# Intent Formalization

*Assessing the Quality of AI-Generated Formal Program Specifications*

**Shuvendu K. Lahiri**

Research in Software Engineering (RiSE),
Microsoft Research, Redmond

ACL2 Workshop 2026

**Contributors:**

**MSR**: Sarah Fakhoury, Saikat Chakraborty, Todd Mytkowicz, Madanlal Musuvathi, Nikolaj Bjorner, Piali Choudhury, Shan Lu, Tahina Ramananandro, Nikhil Swamy, Gabriel Ebner, Tianyu Chen, Shuai Lu, …

**External**: Madeline Endres, Aaditya Naik, Georgios Sakkas, Elizabeth Dinella, Chuyue Sun, Clark Barrett, David Dill, …

[2603.17150] Intent Formalization: A Grand Challenge for Reliable Coding in the Age of AI Agents

***AI can now write (even formally) code — but who checks that it does what you actually meant?***

***More pressing question:***

*If AI can automate program proofs, what do formal verification researchers write papers about :-)*

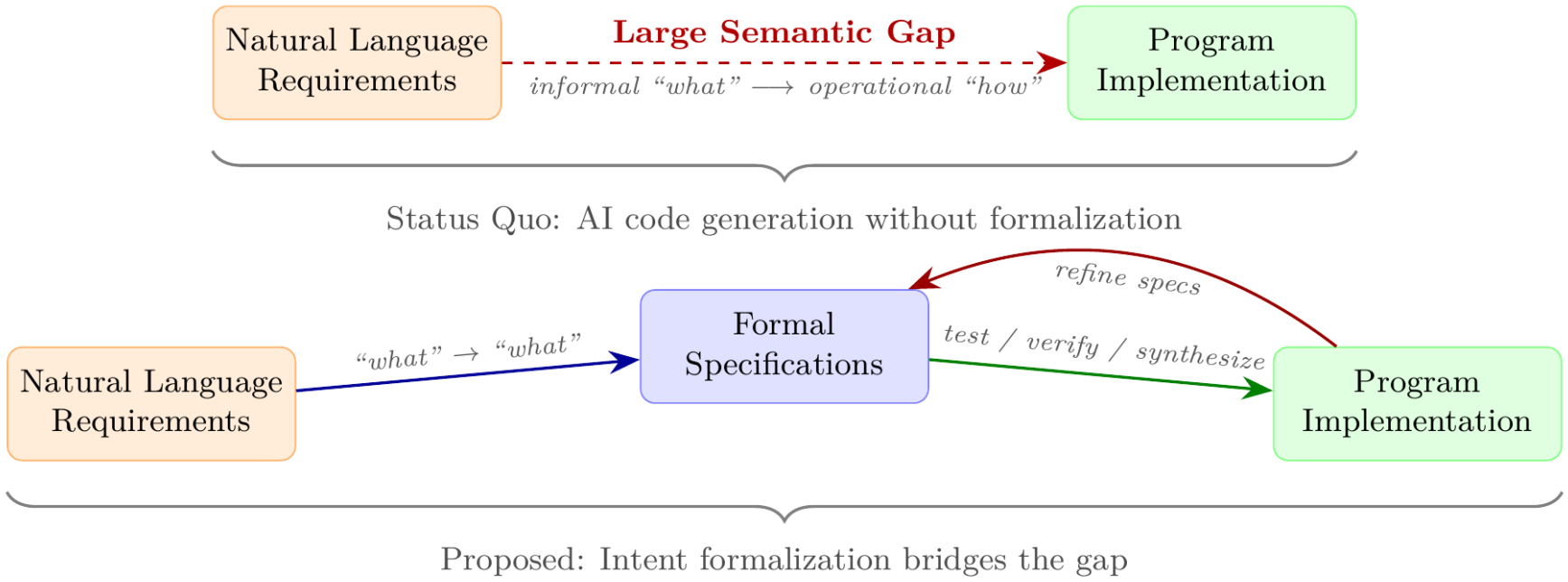
**Quiz**: guess the author?

The choice of functional specifications – and of the notation to write them down in – may be far from obvious, but their rôle is clear: it is to act as a logical firewall between two different concerns. The one is the "pleasantness problem", i.e., the question of whether an engine meeting the specification is the engine we would like to have; the other one is the "correctness problem", i.e., the question of how to design an engine meeting the specification. I firmly believe that whenever we succeed in erecting such a firewall, the effort will pay off handsomely. The reason for this belief of mine is that the two concerns deserve separation because the two problems are most effectively tackled by totally different techniques.



(They are currently psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem.)

# The Intent Gap




**The intent gap**:

- Semantic distance between what a user means and what the program does

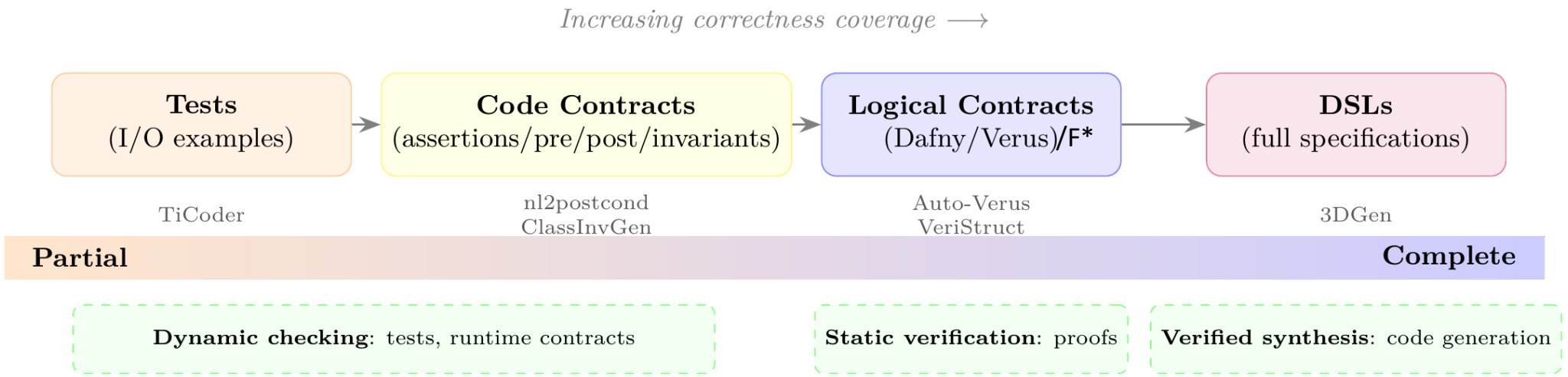
**AI** amplifies the gap:

- "Vibe coding" — AI generates code from NL prompts.
- Code generated faster than humans can review

**Formal Specifications** bridge the gap:

- Formal artifacts that capture user intent
- Checkable — can be validated automatically

# The (Formal) Specification Spectrum




**Intent formalization** = synthesizing (partial) and validating formal specifications from informal intent

Not quite **Auto-formalization** [Wu et al. Autoformalization with Large Language Models. NeurIPS 2022]

- Program specifications can be partial (unlike application in math)
- Users are not expected to have expertise in formal methods to review specifications (unlike mathematicians)

# Big Shift in Research Questions

**Intent Formalization (~2022)**

- [Creativity] Can AI formalize into tests, code contracts, logical contracts, …?
- [Validation] Can symbolic methods validate these specifications?

**Intent Formalization (~2026)**

- [Creativity] Can AI formalize into tests, code contracts, logical contracts, …?
- [Validation] Can symbolic methods validate these specifications?

**This talk**: Ideas from traditional testing (mutation, coverage, metrics) and verification (proofs) have a lot to offer!

# Starting point

## TiCoder: Can AI formalize NL as tests?

- **Main RQ**: Can we automatically evaluate the correctness of a test given an informal natural language intent?
  - **Metrics**
    - [Soundness] How often would a user accept it (accept@k)
      - Passes on all correct implementations
    - [Completeness] How often does it prune away incorrect implementations (pass@k)
      - Fails on many (natural) incorrect implementations
  - **Findings**

    [1] OpenAI Codex can generate a non-trivial functional unit test consistent with the user intent within an average of 1.69 user queries for 90.40% of the examples in the MBPP dataset

    [2] We observe an average absolute improvement of 45.97% in the pass@1 code generation accuracy for {MBPP, HumanEval} datasets and across all LLMs within 5 user interactions

    [2] We find that participants using the proposed workflow are significantly more likely to correctly evaluate AI generated code, and report significantly less task-induced cognitive load.

**[1]** ***Interactive Code Generation via Test-Driven User-Intent Formalization***, Shuvendu K. Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, arXiv2208.05950v1, Aug 2022

**[2]** ***LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation***, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Shuvendu K. Lahiri, TSE 2024

# Intent formalization for programs (symbolic specs)

- **Problem statement**: Can we automatically evaluate the correctness of a formal (requirement) specification given an informal natural language intent?

Hard to define

# Intent formalization for programs (symbolic specs)

- **(Revised) Problem statement**: Can we automatically evaluate the correctness of a formal (requirement) specification given an informal natural language intent + a diverse set of (hidden) validation tests?

- **Why tests?**
  - Don’t need to learn different specification formalisms
  - Intuitive for mainstream developers
  - LLMs were getting good at generating unit tests (e.g., TiCoder, CodaMosa, ..)

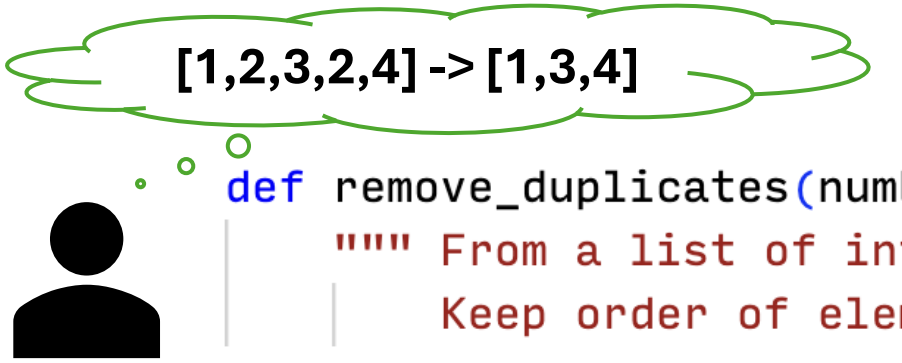
# Overview

- Different ways of testing formal specifications for *correctness*
  - Soundness
  - Completeness
  - Preciseness
- Applications to 3 different **verification-aware languages**
  - **Dafny**: Finding spec-quality issues in a verified artefact
  - Auto**Verus**: Training spec-generation models
  - **F*/Pulse**: AutoCLRS spec-validation with SPOTs
- **Neuro-formal verification**: intent-formalization for environment modeling

# Different metrics for Specification Correctness (given Tests)

1. [**Sound**] I/O Tests satisfy Specs
    1. Input I satisfies each precondition
    2. (Input I, Output O) satisfies each postcondition
2. [**Complete(k)**] Specs rule out a large fraction **k** of code/test mutations
    - Parameterized by a code/test mutator
3. [**Precise**] Specs uniquely determine test outputs given test inputs
    1. E.g., For *sort(*in*):* out
        - *is_ordered*(out) && *is_permutation*(in, out) uniquely determines out given in
    2. Needs Symbolic reasoning
    3. Caveat for non-determinism in output

# Intuition: Example for a **mainstream languages** Python




***Code Contracts in Python***



```
assert len(set(numbers)) == len(set(return_list))
```



```
assert all(numbers.count(i) == 1 for i in return_list)
```

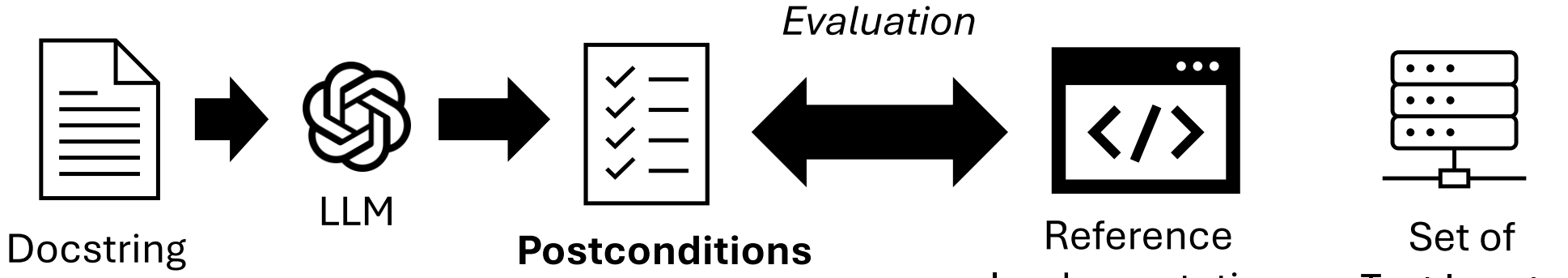


```
assert all(i in return_list for i in numbers if numbers.count(i) == 1)
```

[Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? Endres, Fakhoury, Chakraborty, Lahiri *FSE'24* ]

# Python postconditions: *Soundness*

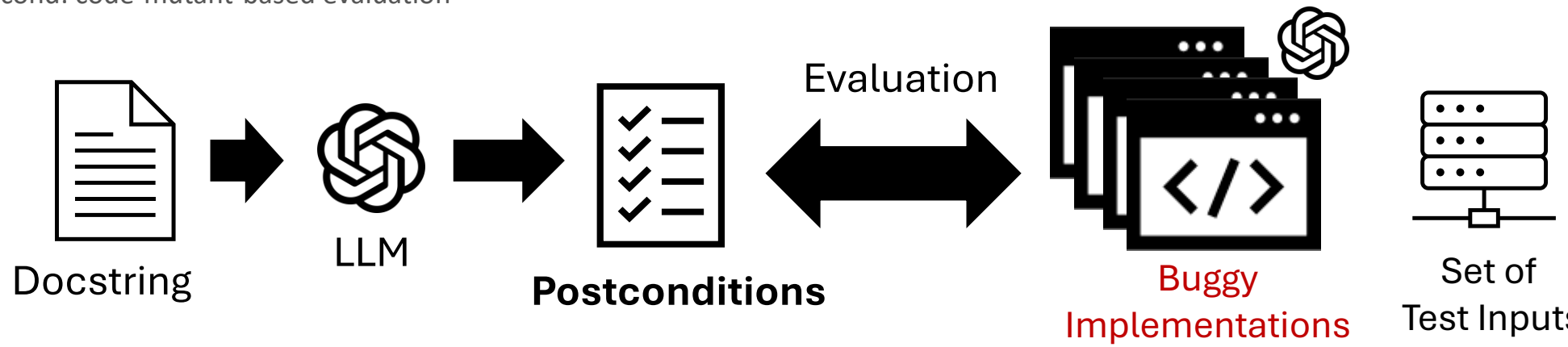
nl2postcond: code-mutant-based evaluation




| Model | Prompt | Prompt has: NL Only=✗ ref code=✓ | Accept @ 1 | Accept @ 5 | Accept @ 10 | x/164 correct |
|---|---|---|---|---|---|---|
| GPT-3.5 | base | ✗ | 0.46 | 0.80 | 0.87 | 143 |
| GPT-3.5 | base | ✓ | 0.49 | 0.81 | **0.88** | **145** |
| GPT-3.5 | simple | ✗ | 0.55 | **0.82** | 0.87 | 143 |
| GPT-3.5 | simple | ✓ | **0.56** | **0.82** | **0.88** | 144 |
| GPT-4 | base | ✗ | 0.63 | 0.83 | 0.88 | 144 |
| GPT-4 | base | ✓ | 0.71 | 0.89 | 0.91 | 150 |
| GPT-4 | simple | ✗ | **0.77** | **0.94** | **0.96** | **158** |
| GPT-4 | simple | ✓ | 0.76 | 0.92 | **0.96** | 157 |
| StarChat | base | ✗ | 0.21 | 0.61 | 0.82 | 134 |
| StarChat | base | ✓ | 0.20 | 0.59 | 0.77 | 126 |
| StarChat | simple | ✗ | **0.25** | **0.69** | 0.85 | 139 |
| StarChat | simple | ✓ | 0.23 | 0.67 | **0.86** | **141** |

# Python postconditions: *Completeness*

nl2postcond: code-mutant-based evaluation




| Model | Prompt | Prompt has: NL Only=✗ ref code=✓ | % bug-complete | % problems with bug-complete | % problems union bug-complete | Avg bug-completeness-score for correct postconditions *Natural bugs* | *All bugs* |
|---|---|---|---|---|---|---|---|
| GPT-3.5 | base | ✗ | 15.4 | 42.1 | 48.2 | 0.62 | 0.72 |
| GPT-3.5 | base | ✓ | **18.5** | **47.0** | **49.4** | **0.70** | **0.76** |
| GPT-3.5 | simple | ✗ | 8.1 | 29.3 | 33.5 | 0.44 | 0.55 |
| GPT-3.5 | simple | ✓ | 14.0 | 37.2 | 41.5 | 0.58 | 0.62 |
| GPT-4 | base | ✗ | **35.1** | **61.6** | **62.2** | **0.81** | **0.85** |
| GPT-4 | base | ✓ | 34.9 | 58.0 | 61.6 | 0.78 | 0.82 |
| GPT-4 | simple | ✗ | 9.2 | 26.2 | 29.3 | 0.40 | 0.52 |
| GPT-4 | simple | ✓ | 8.9 | 29.3 | 36.0 | 0.47 | 0.56 |
| StarChat | base | ✗ | 0.8 | 7.3 | 8.5 | 0.13 | 0.24 |
| StarChat | base | ✓ | 1.4 | 9.1 | 11.0 | **0.23** | 0.30 |
| StarChat | simple | ✗ | 1.5 | 6.7 | 7.3 | 0.16 | 0.24 |
| StarChat | simple | ✓ | **3.0** | **17.1** | **17.7** | 0.23 | **0.36** |

GPT-4 substantially better at complete specs

# From code mutants to test mutants

Soundness accepts valid behavior; completeness rejects incorrect behavior.

## Code-mutant-based

TiCoder / nl2postcond

Vary code; keep test inputs fixed.

### Soundness

The test/spec accepts known-correct implementations on evaluated inputs.

### Completeness

Reject buggy implementations on at least one input; measure mutant rejection.

## Test-mutant-based

Later symbolic-specification evaluation

Keep specs fixed; mutate I/O examples.

### Soundness

Known-valid I/O examples satisfy the specification.

### Completeness

Mutate expected outputs; measure the fraction rejected by the specification.

# Motivating example (Dafny)

"Write a function to find the shared elements from the given two lists.“ [MBPP task#2]

```
"test_cases": {
   "test_1": "var a1 := new int[] [3, 4, 5, 6];
              var a2 := new int[] [5, 7, 4, 10];
              var res1 :=SharedElements(a1,a2);
                        //expected[4, 5];",
   "test_2": “...",
   "test_3": “..."
  }
```

Hidden (not visible to spec generator)

```
predicate InArray(a: array<int>, x: int)
  reads a
  {exists i :: 0 <= i < a.Length && a[i] == x}


method SharedElements(a: array<int>, b: array<int>) returns (result: seq<int>)
  ensures forall x :: x in result ==> (InArray(a, x) && InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==> result[i] != result[j]
```

GPT4 generated specifcation. Labeled as “strong” by authors

[*Towards AI-Assisted Synthesis of Verified Dafny Methods*. Misu, Lopes, Ma, Noble. FSE’24]

# Symbolically validating formal specifications against tests

- An approach to evaluate the correctness of formal program specifications (given tests)
  - Formal specifications may not be executable (ghost code, quantifiers, ..)

- A step towards creating a **benchmark** for user-intent-formalization (specification quality) for programs in Dafny
  - Leverages the dataset from MBPP-DFY
  - “Adapts” the metric from nl2postcond for verification-aware languages

*[Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages,* Lahiri, FMCAD’24]

# Symbolic Spec Validation

- Symbolically test specifications (given tests as input/output examples)
  - Given
    - A method signature

      ```
      method Foo(x): (returns y) requires P(x) ensures Q(x, y)
      ```
    - A set of input/output tests T
- **Specification Soundness** (for a test (i, o)) //Boolean metric
  - `{P} x := i; y := o; {Q}` is valid
- **Specification Completeness** (for a test (i,o)) //Quantitative metric (kill-set from mutation-testing)
  - Fraction of mutants o' of o, s.t. `{P} x := i; y := o'; {Q}` is not valid

*[Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages,* Lahiri, FMCAD'24]

"Write a function to find the shared elements from the given two lists."

```
predicate InArray(a: array<int>, x: int)
  reads a
  {exists i :: 0 <= i < a.Length && a[i] == x}


method SharedElements(a: array<int>, b: array<int>)
returns (result: seq<int>)
  ensures forall x :: x in result ==> (InArray(a, x) &&
InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==>
result[i] != result[j]


method SharedElementsTest(){
  var a1:= new int[] [3, 4, 5, 6];
  var a2:= new int[] [5, 7, 4, 10];
  var res1:=SharedElements(a1,a2);
              //expected[4, 5];
}
```

Should verify

Should fail to verify

```
predicate InArray(a: array<int>, x: int)
reads a
{  exists i :: 0 <= i < a.Length && a[i] == x }

method SharedElements(a: array<int>, b: array<int>)
              returns (result: seq<int>)
  ensures forall x :: x in result ==>
                    (InArray(a, x) && InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==>
                    result[i] != result[j]
{
  var a1 := new int[] [3, 4, 5, 6];
  var a2 := new int[] [5, 7, 4, 10];
  assume {:axiom} a[..a.Length] == a1[..a1.Length];
  assert a[0] == a2[0] && .... && a[3] == a2[3];
  assume {:axiom} b[.. b.Length] == a2[..a2.Length];
  assert b[0] == a2[0] && .... && b[3] == a2[3];
  result := [4, 5];
}
```

```
predicate InArray(a: array<int>, x: int)
reads a
{  exists i :: 0 <= i < a.Length && a[i] == x }

method SharedElements(a: array<int>, b: array<int>)
              returns (result: seq<int>)
  ensures forall x :: x in result ==>
                    (InArray(a, x) && InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==>
                    result[i] != result[j]
{
  var a1 := new int[] [3, 4, 5, 6];
  var a2 := new int[] [5, 7, 4, 10];
  assume {:axiom} a[..a.Length] == a1[..a1.Length];
  assert a[0] == a2[0] && .... && a[3] == a2[3];
  assume {:axiom} b[.. b.Length] == a2[..a2.Length];
  assert b[0] == a2[0] && .... && b[3] == a2[3];
  result :=  [6];
}
```

# Evaluation https://github.com/microsoft/nl-2-postcond

- Evaluate the soundness/completeness metrics against human labels
  - Dataset: 153 Dafny specifications for MBPP-DFY dataset
  - Implemented **mutation operators** for simple output values (Booleans, integers, string, lists)
- Results
  - Automated metrics gets parity with the human-labeling for most examples
  - Soundness check satisfied for **64/150** examples
    - Limited by parsing and proof automation
  - Finds instances where a “strong” specification is not complete
  - Finds instances of “incorrect” specifications

# Result 1: label weak specifications

```
"Write a function to find the shared elements from the given two lists."
predicate InArray(a: array<int>, x: int)
  reads a
  {exists i :: 0 <= i < a.Length && a[i] == x}


method SharedElements(a: array<int>, b: array<int>) returns (result: seq<int>)
  ensures forall x :: x in result ==> (InArray(a, x) && InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==> result[i] != result[j]


method SharedElementsTest(){
  var a1:= new int[] [3, 4, 5, 6];
  var a2:= new int[] [5, 7, 4, 10];
  var res1:=SharedElements(a1,a2);
              //expected[4, 5];
}
```

GPT4 generated. Labeled as "strong"

Our metric marks this as a weak specification (wrt the test)

Changing ==> to <==> makes it a strong invariant by our metric

2 more such examples in the paper detected automatically

# Result 2: label incorrect specifications

```
method CubeVolume(size: int) returns (volume: int)
    requires size > 0
    ensures volume == size * size * size
{
    volume := size * size * size;
}
```

Our metric marks this as a
Incorrect specification (wrt the tests)


```
"234": {
  "task_id": "234",
  "task_description": "Write a method in Dafny to find the volume of a cube given its side length.",
  "method_signature": "method CubeVolume(l:int) returns (int)",
  "test_cases": {
    "test_1": "var out1:= CubeVolume(3);\nassert out1==27;",
    "test_2": "var out2:= CubeVolume(2);\nassert out2==8;",
    "test_3": "var out3:= CubeVolume(5);\nassert out3==25;"
  }
```


Problem: authors introduced error during copy-paste from Python -> Dafny!

2 more such examples in the paper detected automatically

# Challenges: quantifiers, unrolling, … (back in 2024)

- Intermediate assertions and triggers
  - Need redundant assertions to trigger the quantifiers [DONE]

```
assume {:axiom} a[..a.Length] == a1[..a1.Length];
assert a[0] == a1[0] && .... && a[3] == a1[3];
```

  - Trigger for nested quantifiers in specification predicates
    - `method IsNonPrime(n: int) returns (result: bool)`

```
requires n >= 2
ensures result <==> (exists k :: 2 <= k < n && n % k == 0)
```

- Unroll (or partially evaluate) recursive predicates based on concrete input

# IronSpec: Increasing the Reliability of Formal Specifications

Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos, *University of Michigan*

https://www.usenix.org/conference/osdi24/presentation/goldweber

Closely related work (independently done)

# Application: Specification model training for AutoVerus

The approach of scoring specifications given tests has been ported to **Verus** for boosting synthetic (spec, code, proof) generation for model training

**Automated Proof Generation for Rust Code via Self-Evolution**. Chen, Lu, **(Shan) Lu**, Gong, Yang, Li, Misu, Yu, Duan, Cheng, Yang, Lahiri, Xie, Zhou [arXiv:2410.15756], **ICLR 2025**

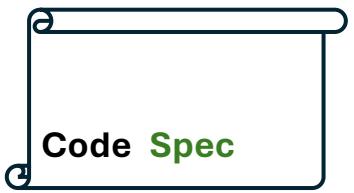
**Problem**: Fine-tune an LLM to produce high-quality Proof given <DocString, Code, Spec>

- Need a high quality < DocString, Code, Spec, Proof > tuples
  - **Sub-Problem**: Need a high quality < DocString, Code, Spec> tuples
    - **Baseline approach:** Specs generated were incorrect or weak leading to trivial proofs
    - **New Approach**: Fine-tune an LLM to produce high quality Spec given DocString and Tests using the metric from earlier slide

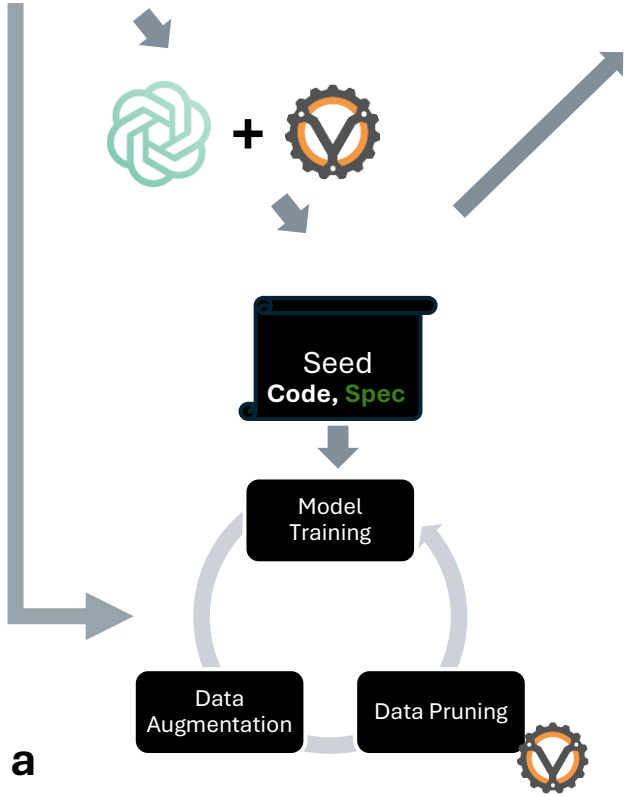
# AutoVerus training workflow




~45K Rust Programs
from code-synthesis dataset

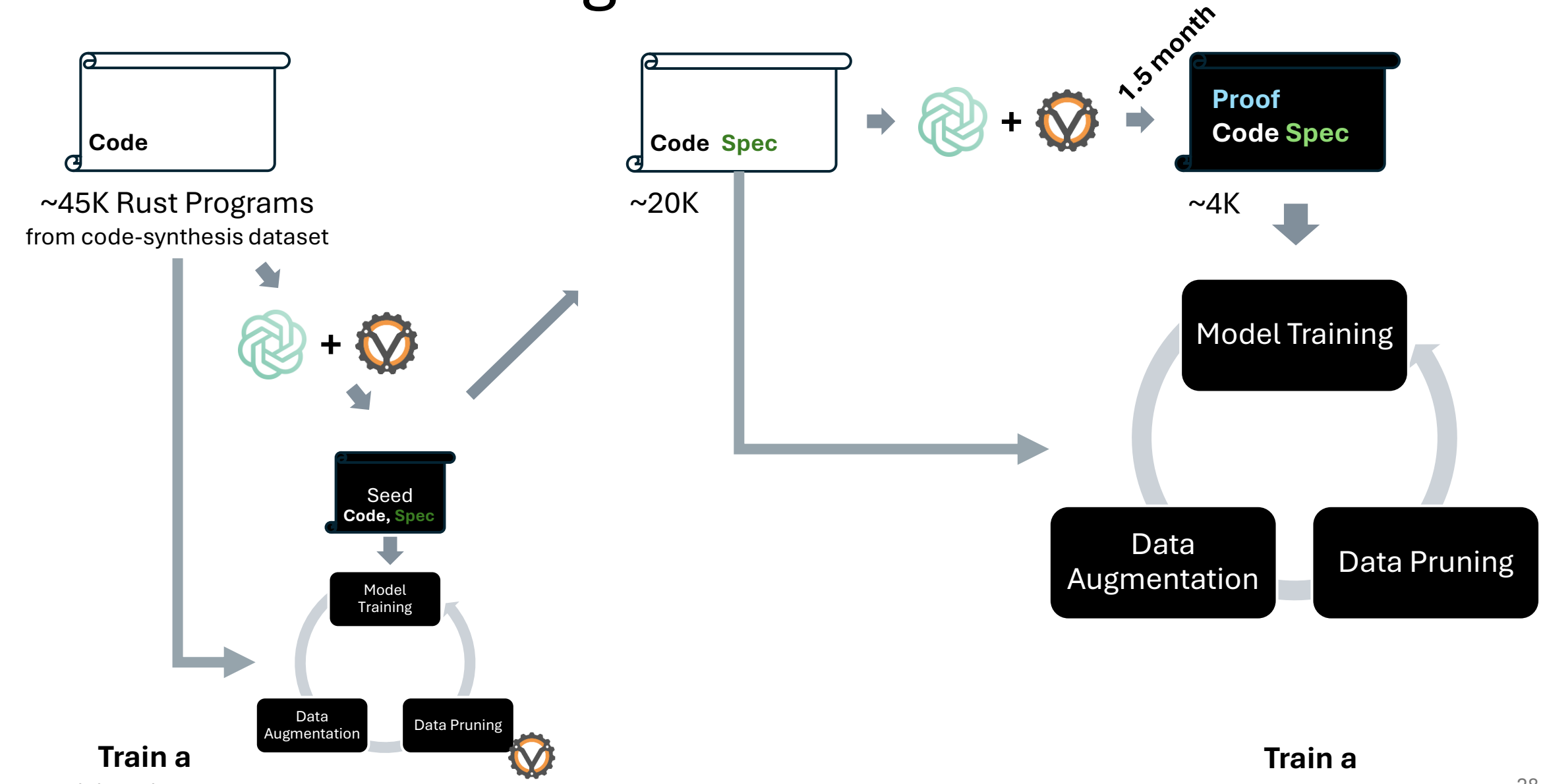



~20K



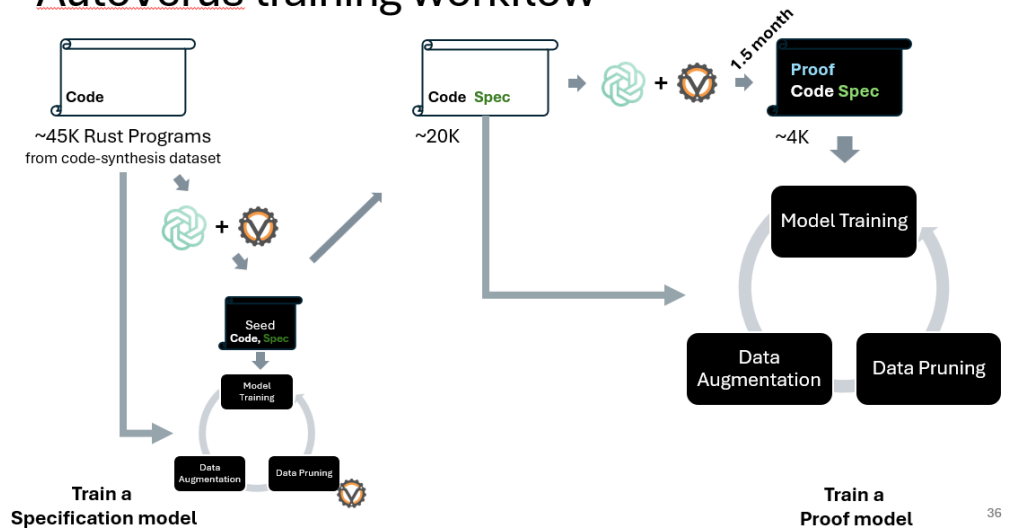

**Train a Specification model**

# AutoVerus training workflow



# Specification model training for Verus

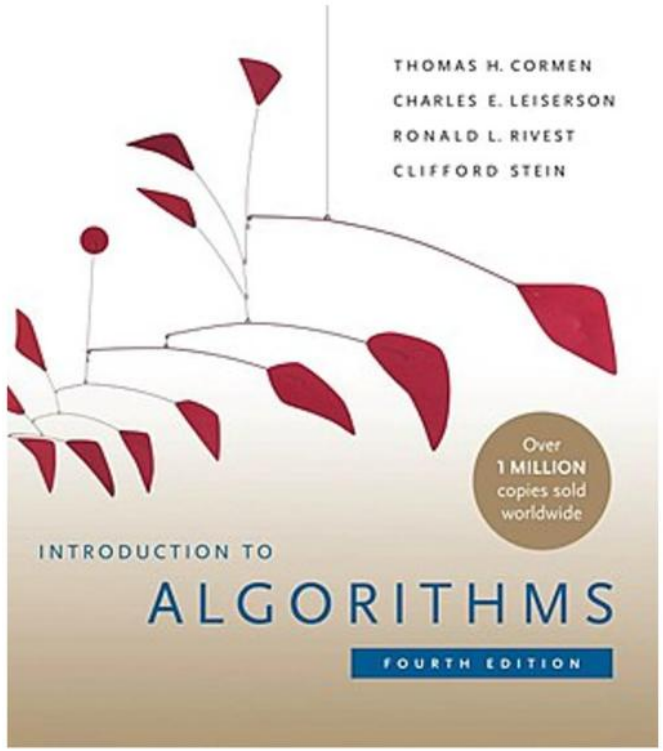
The approach of scoring specifications given tests has been ported to **Verus** for boosting synthetic (spec, code, proof) generation for model training



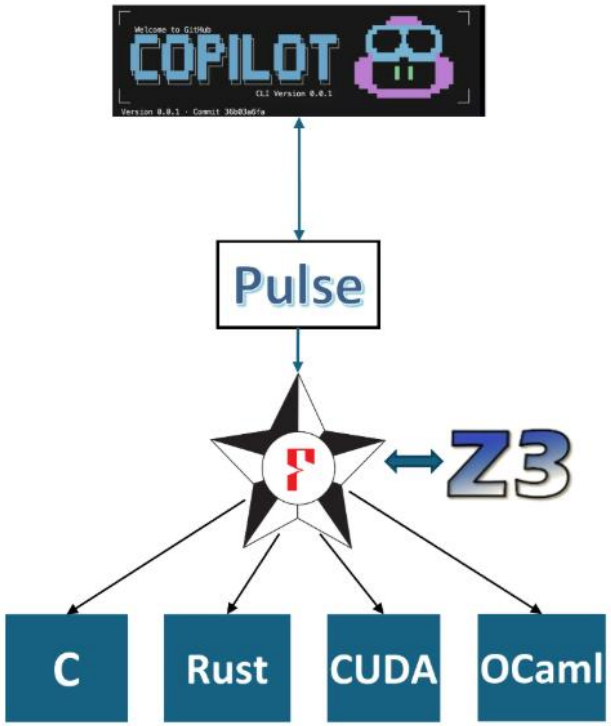

**Automated Proof Generation for Rust Code via Self-Evolution**. Chen, Lu, **(Shan) Lu**, Gong, Yang, Li, Misu, Yu, Duan, Cheng, Yang, Lahiri, Xie, Zhou [arXiv:2410.15756], **ICLR 2025**

In our experiments, SAFE leverages DeepSeekCoder (Guo et al., 2024) as the generator, successfully synthesizing 19,017 formal specifications and 9,706 verified Rust functions, from a dataset comprising 45,395 Rust functions sourced from the MBPP (Austin et al., 2021) training split and the CodeNet (Puri et al., 2021) training dataset. Note that, the initial Rust dataset contains **zero** lines of formal specification or proof. Our evaluation on a human-curated Verus benchmark with human-written specification, **VerusBench**, and a synthetic benchmark, **CodeNet-Test** demonstrate that SAFE empowers DeepSeekCoder, which is initially unacquainted with Verus, to achieve 43.17% and 43.83% accuracy on the two benchmarks by direct generation, far surpassing GPT-4o's performance of 11.51% and 0.28%. Furthermore, the model's accuracy reaches 79.14% in VerusBench and 48.43% in CodeNet-Test once its self-debugging feature is used.

From the results in Table 2, we can see that high-quality specifications contribute substantially to the end-to-end effectiveness of SAFE. If the model is trained without any high quality data ('low-quality spec-input'), the accuracy drops for all three subsets of VerusBench and even drops to 0% Accuracy@1 and 5.26% Accuracy@10 for SV. Probably because more debugging data is created in




&




https://risemsr.github.io/blog/2026-03-06-autoclrs/

RiSE MSR

Search

Blog

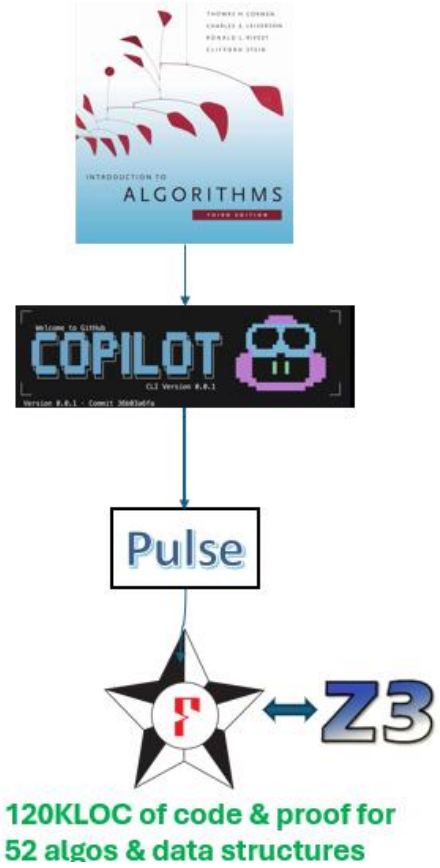
- Welcome to RiSE MSR Blog
- Agentic Proof-Oriented Programming
- How to train your program verifier
- Intent Formalization: A Grand Challenge for Reliable Coding in the Age of AI Agents
- **Formalizing Data Structures and Algorithms with Agents**

# Formalizing Data Structures and Algorithms with Agents

- Author: Nik Swamy with thanks to Gabriel Ebner, Lef Ioannidis, Guido Martinez, Matthai Philipose, Tahina Ramananandro

# AutoCLRS/Algo*

Formalizing data structures and algorithms using agents




| # | Algorithm | CLRS | Spec | Complexity |
|---|---|---|---|---|
| 1 | Insertion Sort | §2.1 | sorted ∧ permutation | Linked O($n^2$) |
| 2 | Merge Sort | §2.3 | sorted ∧ permutation | Linked O(n lg n) |
| 3 | Binary Search | §2.3 | found ⟺ key in array | Linked O(lg n) |
| 4 | Kadane (Max Subarray) | §4.1 | result = max contiguous sum | Linked O(n) |
| 5 | D&C Max Subarray | §4.1 | result = max contiguous sum | Pure O(n lg n) |
| 6 | Matrix Multiply | §4.2 | C = A · B | Linked O($n^3$) |
| 7 | Strassen | §4.2 | Strassen = standard mult | Pure O(n^{2.81}) |
| 8 | Heapsort | §6.4 | sorted ∧ permutation | Linked O(n lg n) |
| 9 | Partition | §7.1 | elements partitioned ∧ perm | Linked Θ(n) |
| 10 | Quicksort | §7.1 | sorted ∧ permutation | Linked O($n^2$) |
| 11 | Counting Sort | §8.2 | sorted ∧ permutation | — |
| 12 | Counting Sort (Stable) | §8.2 | stable sort ∧ perm | — |
| 13 | Radix Sort | §8.3 | sorted ∧ permutation | — |
| 14 | Bucket Sort | §8.4 | sorted ∧ permutation | — |
| 15 | Min / Max | §9.1 | result ∈ array ∧ is min/max | Linked Θ(n−1) |
| 16 | Simultaneous Min-Max | §9.1 | min ∧ max of array | Linked Θ(⌊3n/2⌋) |
| 17 | Quickselect | §9.2 | k-th smallest at position k | Linked O($n^2$) |
| 18 | Partial Selection Sort | §9.2 | first k sorted, k-th correct | Linked O(nk) |
| 19 | Stack | §10.1 | ghost list matches (LIFO) | — |
| 20 | Queue | §10.1 | ghost list matches (FIFO) | — |
| 21 | Singly Linked List | §10.2 | ghost list matches contents | Linked O(n) |
| 22 | Doubly-Linked List | §10.2 | segment pred ∧ ghost list | — |
| 23 | Hash Table (Linear Probing) | §11.4 | key ↦ value lookup | — |
| 24 | BST (Pointer-based) | §12.1–3 | key set ∧ BST invariant | Linked O(h) |
| 25 | BST (Array-based) | §12.1–3 | key set ∧ BST invariant | Linked O(h) |
| 26 | RB Tree (Okasaki) | §13.1–4 | BST ∧ RB invariants | Pure O(lg n) |
| 27 | RB Tree (CLRS-style) | §13.1–4 | BST ∧ RB invariants + delete | — |
| 28 | Rod Cutting | §15.1 | optimal revenue | Linked O($n^2$) |
| 29 | LCS | §15.4 | LCS length | Linked O(mn) |
| 30 | Matrix Chain | §15.2 | optimal parenthesization cost | Pure O($n^3$) |
| 31 | Activity Selection | §16.1 | greedy = optimal count | Linked O(n) |
| 32 | Huffman Coding | §16.3 | WPL-optimal prefix-free tree | Pure O($n^2$) |
| 33 | Union-Find | §21.3 | find/union maintain forest | — |
| 34 | BFS | §22.2 | distances ∧ parent tree | Linked O($V^2$) |
| 35 | DFS | §22.3 | timestamps ∧ classification | Linked O($V^2$) |
| 36 | Kahn Topological Sort | §22.4 | valid topological order | Linked O($V^2$) |
| 37 | Kruskal MST | §23.2 | forest of graph | Pure O($V^3$) |
| 38 | Prim MST | §23.2 | spanning tree + key correct | Pure O($V^2$) |
| 40 | MST Theory | §23.1 | cut/cycle properties | — |
| 41 | Bellman-Ford | §24.1 | SSSP distances | Linked O($V^3$) |
| 42 | Dijkstra | §24.3 | SSSP distances | Linked O($V^2$) |
| 43 | Floyd-Warshall | §25.2 | APSP distances | Linked Θ($n^3$) |
| 44 | Edmonds-Karp (Max Flow) | §26.2 | valid flow ∧ max flow | Pure O($VE^2$) |
| 45 | GCD / Extended GCD | §31.2 | gcd ∧ Bézout coefficients | Linked O(lg b) |
| 46 | Modular Exponentiation | §31.6 | result = b^e mod m | Linked O(lg e) |
| 47 | Naive String Match | §32.1 | all match positions | Linked O(nm) |
| 48 | KMP | §32.4 | all match positions | Linked O(n+m) |
| 49 | Rabin-Karp | §32.2 | all match positions | Pure O(nm) |
| 50 | Segments (Primitives) | §33.1 | correct intersection test | — |
| 51 | Graham Scan | §33.3 | convex hull (all left turns) | — |
| 52 | Jarvis March | §33.3 | convex hull vertices | — |
| 53 | Vertex Cover (2-approx) | §35.1 | cover ≤ 2 · OPT | — |

Question: how do we evaluate the specs?

# Small Proof-Oriented Tests (SPOTs) spec precision in AutoCLRS

| | | | |
|---|---|---|---|
| 02 | Insertion Sort | Deterministic | ✅ Precise |
| 02 | Merge Sort | Deterministic | ✅ Precise |
| 04 | Binary Search | Deterministic | ✅ Precise |
| 04 | Max Subarray (Kadane) | Deterministic | ✅ Precise |
| 04 | Matrix Multiply | Deterministic | ✅ Precise |
| 06 | Heapsort | Deterministic | ✅ Precise |
| 07 | Partition | **Relational** | ✅ Precise |
| 07 | Quicksort | Deterministic | ✅ Precise |
| 08 | Counting Sort (2 variants) | Deterministic | ✅ Precise |
| 09 | Min / Max | Deterministic | ✅ Precise |
| 09 | Simultaneous MinMax | Deterministic | ✅ Precise |
| 09 | Quickselect | Deterministic | ✅ Precise |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 11 | Hash Table | Relational/Precise | ⚠️ Weak insert | **Comprehensive** | 7 tests | 0 | Insert postcondition doesn't guarantee success |
| 12 | BST (Pointer) | Deterministic | ✅ Precise | **Comprehensive** | 14 pure + Pulse | 0 | None |
| 12 | BST (Array) | Relational/Existential | ❌ **Too weak** | Minimal | Partial | 0 | No reachability; insert→search broken |
| 21 | Union-Find | Relational | ⚠️ Moderate | Minimal | 1 union on 3 elts | 0 | Rank bound not preserved in postcondition |
| 22 | BFS | **Relational/Imprecise** | ⚠️ **Weak** | Minimal | 1 (3-vertex chain) | 0 | **No shortest-path optimality in postcondition** |
| 22 | DFS | Relational | ⚠️ **Weak** | Minimal | 1 (3-vertex chain) | 0 | **Spec↔Impl disconnect**; theorems not exposed |
| 22 | Topological Sort | **Relational** | ✅ Precise | Moderate | 1 (3-vertex DAG) | 0 | None — correctly relational |
| 23 | Kruskal | Relational | ⚠️ **Weak** | Minimal | 1 (3-vertex triangle) | 0 | **No spanning tree or MST property** |
| 23 | Prim | Functional but weak | ❌ **Critical** | Minimal | 1 (3-vertex triangle) | 0 | **Postcondition admits incorrect outputs** |
| 24 | Bellman-Ford | Relational/Conditional | ⚠️ Moderate | Adequate | 1 (3-vertex, neg wts) | 0 | Correctness conditional on runtime boolean |

**test(i,o) { y = algorithm(i); assert(y == o) }**

# A typical specification gap: Underspecified error conditions

```
fn hash_insert
  (table: A.array int)
  (#s: erased (Seq.seq int))
  (size: SZ.t)
  (key: int{key >= 0 /\ SZ.fits key})
  (ctr: GR.ref nat)
  (#c0: erased nat)
  requires
    A.pts_to table s **
    GR.pts_to ctr c0 **
    pure (SZ.v size > 0 /\ Seq.length s == SZ.v size /\
          valid_ht s (SZ.v size))
  returns result: bool
  ensures exists* s' cf.
    A.pts_to table s' **
    GR.pts_to ctr cf **
    pure (
      Seq.length s' == SZ.v size /\
      Seq.length s' == Seq.length s /\
      valid_ht s' (SZ.v size) /\
      (if result
       then (key_in_table s' (SZ.v size) key /\
             key_findable s' (SZ.v size) key /\
             (exists (idx: nat). idx < SZ.v size /\
               (Seq.index s idx == -1 \/ Seq.index s idx == -2) /\
               Seq.index s' idx == key /\
               seq_modified_at s s' idx))
       else s' == s) /\
      cf >= reveal c0 /\ cf - reveal c0 <= SZ.v size
    )
```

**Does not say when result is allowed to be false**

```
// Insert key 0
let b = hash_insert table 3sz 0 ctr;

if b {
  // === Insert succeeded ===
  // From insert postcondition:
  //   key_in_table s' 3 0 /\ key_findable s' 3 0 /\ valid_ht s' 3

  // Search for key 0 — should find it
  let r = hash_search table 3sz 0 ctr;

  // Postcondition precision: search must find key 0.
  //
  // From insert: key_in_table s' 3 0
  // From search postcondition: SZ.v r == 3 ==> ~(key_in_table s' 3 0)
  // Contrapositive: key_in_table s' 3 0 ==> SZ.v r != 3
  // Combined with SZ.v r <= 3: SZ.v r < 3
  assert (pure (SZ.v r < 3));

  // Cleanup
  GR.free ctr;
  with s. assert (A.pts_to table s);
  rewrite (A.pts_to table s) as (A.pts_to (V.vec_to_array tv) s);
  hash_table_free tv;
} else {
  // === Insert failed ===
  // Postcondition: s' == s (table unchanged)
  // This branch represents a spec weakness: the postcondition of hash_insert
  // does not guarantee success when empty slots are available.
```

**Unable to prove that this branch is unreachable**

```
  // Cleanup
  GR.free ctr;
  with s. assert (A.pts_to table s);
  rewrite (A.pts_to table s) as (A.pts_to (V.vec_to_array tv) s);
  hash_table_free tv;
}
```

```
//SNIPPET_START: ht_hash_insert
// Insert a key into the hash table
// Returns true if successful, false if table is full
// Proves both correctness and O(n) complexity
fn hash_insert
  (table: A.array int)
  (#s: erased (Seq.seq int))
  (size: SZ.t)
  (key: int{key >= 0 /\ SZ.fits key})
  (ctr: GR.ref nat)
  (#c0: erased nat)
  requires
    A.pts_to table s **
    GR.pts_to ctr c0 **
    pure (SZ.v size > 0 /\ Seq.length s == SZ.v size /\
          valid_ht s (SZ.v size))
  returns result: bool
  ensures exists* s' cf.
    A.pts_to table s' **
    GR.pts_to ctr cf **
    pure (
      Seq.length s' == SZ.v size /\
      Seq.length s' == Seq.length s /\
      valid_ht s' (SZ.v size) /\
      (if result
       then (key_in_table s' (SZ.v size) key /\
             key_findable s' (SZ.v size) key /\
             (exists (idx: nat). idx < SZ.v size /\
               (Seq.index s idx == -1 \/ Seq.index s idx == -2) /\
               Seq.index s' idx == key /\
               seq_modified_at s s' idx))
       else (s' == s /\
             (forall (q: nat). {:pattern (hash_probe_nat key q (SZ.v size))}
               q < SZ.v size ==>
                 Seq.index s (hash_probe_nat key q (SZ.v size)) =!= -1 /\
                 Seq.index s (hash_probe_nat key q (SZ.v size)) =!= -2))) /\
      cf >= reveal c0 /\ cf - reveal c0 <= SZ.v size
    )
//SNIPPET_END: ht_hash_insert
```

**Strengthened postcondition**
**Allows proving that the result must be true in this SPOT**

```
fn test_insert_then_search ()
  requires emp
  returns res: bool
  ensures pure (res == true)
{
  let tv = hash_table_create 3sz;
  let table = V.vec_to_array tv;
  rewrite (A.pts_to (V.vec_to_array tv) (Seq.create 3 (-1)))
          as (A.pts_to table (Seq.create 3 (-1)));
  let ctr = GR.alloc #nat 0;

  // Insert key 0
  let b = hash_insert table 3sz 0 ctr;
  trigger_insert_empty 3 0;
  assert (pure (b == true));

  // Search for key 0 — must find it
  let r = hash_search table 3sz 0 ctr;
  assert (pure (SZ.v r < 3));

  // Compute runtime-observable result
  let ok = (b && (r <^ 3sz));
  assert (pure (ok == true));

  // Cleanup
  GR.free ctr;
  with s. assert (A.pts_to table s);
  rewrite (A.pts_to table s) as (A.pts_to (V.vec_to_array tv) s);
  hash_table_free tv;

  ok
}
```

| | | | |
|---|---|---|---|
| 02 | Insertion Sort | Deterministic | ✅ Precise |
| 02 | Merge Sort | Deterministic | ✅ Precise |
| 04 | Binary Search | Deterministic | ✅ Precise |
| 04 | Max Subarray (Kadane) | Deterministic | ✅ Precise |
| 04 | Matrix Multiply | Deterministic | ✅ Precise |
| 06 | Heapsort | Deterministic | ✅ Precise |
| 07 | Partition | **Relational** | ✅ Precise |
| 07 | Quicksort | Deterministic | ✅ Precise |
| 08 | Counting Sort (2 variants) | Deterministic | ✅ Precise |
| 09 | Min / Max | Deterministic | ✅ Precise |
| 09 | Simultaneous MinMax | Deterministic | ✅ Precise |
| 09 | Quickselect | Deterministic | ✅ Precise |
| 09 | Partial Selection Sort | Deterministic | ✅ Precise |
| 10 | Stack | Deterministic | ✅ Precise |
| 10 | Queue | Deterministic | ✅ Precise |
| 10 | Doubly Linked List | Deterministic | ✅ Precise |
| 10 | Singly Linked List | Deterministic | ✅ Precise |
| 11 | Hash Table | Deterministic | ✅ Precise |
| 12 | BST (Pointer) | Deterministic | ✅ Precise |
| 12 | BST (Array) | Deterministic | ✅ Precise |
| 13 | RB-Tree (Okasaki) | Deterministic | ✅ Precise |
| 13 | RB-Tree (CLRS) | Deterministic | ✅ Precise |
| 15 | Rod Cutting | Deterministic | ✅ Precise |
| 15 | Matrix Chain | Deterministic | ✅ Precise |
| 15 | LCS | Deterministic | ✅ Precise |
| 16 | Activity Selection | **Precise Relational** | ✅ Precise |
| 16 | Huffman Tree | **Precise Relational** | ✅ Precise |
| 16 | Huffman Codec | Deterministic | ✅ Precise |
| 21 | Union-Find | Relational | ✅ Precise |
| 22 | BFS | Relational | ✅ Precise |
| 22 | DFS | Relational | ✅ Precise |
| 22 | Topological Sort | **Relational** | ✅ Precise |
| 23 | Kruskal | Relational | ✅ Precise |
| 23 | Prim | Relational | ✅ Precise |
| 24 | Bellman-Ford | Deterministic (conditional) | ✅ Precise |
| 24 | Dijkstra | Deterministic | ✅ Precise |
| 25 | Floyd-Warshall | Deterministic | ✅ Precise |
| 26 | Max Flow (Edmonds-Karp) | Deterministic + Optimality | ✅ Precise |
| 31 | GCD | Deterministic | ✅ Precise |
| 31 | Extended GCD | Deterministic | ✅ Precise |
| 31 | ModExp (R-to-L) | Deterministic | ✅ Precise |
| 31 | ModExp (L-to-R) | Deterministic | ✅ Precise |
| 32 | Naive String Match | Deterministic | ✅ Precise |
| 32 | KMP | Deterministic | ✅ Precise |
| 32 | Rabin-Karp | Deterministic | ✅ Precise |
| 33 | Segments (primitives) | Deterministic | ✅ Precise |
| 33 | Graham Scan | Deterministic | ✅ Precise |
| 33 | Jarvis March | Deterministic | ✅ Precise |
| 35 | Vertex Cover (2-approx) | **Relational** | ✅ Precise |

[AlgoStar/autoclrs/TEST_REVIEW_REPORT.md at main · FStarLang/AlgoStar]

After iterating with the model to fix specs to prove all SPOTs (**CEGAR for spec repair**)

Enabled by proof automation of SPOTs, previously manual before 2026

# Verification needs an environment model

Given intent formalization and proof automation, what remains?

### What a proof still needs

Library and API behavior
Preconditions and calling context

**The environment model is often present as informal documentation.**

**Can we turn documentation into formal environment contracts, and validate them against tests (given or generated)?**

# Neuro-formal verification

Agentic, language-agnostic formal program reasoning

**Vision: 99.99% accuracy separating correct from incorrect programs, with every verdict backed by a machine-checked proof.**

PROGRAM

## Source code in L

Python, Java, C, ...

PROPERTY

## User intent

Docstring, issue, assertion

ENVIRONMENT

## Informal documentation

Library and framework behavior

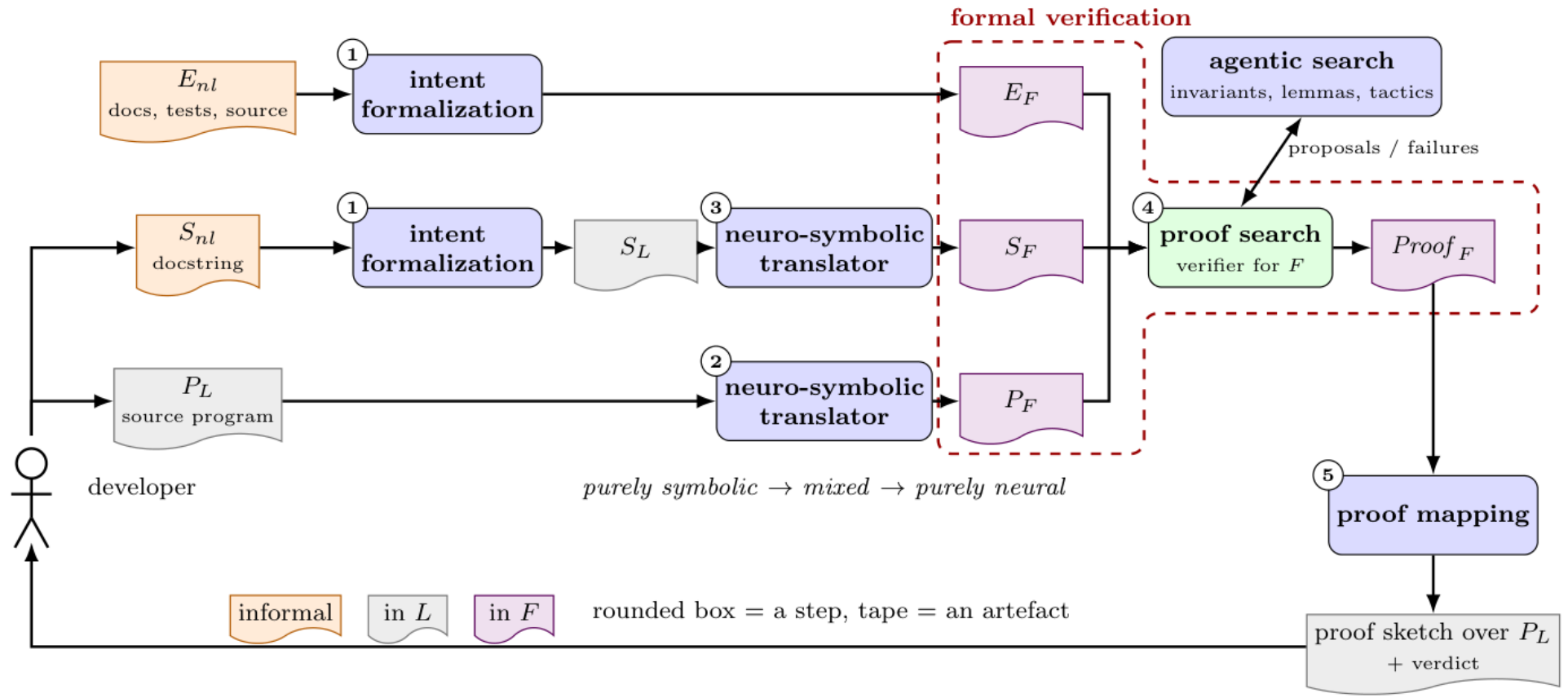
Validate property and environment specs against tests (given or generated).
API/library documentation -> formal environment contracts -> test audit.

**Every verdict is backed by a proof about a model of {program, property, environment}**

https://arxiv.org/abs/2608.21516

# From informal intent to a checked argument

Intent formalization + translation + proof search + proof mapping




The formal proof checks P_F, S_F and E_F; proof mapping reconnects it to the developer.

# Proofs help. Modeling fidelity still matters.

https://github.com/microsoft/neuro-formal-verification

Dafny evaluation: 103 tasks, 206 entries; 5 attempts x 10 proof repair rounds

| Task | Population | Verdicts issued | Wrong verdicts | Precision |
|---|---|---|---|---|
| Verification | 103 | 77 | 5 | 93.5% |
| Refutation | 103 | 53 | 6 | 88.7% |
| **Refute, then verify** | **206** | **128** | **10** | **92.2%** |

## Both verification and refutations are accompanied by proofs

Programs are small (median 7 LOC), but target real Python libraries

**Proofs and Staging helps**

- **LLM-as-judge:** 72% precision while answering every entry, without any checkable artifact.
- **No staging:** an agent-verifier combination proves 98% of both correct and known-buggy programs, yielding only 50% precision.

# Summary

From generated specifications to evidence about intent.

## Synthesize, then validate

Formalize intent as specs; validate them against tests (given or generated).

## Reduce spec audit to test audit

Assess soundness, completeness and preciseness; audit the tests themselves.

## Choose the right mutants

Code mutants: TiCoder / nl2postcond.
Test mutants: symbolic-spec evaluation.

## Applications and the next step

Dafny, AutoVerus and F*/Pulse SPOTs;
NFV targets mainstream developers.

**Proofs validate models of programs. Intent modeling fidelity still needs validation.**

# Related work

- **Expecto: Extracting Formal Specifications from Natural Language Description for Trustworthy Oracles**
  Dongjae Lee & Kihong Heo (PLDI'26)
- **Property-based testing of formal specifications**
  Ilya Sergey (2026, blog)
- **Property templates for proofs and property-based tests**
  Lee, Wu & Miryung Kim (ICSE'27, to appear)
- **MutDafny: A Mutation-Based Approach to Assess Dafny Specifications**
  Amaral, Mendes & Campos (ICSE'26)
- **VERINA: Benchmarking Verifiable Code Generation**
  Ye et al. (ICLR'26)

*And several others... maybe several since last night :-)*

# Questions



Intent formalization





Neuro-formal verification



RiSE