CS1632, Lecture 15: Property-Based Testing

Wonsun Ahn

What is Testing?

- Checking *expected behavior* against *observed behavior*
- What we have been doing so far:
  1. Split the set of input values into equivalence classes
  2. Choose a few representative values from each equivalence class
  3. Write test case for those few values

  … And hope that those few values cover all behavior

- But do they? Are you really confident?

So let’s take a sort function

```
public int[] sort(int[] arrToSort) {
    ...
}
```

Possible test cases

- null
- []
- [1]
- [-1]
- [1, 2, 3, 4, 5]
- [5, 4, 3, 2, 1]
- [-9, 7, 2, 0, -14]
- [1, 1, 1, 1, 1, 1]
- [1, 2, 3, 4 … 99999, 100000, 100001]

At what point would you be satisfied?

- There is a near infinite number of sequences you could test
- Each sequence is a unique equivalence class! (almost)
  - In the sense that each sequence will take a different execution path
  - Each execution path represents a unique behavior
- Verdict: it’s impossible to write enough tests to cover all behavior

- Well.. It’s impossible for a human, but can’t we auto-generate them?

Stochastic Testing

- Stochastic testing: testing using randomly generated input values
  - Note: we are still not testing all input values
  - We are just testing a large number of random values hoping good coverage

- Popularly called “monkey testing” (monkey on the typewriter)
  - Not a good analogy: implies no thought is given to generation of input values
  - Testers should give *a lot* of thought to how input values are generated
  - Values are generated from a distribution, and distribution affects coverage
  - Testers should choose a distribution most likely to uncover defects

Stochastic Testing: Problem

- So now we have a set of random auto-generated input values
- How do we auto-generate test cases out of them?
  - We would also need to add expected behavior to the test cases.
  - As in, an output value for each input value.
  - But how do we auto-generate the output value for each input value?
  - Using the tested method? Yes, if you like circular reasoning. But, no.

What if we tested *properties* of the output values instead?

Property-Based Testing

- Property-Based Testing: testing using properties of output values
  - Does not test output values directly
  - Tests certain properties that must invariably hold in output values
  - These properties are called *invariants*
- Examples of invariants:
  - Program should not crash on input value (obviously)
  - For + operator, if inputs are positive, output must also be positive
  - For + operator, if inputs are negative, output must also be negative
- As you can see, invariants check only a subset of behavior
  - Testing properties of an output is not equivalent to checking its value outright
  - But if you check enough properties, you often can get pretty close

Going back to our sort() example

- What are the invariants?

1. Output array is the same size as input array
2. Every element in input array is in output array
3. No element not in input array is in output array
4. Values in output array are always increasing or staying the same
5. Idempotent - running it again does not change output array
6. Pure – one call of sort() does not impact the next call of sort() in any way

Property-Based Testing

- Advantages
  - Can check behavior without being provided an expected output value
  - Enables stochastic testing
  - Leads programmer to think about invariants and better understand code
- Disadvantages
  - Checking properties of an output value does not guarantee it is correct
  - Hard to use with impure functions where inputs are not clearly specified
    (Hard to come up with invariants when side-effects can change behavior)
  - If used with stochastic testing, tests are not repeatable
    (A pass in one test run does not guarantee a pass in the next run)

QuickCheck

- Presented at ICFP ‘00 in the paper, “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”
- https://www.researchgate.net/publication/2449938_QuickCheck_A_Lightweight_Tool_for_Random_Testing_of_Haskell_Programs
- More popular in functional programming languages
  - Because all functions in functional programming are pure by definition
  - Pure functions are easier to use property-based testing on
- But becoming more mainstream in other languages too

Not just used in functional programming!

- Java: junit-quickcheck
- Ruby: rantly
- Scala: scalacheck
- Python: pytest-quickcheck
- Node.js: node-quickcheck
- Clojure: simple-check
- C++: QuickCheck++
- .NET: FsCheck
- Erlang: Erlang/QuickCheck
- *The only one I couldn't find is a version for PHP.*

What the tester needs to do

- Two simple steps:
    1. Specify the properties of the allowed input
    2. Specify the properties of the output that must hold (invariants)

Example junit-quickcheck tests

```
@Property public void testConcat(String s1, String s2) {
  assertEquals(s1.length() + s2.length(),
               (s1 + s2).length());
}
@Property public void testSqrt(@InRange(minInt=0) int n)
{
  if(n > 1) assertTrue(n > sqrt(n));
}
```

- `@Property`: a property-based test, so multiple random values are passed in
- `@InRange`: constrains range of input values to those acceptable to sqrt()

Write the junit-quickcheck test for sort()

```
@Property public void testSort(int[] arr) {
  int[] result = sort(arr);
  assertEquals(arr.length, result.length);
  …
}
```

Then sit back with a beverage of your choice

- QuickCheck then runs randomized test cases for us!

COMPUTER – DOING HARD WORK!

```
[17, 19, 1] -> [1, 17, 19] OK
[-9, -100] -> [-100, -9] OK
[8, 2, 987, 287, 201] -> [2, 8, 201, 287, 987] OK
[101, 20, 32, -4] -> [-4, 20, 32, 101] OK
[115] -> [115] OK
[2, -9, -9, 1, 2] -> [-9, -9, 1, 2, 2] OK
[8, 3, 0, 4] -> [0, 3, 4, 8] OK
[17, 1009, -2, 413] -> [-2, 17, 413, 1009] OK
[12, 12, 1, 17, -100] -> [-100, 1, 12, 12, 17] OK
[] -> [] OK
...
```

YOU –
lying on
beach
taking
foot selfies!

This is what it sounds like when Invariants fail

```
[17, 19, 1] -> [1, 17, 19] OK
[-9, -100] -> [-100, -9] OK
[8, 2, 987, 287, 201] -> [2, 8, 201, 287, 987] OK
[101, 20, 32, -4] -> [-4, 20, 32, 101] OK
[115] -> [115] OK
[2, -9, -9, 1, 2] -> [-9, -9, 1, 2, 2] OK
[8, 3, 0, 4] -> [0, 3, 4, 8] OK
[17, 1009, -2, 413] -> [-2, 17, 413, 1009] OK
[12, 12, 1, 17, -100] -> [-100, 1, 12, 12, 17] OK
[9, 0, -6, -5, 14] -> [0, -6, -5, 9, 14] FAIL
[] -> [] OK
```

Shrinking

```
[9, 0, -6, -5, 14] -> [0, -6, -5, 9, 14] FAIL
[9, 0, -6] -> [0, -6, 9] FAIL
[-6, -5, 14] -> [-6, -5, 14] OK
[9, 0] -> [0, 9] OK
[0, -6] -> [0, -6] FAIL
[0] -> [0] OK
[-6] -> [-6] OK

Shrunk Failure: [0, -6] -> [0, -6]
```

Shrinking

- Finds the smallest possible failure
- Helps track down actual issue
- A “toy” failure is a great thing to add to a defect report

Think about the levels of automation we achieved, starting from the beginning of the semester!

1. Write and execute tests (manual testing)
2. Write tests, let computer execute
3. Write *invariants*, let computer write tests and execute
    - With shrinking, will even try to track down the problem!

Now Please Read Textbook Chapter 18