Property Based Testing

CS454 AI-based Software Engineering

Shin Yoo | COINSE@KAIST

Let us briefly examine relevant testing ideas.

(Before we discuss PBT in earnest)

- Exhaustive Testing
- Random Testing
- Test Oracle Problem

Exhaustive Testing

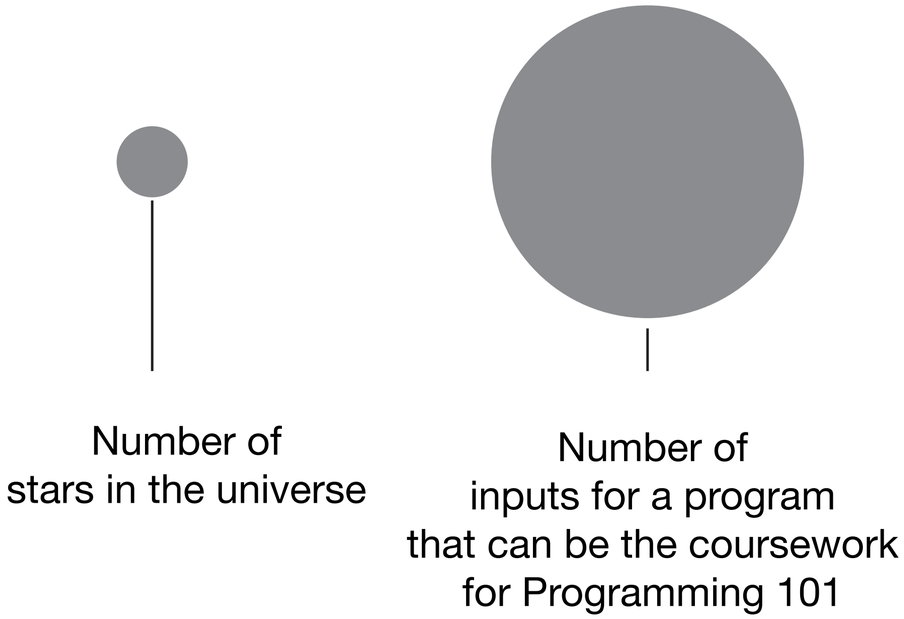
- Testing is a dynamic analysis. Intuitively, it is someone saying that“I know this program is okay because I have tried it myself”.
- Your immediate response should be: “yeah, but, how did you try it? Have you tried this tricky input?”
- The only way to avoid this reaction is: “I have tried EVERYTHING”, i.e., every possible executions.
- Sounds easy?

Exhaustive Testing

- Can we test each and every program with all possible inputs, and guarantee that it is correct every time? Surely then it IS correct.
- In theory, yes - this is the fool-proof, simplest method… or is it?
- Consider the triangle program
  - Takes three 32bit integers, tells you whether they can form three sides of a triangle, and which type if they do.
  - How many possible inputs are there?

Exhaustive Testing

- 32bit integers: between -2^{31} and $2^{31}-1$, there are 4,294,967,295 numbers
- The program takes three: all possible combination is close to 8^{28}
- Approximated number of stars in the known universe is 10^{24}
- Not. Enough. Time. In. The. Whole. World.



“Testing can only prove the presence of bugs, not their absence.”

Edsger W. Dijkstra

Random Testing

- So, all practical testing techniques are input sampling techniques.
- The easiest, and the (sort of) opposite of exhaustive testing would be to sample inputs **randomly**.
- Random testing works better than how it sounds. Sampling randomly means not being constrained by the *happy path* mental model in the developer's head. You may find unexpected behaviour.
- However, there is a little problem… let's first introduce Test Oracles.

Test Oracle

- At the end of execution, we have to check whether the computed results are as expected. This check is called test oracle, and how to (automatically) write them is one of the biggest challenges in software engineering.
  - Implicit Oracles: these are conditions that should always hold, like “program should terminate”, “program should not crash”.
  - Explicit Oracles: these have to be based on user expectations and/or intentions. Very hard to correctly, and formally, capture.

Oracle Problem

- Suppose you can write oracle that can handle any arbitrary input to the program, i.e., the oracle can always correctly decide whether the submitted output is correct or not.
- This implies that, internally, the oracle knows what the correct output should be. In other words, the oracle can compute the correct output.
- If you can compete the correct output, the oracle should have been your program! 🐓 or 🥚?

Oracles for Random Testing

- If inputs are sampled randomly, how can we anticipate the expected output, unless we have fully automated oracle? 🫠
- This is a fundamental wall. There is no solution, just a bunch of (very useful) workarounds.
  - Differential Testing: your are testing a migration, therefore your oracle is the original version (which is fully executable and remains the golden truth).
  - Implicit Oracles only: you are just happy to check for implicit oracles, such as crash. For issues that are closely related to crashes (e.g., memory related vulnerabilities) this is okay (e.g., fuzzing).

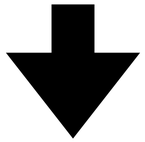
Relaxed Oracle

- Let’s accept that we cannot have the precise expected output. Should we give up testing entirely?
- Perhaps we can write a condition that is tighter than “do not crash” but more relaxed than 1-to-1 input-output correctness matches, so that it has to be met by all program executions?

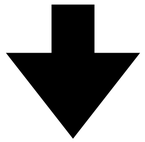
Property Based Testing (PBT)

- Originally invented for Haskell under the name QuickCheck
- PBT consists of three core parts:
  - Assertions that check logical properties a function should satisfy
  - Generators that randomly construct (sample) inputs that meet the valid input specifications
  - Shrinkers that minimise the violating counterexamples

```
def test_reverse():
    assert reverse([3, 2, 1]) == [1, 2, 3]
    assert reverse([]) == []
    assert reverse([42]) == [42]
```



```
def test_reverse(L: list):
    reverse(reverse(L)) == L
    length(reverse(L)) == length(L)
```



```
from hypothesis import given, strategies  as st

@given(st.lists(st.integers()))
def test_reverse(x):
    reverse(reverse(x)) == x
    length(reverse(x)) == length(x)
```

Hypothesis

- We are going to use Hypothesis in our examples.
- Hypothesis is an easy-to-use PBT framework for Python: https://github.com/HypothesisWorks/hypothesis-python

Property Based Oracles

- Suppose we want to test a Python implementation of the absolute function.
- Traditional, example-based oracle requires test engineers to provide input-output pairs
- For complicated functions, this is tedious and error-prone

```
def abs_function(x):
  if x < 0:
    return -x
  else:
    return x
```

```
def test_important_func():
  assert abs_function(1) == 1
  assert abs_function(0) == 0
  assert abs_function(-1) == 1
```

Property Based Oracles

- Hypothesis allows test engineers to write parameterised unit tests.
- Instead of specifying a concrete input-output pair, use a parameterised input and describe the expected properties of the output using the input symbol.
- For example, for any integer x, abs(x) should be greater than or equal to 0.

```
def abs_function(x):
  if x < 0:
    return -x
  else:
    return x
```

```
def test_important_func(x):
  assert abs_function(x) >= 0
```

Some Generic Patterns

- Roundtrip: $f^{-1}(f(x)) = x$
- Idempotence: `sort(sort(l)) == sort(l)`
- Reference/Differential: $f_{own}(x) = f_{reference}(x)$
- Domain-specific Invariants

Input Generator

- Hypothesis uses Python annotation to parameterise the input.
- During the actual test execution, the parameterised input is randomly sampled.
- Anything that violates the assertions will be reported

```
from hypothesis import given
from hypothesis import strategies as st
import unittest

def abs_function(x):
  if x < 0:
    return -x
  else:
    return x

class TestAbs(unittest.TestCase):
   @given(x = st.integers())
   def test_abs_function(self, x):
      assert abs_function(x) >= 0

if __name__ == '__main__':
    unittest.main()
```

Using Hypothesis

- GitHub Repository: https://github.com/HypothesisWorks/hypothesis
- Installation: use PIP (`pip install hypothesis`)
- Documentation is available from: https://hypothesis.readthedocs.io

Hands-on

- Clone the following:
- It contains four subproblems, all requiring you to write Hypothesis test cases.

PBT: Pros and Cons

- Can be super strong when done right
- Makes you think about what the code should do much harder than the conventional, example-based testing
- For very complicated input type, writing the input generator becomes too difficult: eventually test cases require test cases for them.

Implications for Agentic AI Era

- Given the variability and the volume of output generated by agents, 1-to-1 mapping oracles will be difficult to maintain.
- Perhaps PBTs based on specification is easier to apply to the agentic loop?
  - LLMs can already write required properties fluently. However, this is one thing that really needs to be human verified.