Property Based Testing

CS453 Automated Software Testing

Shin Yoo | COINSE@KAIST

Recall Random Testing + Test Oracles

- Random exploration of input space is a good thing to do (assuming no a prior knowledge of the input space)
- However, typically we can only write example-based oracles, i.e., one-to-one mapping between an input and the expected output!
- What are the alternatives?
  - Formal specification: Yes. However, fully automated & executable spec that can produce expected output is 1) very expensive and 2) not very scalable or generalisable.
  - Alternative Implementation: Perhaps we can implement the target system that is slower or less efficient - so that we can still check the output? This is sometimes possible.
  - Reference Implementation: We now to “differential testing” randomly. But what if no such reference exists?

Relaxed Oracle

- Let’s accept that we cannot have the precise expected output. Should we give up testing entirely?
- Perhaps we can write a more relaxed condition that has to be met by all program executions?

An Example

```
import datetime

def check_age(birthday, today):
    return nineteen_day(birthday) <= today

def nineteen_day(birthday):
    return birthday + datetime.timedelta(days=365 * 19)

print(check_age(datetime.datetime(1977, 5, 14), datetime.datetime.today()))
```

Property Based Testing

- The PBT idea is originally from the QuickCheck for Haskell, developed in 1999 (http://www.cse.chalmers.se/~rjmh/QuickCheck/)
- It aims to attack the following problems:
  - Random testing can be highly effective, but is weak against structured inputs.
  - Automated oracle is essential for effective random testing.
  - Structural coverage itself does not guarantee anything.

Property Based Testing

- PBT is the combination of the following:
  - Property based oracles, instead of input-output pair oracles
  - Test input generators that combine low level random generators to build a input generator for complex structured inputs
- Using these two, PBT randomly samples complex, structured inputs, and reports anything that violates the given property.

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
```

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: `git@github.com:coinse-classroom/cs453-pbt-exercise.git`
- 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.