

Properties for the People

Benjamin C. Pierce

University of Pennsylvania

February 2026



The talk in 20 seconds:

Recent progress in property-based testing

and

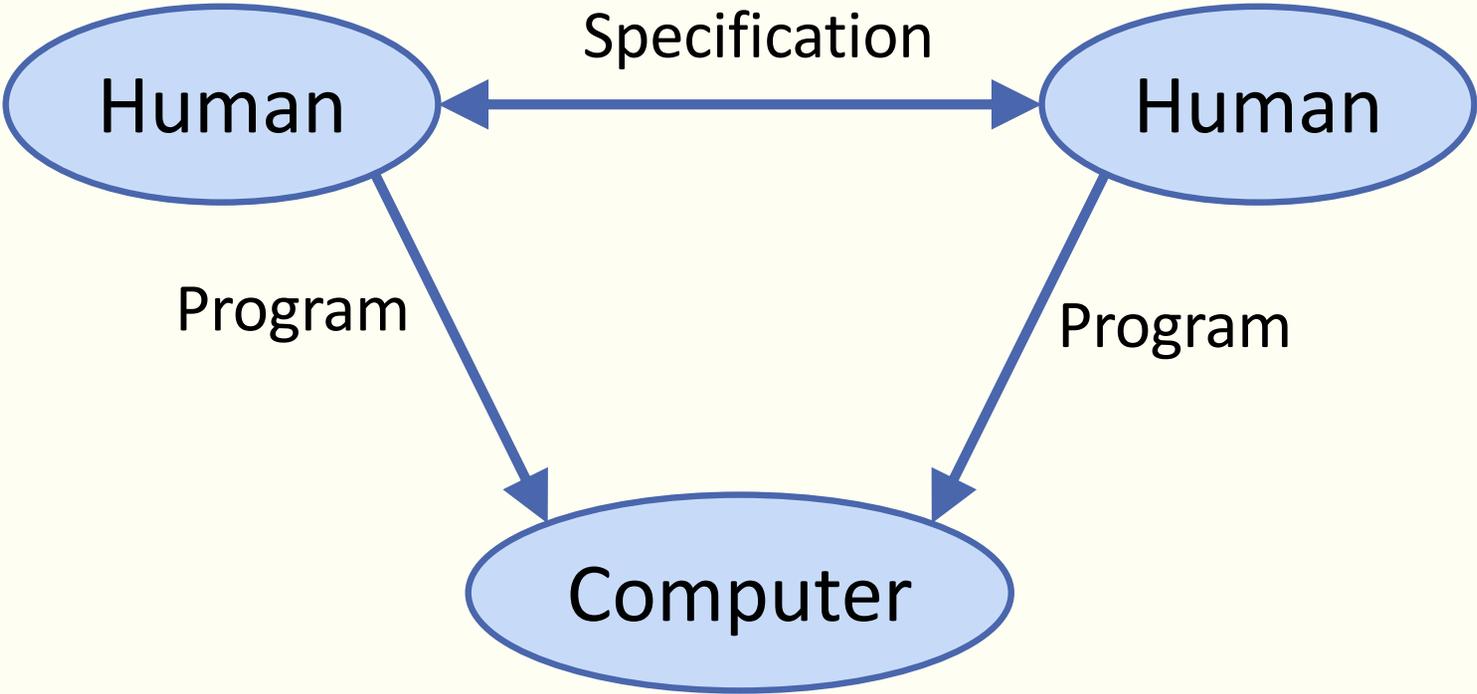
Why this is a good thing for communicating with AI agents

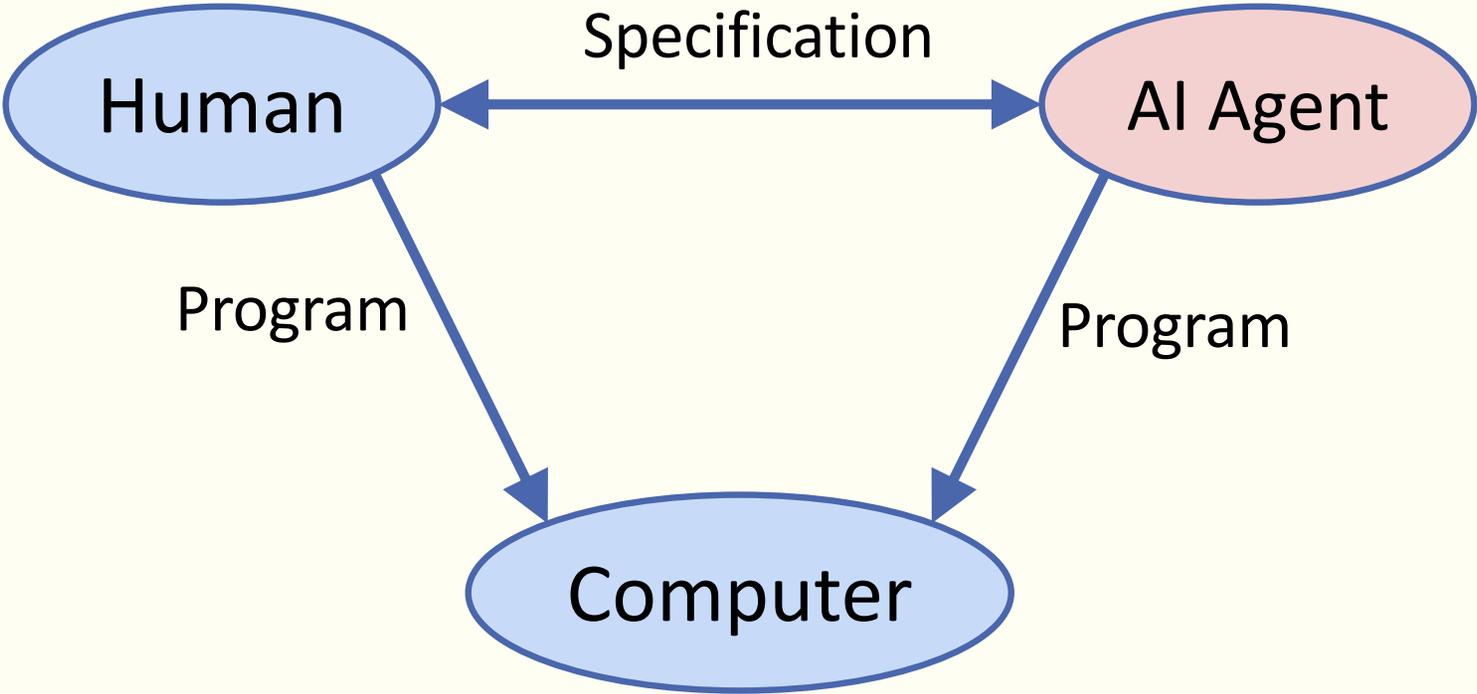
plus

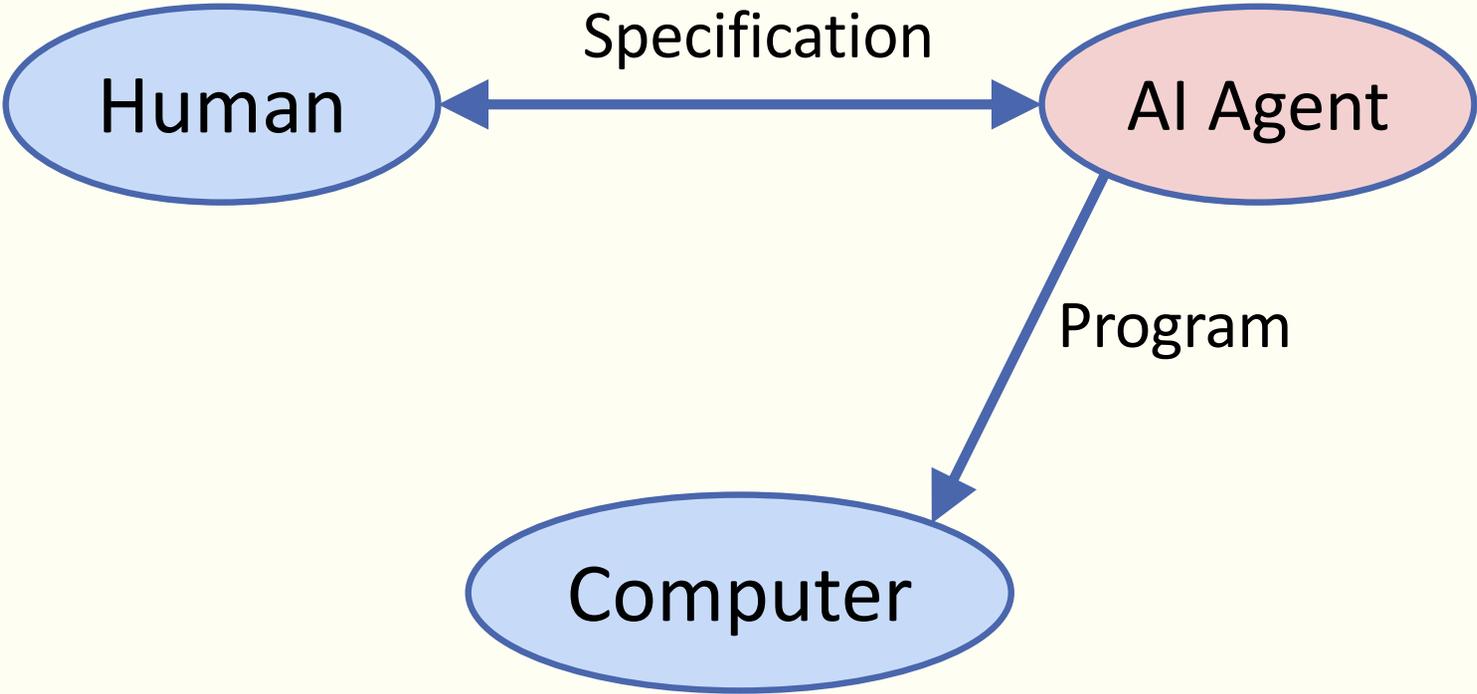
Some challenges

Please interrupt!

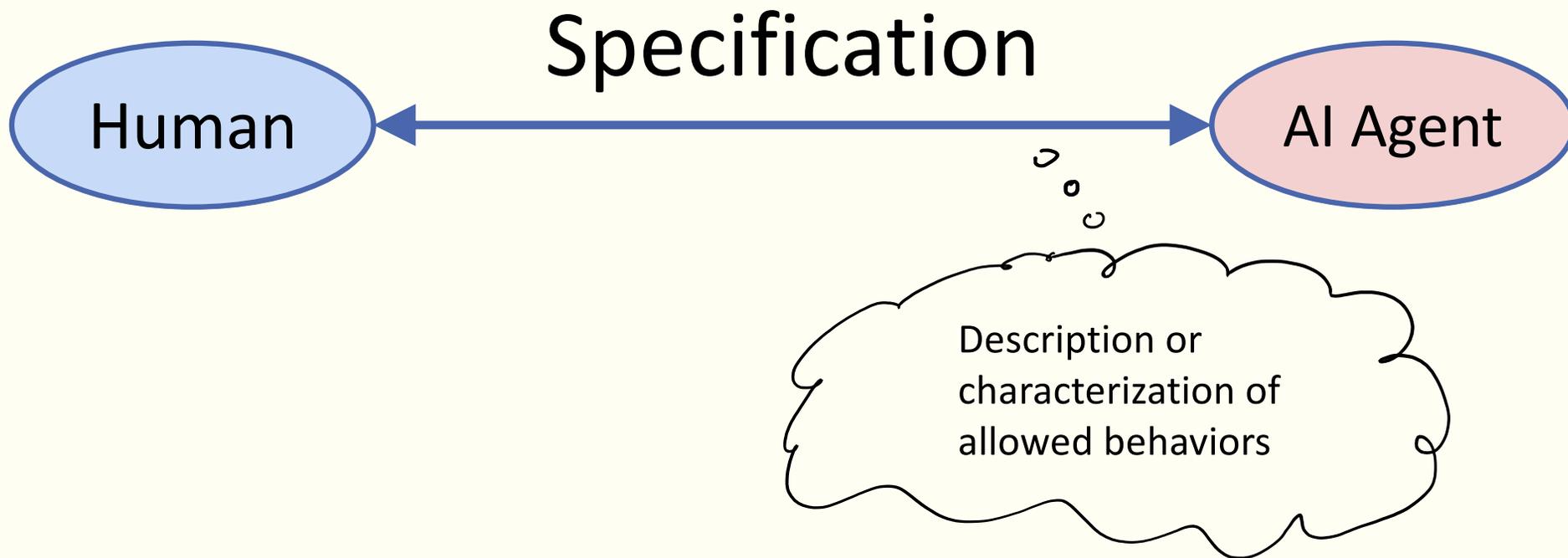
Prologue











Universe: all conceivable behaviors

$$\mathcal{U} = \mathcal{P}(I \times O)$$

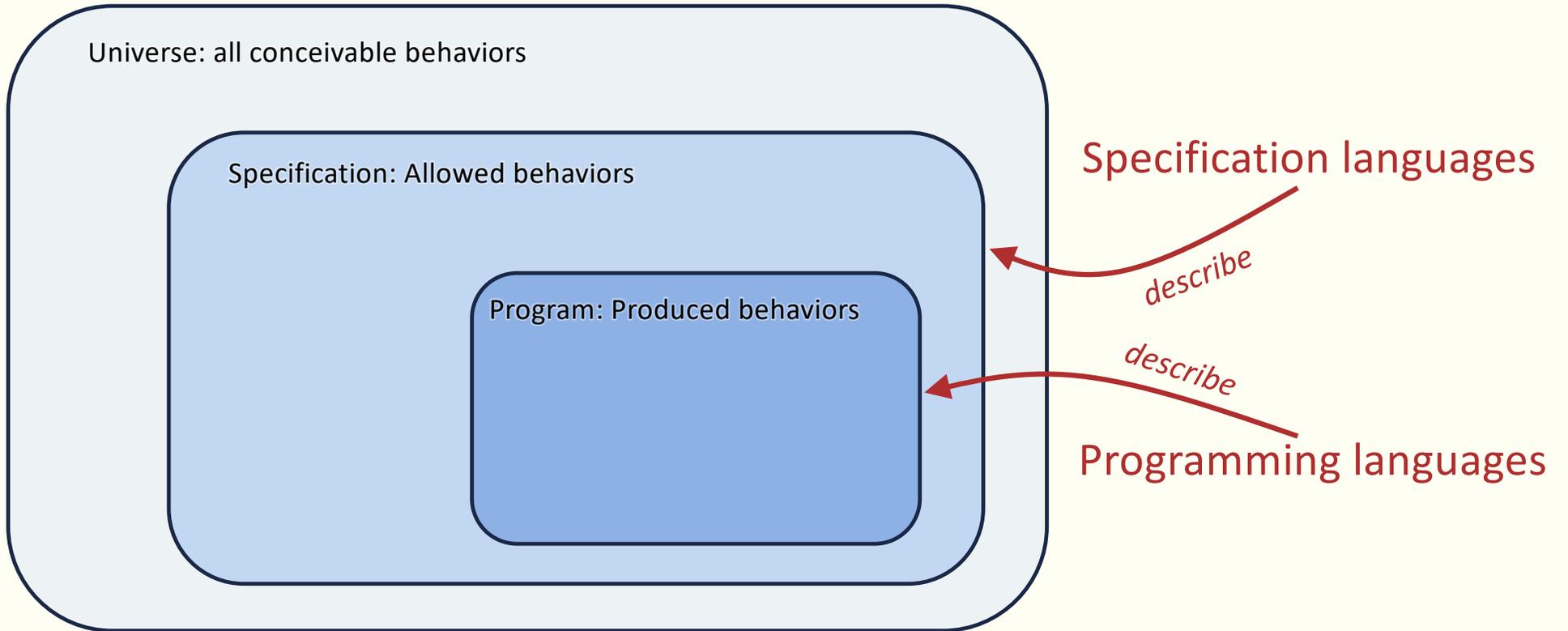
or

$$\mathcal{U} = \mathcal{P}(\text{IO traces})$$

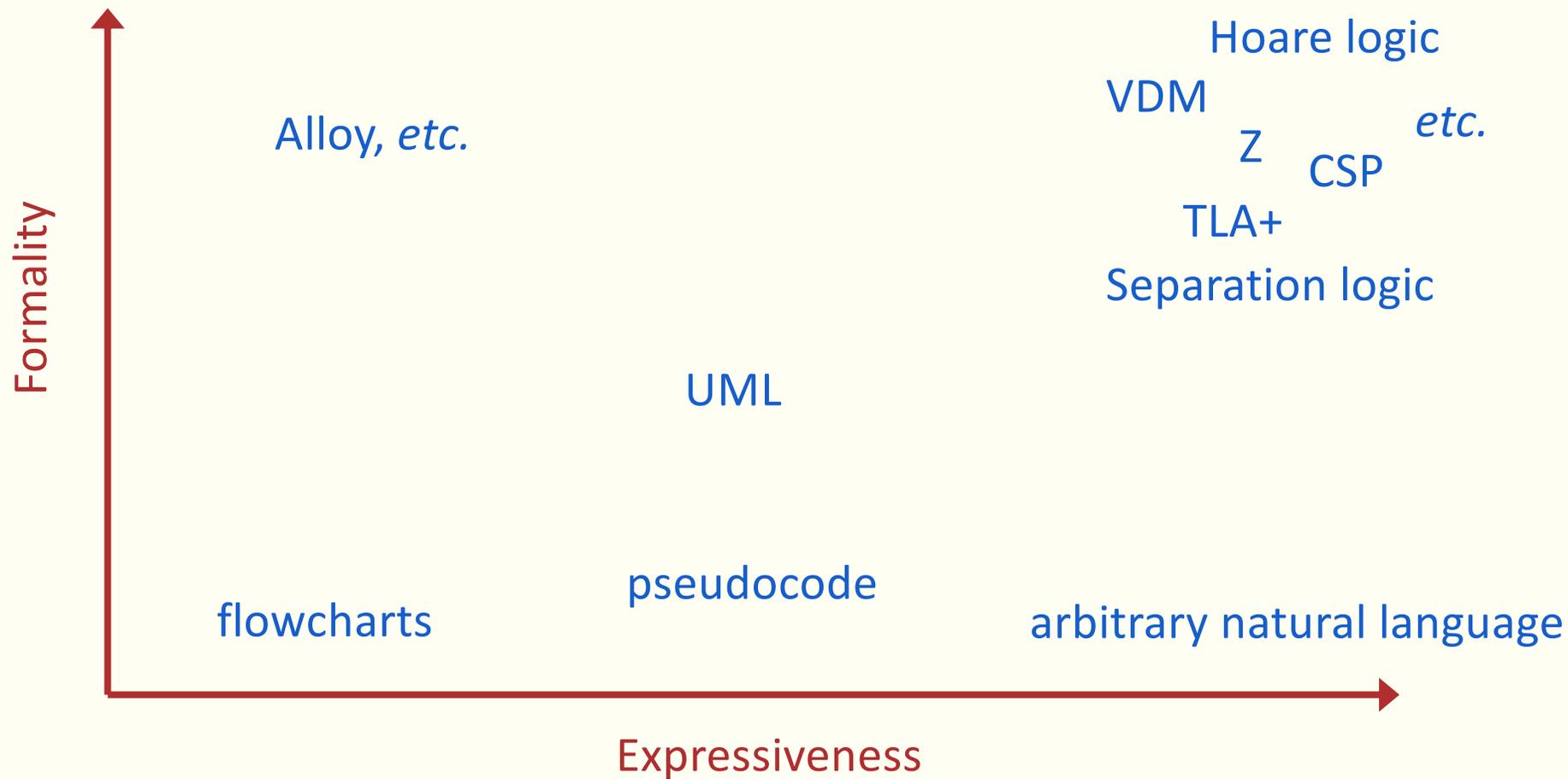
Universe: all conceivable behaviors

Specification: Allowed behaviors

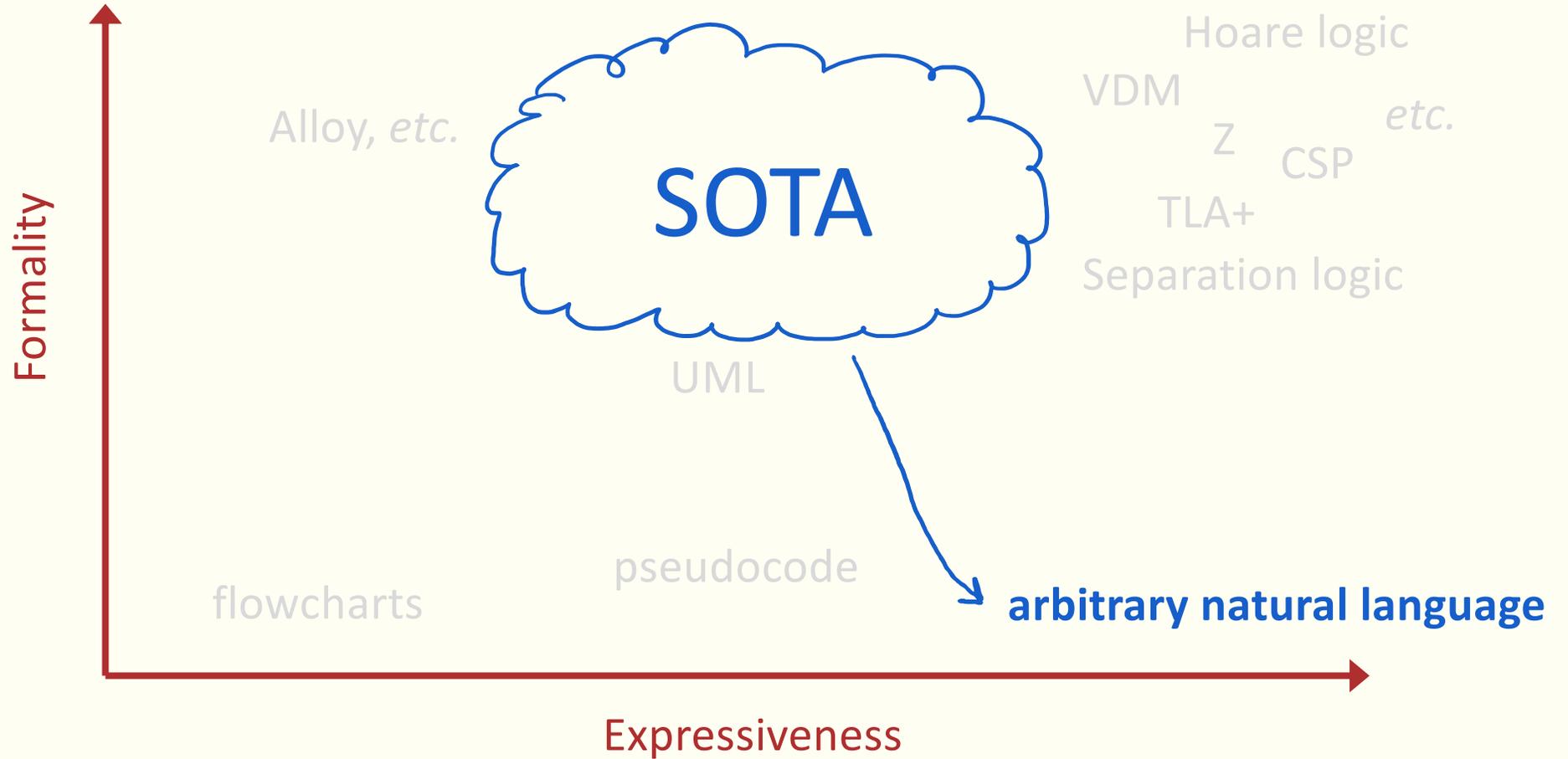
Program: Produced behaviors



Specification languages



Specification languages



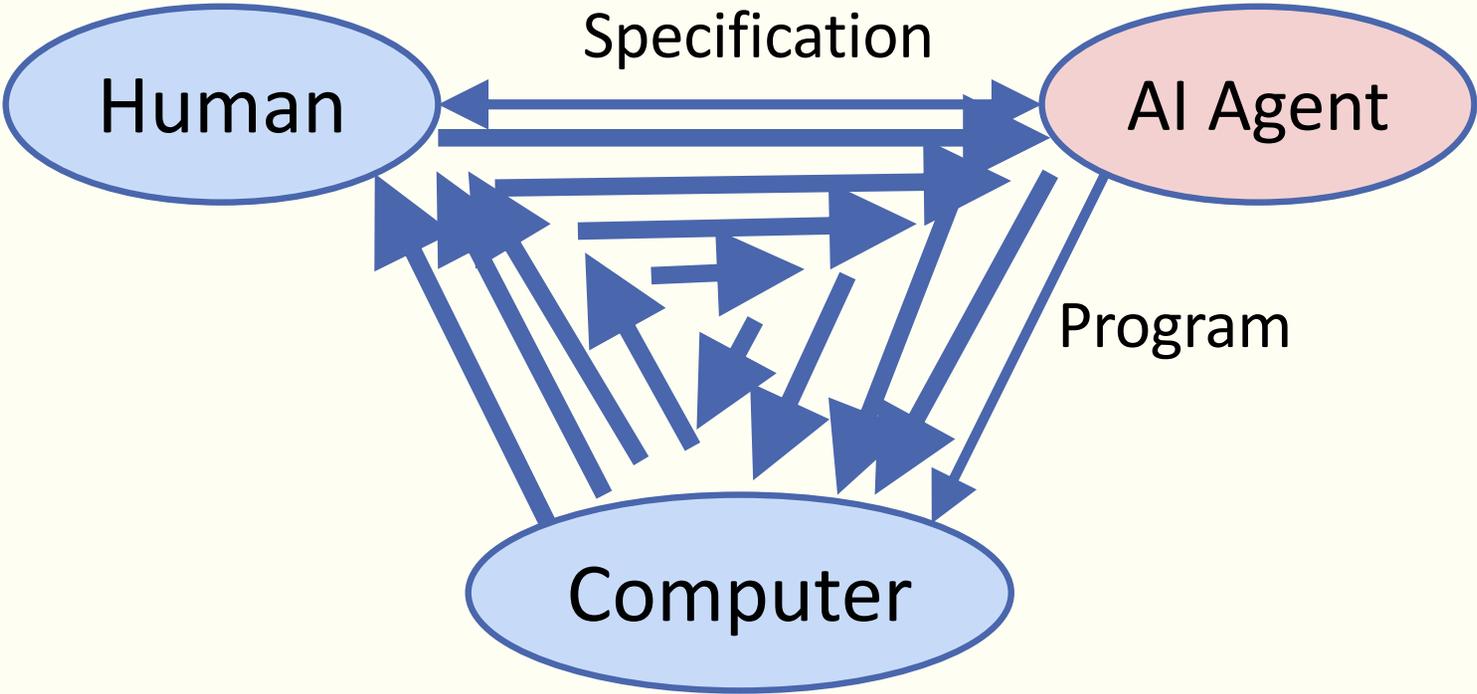
So... is this fine?

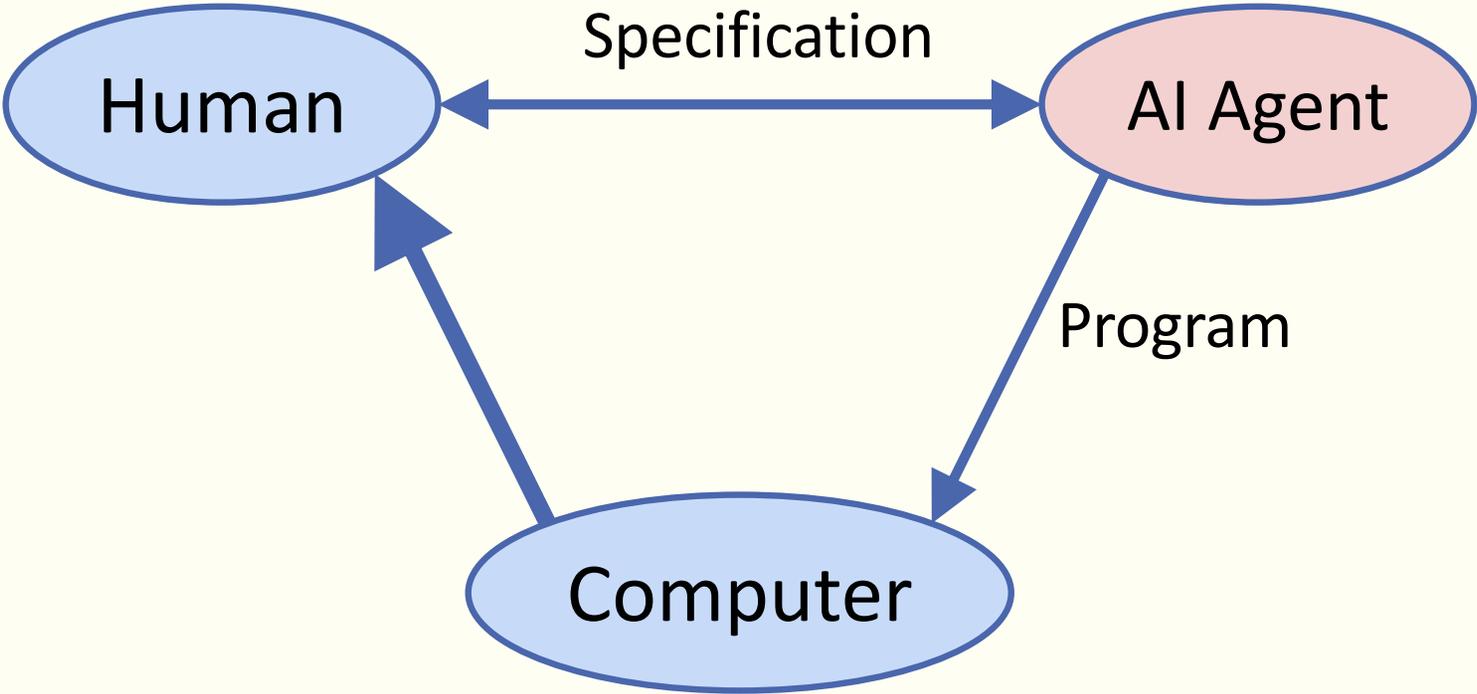
Is natural language the perfect
specification language?

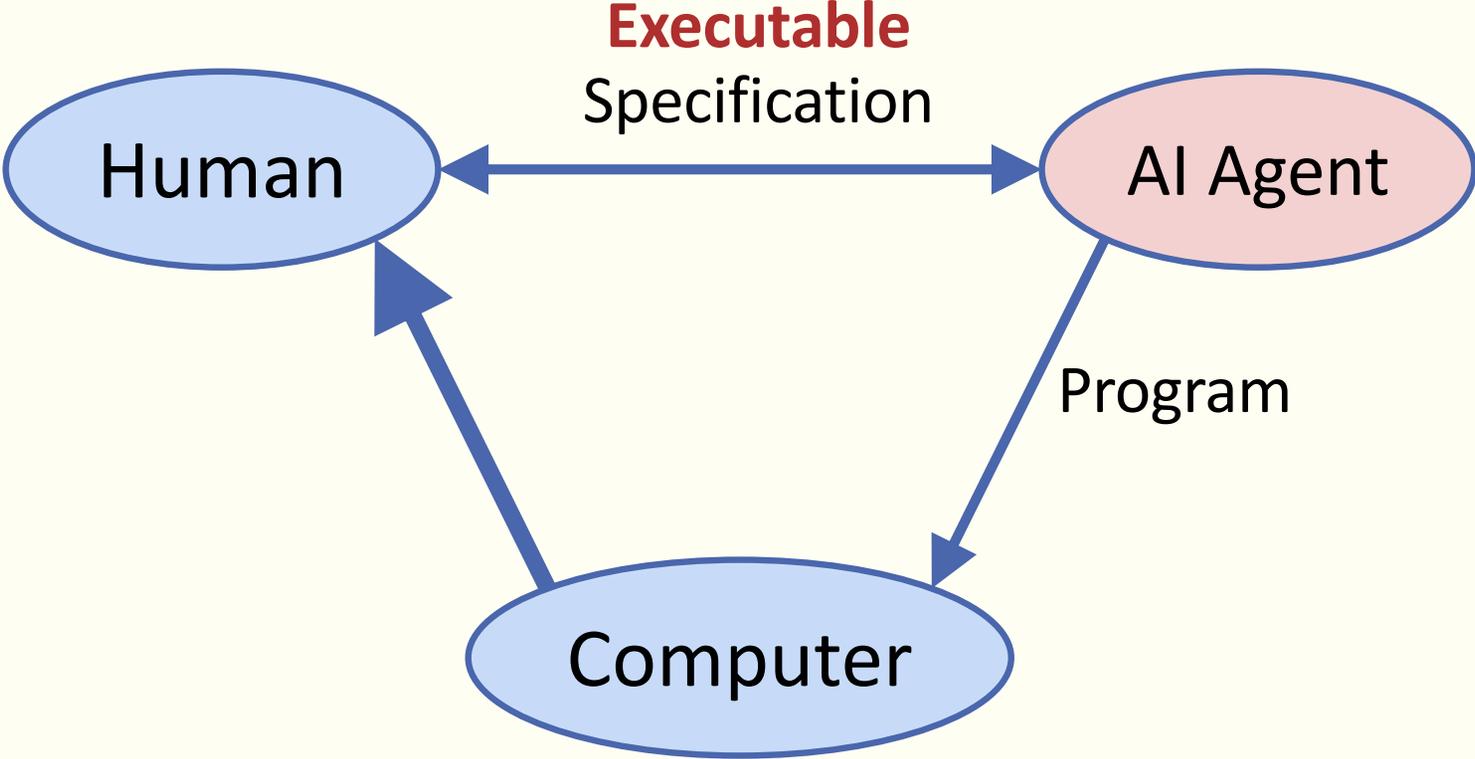
Nope...

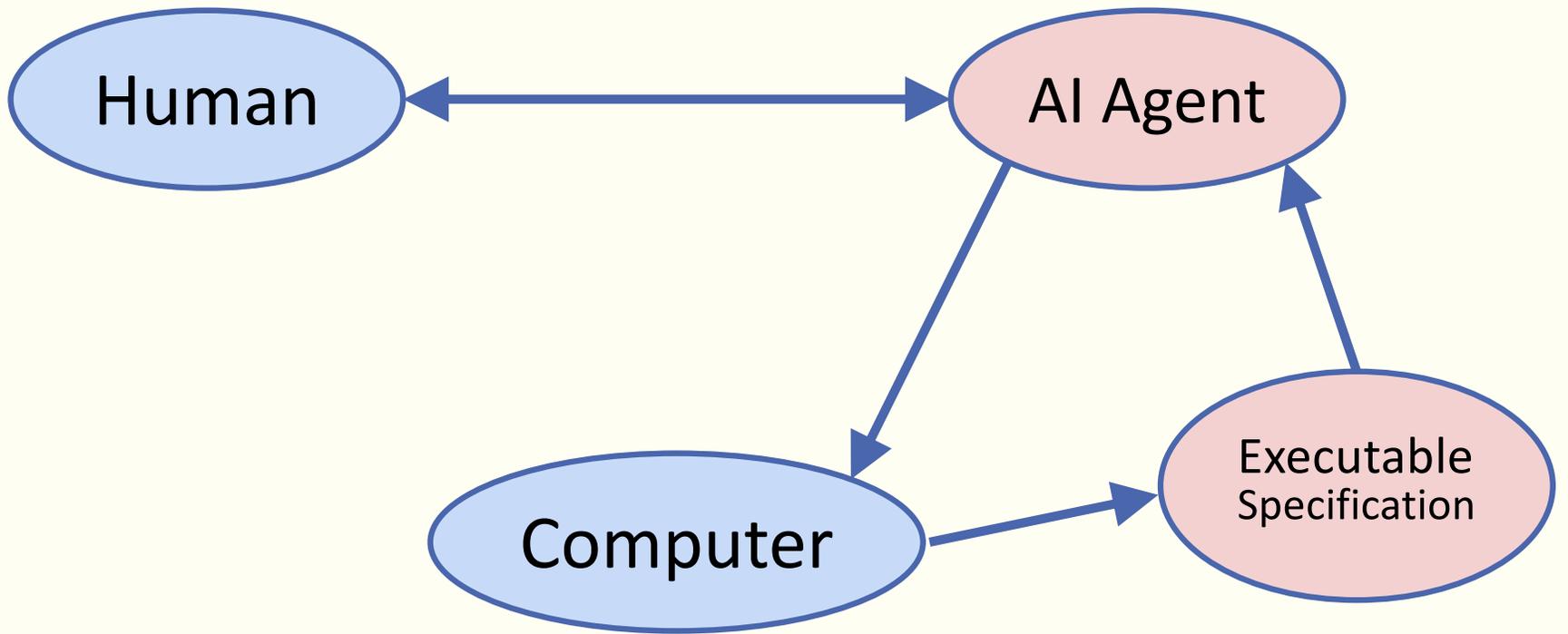
1) Imprecision limits expressiveness

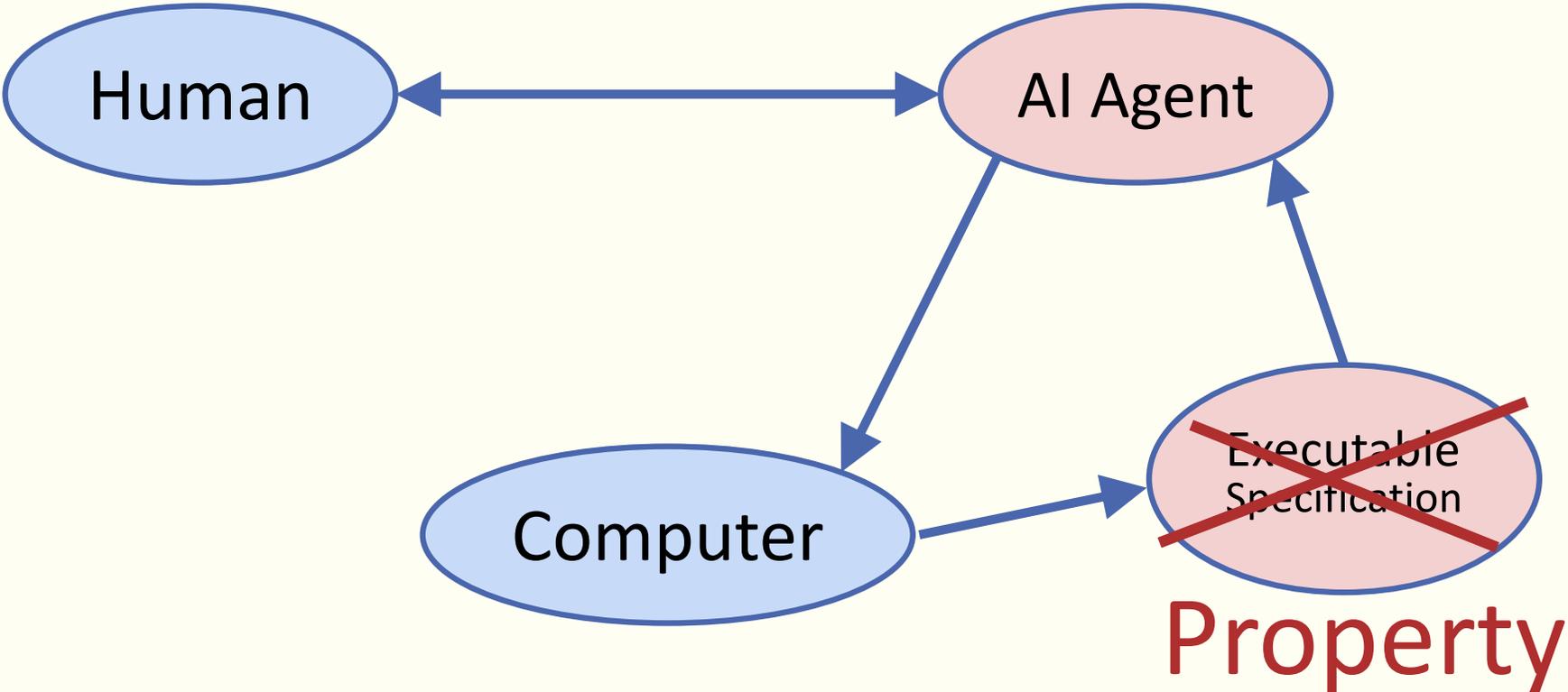
2) Imprecision inhibits iteration



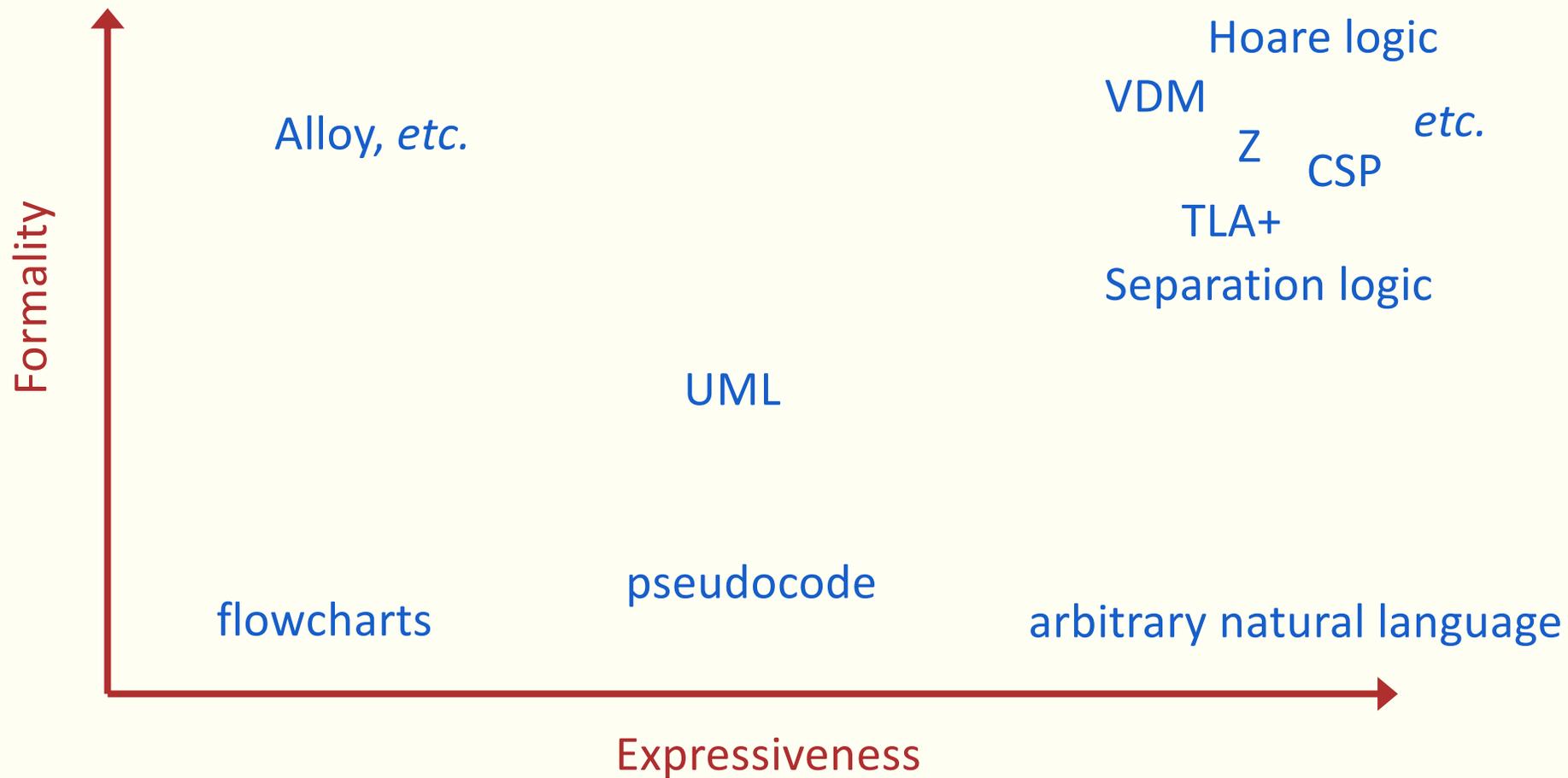




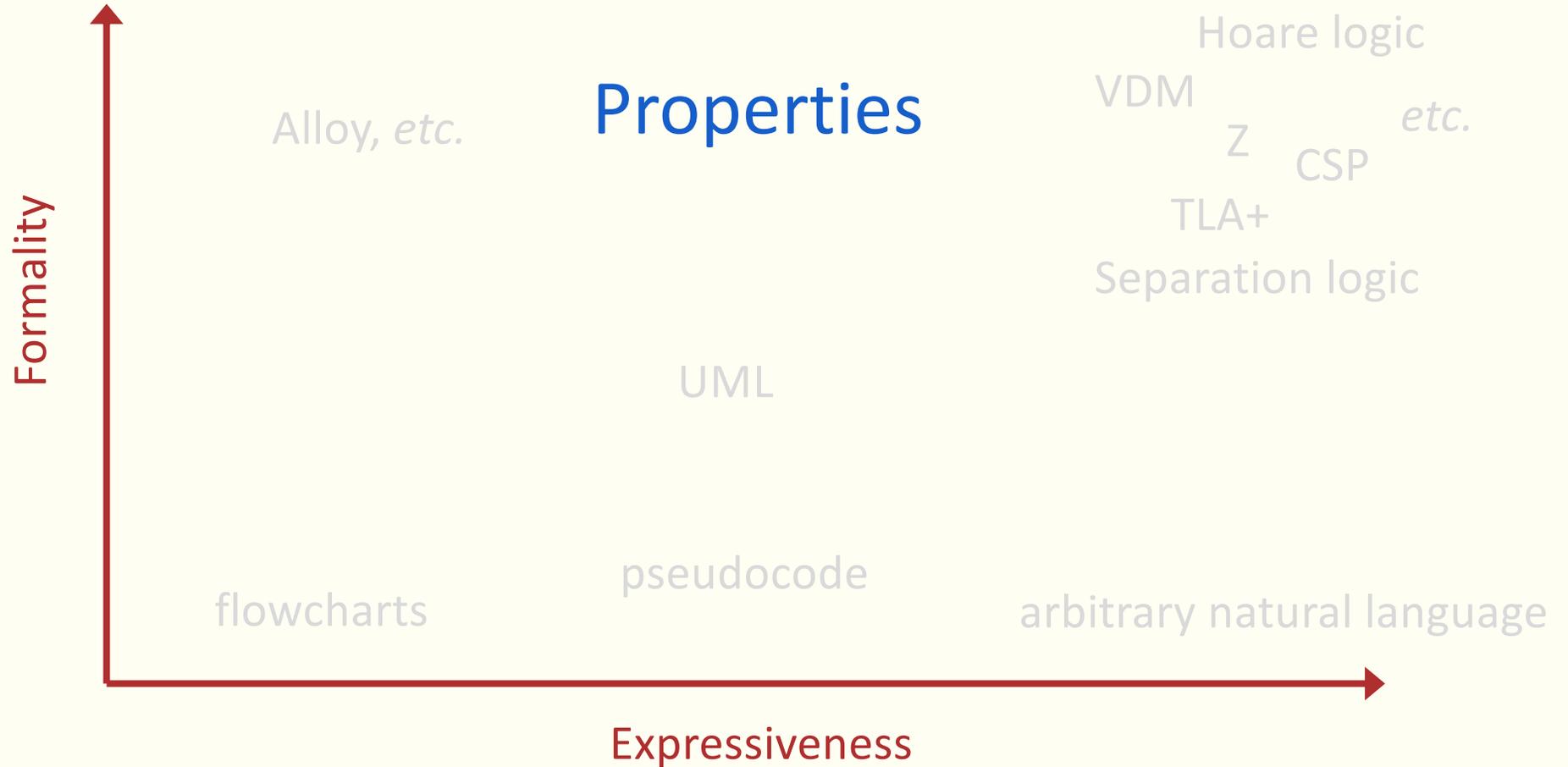




Specification languages

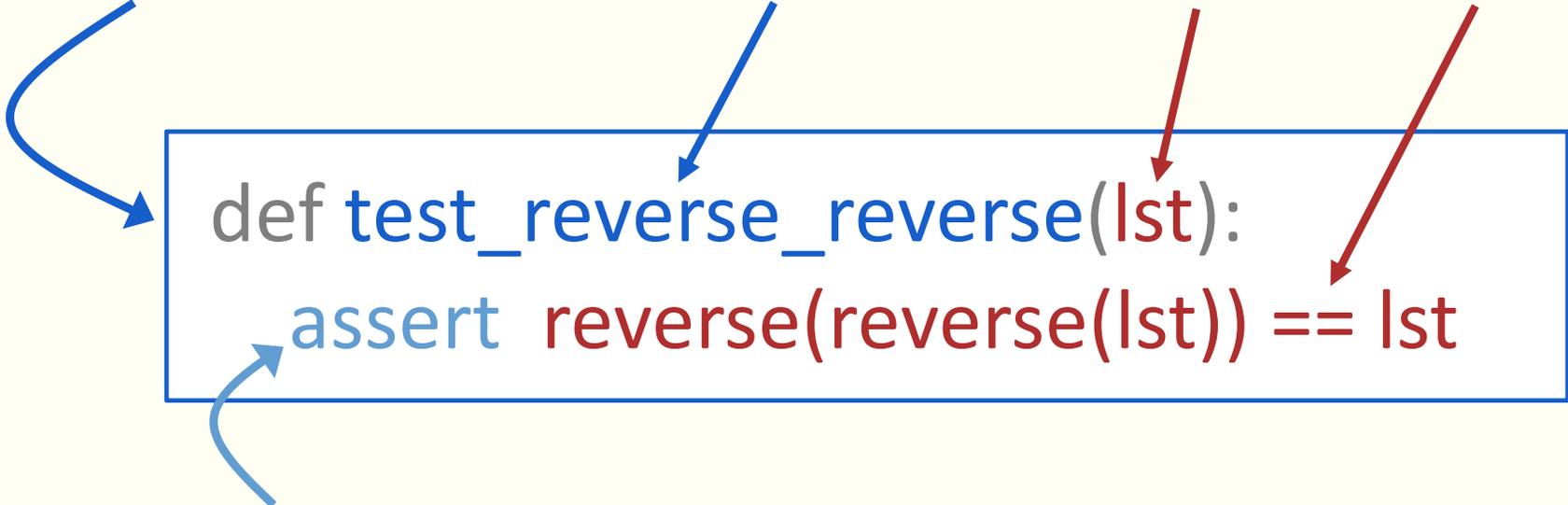


Specification languages



Properties

A **property** is an ordinary **function** from **test inputs** to **booleans**



```
def test_reverse_reverse(lst):  
    assert reverse(reverse(lst)) == lst
```

The diagram shows a blue-bordered box containing the code snippet. A blue arrow points from the word 'property' in the text above to the function name 'test_reverse_reverse'. Another blue arrow points from the word 'function' to the parameter 'lst'. Two red arrows point from the words 'test inputs' and 'booleans' to the parameter 'lst' and the expression 'reverse(reverse(lst)) == lst' respectively. A blue arrow also points from the text above to the 'assert' statement.

Goal of testing: Find a concrete argument **lst** that makes the **assert** fail (i.e., a counter-example to the property)

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

ABSTRACT

QuickCheck is a Haskell library for formulating and testing properties. Properties are described in a declarative style, and are automatically tested. The user can also define custom test cases, and these are also automatically tested. QuickCheck is especially suited to testing properties that can be stated separately from the code to obtain good



in
properties
automati-
to de-
number of
ed, and
g is es-
properties
lt from
suffices

monad and
fine grain.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [11], which competes surprisingly favourably with



done at a
whether a test
by an auto-
have chosen
e have de-
ble specifi-
properties
ks that the
e specifica-
ss system.
odule as the
ckable doc-

1. INTRODUCTION

Testing is by far the most commonly used approach to

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [11], which competes surprisingly favourably with

QuickCheck Family

C
(theft)

C++
(CppQuickCheck)

Clojure
(test.check)

Coq (QuickChick)

F#
(FsCheck)

Go
(gopter)

Haskell
(QuickCheck or
Hedgehog)

Java
(QuickTheories)

JavaScript
(jsverify)

PHP
(Eris)

Python
(Hypothesis)

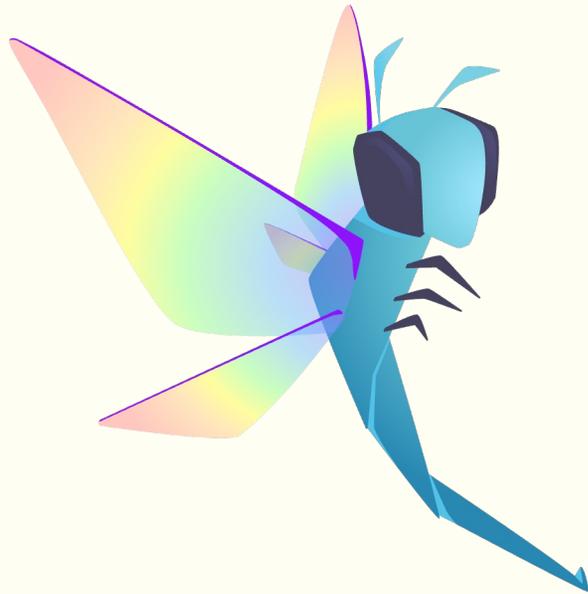
Ruby
(Rantly)

Rust
(Quickcheck)

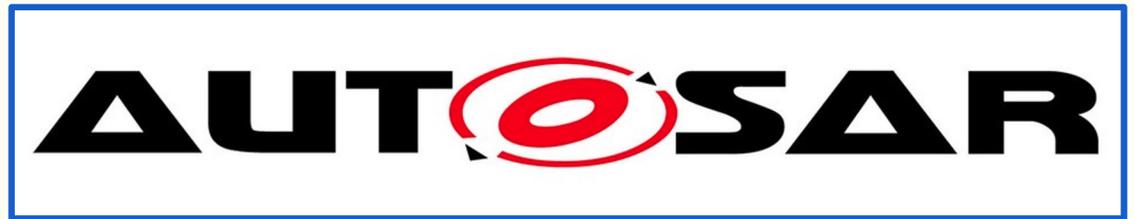
Scala
(ScalaCheck)

Swift
(Swiftcheck)

And more!

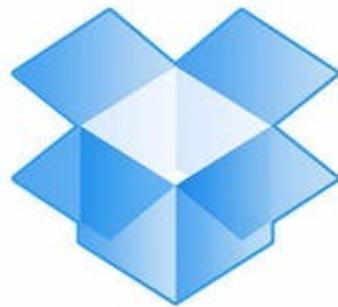


The **Hypothesis** PBT library in **Python** is downloaded over 4 million times each week, and was used by more than 5% of Python users surveyed by the PSF in 2023.



Engineers at Quviq built an executable specification based on the 3000-page AutoSAR standard for automotive software components

QuickCheck-based testing found >200 faults in AutoSAR Basic Software, including >100 inconsistencies in the standard



Dropbox

A QuickCheck specification of DropBox
found several new bugs in its behavior



Rust's PropTest tool was used to test that a new key-value store node implementation for S3 matches a reference implementation.

PBT is used in tandem with other lightweight formal methods like model checking.

[Overview](#)[Learn ▼](#)[Policy playground](#)[Integrations](#)[Cedar SDK ↗](#)

FAST, SCALABLE ACCESS CONTROL

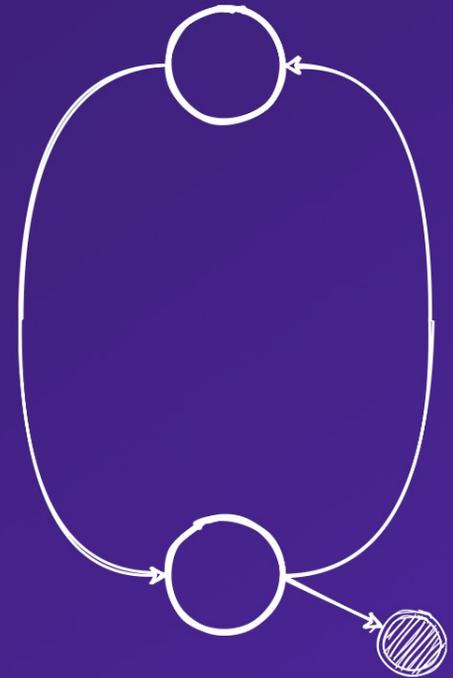
Cedar is a language for defining permissions as policies, and a specification for evaluating those policies. Use Cedar to define who is authorized to do what within your application. Cedar is open source.

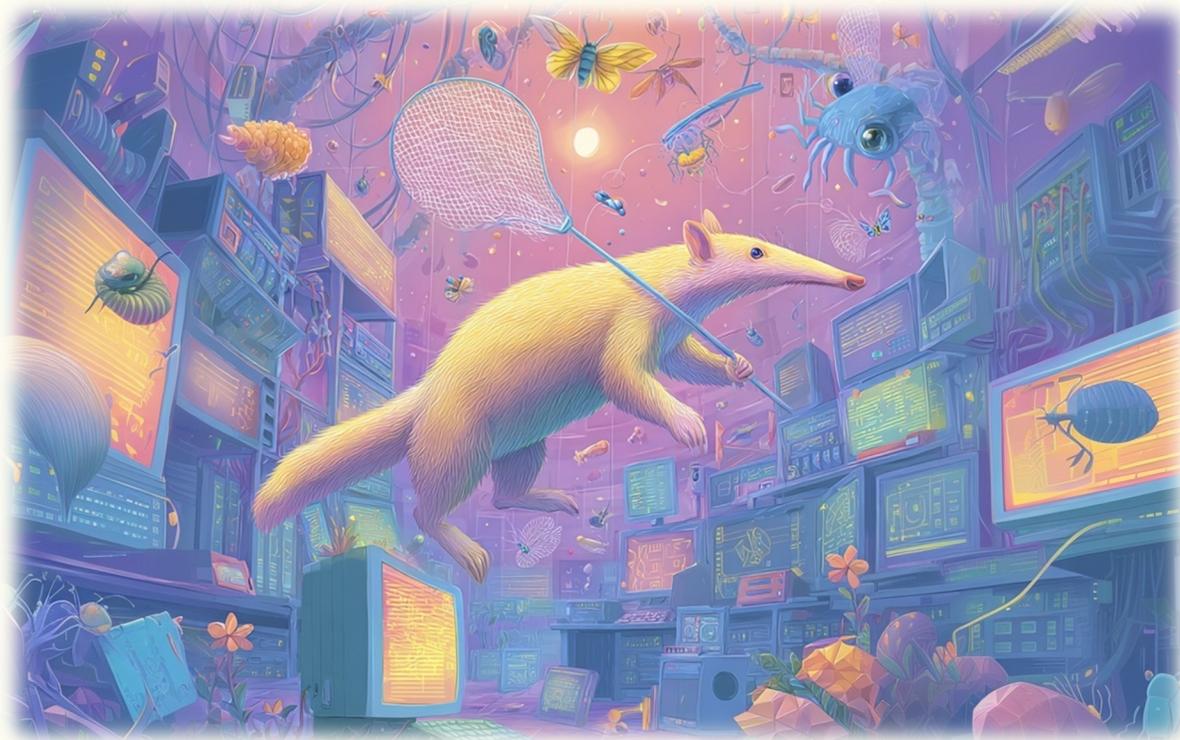
[Learn Cedar](#)[Do the tutorial](#)[Try it out in playground](#)

AWS's **Cedar** implementation was validated against a formal model using PBT

Quickstrom

Quickstrom autonomously checks any type of web application. Focus on understanding and specifying, and let Quickstrom do the testing.





The **Antithesis** end-to-end automated testing platform integrates PBT, fuzzing, and deterministic simulation into a single testing tool

What does all this add up to?

Property-Based Testing



Specification-Driven Development

AI Coding Agents



Property-Based Testing



Autonomous

Specification-Driven Development

Autonomous
Specification-Driven
Development



Scaling Intelligence Lab

Surprisingly Fast AI-Generated Kernels We Didn't Mean to Publish (Yet)



Anne Ouyang
Stanford

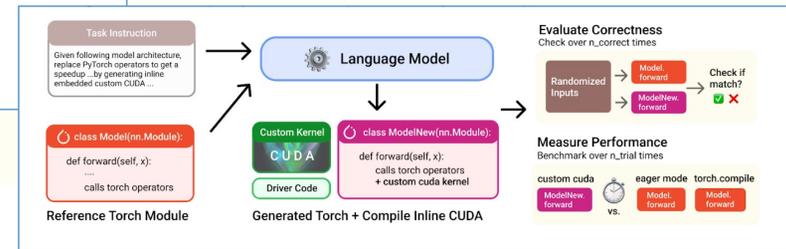


Percy Liang
Stanford



Azalia Mirhoseini
Stanford

- **Matmul (FP32): 101.3%** performance of FP32 torch.matmul; problem size: 4096x4096 square matrices
- **Conv2D: 179.9%** performance of FP32 torch.nn.Conv2D; problem size: (100, 3, 224, 224) input tensor, conv(in_channels=3, out_channels=96, kernel_size=11, stride=4, padding=2)
- **Softmax: 111.8%** performance of FP32 torch.softmax; problem size: (4096, 65536) input tensor
- **LayerNorm: 484.4%** performance of FP32 torch.nn.LayerNorm; problem size: (16, 64, 256, 256) input tensor
- **Conv2D + ReLU + MaxPool: 290.1%** performance of FP32 torch reference, 189.0% performance of FP32 torch.compile() reference; problem size: (100, 3, 224, 224) input tensor, conv(in_channels=3, out_channels=96, kernel_size=11, stride=4, padding=2), maxpool(kernel_size=3, stride=2)



1 # Implementation Plan

🔄 Task in progress

- 2 [] 1. Set up backend API foundation
- 3 - Create subscription data types and inter
- 4 - Set up in-memory storage structure for s
- 5 - Configure CORS middleware for frontend
- 6 - *_Requirements: 2.1, 2.2, 2.3_*
- 7

🔗 Start task

- 8 [] 2. Implement backend subscription endpoint

🔗 Start task

- 8 [] 2.1 Create POST /api/subscribe endpoint with request validation
- Implement email format validation
- Add request body parsing and validation
- *_Requirements: 2.1, 2.2_*

🔗 Start task

- [] 2.2 Implement subscriber storage and duplicate prevention
- Create subscriber storage functions



Including Steering Documents

product.md structure.md tech.md

I'll implement task 1 to set up the backend API foundation. Let me start by examining the current backend structure.

👁 Read file(s) index.ts

👁 Read file(s) package.json

Now I'll update the task status and implement the backend API foundation with the required data types, in-memory storage, and CORS middleware.

📝 Task: 1. Set up backend API foundation
 from index.ts
 Status: In Progress

🔄 Editing

Working.

Cancel

👁 Follow

Ask a question or describe a task



00

Taking stock...

Preview

Recent progress in property-based testing

and

Why this is a good thing for communicating with AI agents

plus

Some challenges

Preview

Recent progress in property-based testing

and

~~Why this is a good thing for communicating with AI agents~~

plus

Some challenges

Preview

~~Recent progress in property-based testing~~

and

~~Why this is a good thing for communicating with AI agents~~

plus

Some challenges

Main challenge:

Writing properties can be hard

Properties for the People!

Why is writing properties hard?

1. Requires abstract reasoning
2. Not a natural fit for every situation

How do we Teach People to Write Properties?

- Thinking explicitly at the level of specifications is unfamiliar to many programmers
- At what level should PBT be taught?
 - What would a software engineering course focused on PBT look like?
 - Can it be taught in an intro programming course?



Using Relational Problems to Teach Property-Based Testing

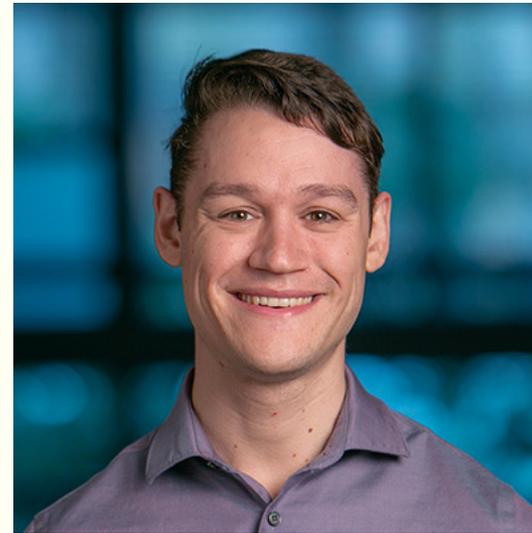
John Wrenn^a, Tim Nelson^a, and Shriram Krishnamurthi^a

^a Brown University, USA

Current experiment:
PBT in CIS 1200 at Penn



Xiaorui Liu

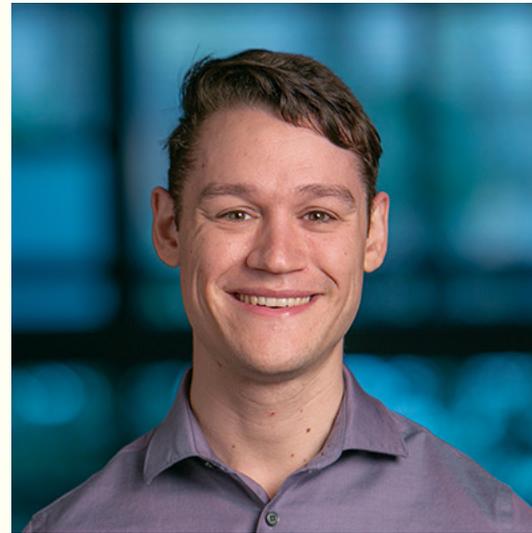


Andrew Head

How do we Teach People to Find Properties?



Harrison Goldstein



Andrew Head

How do we Teach People to Find Properties?

Observation study of PBT use at Jane Street

Key finding: Developers gravitate toward just a few PBT idioms most of the time:

- Differential tests
- Validity / invariants
- Round tripping
- Simple commuting diagrams



Andrew Head



Harrison Goldstein

Can AI Tools Help?

Some examples...

- Kiro uses a language model to translate informal requirements into executable tests
- Saketh Kasibatla (UCSD) is building an interactive environment for collaboratively developing executable specifications



Can AI Tools Help? Yes, but...

We do not want AI tools
writing specifications all by
themselves!

Second challenge:

Good generators are required

Or...

Generators

```
def test_reverse_reverse(lst):  
    assert reverse(reverse(lst)) == lst
```

```
def test_reverse_reverse(lst):  
    assert reverse(reverse(lst)) == lst
```

```
$ pytest
```

```
===== test session starts =====
```

```
collected 1 item
```

```
test_reverse.py . [100%]
```

```
===== 1 passed in 0.12s =====
```

```
def test_reverse_reverse(lst):  
    assert reverse(reverse(lst)) == lst
```

Or perhaps...

```
Falsifying example: test_reverse_reverse(lst=[1, 2])  
AssertionError: assert [1] == [1, 2]
```

These test inputs come from **generators**...

QuickCheck

Type-based generators

No control over distribution

Hand-rolled generators

Distracting

But we can do better...

Tyche

VSCoDe-based tool for **visualizing generator distributions**

The screenshot displays the Tyche tool interface within VS Code. On the left, a code editor shows Python code for testing a binary search tree. The right panel provides a summary of test results, including a pie chart for 'Categorized by t_is_bst' and a bar chart for 'Distribution of t_size'.

```
bst_tests.py > test_insert_valid
from bst import *
from hypothesis import assume, given, strategies as st

import tyche

tyche.features[Tree] = {
    "size": lambda t: t.size(),
    "is_bst": lambda t: "valid" if t.is_bst() else "invalid",
}

@st.composite
def trees(draw, max_depth=3):
    if max_depth == 0:
        return Leaf()
    else:
        if not draw(st.integers(min_value=0, max_value=max_depth)):
            return Leaf()
        return Node(draw(st.integers()), draw(trees(max_depth - 1)))

Tyche: Run Property and Visualize
@given(trees(), st.integers())
def test_insert_valid(t, x):
    assume(t.is_bst())
    assert t.insert(x).is_bst()
```

test_insert_valid

Unique 263 Total 281

Coverage

/Users/harrison/Misc/python_bst_demo/bst_tests.py	100%
/Users/harrison/Misc/python_bst_demo/bst.py	70%

Categorized by t_is_bst

Distribution of t_size

t_size	Count
0	60
1	18
2	35
3	58
4	50
5	35
6	15
7	5

Minimum by t_size

```
{'t': Leaf(), 'x': 1576365896}
```



Harrison Goldstein

Palamedes



Synthesizing generators from logical specifications...

The Search for Constrained Random Generators

HARRISON GOLDSTEIN, University at Buffalo, SUNY, USA

HILA PELEG, Technion, Israel

CASSIA TORCZON, University of Pennsylvania, USA

DANIEL SAINATI, University of Pennsylvania, USA

LEONIDAS LAMPROPOULOS, University of Maryland, USA

BENJAMIN C. PIERCE, University of Pennsylvania, USA

Thank you!!

Questions, discussion, ...?