04834580 Software Engineering (Honor Track) 2024-25

Property-Based Testing

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Parameterized Tests

Parameterized tests make it possible to run a test multiple times with different arguments. An example with JUnit:

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

“racecar”, “radar”, ... are inputs, `isPalindrome` is a property.

```python
from parameterized import parameterized
from my_program import my_function

class TestMyFunction(unittest.TestCase):
    @parameterized.expand([
        ["foo", "a", "a",],
        ["bar", "b", "c"],
        ["lee", "d", "e"],
    ])
    def test_a_property_of_my_function(self, name, a, b):
        result = my_function(a)
        self.assertEqual(result, b)
```

Definition

Property-based testing works by testing an executable predicate (a property) on a stream of randomly generated inputs.

Popular implementations:

- QuickCheck [1] for Haskell
- Hypothesis for Python
- Proptest for Rust

Hypothesis

Generates random integers for the argument n when testing:

```
from hypothesis import given, strategies as st

@given(st.integers())
def test_is_integer(n):
    assert isinstance(n, int)
```

assume filters out unwanted outputs:

```
from hypothesis import assume, given, strategies as st


@given(st.integers(), st.integers())
def test_floor_division_lossless_when_b_divides_a(a, b):
    assume(b != 0)
    assume(a % b == 0)
    assert (a // b) * b == a
```

Defining custom strategies through composition:

```
from hypothesis import strategies as st

@st.composite
def sums_to_one(draw):
    l = draw(st.lists(st.floats(0, 1)))
    return [f / sum(l) for f in l]
```

Hypothesis

Limitations of assume:

```
from hypothesis import HealthCheck, settings, assume, given, strategies

@given(st.lists(st.integers()))
@settings(suppress_health_check=[HealthCheck.filter_too_much])
def test_sort_correct(lst):
    assume(len(lst) > 3 and lst[3] == 12345)
    assert False
```

Defining custom strategies through composition:

```
from hypothesis import strategies as st

@st.composite
def sums_to_one(draw):
    l = draw(st.lists(st.floats(0, 1)))
    return [f / sum(l) for f in l]
```

References I

[1] Koen Claessen and John Hughes.
Quickcheck: a lightweight tool for random testing of haskell programs.
In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.