Certified program synthesis with a multi-modal verifier


Yueyang Feng[1,*]


Dipesh Kafle[1,*]


Vladimir Gladshtein[1]


Vitaly Kurin[2]


George Pîrlea[1]


Qiyuan Zhao[1]


Peter Müller[3]


Ilya Sergey[1]

[1]National University of Singapore [2]Neapolis University Pafos [3]ETH Zurich

[*]Joint first authors

ASE '26 · October 15, 2026 · Munich, Germany

Motivation

AI writes code for us. An idea becomes an implementation in minutes.



Images are AI-generated

Motivation

AI writes code for us. An idea becomes an implementation in minutes.

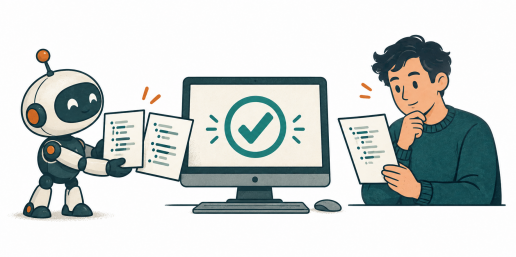


How do we know it does what we asked?

Establishing confidence still takes a tremendous amount of work, despite code generation being fast.

Images are AI-generated

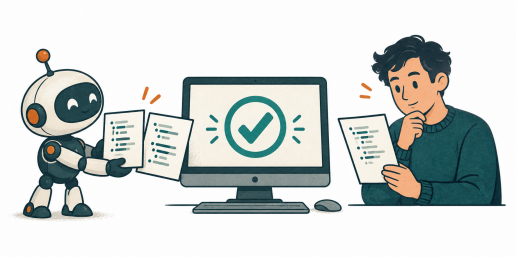
Motivation



Ask the AI for a **checkable** proof, too.

Images are AI-generated

Motivation



Ask the AI for a **checkable** proof, too.

This is not a new problem: many scientific communities have struggled with it.

A recent AI-for-math result: Fermat's Last Theorem in Lean Anthropic, September 2026

$$\forall\, a\, b\, c\, n : \mathbb{N},\ a \neq 0 \to b \neq 0 \to c \neq 0 \to n \geq 3 \to a^n + b^n \neq c^n$$

Specification	1–2 lines, read by a human
Proof	13 million lines of Lean, checked by Lean

A recent AI-for-math result: Fermat's Last Theorem in Lean Anthropic, September 2026

$$\forall\, a\, b\, c\, n : \mathbb{N},\ a \neq 0 \to b \neq 0 \to c \neq 0 \to n \geq 3 \to a^n + b^n \neq c^n$$

Specification	1–2 lines, read by a human
Proof	13 million lines of Lean, checked by Lean

We want the same luxury for programs.

User gives a task; AI gives a specification + implementation + proof.

A recent AI-for-math result: Fermat's Last Theorem in Lean Anthropic, September 2026

$$\forall\, a\, b\, c\, n : \mathbb{N},\ a \neq 0 \to b \neq 0 \to c \neq 0 \to n \geq 3 \to a^n + b^n \neq c^n$$

Specification	1–2 lines, read by a human
Proof	13 million lines of Lean, checked by Lean

We want the same luxury for programs.

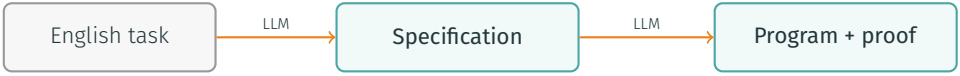
User gives a task; AI gives a specification + implementation + proof.

But LLMs are notoriously good at cheating: `sorry` trivial specifications ...

"Make no mistakes" is not enough. We need to be cleverer.

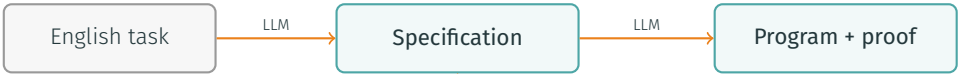
Our goal: Certified synthesis from natural language

Generate a runnable program and a checked correctness proof from an English task.



Our goal: Certified synthesis from natural language

Generate a runnable program and a checked correctness proof from an English task.



Challenge 1: is the specification right?

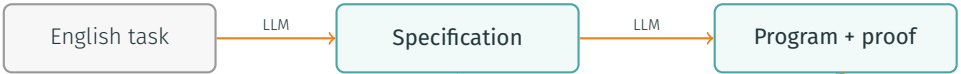
```
-- postcondition
def isSorted (arr : Array Int) (res : Array Int) :=
  ∀ i j, i < j → j < res.size → res[i]! ≤ res[j]!

method sort (arr : Array Int)
    return (res : Array Int)
  ensures isSorted arr res
  do return #[]
```

✓ verified!

Our goal: Certified synthesis from natural language

Generate a runnable program and a checked correctness proof from an English task.



Challenge 1: is the specification right?

```
-- postcondition
def isSorted (arr : Array Int) (res : Array Int) :=
  ∀ i j, i < j → j < res.size → res[i]! ≤ res[j]!

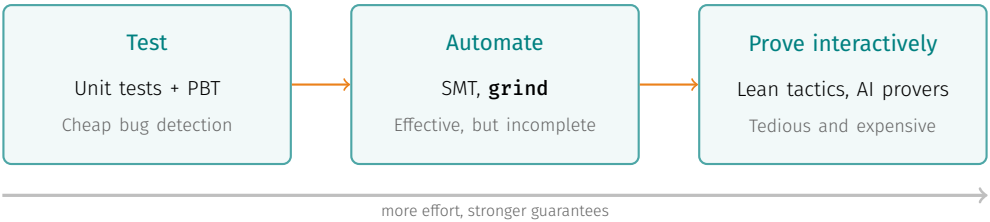
method sort (arr : Array Int)
    return (res : Array Int)
  ensures isSorted arr res
  do return #[]
```

✓ verified!

Challenge 2: can we finish the proof?

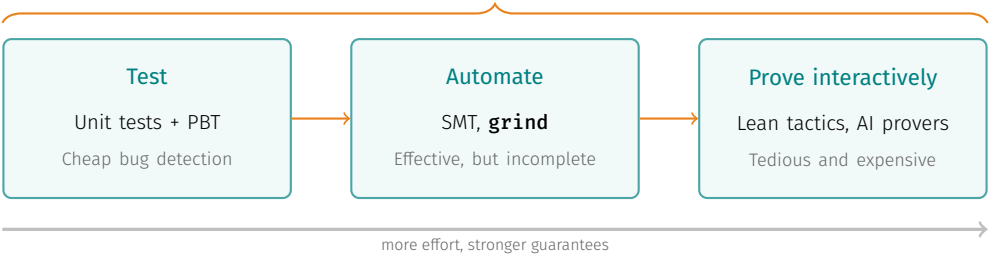
Proofs are long and expensive, and we want to minimize the tokens spent on them.

Our approach: Multi-modal verification



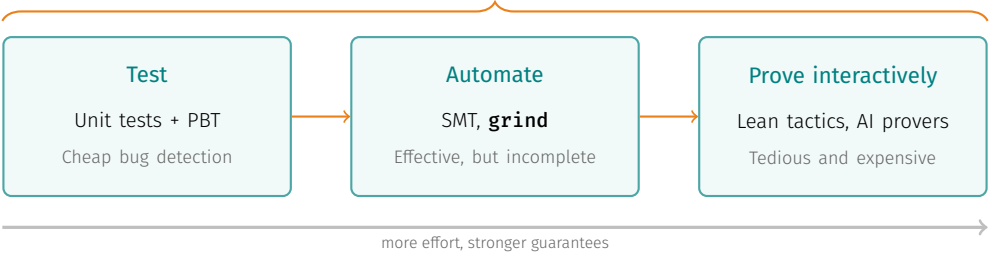
Our approach: Multi-modal verification

Velvet: a Lean library for imperative programs that combines these ***modes***



Our approach: Multi-modal verification

Velvet: a Lean library for imperative programs that combines these ***modes***



Our agentic pipeline, LeetProof, uses Velvet's multi-modality
to turn English tasks into verified programs efficiently.

Velvet 101

```
method IsNonPrime (n : Nat) return (result : Bool)
  ensures result = true ↔ ¬isPrime n
  do
    if n ≤ 1 then return true
    let mut i : Nat := 2
    let mut ret : Bool := false
    while i * i ≤ n
    invariant ¬ret ↔ ∀d, 2 ≤ d ∧ d < i → n % d ≠ 0
    invariant i ≥ 2 ∧ (i - 1) * (i - 1) ≤ n
    do
      if n % i = 0 then ret := true
      i := i + 1
    return ret

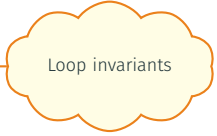
prove_correct IsNonPrime by
  loom_solve  -- automated VC discharge
  ...         -- remaining interactive proof
```

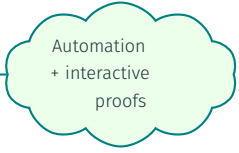
Velvet 101

```
method IsNonPrime (n : Nat) return (result : Bool)
  ensures result = true ↔ ¬isPrime n
  do
    if n ≤ 1 then return true
    let mut i : Nat := 2
    let mut ret : Bool := false
    while i * i ≤ n
    invariant ¬ret ↔ ∀d, 2 ≤ d ∧ d < i → n % d ≠ 0
    invariant i ≥ 2 ∧ (i - 1) * (i - 1) ≤ n
    do
      if n % i = 0 then ret := true
      i := i + 1
    return ret

prove_correct IsNonPrime by
  loom_solve  -- automated VC discharge
  ...         -- remaining interactive proof
```

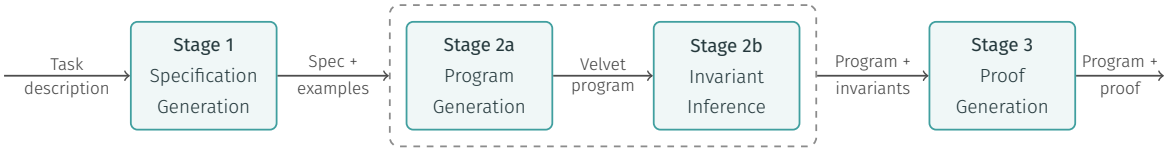




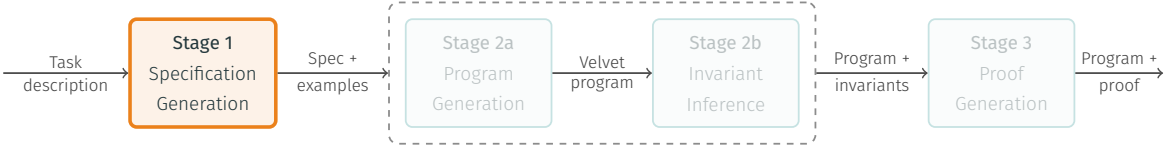


LeetProof pipeline

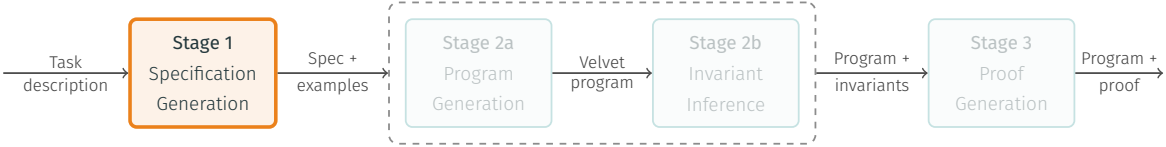
LeetProof turns an English task into a verified Velvet program in stages.



Specification generation



Specification generation



Intent capture from English cannot be guaranteed,
but we can effectively test for mismatches.

Step 1: Check the LLM's understanding

Task description

Sort the array in non-decreasing order.

Example: `[1, 3, 2]` ↦ `[1, 2, 3]`



LLM

Specification

```
def precondition (arr : Array Int) :=
  arr.size > 0

-- postcondition
def isSorted (arr : Array Int) (res : Array Int) :=
  ∀ i j, i < j → j < res.size → res[i]! ≤ res[j]!
```

Input	Expected output
`[1, 3, 2]`	`[1, 2, 3]`
`[5, 5, 1]`	`[1, 5, 5]`
`[]`	`[]`

... about 10 more

The LLM writes the specification, plus examples from its own reading of the task.

Step 2: Test the specification against the examples

Proving every example is expensive, so we search for counterexamples with property-based testing (PBT).

```
def precondition (arr : Array Int) := arr.size > 0
-- postcondition
def isSorted (arr : Array Int) (res : Array Int) :=
  ∀ i j, i < j → j < res.size → res[i]! ≤ res[j]!
```

`[1,3,2]` ↦ `[1,2,3]` `[]` ↦ `[]`

Step 2: Test the specification against the examples

Proving every example is expensive, so we search for counterexamples with property-based testing (PBT).

```
def precondition (arr : Array Int) := arr.size > 0
-- postcondition
def isSorted (arr : Array Int) (res : Array Int) :=
  ∀ i j, i < j → j < res.size → res[i]! ≤ res[j]!
```

		`[1,3,2]` ↦ `[1,2,3]`	`[]` ↦ `[]`
Input validity	Does the input satisfy **`precondition`**?	✓	✗

Sorting **`[]`** is a valid task, but **`precondition`** rejects it: it is too strong.

Step 2: Test the specification against the examples

Proving every example is expensive, so we search for counterexamples with property-based testing (PBT).

```
def precondition (arr : Array Int) := arr.size > 0
-- postcondition
def isSorted (arr : Array Int) (res : Array Int) :=
  ∀ i j, i < j → j < res.size → res[i]! ≤ res[j]!
```

		`[1,3,2]` ↦ `[1,2,3]`	`[]` ↦ `[]`
Input validity	Does the input satisfy **`precondition`**?	✓	✗
Expected output	Does the output satisfy the postcondition?	✓	✓

Sorting **`[]`** is a valid task, but **`precondition`** rejects it: it is too strong.

Step 2: Test the specification against the examples

Proving every example is expensive, so we search for counterexamples with property-based testing (PBT).

```
def precondition (arr : Array Int) := arr.size > 0
-- postcondition
def isSorted (arr : Array Int) (res : Array Int) :=
  ∀ i j, i < j → j < res.size → res[i]! ≤ res[j]!
```

		`[1,3,2]` ↦ `[1,2,3]`	`[]` ↦ `[]`
Input validity	Does the input satisfy **`precondition`**?	✓	✗
Expected output	Does the output satisfy the postcondition?	✓	✓
Output uniqueness	Is every other output rejected?	?	–

Sorting **`[]`** is a valid task, but **`precondition`** rejects it: it is too strong.

Output uniqueness: mutate the expected output

If the spec admits a wrong output, it is usually close to the expected one.

So PBT mutates the expected output and checks the postcondition again.

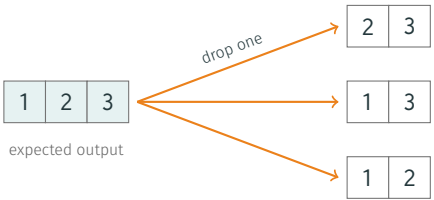


expected output

Output uniqueness: mutate the expected output

If the spec admits a wrong output, it is usually close to the expected one.

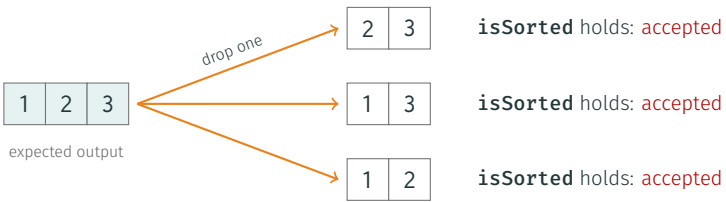
So PBT mutates the expected output and checks the postcondition again.



Output uniqueness: mutate the expected output

If the spec admits a wrong output, it is usually close to the expected one.

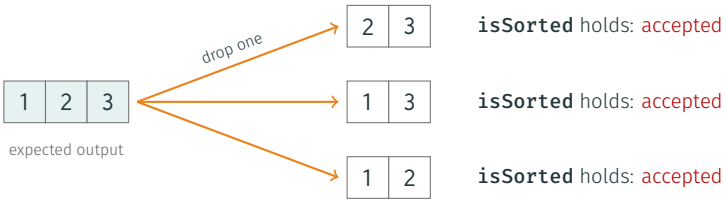
So PBT mutates the expected output and checks the postcondition again.



Output uniqueness: mutate the expected output

If the spec admits a wrong output, it is usually close to the expected one.

So PBT mutates the expected output and checks the postcondition again.



Rejected by PBT: the postcondition is too weak.

The LLM revises the specification

```
def precondition (arr : Array Int) := True
-- postcondition
def sortedPerm (arr : Array Int) (res : Array Int) :=
  isSorted arr res ∧ isPerm res arr    -- same elements as the input
```

	`[1,3,2]` ↦ `[1,2,3]`	`[]` ↦ `[]`
Input validity	✓	✓
Expected output	✓	✓
Output uniqueness	✓	✓

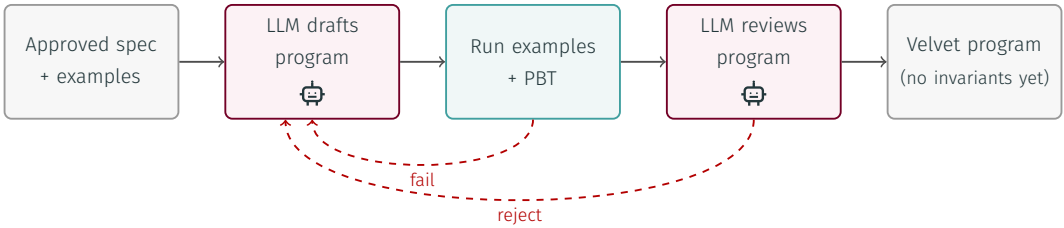
The LLM revises the specification

```
def precondition (arr : Array Int) := True
-- postcondition
def sortedPerm (arr : Array Int) (res : Array Int) :=
  isSorted arr res ∧ isPerm res arr    -- same elements as the input
```

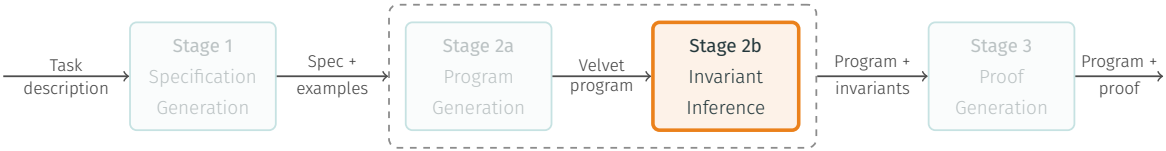
	`[1,3,2]` ↦ `[1,2,3]`	`[]` ↦ `[]`
Input validity	✓	✓
Expected output	✓	✓
Output uniqueness	✓	✓

Approved by the pipeline

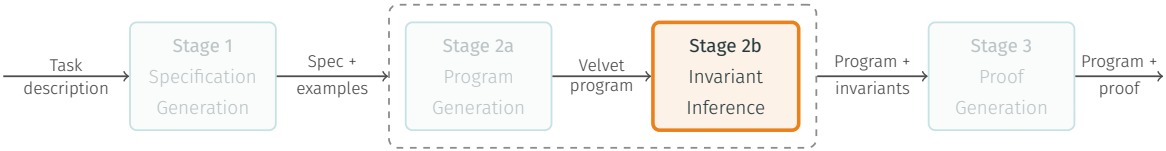
Program generation



Invariant inference

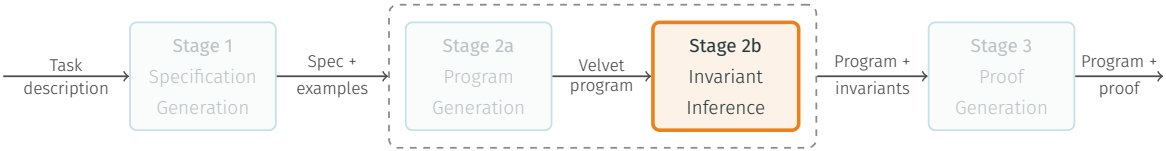


Invariant inference



The program is generated. To prove it correct, every loop needs **invariants**, facts that hold before and after every iteration.

Invariant inference



The program is generated. To prove it correct, every loop needs **invariants**, facts that hold before and after every iteration.

Finding inductive invariants is a notoriously difficult problem.

How do we do this effectively?

Step 1: The LLM proposes loop invariants

```
let mut res := arr
let mut n : Nat := 1
while n ≠ res.size
  invariant n_le: n ≤ res.size
  invariant sorted: isSorted arr res
  invariant same_elems: isPerm res arr
do
  -- insert res[n] into the sorted prefix (inner loop)
  ...
  n := n + 1
return res
```

Invariant inference

Step 1: The LLM proposes loop invariants

```
let mut res := arr
let mut n : Nat := 1
while n ≠ res.size
  invariant n_le: n ≤ res.size
  invariant sorted: isSorted arr res
  invariant same_elems: isPerm res arr
do
  -- insert res[n] into the sorted prefix (inner loop)
  ...
  n := n + 1
return res
```

Looks plausible… but is it true?

Step 2: Validate the invariants

Automation (SMT, **grind**) tries to prove the obligations; for the rest, PBT searches for a counterexample.

```
invariant sorted: isSorted arr res
```

Step 2: Validate the invariants

Automation (SMT, **grind**) tries to prove the obligations; for the rest, PBT searches for a counterexample.

```
invariant sorted: isSorted arr res
```

```
[velvet_plausible_test] FAIL:
invariant "sorted" doesn't hold
arr = #[2, 1]
```

Step 2: Validate the invariants

Automation (SMT, **grind**) tries to prove the obligations; for the rest, PBT searches for a counterexample.

```
invariant sorted: isSorted arr res
```

```
[velvet_plausible_test] FAIL:
invariant "sorted" doesn't hold
arr = #[2, 1]
```

	n	res	n_le	sorted	same_elems
loop entry	1	2 1	✓	✗	✓
after iteration 1	2	1 2	✓	✓	✓

Step 2: Validate the invariants

Automation (SMT, **`grind`**) tries to prove the obligations; for the rest, PBT searches for a counterexample.

```
invariant sorted: isSorted arr res
```

```
[velvet_plausible_test] FAIL:
invariant "sorted" doesn't hold
arr = #[2, 1]
```

	`n`	`res`	`n_le`	`sorted`	`same_elems`
loop entry	1	2 1	✓	✗	✓
after iteration 1	2	1 2	✓	✓	✓

`isSorted` only holds once the loop finishes; during the loop only the prefix **`res[0..n)`** is sorted.

Only **`sorted`** goes back to the LLM, with this counterexample. The rest are kept.

Step 2: Validate the invariants

Automation (SMT, **grind**) tries to prove the obligations; for the rest, PBT searches for a counterexample.

```
invariant sorted_prefix: isSorted arr res[0..n)
```

PBT finds no counterexample ✓

	n	res	n_le	sorted_prefix	same_elems
loop entry	1	2 1	✓	✓	✓
after iteration 1	2	1 2	✓	✓	✓

The LLM revises **sorted** to the sorted prefix, which holds at every iteration.

Prove the final program

With good invariants, Velvet generates the **verification conditions** (VCs) and **tries the cheapest mode first**.

Insertion sort: **34 VCs**



33 closed by automation (SMT, **`grind`**)

Prove the final program

With good invariants, Velvet generates the **verification conditions** (VCs) and **tries the cheapest mode first**.

Insertion sort: **34 VCs**

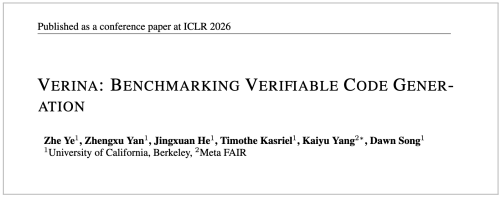


- **33** closed by automation (SMT, **`grind`**)
- **1** left → interactive proof by an LLM agent with tools to *search lemmas* and *recursively decompose* the proof

Evaluation: PBT finds bugs in benchmarks

We ran our specification checks on human-written Lean benchmarks

VERINA (ICLR 2026)



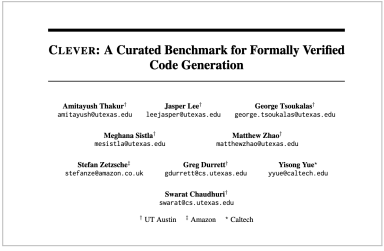
Published as a conference paper at ICLR 2026

VERINA: BENCHMARKING VERIFIABLE CODE GENERATION

Zhe Ye[1], Zhengxu Yan[1], Jingxuan He[1], Timothe Kasriel[1], Kaiyu Yang[2*], Dawn Song[1]
[1]University of California, Berkeley, [2]Meta FAIR

13 issues

CLEVER (NeurIPS 2025)



CLEVER: A Curated Benchmark for Formally Verified Code Generation

Amitayush Thakur[†] amitayush@utexas.edu
Jasper Lee[†] leejasper@utexas.edu
George Tsoukalas[†] george.tsoukalas@utexas.edu
Meghana Sistla[†] msistla@utexas.edu
Matthew Zhao[†] matthewzhao@utexas.edu
Stefan Zetzsche[‡] stefanze@amazon.co.uk
Greg Durrett[†] gdurrett@cs.utexas.edu
Yisong Yue[*] yyue@caltech.edu
Swarat Chaudhuri[†] swarat@cs.utexas.edu
[†] UT Austin [‡] Amazon [*] Caltech

18 issues

Found by PBT alone, without any LLM assistance.

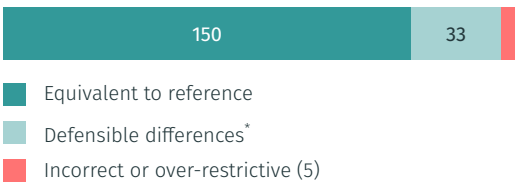
Evaluation: Specification quality

Generating specifications from English tasks

97.4%

Equivalent or defensible*

188 VERINA problems



*Defensible: justified differences due to reference errors, valid precondition variations, or ambiguous descriptions.

I left something out...

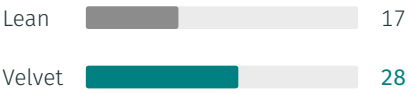
LeetProof can also generate a **functional program in plain Lean** and prove it directly.

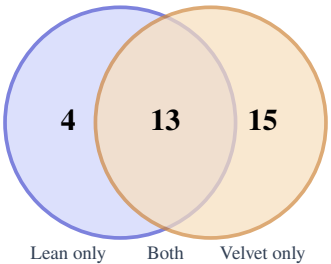
This is our **baseline**: a single mode, proving the whole program directly.

I left something out...

LeetProof can also generate a **functional program in plain Lean** and prove it directly.
This is our **baseline**: a single mode, proving the whole program directly.

Fully proven (GPT-5.2, 50 problems, $5 each)

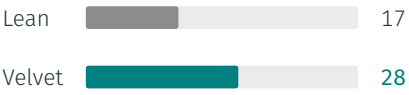




I left something out...

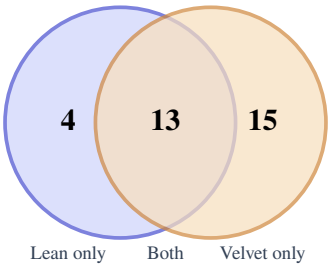
LeetProof can also generate a **functional program in plain Lean** and prove it directly.
This is our **baseline**: a single mode, proving the whole program directly.

Fully proven (GPT-5.2, 50 problems, $5 each)



Partially proven, finished by Aristotle





Takeaways

- PBT is a **cheap, effective and versatile** validation technique: it filters out wrong specifications, invariants and programs, and even caught bugs in **published benchmarks**.
- Combining testing, SMT and interactive proof (multi-modality) **certifies more programs at the same cost**.

Summary

Takeaways

- PBT is a **cheap, effective and versatile** validation technique: it filters out wrong specifications, invariants and programs, and even caught bugs in **published benchmarks**.
- Combining testing, SMT and interactive proof (multi-modality) **certifies more programs at the same cost**.

Future directions

- Certified synthesis of protocols and distributed systems.
- Going beyond functional correctness.
- Synthesizing verified low-level systems code.



Paper



Velvet

Thank you!