Property-based Testing: Getting Computers to Test Your Code For You

Tunan Shi

CompSci Student Talks, Churchill College, Cambridge

17 November 2021

Software safety

- You want your code to behave the way that you expect it to
- Especially for safety-critical systems
  - Medical life-support systems
  - Control systems for nuclear reactors
  - Command and control systems in spacecraft
  - Automated trading programs

Example

Figure: *Wikimedia Commons*

Methods for software safety

- Using a language with a type system designed for safety
- Formal verification
- Testing ← our focus

We will get back to the first two later!

Review of unit (example-based) testing

- Writing examples of inputs and expected output to test each function you write
- Small $\rightarrow$ very easy to debug
- Easily tell when your code doesn't output the correct answer
- System complexity $\rightarrow$ much more time consuming to write
- Easy to miss failing cases from human error
- Kind of boring and brainless a lot of the time

Generative (random) testing

- Throw large, random cases at our code?

The Oracle Problem

Even if we can generate large, high quality test cases, how would we be able to tell if the code's output was correct?

We introduce *property-based testing* as a testing method which combines the best of both worlds.

Programming language for the talk

- Most talks and articles on property-based testing use some sort of FP language (including the original paper [1], which used Haskell)
- This testing method not exclusive to FP
- Implementations available in a lot of languages (Haskell, Erlang, OCaml, Python, F#, C++, Prolog, Java etc.)
- We will use Python 3 with the `hypothesis` library

What is a property-based test?

Each test is meant to programmatically check if a function behaves in a certain way.

Example

```
def my_reverse(xs):
    # My implementation...
```

A property: For every possible list of integers `xs`, reversing a list twice will always give back the original list.

```
from hypothesis import given, strategies as st

@given(st.lists(st.integers()))
def test_reversing_twice_gives_same_list(xs):
    assert xs == my_reverse(my_reverse(xs))
```

Let the testing begin

- `hypothesis` will begin generating lists of integers to test
- `hypothesis` marks the test as passed after a lot (e.g. 100) of passed tests
- *What if a test case is found that breaks the property?*
- *Give it back to the developer, right?*

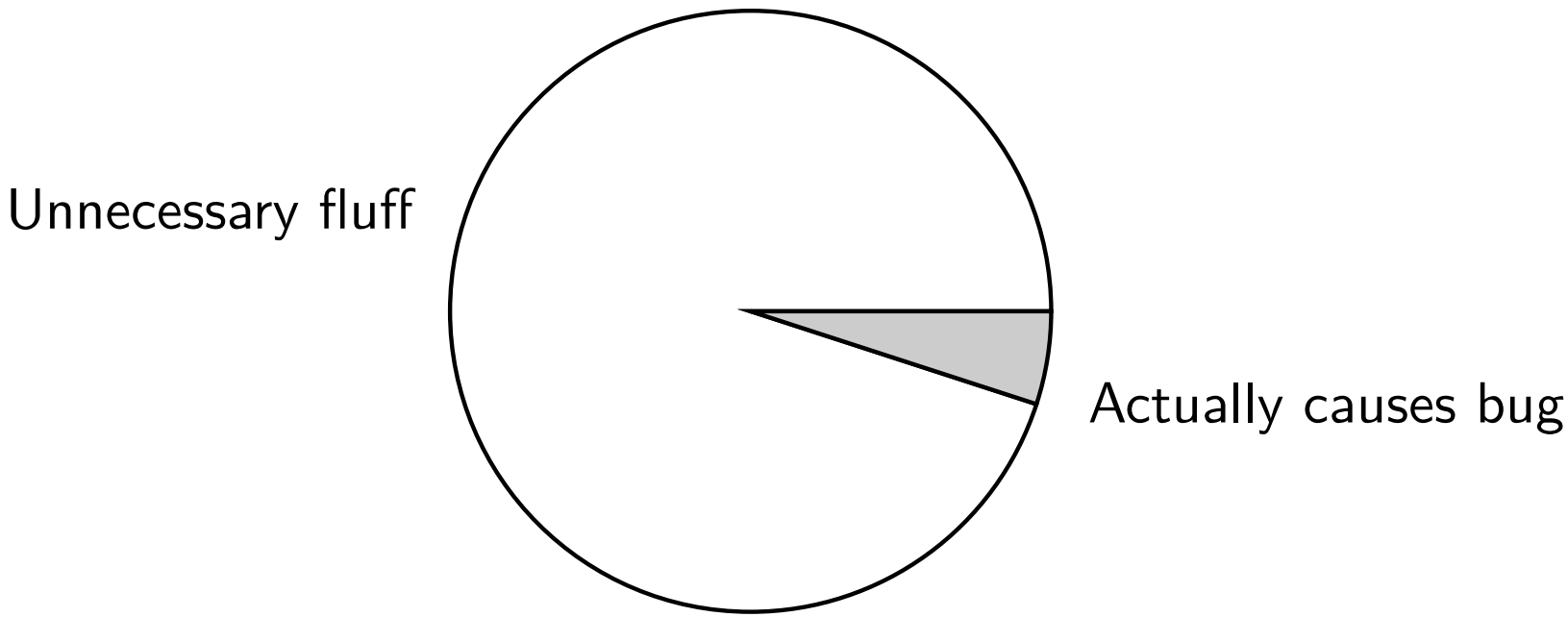
Let the testing begin

```
Falsifying example:
test_reversing_twice_gives_same_list(
    xs=[929, -31594, -30016, 22329, 6252,
    2677217053071645725, -20903, 58, -36,
    -16, 5699505368379265500],
)
Traceback (most recent call last):
  File "pbt.py", line 12, in [...]
    assert xs == my_reverse(my_reverse(xs))
AssertionError
```

Dealing with large unwieldy cases

- Generate only smaller cases instead?
  - But larger cases are more likely to detect bugs in code...

Observation

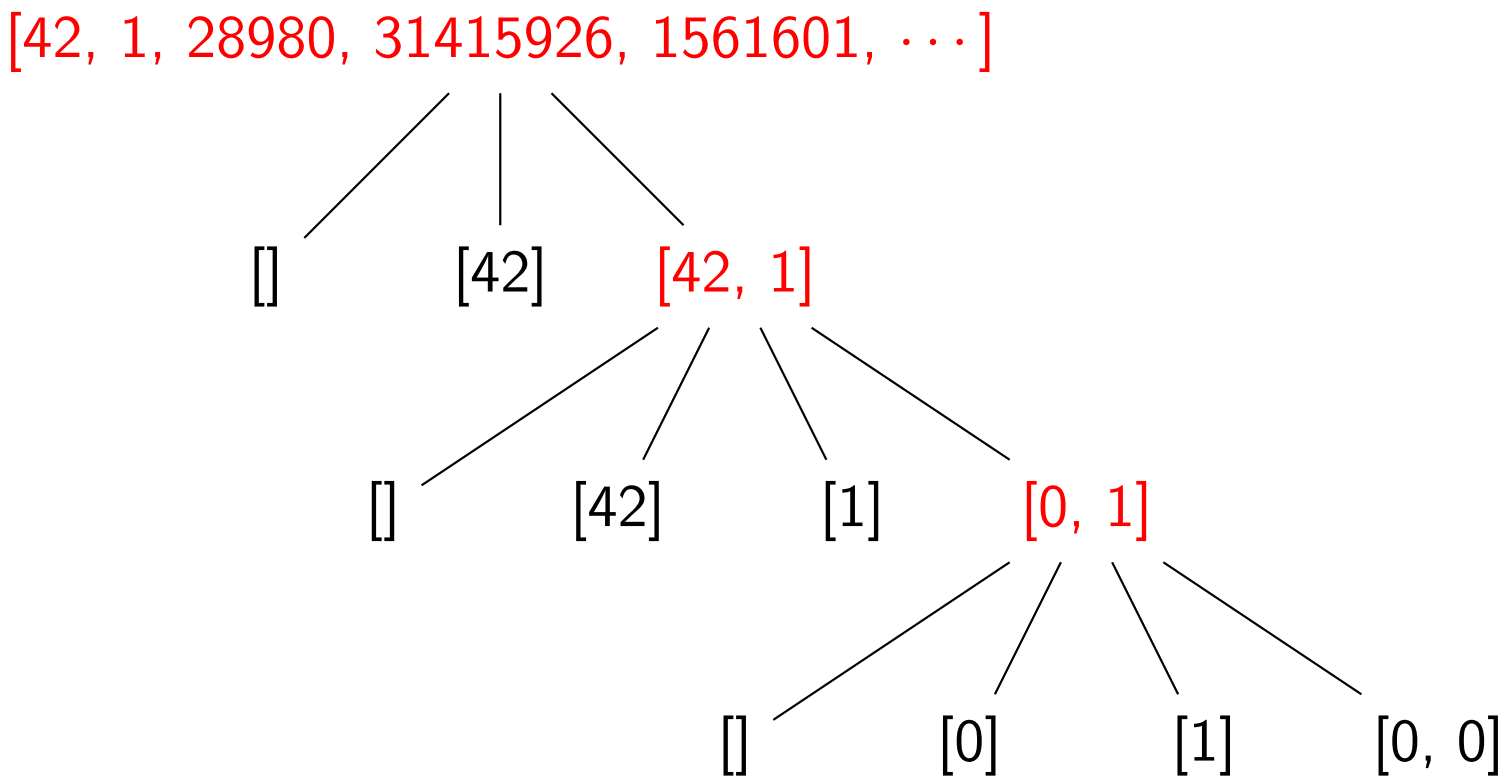


Shrinking our test case

1. Generate alternate (smaller) test cases
2. Run each alternate case against the original function to check if it fails
3. As soon as one fails, use it to generate even smaller alternate cases
4. If none of the alternate cases fail, stop

Like a “greedy hill climbing” algorithm

Shrinking our test case



Shrunk counterexample: [0, 1]

Shrinking our test case

Let's use it on our Python code...

```
Falsifying example:
test_reversing_twice_gives_same_list(
    xs=[0, 0, 1],
)
Traceback (most recent call last):
  File "pbt.py", line 12, in [...]
    assert xs == my_reverse(my_reverse(xs))
AssertionError
```

Debugging

So, what was my buggy implementation?

```
def my_reverse(xs):
    if len(xs) == 0:
        return []
    result = xs[1:] + xs[:1]
    return result
```

How shrinking works

- Shrinking is automatically implemented by libraries for most common types
- Getting to know the underlying algorithm will help guide your decisions on how best to use PBT
- So, how does one generate counterexamples?

Shrinking an integer

- What's the simplest example of an integer you can think of?
- Given two integers, how should we determine if one is simpler than the other?

Possible algorithm

Suppose the integer to shrink is n. Then, try:

- 0
- $\left\lfloor \frac{n}{2} \right\rfloor$
- If $n < 0$, then
  - $|n|$
  - $n + 1$
- If $n > 0$, then $n - 1$

Other shrinking examples

Example: List of integers

1. Try the empty list
2. Take one half of the list
3. Remove a random element of the list
4. Shrink one of the integers of the list

Example: Binary search tree

1. Only the root node
2. Only the left/right subtree
3. Remove any one of the descendant subtrees
4. Shrink any one of the elements

Alternatives to greedy shrinking

Clearly, the "greedy hill-climb" algorithm can be improved

Improvements

- Generate a few large failing cases before shrinking them in parallel
- Once unshrinkable, generate failing cases around same size and try shrink those instead
- Small amounts of backtracking to see if you can find a better shrink

In most cases, the simple greedy hill-climb will give you a counterexample small enough to work with.

How not to generate test cases

Example

Generating random lists of (key, value) updates to test your dictionary:

```
[('\x1cæÙ\U0003cba9\x859nôÕ', '¹'),
('\U0010c79fD¬+∅ Z\x0b', '³\x00øÚûJ5\U000d4ee4oªq'),
('', ':\x9e=\U0005fa13\x9c'),
('\x12»pb\U000351e9È\U00102e28\x82', ''),
('Ñ\x8e}*', '&£'),
('\U000c33e7\x02Üä\U0005627c\U000de36f', 'AS')]
```

What can we see?

Writing better generators

How do we give the dictionary a suitable number of key-value collisions?

Possible strategy

Create a random list of at least 5 strings and then for all of our key queries, pick only from the list of strings

Writing the generator

```
@st.composite
def gen_tests_for_dict(draw):
    key_strategy = st.lists(st.text())
    keys = draw(key_strategy)
    assume(len(keys) >= 5)
    return draw(st.lists(
        st.tuples(st.sampled_from(keys), st.text())
    ))
```

How do I write good properties?

It is hard to come up with good properties to test your code

```
def test_reversing_twice_gives_same_list(xs):
    assert xs == my_reverse(my_reverse(xs))
```

Counterexample?

```
def my_reverse(xs):
    return xs
```

Summed up by John Hughes [3], there are several design patterns for writing properties

Invariant (sanity) checks

- When testing data structures, check that your data structure is well-formed
- Easy entry-level check that indicates catastrophic failure if broken

Invariant check example: Binary search tree

```
class BST:
    def __init__(self, value, left, right):
        self.value = value
        self.left = left
        self.right = right

    def insert(self, elem):

    def to_list(self):

@st.composite
def gen_bsts(draw):
    # Generate BSTs of integers
```

Invariant check example: BST checker

```
def check_well_formed(bst):
    if bst.left is not None:
        assert all(x < bst.value
            for x in bst.left.to_list())
        check_well_formed(bst.left)
    if bst.right is not None:
        assert all(x > bst.value
            for x in bst.right.to_list())
        check_well_formed(bst.right)
```

Invariant check example: BST tests

```python
@given(gen_bsts())
def test_bst_generation_is_valid(bst):
    # Check generator satisfies invariant
    check_well_formed(bst)

@given(gen_bsts(), st.integers())
def test_bst_insert_is_valid(bst, val):
    # Check insert operation satisfies invariant
    bst.insert(val)
    check_well_formed(bst)
```

Postcondition checks

- For functions or operations on data structures, what about the result or output is *always true*?
- Extension on sanity checks

Postcondition check example: Dictionary durability

```
@given(st.dictionaries(st.text(), st.text()),
    st.text(), st.text())
def test_dict_insert_findable(dictionary, key, value):
    # If I insert (key, value) into the dictionary, I
    # should be able to find it again
    dictionary[key] = value
    assert dictionary[key] == value
```

Postcondition check example: Function check

```
def gcd(x, y):
    # Only works on positive numbers

@given(st.integers(min_value=1),
       st.integers(min_value=1))
def test_gcd_outputs_common_divisor(x, y):
    # gcd(x, y) must actually be a common
    # factor of x and y
    g = gcd(x, y)
    assert x % g == 0
    assert y % g == 0
```

Metamorphic properties

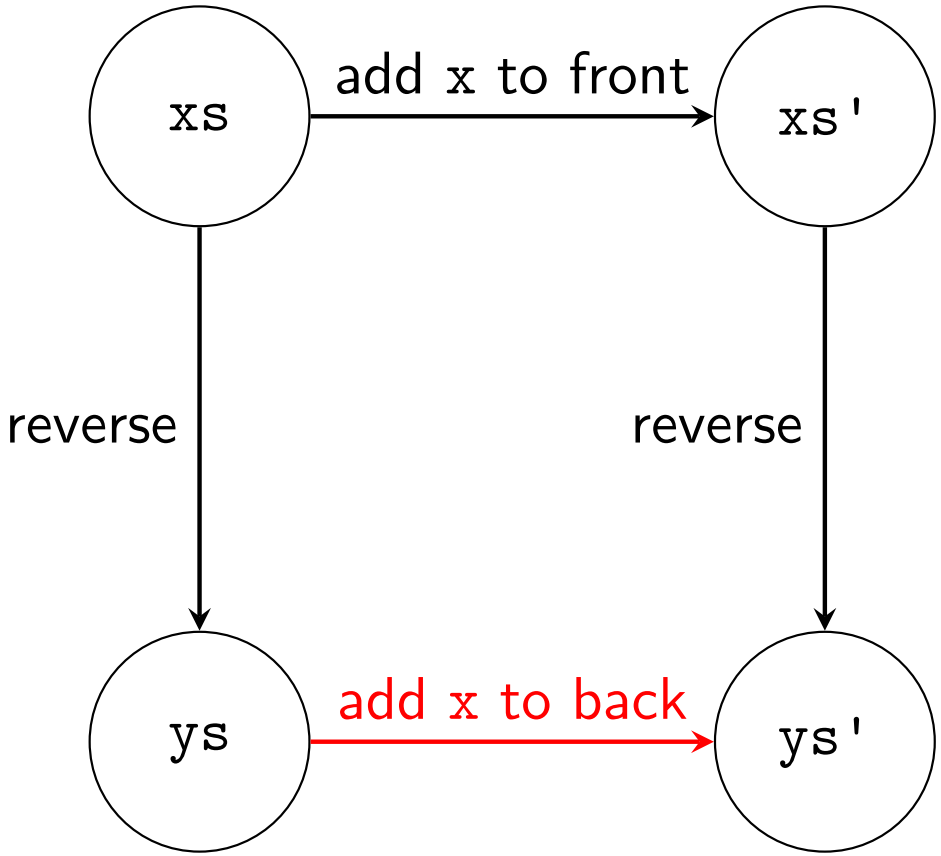
- When testing functions, determine what changes in the output if we *mess with* the input a bit

Example

Given a list `xs`, how is the result `ys` affected when you add an element `x` to the front of `xs` before reversing it?

- Increases the number of possibilities for properties to write
- Increases the number of ways that counterexamples could pop up

Metamorphic properties



Metamorphic properties

Code:

```
@given(st.integers(), st.lists(st.integers()))
def test_prepend_reverse_becomes_append(x, xs):
    assert my_reverse([x] + xs) == my_reverse(xs) + [x]
```

Challenge: Malicious compliance

We could easily break the previous property for defining reverse. Can you provide a terminating function on lists of integers that satisfies the above property but doesn't reverse the list?

1 minute break... You're 60% of the way through the talk...

Coming up with metamorphic properties

A few ideas:

- Is there a way to mess with the input such that the output **stays the same**?
- Is there a way to do two operations in a **different order** and end up with similar results?
- Is there a way to **do** an operation and then **immediately undo** that operation?

Metamorphic properties can be weak by themselves, but very powerful when multiple are combined together.

Describing behaviour

- Combining properties can even be so powerful as to specify the behaviour for the function entirely.
- If we have enough properties that can directly describe the requirements of the function in all cases, then we have a specification.

The Oracle Problem (revisited)

Even if we can generate large, high quality test cases, how would we be able to tell if the code's output was correct?

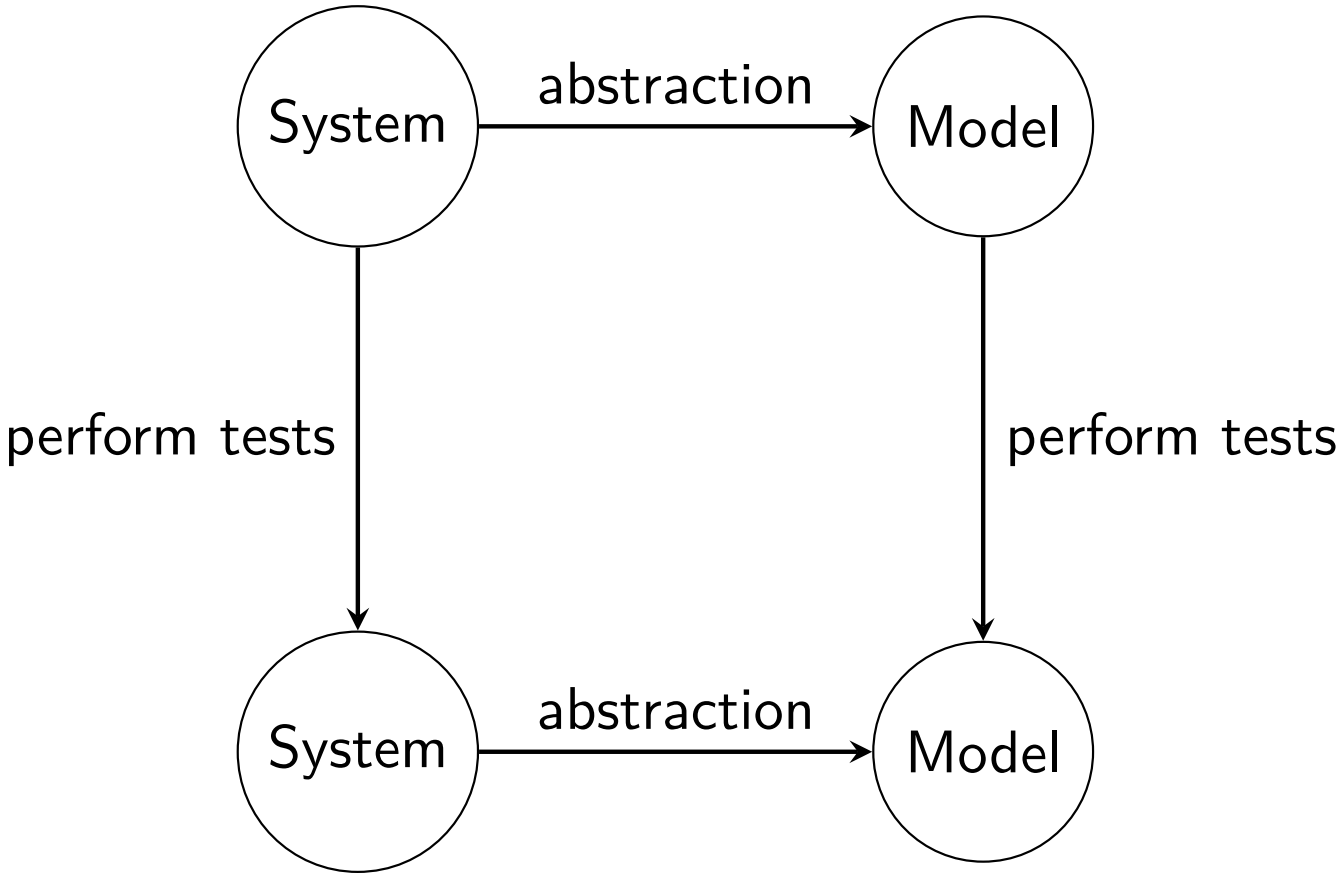
What if we checked our code against a simpler implementation?

Model-based testing

System: What actually happens
Model: What we *expect to* happen

- Developed by Tony Hoare [2]
- The model supports the same functionality as the system, except perhaps less efficient

System against Model



Model-based testing example

Suppose I had a data structure for storing a set of numbers.

```
class NumberSet:
    def __init__(self):

    def in_set(self, num):

    def insert(self, num):

    def remove(self, num):

    def to_list(self):
```

We will use a list of integers as our model.

Defining a model: abstraction and refinement

Define functions to convert *to and from* the two representations...

```
# Convert from NumberSet to list
def abstract(s):
    return s.to_list()

# Convert from list to NumberSet
def refine(model):
    result = NumberSet()
    for i in model:
        result.insert(i)
    return result
```

Defining a model: Operations

Implement equivalent operations on the model...

```
def model_in_set(model, num):
    return num in model

def model_insert(model, num):
    if num not in model:
        model.append(num)

def model_remove(model, num):
    if num in model:
        model.remove(num)
```

Defining a model: Generation strategy

```
@st.composite
def generate_numbersets(draw):
    model = draw(st.lists(st.integers()))
    return refine(model)
```

Defining a model: Equivalence

```
def equiv(model1, model2):
    for i in model1:
        if i not in model2:
            return False
    for i in model2:
        if i not in model1:
            return False
    return True
```

Defining a model: Basic properties

```
@given(generate_numbersets(), st.integers())
def test_insert_matches_model(nset, x):
    model = abstract(nset)
    nset.insert(x)
    model_insert(model, x)
    assert equiv(abstract(nset), model)

@given(generate_numbersets(), st.integers())
def test_remove_matches_model(nset, x):
    model = abstract(nset)
    nset.remove(x)
    model_remove(model, x)
    assert equiv(abstract(nset), model)
```

Defining a model: Generating operations

```
# For generating inputs, pick random small integers
# and random operations
@st.composite
def generate_operation(draw):
    update_type = draw(
        st.sampled_from(["Check", "Insert", "Remove"])
    )
    update_value = draw(st.one_of(
        st.integers(-50, 50),
        st.integers()
    ))
    return (update_type, update_value)
```

Defining a model: Combining many queries

```
@given(generate_numbersets(), st.lists(generate_operation()))
@settings(verbosity=Verbosity.verbose)
def test_all_operations(nset, ops):
    model = abstract(nset)
    for op in ops:
        if op[0] == "Check":
            assert model_in_set(model, op[1]) == nset.in_set(op[1])
        elif op[0] == "Insert":
            model_insert(model, op[1])
            nset.insert(op[1])
            assert equiv(abstract(nset), model)
        elif op[0] == "Remove":
            model_remove(model, op[1])
            nset.remove(op[1])
            assert equiv(abstract(nset), model)
        else:
            # If we generated operations properly,
            # we shouldn't get here
            assert False
```

Comparison of property design patterns

Property type	Min	Max	Mean
Postcondition	7.1	245	77
Metamorphic	2.4	714	56
Model-based	3.1	9.8	5.8

Figure: Mean number of tests to make a property of each type fail

Source: Hughes [3]

Pitfalls

Tempting to use model-based testing for *everything*

```
def test_reversing_twice_gives_same_list(xs):
    assert my_reverse(xs) == ...
```

- Repeating code is very bad
- No simpler way to implement the function you are writing
- Use the other techniques instead! Model-based testing cannot solve everything.

Ensuring software safety

- Property-based testing is not going to be the only thing that is used on a project to ensure software safety
  - Using a language with a type system designed for safety
  - Formal verification
- We will discuss how PBT can complement these other methods

PBT and safe type systems

- Many languages have type systems that can get rid of most of the bugs before they even get compiled
- Can get difficult to encode *all* of your requirements in types without complex type hackery
- Encoding some of the more complicated ones as PBTs can greatly reduce complexity
  - Less bugs in the long run?

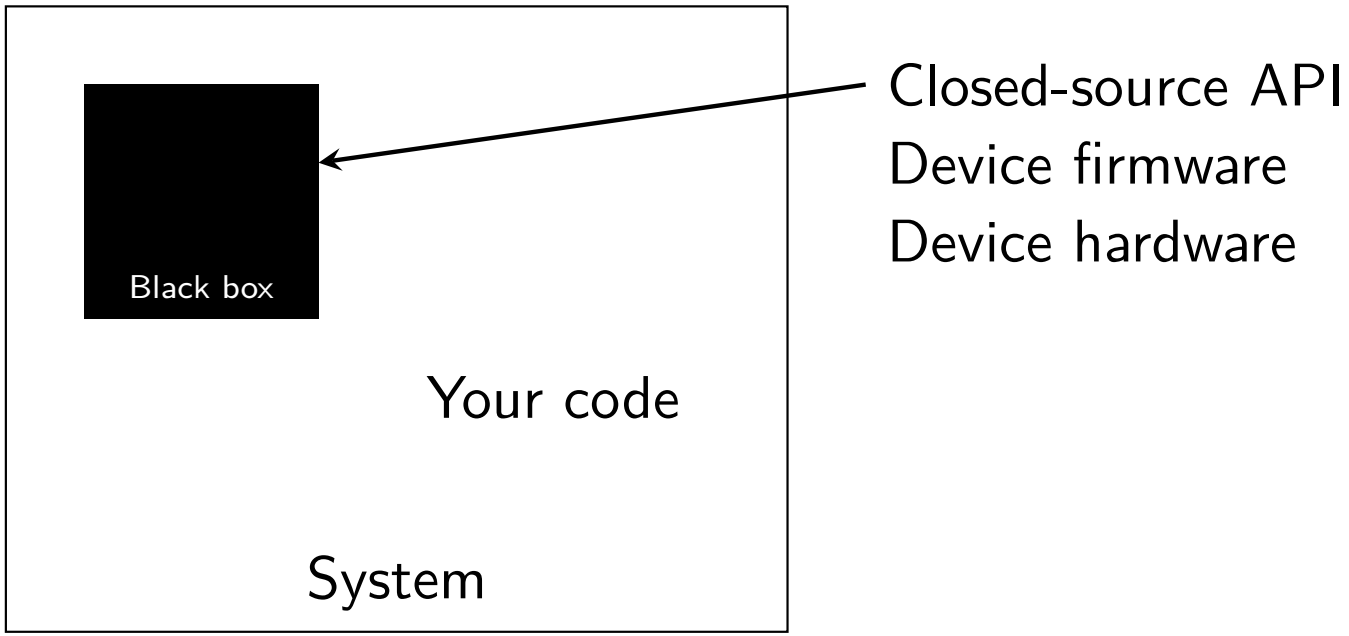
PBT in relation to formal verification

Formal verification: *Proving the correctness of your code with rigorous mathematics*

> "Why do I need property-based tests if I can just prove my program works?"

- What if your proof gets stuck?
- Property-based tests provide a good sanity check
- Great preparatory step before formal verification

PBT as a black-box testing method



□ = Can prove, ■ = Can't prove

We can test the *whole system* and even check for potential bugs from code we didn't write.

Conclusion

Property-based testing presents many advantages

- Code can be tested on large numbers of cases per second
  - One property can replace many individual example tests
  - Can continually be running tests in the background
- Can help people find problems they may have overlooked in their code/unit tests
- More helpful for debugging
- More interesting to write, because...
- Property-based tests force you to think about how your code behaves!

References and further readings

The majority of this talk was sourced from these papers:

Koen Claessen and John Hughes.

Quickcheck: a lightweight tool for random testing of haskell programs.
In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

C. A. R. Hoare.

Proof of correctness of data representations.
Acta Informatica, 1:271–281, 1972.

John Hughes.

How to specify it! - A guide to writing properties of pure functions.
In William J. Bowman and Ronald Garcia, editors, *Trends in Functional Programming - 20th International Symposium, TFP 2019, Vancouver, BC, Canada, June 12-14, 2019, Revised Selected Papers*, volume 12053 of *Lecture Notes in Computer Science*, pages 58–83. Springer, 2019.

Further resources: https://spdskatr.github.io/misc/pbt

Q&A + Challenge solutions?

```python
@given(st.integers(), st.lists(st.integers()))
def test_prepend_reverse_becomes_append(x, xs):
    assert my_reverse([x] + xs) == my_reverse(xs) + [x]
```

Challenge: Malicious compliance

Can you provide a terminating function on lists of integers that satisfies the above property but doesn't reverse the list?

Challenge Solution

Why does this code work?

```
def my_reverse(x):
    return [0] + list(reversed(x))
```

The property we used does not specify a base case!

Extra extension: Suppose the function `my_reverse` didn't just work with lists of integers.
If `my_reverse` was parametrically polymorphic i.e. in OCaml:

```
my_reverse : 'a list -> 'a list
```

Clearly we can't just add an integer to the front anymore, since it won't type check. Is it still possible to satisfy the property with a terminating function that doesn't actually reverse?