking the rules. An example is shown in Figure 9. Secondly, in case there is destructive update, the method keeps the mapping between objects with uninitialized fields and when they are initialized for later recovery. Lastly, the authors describe how to solve the constraints to get the actual test input.

Evaluation. The authors provide testing results for the above-mentioned three generators. For model checking, the performance does not go well – it runs out of resource at N=6. It only achieves good performance when testing small input. Black-box approach takes only a minute to generate all the test cases based on the repOk method. The coverage is decent at >80%. Last but not least, white-box has the optimal code coverage in the experiment.

2.3 Randoop

Motivation. The idea of this paper is very unique: instead of keep generating test inputs randomly, or systematically generate all the "useful" samples, Randoop uses a feedback mechanism to utilize the result of the previous (random) test results. By doing this, the tool generates method sequences as the test cases, and is able to find numerous bugs in the existing commercial and open source systems.

Technique. The core of this paper is the algorithm described in Figure 3. The sequence generators take the following as the arguments

- Classes: What are used to create the sequences eventually
- Contracts: A set of rules to test with (there are some default contracts defined already)
- Filters: The filters which are used to process the feedback from the previous round for the future round
- timeLimit: The max time of this method being executed

From the high level, the algorithm works as follows: Firstly, there is a function called *randomPublicMethod* to generate a random method out of the classes in the argument; yet another function called *randomSeqsAndVals* takes care of the random parameter and sequence generation. Then we use an extend operation on the sequence, method, and the variables.

Note that the extend operation is described in detail in §2.1. Essentially, it picks some methods from the previous sequences and combine them together. It as well performs some extra methods after combining them. This could be think of generating a new sequence by mutating the previous sequences. Note that these sequences from which to generate new sequence is filtered, as we will discuss later.

After we get a hold of the new sequence, we evaluate this new sequence. The basic idea is, after executing all the methods specified in the sequence, we check the "contracts" after this series of operations. These contracts could be regarded as the properties of the objects and methods that must been hold whatsoever (e.g. some methods throw no exception, some method throw no exception). The programmer is also empowered to write his own contracts.

Now we get one of the two results: violated or not violated. Here comes the filter: we shall determine which values of the sequence could be used for the future sequence generation. By default there are some filters used. For example, one is called equality. This filter will eliminate all the object that create the same abstract of the previous sequences. By applying filters, we keep generating "useful" and "different" sequences.

Evaluation. The authors compare Randoop with other methods in three experiments. First, the authors show that Randoop has similar coverage compared to systematic test generators on multiple test cases. Besides, Randoop is tested on a few libraries. The authors show that the real strength of Randoop is bug finding: Randoop is able to discover tens of bugs in the existing popular libraries. On the other hand, the systematic testing (after the authors write a driver for it) cannot find anything before it runs out of memory. Also, only using undirected random testing is not enough as well, as JCrasher finds much less bugs compared to Randoop. The authors also illustrate the superiority of the approach when testing inconsistencies between implementations.

3 Comparison and Discussion

In this report, we see the descriptions of the three approaches. After writing and reading this, the difference of the three techniques are very clear:

- The first paper, QuickCheck, is a tool to support random testing generation. The second paper, Java PathFinder, is more focused on a systematic way by combing model checking and symbolic execution. While the third paper, Randoop, kind of sits in the middle – it randomly generates the test cases, but systematically based on the previous test results. For the random input generating, it is easier to implement and scale, while could have low code coverage. On the other hand, systematic way is more heavy-weighted and have good code coverage, which can yet cause trouble in scalability in some cases, as we see in the Randoop paper.

- The random test input generation is usually more lightweight, as intuitively it is fairly simple to write something that generates something "random". As we can see in the QuickCheck paper, the authors' implementation is only a 300-line module. As in the Randoop paper, the core algorithm is also not difficult to understand. Compared to those, the Java PathFinder paper takes me huge amount of time to understand, even superficially (as a non software engineering major).

- Both QuickCheck and PathFinder paper depends on the "properties" of the module being tested. They utilize this property function to either generate test input or perform testing. On the other hand, the Randoop more rely on the "contracts", which are some rules that are "obvious" or "intuitive". However, these rules are extremely

powerful in finding bugs. To find bugs with the previous two works, we need to modify their approach so that they explore bugs instead of just testing the module property. However, this is still not sufficient because the pure random approach has lots of useless test cases, while PathFinder could hardly scale to a complex program.

References

[1] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: hardware design in haskell. In *ACM SIGPLAN Notices* (1998), vol. 34, ACM, pp. 174–184.

[2] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices 46*, 4 (2011), 53–64.

[3] Khurshid, S., Păsăreanu, C. S., and Visser, W. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2003), Springer, pp. 553–568.

[4] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 75–84.

[5] Visser, W., Pasareanu, C. S., and Khurshid, S. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes 29*, 4 (2004), 97–107.