

# Why property-based testing matters

Pedro Vasconcelos  
pbvascon@fc.up.pt

DCC/FCUP & LIACC



18th November 2024

# Overview

- ▶ A long-standing challenge for software engineering is ensuring software correctness
- ▶ Formal verification is (still) expensive and rarely used
- ▶ *Tests* are the most commonly used practical technique
- ▶ *Unit tests* are the industry-standard for verification “in the small”

# This talk

- ▶ *Property-based testing*: an automatic testing alternative to unit tests
- ▶ A “lightweight” formal method
- ▶ Available for many programming languages
- ▶ Many successful applications in open-source and some industrial projects
- ▶ But still not commonly taught and under-utilized in practice

Slides and demo code:

<https://github.com/pbv/why-pbt-matters>

# “Lightweight” formal method?

## Formal method

A mathematically rigorous technique for validating the actual behaviour of a program against a description of desired behaviours

## Lightweight formal method

One that can be applied successfully by someone who doesn't fully understand it 😊

# “Lightweight” formal method?

supports automation

## Formal method

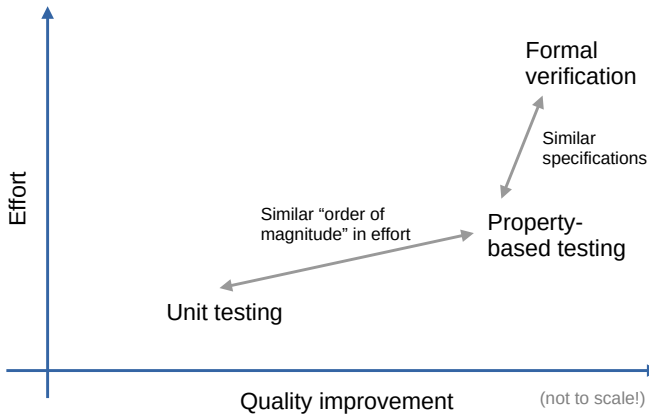
A mathematically rigorous technique for validating the actual behaviour of a program against a description of desired behaviours

requires automation

## Lightweight formal method

One that can be applied successfully by someone who doesn't fully understand it 😊

# PBT vs unit tests vs formal verification



# Unit tests

- ▶ Code fragments for testing functions, classes, libraries, etc.
- ▶ Express the expected outputs for specific combinations of inputs
- ▶ Example: testing an integer square root function in Python

```
def test_isqrt():  
    assert isqrt(0) == 0  
    assert isqrt(2) == 1  
    assert isqrt(4) == 2  
    assert isqrt(5) == 2  
    assert isqrt(9) == 3
```

# Problems with unit tests

Cognitive bias:

- ▶ how can we include an edge case in the tests that we didn't consider in the code?

Poor scaling:

- ▶ a few unit tests per feature
- ▶ for  $n$  features,  $O(n)$  unit tests
- ▶ but testing *interactions* between features requires  $O(n^2)$ ,  $O(n^3)$ , ... unit tests



# Problems with unit tests

Cognitive bias:

- ▶ how can we include an edge case in the tests that we didn't consider in the code?

Poor scaling:

- ▶ a few unit tests per feature
- ▶ for  $n$  features,  $O(n)$  unit tests
- ▶ but testing *interactions* between features requires  $O(n^2)$ ,  $O(n^3)$ , ... unit tests

Solution:

*"Don't write tests — generate them!"*

John Hughes, co-author of the *QuickCheck* PBT library



# Property-based testing

- ▶ Write *properties* instead of specific tests
  - ▶ should be universal, i.e. hold for all values
  - ▶ should define the expected behaviour for *all* cases
- ▶ Specify *generators* for the inputs
- ▶ The testings framework runs the property with a large number of inputs
  - ▶ testing fails if a **counter-example** is found
  - ▶ otherwise, testing succeeds

## Property-based testing (cont.)

- ▶ QuickCheck (2000): first PBT library (for Haskell)
- ▶ Other implementations:
  - PropEr for Erlang
  - ScalaCheck for Scala
  - Hypothesis for Python
  - FsCheck for F#
  - JUnit-QuickCheck for Java
  - RapidCheck for C++

Many others: <https://en.wikipedia.org/wiki/QuickCheck>

## Example property

What can we say about the integer square root function?

## Example property

What can we say about the integer square root function?

Let  $n$  be an arbitrary non-negative number; let  $r = \text{isqrt}(n)$ ; then

$$r \geq 0 \wedge r^2 \leq n \wedge (r + 1)^2 > n$$

i.e.  $r$  should be *largest non-negative integer* such that  $r^2 \leq n$ .

## Example property

What can we say about the integer square root function?

Let  $n$  be an arbitrary non-negative number; let  $r = \text{isqrt}(n)$ ; then

$$r \geq 0 \wedge r^2 \leq n \wedge (r + 1)^2 > n$$

i.e.  $r$  should be *largest non-negative integer* such that  $r^2 \leq n$ .

In Python:

```
from hypothesis import given
import hypothesis.strategies as st
@given(st.integers(min_value=0))
def test_isqrt(n):
    r = isqrt(n)
    assert r >= 0 and r**2 <= n and (r+1)**2 > n
```

## Example property

What can we say about the integer square root function?

Let  $n$  be an arbitrary non-negative number; let  $r = \text{isqrt}(n)$ ; then

$$r \geq 0 \wedge r^2 \leq n \wedge (r + 1)^2 > n$$

i.e.  $r$  should be *largest non-negative integer* such that  $r^2 \leq n$ .

In Python:

```
from hypothesis import given
import hypothesis.strategies as st
@given(st.integers(min_value=0))
def test_isqrt(n):                                # for all n
    r = isqrt(n)
    assert r >= 0 and r**2 <= n and (r+1)**2 > n
```

## Example property

What can we say about the integer square root function?

Let  $n$  be an arbitrary non-negative number; let  $r = \text{isqrt}(n)$ ; then

$$r \geq 0 \wedge r^2 \leq n \wedge (r + 1)^2 > n$$

i.e.  $r$  should be *largest non-negative integer* such that  $r^2 \leq n$ .

In Python:

```
from hypothesis import given
import hypothesis.strategies as st
@given(st.integers(min_value=0)) # non-negative integer
def test_isqrt(n):               # for all n
    r = isqrt(n)
    assert r >= 0 and r**2 <= n and (r+1)**2 > n
```



## Example property

What can we say about the integer square root function?

Let  $n$  be an arbitrary non-negative number; let  $r = \text{isqrt}(n)$ ; then

$$r \geq 0 \wedge r^2 \leq n \wedge (r + 1)^2 > n$$

i.e.  $r$  should be *largest non-negative integer* such that  $r^2 \leq n$ .

In Python:

```
from hypothesis import given
import hypothesis.strategies as st
@given(st.integers(min_value=0))  # non-negative integer
def test_isqrt(n):                # for all n
    r = isqrt(n)
    assert r >= 0 and r**2 <= n and (r+1)**2 > n  # assertion
```

## Example property

What can we say about the integer square root function?

Let  $n$  be an arbitrary non-negative number; let  $r = \text{isqrt}(n)$ ; then

$$r \geq 0 \wedge r^2 \leq n \wedge (r + 1)^2 > n$$

i.e.  $r$  should be *largest non-negative integer* such that  $r^2 \leq n$ .

In Python:

```
from hypothesis import given
import hypothesis.strategies as st
@given(st.integers(min_value=0))    # non-negative integer
def test_isqrt(n):                  # for all n
    r = isqrt(n)
    assert r >= 0 and r**2 <= n and (r+1)**2 > n    # assertion
```

(Cue demo.)

# Properties in Hypothesis

- ▶ Properties are *functions*...
- ▶ ...that should fail if the expected condition is not met
- ▶ Arguments are *universally quantified*
- ▶ For each property:
  - ▶ test with a large number of random values (100 by default)
  - ▶ generated using *strategies* (defined by `@given`)
- ▶ Module `hypothesis.strategies` provides:
  - ▶ *predefined strategies* for basic types
  - ▶ methods for *modifying* and *combining* strategies

# Strategies

`floats()` generate floating-point numbers  
`integers()` generate integers  
`booleans()` generate logical values  
`text()` generate Unicode strings  
`lists(s)` lists of elements given by strategy `s`  
... many others

We can also:

- ▶ customize strategies using *parameters* (e.g. `min_value`)
- ▶ modify strategies by *mapping* and *filtering*
- ▶ combine strategies using *combinators*

## Generating data

```
>>> integers().example()
```

```
848041
```

```
>>> lists(integers(min_value=0, max_value=100)).example()
```

```
[2, 29, 54, 66, 1, 27, 77, 81, 51, 18, 18]
```

```
>>> lists(integers().map(lambda x:x*2)).example()
```

```
[6668, -38, 1081651134, -6590]
```

```
>>> lists(integers()).map(sorted).example()
```

```
[-6913, -59, 37, 77, 90, 25088]
```

```
>>> lists(one_of(integers(), booleans())).example()
```

```
[True, True, -1318, True, True, -46, -46, True, -46]
```

## Another example

Let's test the interaction between *list reverse* and *append*.

Consider  $x, y$  two arbitrary lists:

$$\text{reverse}(x + y) = ???$$

## Another example

Let's test the interaction between *list reverse* and *append*.

Consider  $x, y$  two arbitrary lists:

$$\text{reverse}(x + y) = \text{reverse}(y) + \text{reverse}(x)$$

Example:

$$\begin{aligned}\text{reverse}([1, 2] + [3, 4]) &= \text{reverse}([3, 4]) + \text{reverse}([1, 2]) \\ &= [4, 3] + [2, 1] \\ &= [4, 3, 2, 1]\end{aligned}$$

## Testing with lists of integers

```
intlist = st.lists(st.integers())  
@given(intlist, intlist)  
def test_reverse_append(x, y):  
    assert reverse(x + y) == reverse(x) + reverse(y)
```



## Testing with lists of integers

```
intlist = st.lists(st.integers())
@given(intlist, intlist)
def test_reverse_append(x, y):
    assert reverse(x + y) == reverse(x) + reverse(y)
```

```
pytest basic.py -k test_reverse_append
```

```
===== FAILURES =====
----- test_reverse_append -----
```

```
x = [0], y = [1]
```

What happened...?

# Checking expectations

- ▶ We've written the property incorrectly!

```
assert reverse(x + y) == reverse(x) + reverse(y)
```

instead of

```
assert reverse(x + y) == reverse(y) + reverse(x)
```

- ▶ Hypothesis found a counter-example for the wrong property:

$$\text{reverse}([0] + [1]) \neq \text{reverse}([0]) + \text{reverse}([1])$$

- ▶ This is the *smallest* counter-example that falsifies the property
- ▶ Hypothesis *always* find this counter-example regardless of random generation!

# Shrinking

- ▶ Hypothesis attempts to simplify counter-examples before presenting; e.g.:
  - ▶ removing elements from the lists
  - ▶ shrinking elements inside the lists
- ▶ This is useful to remove “noise” from randomly generated data
- ▶ For the previous example we obtain the *minimal* counter-example
- ▶ In general, shrinking only finds a *local minimum*

## A real-world example

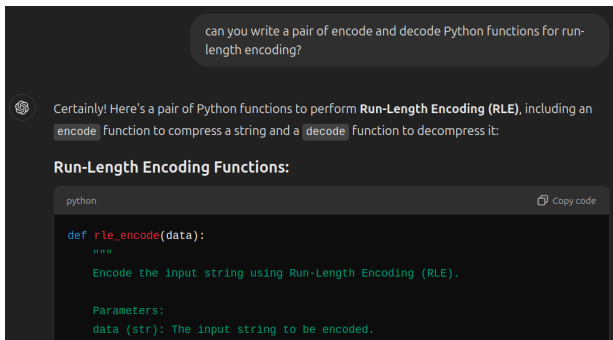
- ▶ Erlang code for SMS text packing at Ericsson
- ▶ 7-bit characters transmitted using 8-bit *bytes*
- ▶ GSM standard: pack 8 caracteres into 7 bytes
- ▶ Two functions (translated to Python):  
    `pack(seq: bytes) -> bytes`  
    `unpack(seq: bytes) -> bytes`
- ▶ Roundtrip property: `unpack` is the inverse of `pack`

$$\text{unpack}(\text{pack}(\text{seq})) = \text{seq}, \text{ for all } \text{seq}$$

## A real-world example (cont.)

- ▶ John Hughes's company (QuviQ) found a subtle bug affecting strings of length multiple of 8 ending in a `\NUL` character
- ▶ The code had been unit tested and was in production

# Validating AI generated code



- ▶ Let us use Hypothesis to validate AI generated code
- ▶ Problem: write a pair of encode/decode functions for **run-length encoding**

'AAABBCAAAAA'  $\xrightarrow{\text{encode}}$  'A3B2C1A5'  
 $\xleftarrow{\text{decode}}$

# Testing ChatGPT solution

“Roundtrip” property i.e. decode is the inverse of encode

```
@given(st.text())  
def test_decode_encode_1(s):  
    assert rle_decode(rle_encode(s)) == s
```

# Testing ChatGPT solution

“Roundtrip” property i.e. decode is the inverse of encode

```
@given(st.text())
def test_decode_encode_1(s):
    assert rle_decode(rle_encode(s)) == s
```

```
pytest rle_tests.py -k test_decode_encode_1
```

```
===== FAILURES =====
```

```
----- test_decode_encode_1 -----
```

```
s = '01'
```

The solution doesn't work for strings with digits!



# Characterizing failure

We can check if the generated code works when the string contains no digits:

```
no_digits = st.characters(exclude_categories='N')
@given(st.text(alphabet=no_digits))
def test_decode_encode_2(s):
    assert rle_decode(rle_encode(s)) == s
```

- ▶ The property now passes the default 100 tests
- ▶ We could now try to fix the solution for digits
- ▶ But first: perform some statistics on the testing data

## Characterizing test data

```
def longest_count(s: str) -> int:
    "Compute the maximum length of repeated chars."
    ...

@given(st.text(alphabet=no_digits))
def test_decode_encode_3(s):
    event(f'longest = {longest_count(s)}')
    assert rle_decode(rle_encode(s)) == s
```

## Characterizing test data (cont.)

```
pytest rle_tests.py -k test_decode_encode_3 \  
    --hypothesis-show-statistics
```

```
===== Hypothesis Statistics =====  
    * 75.70%, longest = 1  
    * 22.69%, longest = 2  
    * 1.00%, longest = 0  
    * 0.60%, longest = 3
```

- ▶ 75% of the test cases had no repeats
- ▶ 22% had maximum of 2 repeats
- ▶ No test case had more than 3 repeats
- ▶ Why? Because `text()` chooses each character independently

# Improving test data generation

We will use **combinators** to write a strategy that:

1. generates a long sequence of a *single repeated* character;
2. *or* generates text as before.

```
many_no_digits
  = st.builds(lambda count, char: count*char,
              st.integers(min_value=0, max_value=20),
              no_digits)

combined = st.one_of(many_no_digits,
                     st.text(alphabet=no_digits))
```

## Improving test data generation (cont.)

This gives much better test data distribution:

* 35.69%, longest = 1	* 2.82%, longest = 14
* 12.90%, longest = 2	* 2.82%, longest = 17
* 10.48%, longest = 3	* 1.81%, longest = 9
* 5.44%, longest = 0	* 1.61%, longest = 10
* 4.23%, longest = 20	* 1.61%, longest = 13
* 4.23%, longest = 6	* 1.41%, longest = 15
* 3.63%, longest = 4	* 1.21%, longest = 19
* 3.23%, longest = 7	* 0.60%, longest = 5
* 3.02%, longest = 8	* 0.40%, longest = 18
* 2.82%, longest = 12	






# Conclusion

- ▶ PBT philosophy: write *properties* and *generate* tests
- ▶ Lightweight: implemented as libraries
- ▶ Flexible: domain-specific languages (DSLs) for writing data generators and properties
- ▶ Scales to realistic software
- ▶ Couples executable specifications with code
- ▶ Useful for communicating expectations among developers
- ▶ Useful for finding subtle bugs in complex systems

# Challenges

- ▶ Writting *effective* properties
  - ▶ training software engineers to think about pre- and post-conditions, invariants, etc.
  - ▶ universities can play a significant role here
  - ▶ helping industry adopt a higher-skill technology
- ▶ PBT works best with software that is well structured
- ▶ Design systems around properties and not the other way around

# References

-  K. Claessen and J. Hughes. *QuickCheck: A lightweight tool for random testing of Haskell programs*, ACM ICFP 2000
-  D. R. MacIver *et al.* *Hypothesis: A new approach to property-based testing*, The Journal of Open Source Software, 2019. See also <https://hypothesis.readthedocs.io/>
-  T. Arts, J. Hughes and J. Johansson *Testing Telecoms Software with Quviq QuickCheck*, ACM Workshop on Erlang, 2006
-  T. Arts, J. Hughes, U. Norell and H. Svensson, *Testing AUTOSAR software with QuickCheck*, IEEE ICSTW 2015
-  H. Goldstein, *et al.* *Property-Based Testing in Practice*, IEEE/ACM ICSE 2024



# Extra slides

# Writing properties

**equivalence**  $f(x) = f_{spec}(x)$  e.g.  $f(x)$  is an optimized implementation and  $f_{spec}(x)$  is a reference implementation.

**idempotency**  $f(f(x)) = f(x)$

**inverse**  $g(f(x)) = x$

**associativity**  $f(x, f(y, z)) = f(f(x, y), z)$

**commutativity**  $f(x, y) = f(y, x)$

**right identity**  $f(x, zero) = x$

**left identity**  $f(zero, x) = x$

Hypothesis can write these kind of properties for you — see `hypothesis.ghostwriter`.

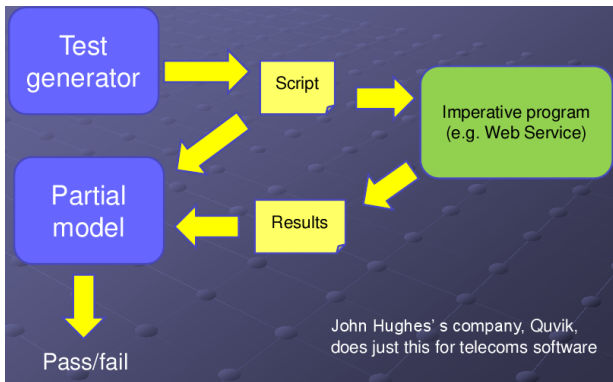
# Stateful programs

We can also use PBT to test programs that:

1. modify state;
2. read and write files;
3. use network services, databases, etc.

# Testing stateful programs

- ▶ Generate sequences of commands
- ▶ Specify behaviour using a functional model (state machine)
- ▶ Compare the execution against the model



# Industrial use example

- ▶ Ericsson Media proxy (Java and C++)
- ▶ Establish telephony connection through a firewall
- ▶ Tested with Erlang QuickCheck (Quviq.com)
- ▶ Adding and removing participants in a call
- ▶ Random counterexample with 160 commands
- ▶ Shrunk automatically to 7 commands

