Certified Program Synthesis with a Multi-modal Verifier

Yueyang Feng*
National University of Singapore
Singapore, Singapore
yueyangfeng@u.nus.edu

Dipesh Kafle*
National University of Singapore
Singapore, Singapore
dipesh@u.nus.edu

Vladimir Gladshtein
National University of Singapore
Singapore, Singapore
vovaglad@u.nus.edu

Vitaly Kurin
Neapolis University Pafos
Paphos, Cyprus
v.kurin@nup.ac.cy

George Pîrlea
National University of Singapore
Singapore, Singapore
gpirlea@u.nus.edu

Qiyuan Zhao
National University of Singapore
Singapore, Singapore
zhaoqiyuan@u.nus.edu

Peter Müller
ETH Zurich
Zürich, Switzerland
peter.mueller@inf.ethz.ch

Ilya Sergey
National University of Singapore
Singapore, Singapore
ilya@nus.edu.sg

Abstract

Certified program synthesis (*a.k.a. vericoding*) is the process of automatically generating a program, its formal specification, and a machine-checkable proof of program/specification alignment from a natural-language task description. Two key challenges make vericoding difficult. First, specifications synthesised from natural language descriptions are often either too weak to be meaningful or too strong to be implementable, yet existing approaches lack systematic means to detect such defects. Second, the program verifiers used to validate results are fragmented: each tool supports a particular reasoning mode—*auto-active* (*e.g.*, Dafny, Verus) or *interactive* (*e.g.*, Rocq, Lean)—with its own trade-off between automation and expressivity. This forces each synthesis methodology to target a single paradigm, limiting the tasks it can handle.

We overcome both challenges by structuring the synthesis workflow in stages around a *multi-modal verifier*—a single tool that combines dynamic validation, automated proofs, and interactive proof scripting within one foundational framework. We realise this idea in LeetProof, a new agentic pipeline built on Velvet, a multi-modal program verifier built in the Lean theorem prover. Multi-modality enables LeetProof to validate generated specifications via randomised *property-based testing* before any code is synthesised, decompose the synthesis task into sub-problems guided by verification conditions, and delegate residual proof obligations to frontier AI provers specialised for Lean. We evaluate LeetProof on an extensive benchmark suite derived from prior work on certified synthesis. Our specification validation uncovers defects in existing reference benchmarks, and LeetProof's staged pipeline achieves a significantly higher rate of fully certified solutions than a single-mode baseline at the same fixed budget—consistently across two different frontier LLM backends.

*Joint first authors.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837559

CCS Concepts

• **Software and its engineering** → **Formal software verification**; *Automatic programming*.

Keywords

program synthesis, formal verification, property-based testing, large language models, Lean, Velvet, multi-modal verification

ACM Reference Format:
Yueyang Feng, Dipesh Kafle, Vladimir Gladshtein, Vitaly Kurin, George Pîrlea, Qiyuan Zhao, Peter Müller, and Ilya Sergey. 2026. Certified Program Synthesis with a Multi-modal Verifier. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837559

1 Introduction

Certified program synthesis, or *vericoding* [8], is the task of producing, from a description in a natural language, a program together with a formal specification and a machine-checkable proof that the program meets it. This is inherently difficult because it weaves together two problems that are hard in isolation—(a) translating an informal intent into a precise formal specification and (b) proving that a synthesised implementation satisfies it—and each feeds back into the other: a faulty specification dooms even a correct program, while an inadequate proof strategy leaves a correct specification unverified. Recent advances in LLM-based code generation have made vericoding feasible by enabling models to produce formal specifications and proof scripts [9, 25, 41, 58, 65], while *program verifiers* serve as trustworthy oracles that check the results.

The landscape of modern program verifiers broadly splits into two families. *Auto-active* verifiers, such as Dafny [32], Viper [45], Verus [28], and F* [9], ask the programmer to annotate code with pre/postconditions, assertions, and loop invariants, which an SMT solver [6, 13] then checks automatically. *Interactive* provers, such as Rocq [55] and Lean [14] instead require the user to construct proofs, step by step, using *proof scripts*, offering full expressivity at the cost of greatly increased manual effort.

A growing body of work has tackled vericoding for each individual verifier and paradigm. On the auto-active side, LLM-based approaches have been developed for Dafny [4, 5, 35, 41, 43, 58],

Verus [2, 10, 65], and F* [9]. On the interactive side, analogous efforts target Lean [57] and Rocq [25, 36, 61]. Dedicated benchmarks have accompanied each line of work: DafnyBench [35] for Dafny, and VERINA [67], CLEVER [60], and VeriBench [40] for Lean.

This fragmentation of the verifier landscape, and of the vericoding efforts built atop it, is not merely an inconvenience. Because each approach is built around the idioms of a single verifier, the resulting workflows, prompting strategies, and feedback loops are deeply entangled with tool-specific details, making it difficult to distil reusable principles that transfer across different verifiers or LLM backends. Moreover, *validating* formal program specifications in most of these pipelines is typically limited to human inspection [8, 37, 41, 67], with no systematic way to detect whether a generated specification is too weak (admitting incorrect implementations) or too strong (ruling out every correct one).

The specification-quality problem is not hypothetical. During this work, we discovered that roughly 10% of the reference specifications in two state-of-the-art benchmarks, VERINA [67] and CLEVER [60], are defective—they either under-constrain or incorrectly express the intended properties of the expected output (more on that in Sec. 5). This reinforces the need for automated specification quality checks as a component of any vericoding pipeline.

In this work, we propose to address both the tool-fragmentation and the specification-quality problems by designing the vericoding workflow around a *multi-modal verifier*—a single tool that supports dynamic validation (testing), automated verification (SMT), and interactive proof scripting within one *foundational* framework—one whose reasoning principles are themselves mechanically verified, rather than trusted axioms. Traditional *single-mode* approaches commit to one paradigm: auto-active tools such as Dafny [32] offer fast SMT-backed feedback but cannot express proofs outside the solver's reach, while interactive provers such as Lean are more expressive but provide less automation. A multi-modal verifier combines both, letting the pipeline choose the most effective mode at each step: automation for routine obligations, interactive mode for harder cases, and testing as a fast oracle.

We implement this idea on top of Velvet [18], a new verifier for imperative programs embedded as a library in Lean. Velvet is built on Loom [19], a general framework for foundational multi-modal verification in Lean. It integrates SMT-based automation [42, 53] with interactive Lean proofs and supports QuickCheck-style [11, 20, 29] property-based testing (PBT) of programs, specifications, and loop invariants. Since Velvet programs are ordinary Lean programs, the entire Lean ecosystem—type checker, automation tactics, and the Mathlib library [38]—is available at every stage, and tricky verification goals can be delegated to frontier AI provers [1].

Working with a foundational yet executable language (*i.e.*, Lean) also lets us address the specification-quality problem. Lahiri [27] proposed *symbolic specification testing*: using specifications and an SMT-based verifier to prove that concrete tests do not fail, without executing code. While effective for Dafny, purely symbolic checking becomes prohibitively expensive in Lean, where SMT covers a smaller fraction of proof obligations. What turned out to work surprisingly well is *randomised specification testing*—using property-based testing [20] to validate specifications against test cases instead of formal proofs. PBT is fast, requires no proof engineering, and catches the same class of defects at a fraction of the cost. A specification that fails is rejected *before* any code synthesis, catching under-specification early and cheaply.

Combining multi-modal verification with randomised specification testing, we present LeetProof: an AI-assisted agentic pipeline for end-to-end vericoding that decomposes synthesis into independently validated *stages*—specification generation, program/invariant synthesis, and last-mile proof construction. This structure maps onto an agentic architecture [66, 68], in which independent AI systems work in tandem with deterministic symbolic tools, allowing the pipeline to be composed modularly and optimised for costs by fine-tuning its specific components. It was only by experimenting with different modes at each stage that we identified PBT as the most cost-effective approach for specification validation—illustrating the benefit of the staged, multi-modal design.

Contributions. This work makes the following contributions:

- LeetProof: the first agentic pipeline for end-to-end vericoding built around a foundational multi-modal verifier, combining testing, automated SMT-based proofs, and AI-assisted interactive proof scripting in a unified, staged synthesis pipeline (Sec. 3).
- A new *testing infrastructure* for Lean-based vericoding (Sec. 4): type class-based output mutation for specification completeness checking, bounded enumeration for existential quantifiers in verification conditions, and a meta-programmed harness for randomised testing of synthesised programs and invariants.
- An evaluation of LeetProof *specification inference* (Sec. 5) demonstrating that 97.4% of our generated specifications are logically equivalent or represent justified divergence points from the VERINA benchmark [67], while randomised specification testing uncovers defects in ∼10% of two published benchmark suites: VERINA and CLEVER [60].
- A new benchmark of 50 imperative-style LeetCode problems with complexity annotations, and an evaluation of LeetProof *synthesis pipeline* (Sec. 6) showing that: (a) the multi-modal LeetProof pipeline produces significantly more fully certified solutions than a single-mode Lean baseline at the same fixed budget (Sec. 6.2); (b) all partially verified Velvet solutions are dischargeable with additional interactive effort, confirming the correctness of the synthesised artefacts (Sec. 6.3); and (c) these gains are consistent across different frontier LLM backends (Sec. 6.4).

2 Background

We start by briefly introducing the two systems that LeetProof builds on: the Lean theorem prover and the Velvet verifier.

2.1 Lean

Lean [12] is an open-source theorem prover and dependently typed programming language. Its expressive type system allows users to state and prove theorems (including statements about pure *functional* programs) interactively using *proof scripts*. Lean's mathematical library Mathlib [38] contains over 210,000 formalised theorems, making it one of the most extensive such libraries in any proof assistant. This rich ecosystem has made Lean the platform of choice for major AI-assisted mathematical reasoning efforts, including AlphaProof [23], which achieved silver-medal performance at the 2024 International Mathematical Olympiad (IMO), and Aristotle [1], which reached gold-medal level at the 2025 IMO.

```
1  def countDivisors (n: ℕ) : ℕ :=
2    ((List.range (n + 1)).filter
3      (fun d => d > 0 ∧ n % d = 0)).length
4
5  def isPrime (n: ℕ) : Prop := n > 1 ∧ countDivisors n = 2
6
7  method IsNonPrime (n: ℕ) return (result: Bool)
8    ensures result = true ↔ ¬isPrime n
9    do
10     if n ≤ 1 then return true
11     let mut i : ℕ := 2
12     let mut ret : Bool := false
13     while i * i ≤ n
14     invariant ¬ret ↔ ∀d, 2 ≤ d ∧ d < i → n % d ≠ 0
15     invariant i ≥ 2 ∧ (i - 1) * (i - 1) ≤ n
16     do
17       if n % i = 0 then ret := true
18       i := i + 1
19     return ret
```

Fig. 1: A Velvet method for checking non-primality.

Beyond mathematics, Lean also serves as a *meta-verifier*: its hygienic macro system and metaprogramming facilities [49] allow users to embed domain-specific reasoning frameworks as libraries.

2.2 Velvet

Velvet [18] is a Hoare-style [22] program verifier for *imperative* programs, embedded as a library in Lean via Loom framework [19]. Programs in Velvet are annotated with pre/postconditions and loop invariants, and Loom generates verification conditions (VCs) whose validity implies program correctness. Because Loom is formalised in Lean, this implication is a machine-checked theorem—making Velvet a *foundational* verifier that need not be trusted.

Fig. 1 shows a complete Velvet example: a method that decides whether a natural number is *not* prime. The specification relies on two auxiliary Lean definitions, `countDivisors` and `isPrime` (lines 1–5). These are ordinary Lean functions that use list filtering and quantification; they are executable but would be inefficient to run on large inputs. This is a deliberate design choice: specifications in Velvet are Lean propositions and *need not* be executable.

The imperative method `IsNonPrime` (lines 7–19) comes with the postcondition (`ensures` clause at line 8), which states that its Boolean result corresponds exactly to `¬isPrime n`. The loop (lines 13–18) carries two invariants: `ret` is false iff no divisor of n has been found in $[2, i)$, and $i \geq 2$ with $(i-1)^2 \leq n$. Velvet programs are ordinary Lean programs (with monadically encoded effects [59]): one can execute, *e.g.*, `#eval (IsNonPrime 42).run` to test the method.

The Velvet command `prove_correct` triggers VC generation. For this example, it produces 15 verification conditions—plain Lean theorems that together imply correctness of `IsNonPrime` *w.r.t.* the ascribed specification. Of these, 14 are discharged fully automatically with the help of SMT-based automation (via lean-auto [53]) and Lean tactics such as `grind` [30] and `aesop` [34]. The single remaining VC requires an *interactive* proof: it is the number-theoretic fact stating that a number is prime if and only if it has no divisors between 2 and its integer square root. This proof can be written in Lean or delegated to an AI prover such as Aristotle.

3 A Tour of LeetProof

LeetProof takes a programming task in natural language and produces a verified Velvet program with a machine-checkable correctness proof in Lean. The task is decomposed into three stages: specification (Sec. 3.1), program synthesis (Sec. 3.2), and proof (Sec. 3.3); each producing a validated intermediate artefact. Fig. 2 illustrates the pipeline on LeetCode problem 1752:[1] the natural language statement (Fig. 2a) becomes a formal specification (Fig. 2b), then a Velvet implementation (Fig. 2c) with a correctness proof (Fig. 2d).

3.1 Specification Synthesis

The first stage translates a natural-language problem description into a formal Lean specification (Fig. 2b). The LLM also generates several concrete test cases alongside the specification; these serve both as validation inputs and as human-readable documentation for the formal definitions. Since every subsequent stage—program synthesis, invariant inference, and correctness proof—depends on this specification, ensuring its quality is crucial. An LLM-generated specification can go wrong in several ways: it may fail to type-check, admit trivially correct implementations by being too weak, impose constraints that no implementation can satisfy by being too strong, or simply misinterpret the problem. Our pipeline addresses these risks in steps, as shown in the top row of Fig. 3.

The process begins with an LLM proposing a draft specification together with ∼10 test cases (some derived from the problem statement, some are synthesised). The draft must type-check in Lean; an LLM judge then reviews it for common issues. If either check fails, the LLM revises the draft. To gain further confidence, we apply our new take on *randomised specification testing* using Lean's Plausible PBT library [29]. For each test case, LeetProof programmatically generates three checks: (a) the test input satisfies the `precondition`, filtering irrelevant inputs; (b) the intended input/output pair satisfies the `postcondition`, catching specs that reject correct answers; and (c) no alternative output satisfies the `postcondition` for the same input, ruling out under-specification [4, 27].

Fig. 4 illustrates this on our running example. The specification defines a `precondition` (①) requiring the array size to exceed 1, and a `postcondition` (②) stating that `result = true` *implies* the array is sorted-and-rotated. A test case (③) is generated alongside. Each PBT check is a Lean definition typed as `Prop`—syntactically identical to a theorem. Normally this requires a deductive proof; however, the tactic `plausible'` (a non-failing variant of Lean's Plausible tactic [29]) instead searches for counterexamples by generating random inputs. If a counterexample is found, the check fails and the spec or test case is flagged. If none is found, `plausible'` silently admits the goal (*i.e.*, inserts a `sorry`), letting the pipeline proceed. This is safe in terms of verification because these checks are validation guards, not part of the final correctness certificate.

In this example, checks ④ and ⑤ both pass: the input `#[1, 2, 3]` satisfies the precondition (size $3 > 1$), and the expected output `true` satisfies the postcondition. However, the uniqueness check ⑥ fails: PBT finds `result = false` as a counterexample. Because the postcondition uses an implication (→ rather than ↔), setting `result` to `false` makes it *vacuously true*—so the specification admits a

[1] https://leetcode.com/problems/check-if-array-is-sorted-and-rotated/

> Given an array `nums`, return `true` if the array was originally sorted in non-decreasing order, then rotated some number of positions (including zero). Otherwise, return `false`. There may be duplicates in the original array. An array `A` rotated by `x` positions results in an array `B` of the same length such that `B[i] == A[(i+x) % A.length]` for every valid index `i`.

(a) LeetCode problem 1752 statement in plain English

```
-- A "drop" is a strict decrease from an element to
-- its cyclic successor.
def isDrop (nums : Array Int) (i : Nat) : Prop :=
  nums.size > 0 ∧ i < nums.size ∧
  nums[(i + 1) % nums.size]! < nums[i]!

-- A sorted-and-rotated array has at most one cyclic drop.
def rotSortedProp (nums : Array Int) : Prop :=
  nums.size ≤ 1 ∨
  (∀ i j : Nat, isDrop nums i → isDrop nums j → i = j)

-- No extra assumptions are needed for this problem.
def precondition (nums : Array Int) : Prop := True

-- The returned boolean should exactly decide "rotSortedProp"
def postcondition (nums : Array Int) (result : Bool) :=
  result = true ↔ rotSortedProp nums
```

(b) Lean specification: auxiliary functions and pre/postconditions

```
-- One of the residual goals to prove interactively
theorem goal_2
  (nums : Array Int)
  (i_2 : Nat)
  (invariant_inv_bounds : i_2 ≤ nums.size)
  -- other invariants are omitted for brevity
  (if_pos : {k ∈ Finset.range i_2 | nums[(k + 1) % nums.size]!
      < nums[k]!}.card ≤ 1)
  (done_1 : nums.size ≤ i_2)
  : postcondition nums true := by sorry

prove_correct CheckSortedAndRotated by
  loom_solve <;> ...
  exact (goal_2 ...)
```

(d) Proof script and an example extracted residual goal

```
method CheckSortedAndRotated (nums : Array Int)
return (result : Bool)
require precondition nums
ensures postcondition nums result
do
  let n := nums.size
  if n ≤ 1 then return true
  else
    let mut drops : Nat := 0
    let mut i : Nat := 0
    while i < n
      -- The loop index stays within bounds.  ①
      invariant "inv_bounds" (i ≤ n)
      -- The array size used in the loop is fixed.
      invariant "inv_n_def" (n = nums.size)
      -- Modulo indexing is safe because n > 0.
      invariant "inv_n_pos" (n > 0)
      -- drops counts the cyclic decreases
      invariant "inv_drops_count"
        (drops = (Finset.filter
          (fun k : Nat => nums[(k + 1) % n]! < nums[k]!)
          (Finset.range i)).card)
      decreasing n - i
    do
      let a := nums[i]!
      let b := nums[(i + 1) % n]!
      if b < a then drops := drops + 1
      i := i + 1
    return (drops ≤ 1)

#guard (CheckSortedAndRotated #[4, 1, 2, 3]).extract  ②
velvet_plausible_test CheckSortedAndRotated  ③
```

(c) Executable Velvet implementation and tests

Fig. 2: LeetProof pipeline: from the task description in a natural language to formally specified and verified Velvet code.

degenerate implementation that always returns `false`. This is exactly the class of under-specification that randomised testing is designed to catch. Beyond filtering, the generated tests serve as human-readable documentation for the formal specification: a user can inspect the accepted concrete input/output pairs to understand what the spec means without reading the formal Lean definitions, closing an important gap between the informal task description and the rigorous specification used in subsequent stages. An alternative is *symbolic specification testing* [27], which *proves* that each test passes under the specification via an SMT-based verifier. We experimented with this and found it ineffective in Lean: even simple proof obligations were expensive to construct, making PBT the clear winner in cost. Once a specification and its tests pass all checks, they serve as the ground truth for the subsequent stages.

3.2 Program and Invariant Synthesis

Velvet is an *intrinsic* program verifier: correctness proofs rely on program-level annotations—most importantly, *loop invariants*—that drive the verification condition (VC) generation covered in Sec. 3.3. Consequently, this stage couples two sub-tasks: synthesising the program code and inferring its loop invariants.

Code generation. Given a validated specification from Stage 1, the LLM generates a candidate Velvet program (middle row of Fig. 3). For our running example, the result is shown in Fig. 2c. Because Velvet programs are ordinary Lean programs, they can be tested immediately: ② shows a concrete test case assertion (via Lean's `#guard` command), and ③ invokes PBT via a helper macro. PBT checks the program against both the specification and concrete test cases by running it on random inputs satisfying the precondition and verifying the postcondition holds. Any failure is fed back to the LLM for revision. An LLM judge additionally reviews

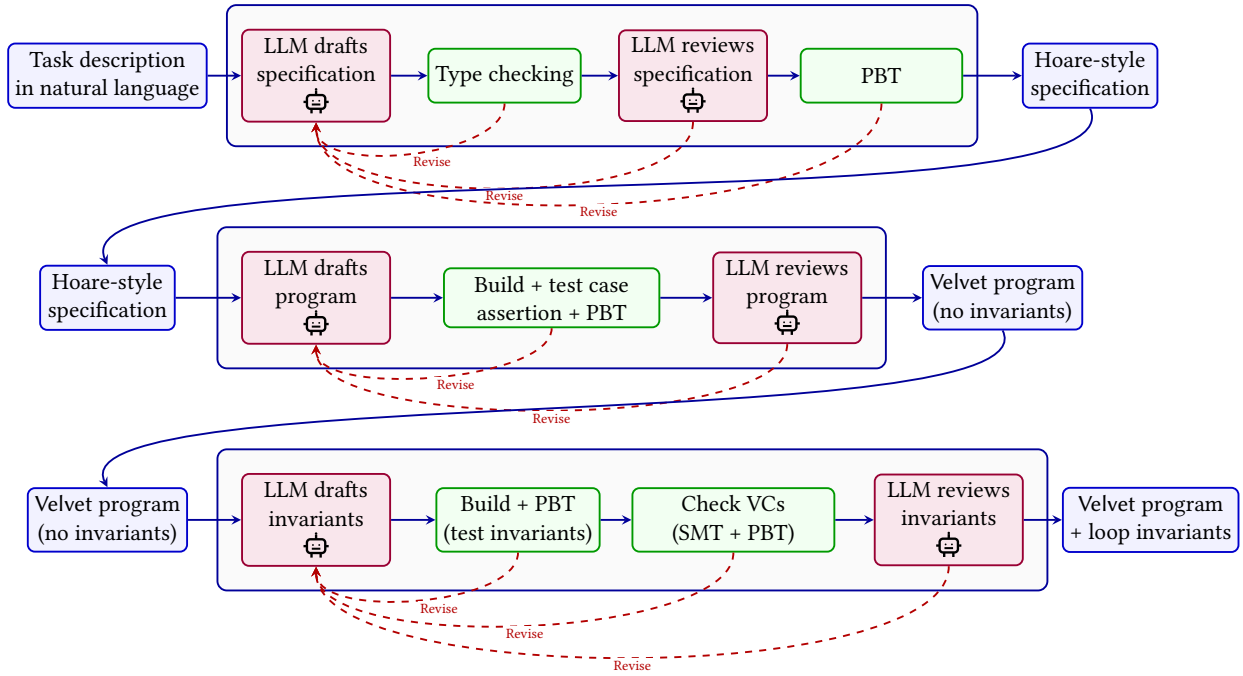

Fig. 3: Partial LeetProof pipeline: specification generation (top), program synthesis (middle), and invariant inference (bottom). Solid blue arrows show artefacts flowing between stages; dashed red arrows show revision loops within each stage.

```
def precondition (nums : Array Int) : Prop :=                    ①
  nums.size > 1 -- not the final one, too restrictive
                                                                 ②
def postcondition (nums : Array Int) (result : Bool) :=
  result = true → rotSortedProp nums
                                                                 ③
def test1_nums : Array Int := #[1, 2, 3]
def test1_Expected : Bool := true

def precondition_test1 : precondition test1_nums :=              ④
  by simp; plausible'

def postcondition_test1 :                                        ⑤
  postcondition test1_nums test1_Expected := by
  simp; plausible'

-- counterexample to uniqueness: result = false                  ⑥
def uniqueness_test1 (result : Bool) :
   result ≠ test1_Expected →
  ¬ postcondition test1_nums result := by
  simp ; plausible'
```

Fig. 4: PBT checks for generated specifications and test cases.

the output for common issues (*e.g.*, producing pure functional Lean code rather than imperative Velvet code, which would bypass invariant inference and undermine the multi-modal design).

Invariant inference. Loop invariant inference—finding inductive properties that hold at every iteration and are strong enough to imply the postcondition—is a notoriously difficult problem, closely intertwined with the specification inference and program correctness proof [4, 17, 43]. In auto-active verifiers like Dafny, candidate

```
invariant "inv_drops_count"
    (drops = < (Finset.filter
       (fun k : Nat => nums[(k + 1) % n]! < nums[k]!)
         (Finset.range i)).card)
```

Fig. 5: Incorrect invariant caught by PBT.

invariants can be checked cheaply via SMT. In Lean, SMT covers a smaller fraction of obligations (though it remains effective for linear arithmetic), so we again exploit multi-modality.

The invariant inference loop (bottom row of Fig. 3) proceeds as follows. The LLM proposes candidate invariants (① in Fig. 2c). LeetProof then triggers VC generation via Loom and attempts to discharge the resulting VCs with automated tactics. For VCs that remain, PBT searches for counterexamples. If a counterexample is found—or if the LLM, inspecting the remaining VCs, judges one to be unprovable given the current invariants—the feedback is propagated back and the LLM revises its invariants.

Fig. 5 illustrates this: replacing `=` with `<` in the `inv_drops_count` invariant causes PBT to immediately produce a counterexample:

```
[velvet_plausible_test] FAIL: invariant "inv_drops_count"
 ↪  doesn't hold
nums = #[-18, 10, -13, 11, 8, 17, 13, -19, 15, -1, -27, -25]
```

The buggy invariant claims `drops` is *strictly less* than the number of cyclic decreases seen so far. For this input, the two quantities are equal at some iteration, violating the strict inequality.

The `decreasing` clause (line 23 of Fig. 2c) specifies a *termination measure* for the loop, which is also inferred by LLM at this stage and formally verified in the subsequent proof synthesis stage. This clause is optional: Velvet supports both *partial correctness* (the

postcondition holds *if* the program terminates) and *total correctness* (the program terminates and satisfies its postcondition) [19]. The choice can be configured per task in the vericoding pipeline.

Guarantees at the end of this stage. PBT can refute incorrect invariants but cannot prove that the surviving ones are inductive or sufficiently strong. The goal of this stage is therefore *high-confidence* invariants: candidates that pass all automated checks (PBT, LLM review) without yet having been formally proved correct. This leaves room for incompleteness: an invariant may turn out to be too weak during the proof stage. In practice, the combination of PBT filtering and LLM-based VC assessment produces invariants that rarely need revision (*cf.* Sec. 6). A program that passes all checks is forwarded to the next stage for the final formal proof.

3.3 Proof Synthesis and Residual Obligations

Once a Velvet program with high-confidence invariants is ready, the `prove_correct` command triggers VC generation via Loom. Each VC is an ordinary Lean theorem, which means we can dispatch it using *any* method available in the Lean ecosystem—a key advantage of working inside a foundational proof assistant.

LeetProof first attempts to close every VC automatically using a combination of SMT-based tactics [42, 53] and built-in Lean tactics such as `grind` [30] and `aesop` [34]. For our running example, this step discharges 14 out of 18 VCs, leaving 4 *residual obligations*.

A residual obligation is a VC that automation cannot close. In the listing, these appear as Lean theorems with `sorry` placeholders (Fig. 2d), which are then plugged into the `prove_correct` block. The goal `goal_2` in Fig. 2d is one such obligation: it requires showing that the loop's postcondition follows from the invariants at termination—a mathematical fact that SMT cannot discharge.

To close residual obligations, LeetProof employs an LLM-based agent inspired by the decomposition technique of Hilbert [63]. The prover agent has access to Mathlib [38] search, can register auxiliary lemmas, and can decompose a goal into smaller sub-goals that it attempts recursively. For particularly hard obligations, LeetProof can delegate to specialised AI provers such as Aristotle [1]. The agent operates autonomously within a fixed token budget.

In principle, the PBT-based checks from earlier stages (specification validation, invariant inference) could be all replaced by a loop over AI-assisted interactive proof attempts. We experimented with this alternative and found it impractical: even for simple tasks, the token and time costs of attempting formal proofs vastly exceeded those of PBT, which provides the same filtering effect at a fraction of the cost. This confirms our design choice of reserving interactive proving for the final stage, where it is strictly necessary.

4 LeetProof Testing Infrastructure

This section details the PBT machinery that underpins the pipeline stages described in Sec. 3: generating specification tests, handling existential quantifiers in statements, testing programs and invariants, and the role of Lean meta-programming. All PBT checks in LeetProof are built on Plausible [29], a property-based testing framework for Lean 4 that integrates with the tactic system. Given a theorem statement, Plausible generates random inputs and attempts to refute the theorem by finding counterexamples.

4.1 Handling Existential Quantification

PBT fundamentally struggles with existential quantification. Universally quantified properties can be tested by *sampling* inputs and checking the predicate, but negating an existential postcondition requires showing that *no* witness exists—which in general entails reasoning over the entire domain. This pattern is pervasive in verification conditions for array programs. For example, to test

$$(\forall i,\ i < \texttt{arr.size} \to \texttt{arr}[i] \geq 0) \to \texttt{sum}(\texttt{arr}) \geq 0,$$

one must handle the logically equivalent form

$$(\exists i,\ i < \texttt{arr.size} \wedge \texttt{arr}[i] < 0) \vee \texttt{sum}(\texttt{arr}) \geq 0.$$

(The equivalence follows from rewriting the implication $A \to B$ as $\neg A \vee B$ and pushing the negation through the universal quantifier.) Plausible cannot test the existential disjunct directly, since refuting it would require enumerating the entire domain of i.

We exploit a structural property of the verification conditions that arise in practice: existential variables typically admit finite bounds inferable from the theorem structure. In the example above, the constraint $i < \texttt{arr.size}$ bounds the existential variable i, so the disjunct can be checked by enumerating $i \in [0, \texttt{arr.size})$. We extend Plausible with a lightweight heuristic that extracts subexpressions appearing in inequality constraints as candidate bounds for existential variables and uses Lean tactics (*e.g.*, `grind`, `omega`) to discharge the resulting bound obligations—*i.e.*, to prove that the candidate expression is indeed an upper bound, which is necessary for the enumeration to be sound. When bounds can be established, existential quantification reduces to bounded enumeration during testing. While this does not solve the challenge of testing arbitrary existential quantifiers, it covers the common patterns that arise in program verification and is effective in practice.

4.2 Testing Specifications

Sec. 3.1 described the three PBT checks—pre/postcondition soundness, and output uniqueness—that LeetProof generates for each test case. Here we formalise the underlying definitions. Let I denote a set of inputs, let $\psi(i)$ be the precondition, and let $\varphi(i, o)$ be the postcondition. For an input $i \in I$, let $\widehat{O}(i)$ denote the set of intended outputs. A postcondition $\varphi(i, o)$ is *precise* on I if it satisfies:

(1) *Soundness*: all intended outputs are accepted, *i.e.*, $\forall i \in I,\ \forall o \in \widehat{O}(i),\ \varphi(i, o)$. A violation means the spec is too strong.
(2) *Completeness*: all unintended outputs are rejected, *i.e.*, $\forall i \in I,\ \forall o \notin \widehat{O}(i),\ \neg\varphi(i, o)$. A violation means the spec is too weak.

During spec generation, the LLM produces concrete test cases alongside the spec. Each test case consists of an input i together with a representative intended output $\hat{o} \in \widehat{O}(i)$. To validate soundness, for each test case $(i, \hat{o})$, Plausible checks (a) the input satisfies the precondition and (b) the input/output pair satisfies the postcondition. Failure in either check indicates a flaw in either the generated specification or the tests; the failing check and counterexample are fed back to the LLM, which revises the spec (*cf.* Sec. 3.1).

Testing completeness is more challenging, because it requires checking that *no* unintended output satisfies the postcondition—a universal statement over an unbounded output domain. However, for many algorithmic problems (*e.g.*, typical LeetCode tasks),

the intended output is deterministic: $\widehat{O}(i) = \{\hat{o}\}$. Under this assumption, completeness simplifies to verifying the *uniqueness* of the intended output, yielding the testable form $\forall o \neq \hat{o}, \neg\varphi(i, o)$. Checking completeness thus becomes a search for a spurious output $o \neq \hat{o}$ that erroneously satisfies φ. In practice, spurious outputs often share structure with the intended one. We therefore first sample candidate outputs randomly, then apply small mutations to $\hat{o}$, checking via Plausible whether any candidate inadvertently satisfies φ. The mutations are type-directed: Booleans are flipped; numeric types (`Nat`, `Int`, `Char`) are perturbed by a small additive delta (with `Int` additionally supporting negation); pairs $\alpha \times \beta$ have one component mutated (or, when $\alpha = \beta$, swapped); collections (`Array` α, `List` α, `String`) undergo element-level mutation, random deletion, or random insertion. These operators target common classes of specification errors: small numeric perturbations expose boundary and off-by-one errors, swaps expose unintended order sensitivity, and insertion or deletion exposes incorrect length or cardinality constraints.

All mutations are implemented via Lean *type classes*—a mechanism for expressing constrained polymorphism, similar to Haskell's type classes or Rust's traits [31]. A type class `Mutatable` α declares a single operation `mutate` : $\alpha \to$ `Gen` α, where `Gen` is Plausible's randomised generation monad. The operation takes a value and returns a "nearby" variant, enabling corpus-guided fuzzing on top of Plausible. Concrete instances provide the implementation for each base type listed above. Composite types such as multi-dimensional arrays are supported automatically through recursive instance resolution: the instance for `Array` α delegates element-level mutations to the `Mutatable` α instance, which in turn may recurse further.

4.3 Testing Programs and Invariants

We adapt PBT to test both synthesised programs and their loop invariants, as outlined in Sec. 3.2, by transforming a Velvet method into a testing harness that interleaves execution with checks done at runtime. Fig. 6 illustrates the result for our running example.

The harness has three noteworthy components (highlighted in Fig. 6). First, the input is sampled randomly subject to the precondition (line 3): Plausible generates candidate arrays and retains only those satisfying `precondition`. Second, each loop invariant is checked both at loop entry (lines 9–11) and after every iteration (lines 17–19). A failure pinpoints the exact invariant and the concrete input that violates it, giving the LLM targeted feedback for revision. Third, the postcondition is checked on the final result (lines 21–22), catching implementations that compute an incorrect answer despite maintaining all invariants.

4.4 Using Lean Meta-Programming

The testing harness of Fig. 6 must inspect a method's structure to instrument each invariant, so it cannot be an ordinary function. We implement it via Lean's meta-programming facilities [49], which manipulate program components during compilation. Velvet is shallowly embedded in Lean: a method is a monadic computation in `VelvetM` [19], modelling mutable state and imperative control flow. We enrich it with two monad transformers [33] for failure reporting and randomised input generation:

```
1  method CheckSortedAndRotatedTesting
2  do
3    nums ← sample(Array Int) satisfying precondition
4    let n := nums.size
5    if n ≤ 1 then return true
6    else
7      let mut drops : Nat := 0
8      let mut i : Nat := 0
9      if not (plausible_test "inv_bounds") then
10       throwError "inv_bounds failed at entry"
11     -- (other invariants checked similarly)
12     while i < n do
13       let a := nums[i]!
14       let b := nums[(i + 1) % n]!
15       if b < a then drops := drops + 1
16       i := i + 1
17       if not (plausible_test "inv_bounds") then
18         throwError "inv_bounds not preserved"
19       -- other invariants are checked similarly
20     let result : Bool := drops ≤ 1
21     if not (plausible_test "post") then
22       throwError "postcondition failed"
```

Fig. 6: A testing procedure for `CheckSortedAndRotated`

```
def VelvetTestingM := ExceptT String (StateT StdGen VelvetM)
```

Here, `ExceptT String` adds the ability to abort with an error message (used when a check fails), and `StateT StdGen` threads a pseudo-random generator through the computation (used by Plausible for sampling). During *elaboration*—the phase in which Lean resolves implicit arguments, synthesises type class instances, and type-checks terms—our metaprogramming code embeds every `VelvetM` operation into `VelvetTestingM` (via the standard monad transformer `lift`) and replaces each invariant annotation with an inlined Plausible check. The result is exposed as the `velvet_plausible_test` command (*cf.* Fig. 2c), which runs property-based tests on Velvet methods and reports any counterexamples.

5 Specification Inference Evaluation

Specification quality is the bedrock of the vericoding pipeline: a flawed spec renders every subsequent stage vacuous. To the best of our knowledge, there is no established method to rigorously prove that a formal specification perfectly captures the subjective intent of a natural language description. Therefore, we instead evaluate the accuracy of LLM-generated specifications compared to human-written references and whether PBT can serve as a quality oracle, which is *a priori* not obvious, since specs are logical formulas whose counterexamples may be hard to find by random sampling. We test against the VERINA benchmark [67]; the results are surprising—both in accuracy and in what PBT reveals about existing benchmarks.

5.1 Assessing Specification Accuracy

> **RQ1**: How accurate are the specifications generated by LeetProof compared to human-written reference specifications?

To answer this question, we need a benchmark with ground-truth specifications against which we can compare. We choose

Tab. 1: Specification issues discovered in published benchmark suites for vericoding in Lean.

Benchmark	Issue type	#	Instances
VERINA [67]	Underspecified postcondition	12	adv.8, adv.47, adv.71, bas.22, bas.34, bas.37, bas.46, bas.77, bas.84, bas.95, bas.97, bas.103
	Incorrect postcondition	4	adv.10, adv.12, adv.13, bas.79
CLEVER [60]	Underspecified postcondition	16	P7, P24, P27, P29, P30, P44, P49, P68, P69, P79, P84, P88, P89, P96, P156, P161
	Implementation issue	1	P9
	Possible incorrect specification	1	P94

VERINA, the largest available Lean-based vericoding benchmark to date [67], which pairs 189 natural-language problem descriptions with manually curated formal specs. We excluded one problem (`basic_104`) from our experiment because its reference specification relies on a custom data structure that prevents automated comparison. Data leakage does not readily explain our results: although VERINA is public, a model merely recalling its reference specifications would also reproduce their defects, whereas our generator diverges from and corrects them (*cf.* Sec. 5.2).

For each of the remaining 188 problems, we run our specification generator (Sec. 3.1) to produce a candidate $(pre_{\text{gen}}, post_{\text{gen}})$ and compare it against the reference $(pre_{\text{ref}}, post_{\text{ref}})$. We assess semantic accuracy via formal equivalence checking in Lean: (i) the preconditions are equivalent, $pre_{\text{gen}} \leftrightarrow pre_{\text{ref}}$, and (ii) under the precondition the postconditions agree, $pre_{\text{gen}} \rightarrow (post_{\text{gen}} \leftrightarrow post_{\text{ref}})$. We discharge these obligations using Aristotle AI prover [1], which was freely available at the time of submission, though high demand for the system led to long waiting times. We rely on an AI prover rather than SMT alone here because Lean's SMT bridge (lean-auto [53]) cannot translate every Lean construct, and these equivalence checks frequently involve functions that resist such translation. When equivalence cannot be established, the authors manually inspect the specifications to categorise the discrepancy. The 188 problems break down as follows:

(1) *Equivalent (150):* generated specifications are logically equivalent to the reference ones.
(2) *Reference issues (16):* inconsistencies between the benchmark's reference specs and the intended semantics (*cf.* Sec. 5.2).
(3) *Precondition variations (14):* preconditions differ slightly (*e.g.*, admitting broader input sets) but remain valid semantic extensions. These do not affect correctness of synthesised programs.
(4) *Ambiguous language (3):* vague descriptions admit multiple interpretations, leading to divergent but defensible specs.
(5) *Over-restrictive postconditions (2):* our postconditions impose unstated constraints to ensure determinism (*e.g.*, a tie-breaking rule not mandated by the problem). While stricter than necessary, these specs are still sound and lead to correct, if slightly constrained, implementations.
(6) *Incorrect specifications (3):* the specifications generated by LeetProof are semantically inconsistent with the problem: incorrect preconditions (2) and an incorrect postcondition (1).

Categories (1)-(4) are logical equivalents or defensible specifications (183/188, 97.4%). The remaining 5/188 instances (2.6%) consist of genuine generation errors (category 6) and over-restrictive cases that stem from a preference for deterministic specifications (category 5). Notably, when semantic disparities occur, they are more frequently attributable to inconsistencies in the human-written benchmark itself (16/188, or 8.5%, category (2), *cf.* Sec. 5.2) than to limitations in our generation process (2.6%). The PBT techniques from Sec. 4.2 played a critical role: PBT caught errors in two specifications that had already passed the LLM judge, reducing incorrect specifications from 7 (3.7%) to 5 (2.6%). It also uncovered one case where the generated test cases did not satisfy the stated preconditions despite passing the judge.

The three incorrect specs (category 6) are all *over-constraining* output in some edge cases: a spurious frequency bound, and an input bound mistakenly imposed on the output. Because the generated test cases happen to satisfy these constraints, PBT cannot detect them—a fundamental limitation of testing-based validation.

Answer to RQ1. Specifications generated by LeetProof are logically equivalent to or represent justified divergence points from human-written baselines in 97.4% of cases. Furthermore PBT filtering caught 2 of the 7 errors that pass the LLM judge.

5.2 Specification Defects in Benchmarks

Beyond evaluating our own specifications, we applied specification-level PBT to the reference specs of both VERINA and CLEVER [60]. For each benchmark problem, we run the three PBT checks of Sec. 4.2—precondition soundness, postcondition soundness, and output uniqueness—using the test cases provided by the benchmarks.

Without any LLM assistance, this process uncovered 13 issues in VERINA and 18 in CLEVER within one day on a single Apple M4 MacBook Pro (14-core CPU, 24 GB RAM). Specification-level PBT is dramatically more cost-effective than LLM-based equivalence checking in Lean: the 13 VERINA issues require only 9 minutes of PBT, whereas Aristotle-based equivalence checking takes approximately 8 hours (both are run with GPT-5.2 at a USD $1 budget per problem, though most problems consume less than $0.2).

In total, we identify 16 issues in VERINA (out of 189 specifications, 8.5%) with the help of Aristotle, and 18 in CLEVER (out of 161 specifications, 11.2%) purely via PBT, summarised in Tab. 1. Although our primary focus was testing specs, sampling inputs and executing reference implementations can also surface implementation bugs (*e.g.*, problem P9 in CLEVER). We reported all findings to the benchmark authors. The VERINA authors acknowledged and addressed 15; the remaining one is under review. The CLEVER authors acknowledged the 18 issues but have not yet released fixes.

6 Certified Program Synthesis Evaluation

We evaluated our approach to program synthesis on a new benchmark of imperative-style problems (Sec. 6.1), comparing the multi-modal LeetProof pipeline against a single-mode Lean baseline

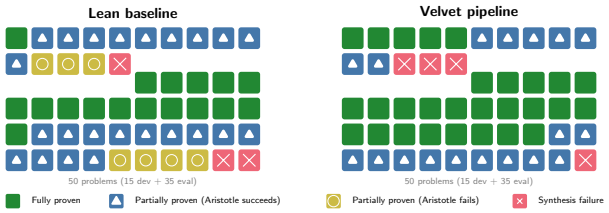

Fig. 7: Synthesis results for single-mode Lean and Velvet.

(Sec. 6.2), assessing the provability of residual obligations (Sec. 6.3), and testing robustness across different LLMs (Sec. 6.4).

6.1 Benchmark Selection

Existing Lean benchmarks (VERINA [67], CLEVER [60]) favour functional patterns (*e.g.*, recursive list traversals), omit complexity constraints, and risk training-data contamination due to long public availability. We therefore construct a benchmark of 50 LeetCode problems targeting imperative algorithms (two-pointer, sliding window, random index access). We manually selected 15 problems during development, deliberately including harder cases to stress-test individual components, which explains their lower success rate in Fig. 7; 35 more were sampled using Claude Opus 4.6 requiring: simple input types, no complex data structures, deterministic specifications, a mix of classical and recent problems, and easy-to-medium difficulty. For each problem, Claude suggested a target time complexity based on input constraints; we manually verified all annotations and applied minor simplifications where needed. Below, we report results on all 50 problems, distinguishing the development and evaluation sets where relevant (*e.g.*, Fig. 7).

6.2 Multi-Modal vs. Single-Mode Synthesis

One might expect that handing a formal specification to a frontier LLM and asking it to produce executable Lean—without staging through an imperative DSL or inferring invariants—would suffice. We call this *single-mode synthesis*: the LLM directly produces a Lean definition with its proof, verified solely by the type checker. Does LeetProof's staged pipeline offer a measurable advantage?

> **RQ2**: Does the LeetProof pipeline produce more certified solutions than a single-mode Lean baseline at the same cost?

Setup. We compare the full LeetProof pipeline (Velvet code + invariants + multi-modal proof) against a single-mode Lean baseline in which the LLM directly synthesises Lean programs verified by the same prover. Both configurations use the same LLM (GPT-5.2) and the same specifications produced in Sec. 5; the USD $5-per-problem budget covers only code synthesis and proving, excluding specification generation (which is identical for both). Fig. 7 shows the results; the development set (15 problems, top part) and evaluation set (35 problems, bottom part) are visually distinguished.

Results. For each problem we record one of three outcomes. *Fully proven:* the pipeline synthesises an implementation (with loop invariants for Velvet) and produces a complete machine-checked correctness proof. Overall, Velvet fully proves 28/50 problems, significantly outperforming single-mode Lean with 17/50. (Development: 5/15 vs. 1/15; Evaluation: 23/35 vs. 16/35.) *Partially proven:* an implementation is synthesised but the proof is incomplete—some verification conditions remain open after GPT-5.2 exhausts its budget. In Sec. 6.3 we further attempt to discharge these residual obligations with Aristotle; here we report both sub-categories together. Velvet yields 18 partially proven cases, while single-mode Lean produces 30. (Development: 7 vs. 13; Evaluation: 11 vs. 17.) *Synthesis failure:* no implementation passes the pipeline's checks. Velvet fails on 4 problems, compared to 3 for single-mode Lean. (Development: 3 vs. 1; Evaluation: 1 vs. 2.) We attribute the higher failure rate of Velvet to the stricter validation imposed by invariant checking.

Tab. 2: Overlap between Velvet and single-mode Lean.

Velvet / Lean	Fully proven	Partial / Aristotle ok	Partial / Aristotle fail	Synthesis failure	Total
Fully proven	13	13	1	1	28
Partial (Aristotle ok)	4	9	4	1	18
Partial (Aristotle fail)	0	0	0	0	0
Synthesis failure	0	1	2	1	4
Total	17	23	7	3	50

Overlap analysis. Tab. 2 breaks down the per-problem overlap. For example, the entry in the first row, second column indicates that 13 problems are fully proven by Velvet but only partially proven by Lean. While Velvet outperforms Lean overall, it does not uniformly dominate: 15 problems are fully proven only by the multi-modal pipeline, whereas 4 are fully proven only by the single-mode baseline. Because the outcomes are paired per problem, we compare fully proven versus not fully proven outcomes using an exact two-sided McNemar test. The difference is statistically significant ($p = 0.0192 < 0.05$). A representative example is LeetCode 917 ("reverse only the English letters in a character sequence"). The idiomatic functional recursive implementation in Lean handles skipping non-letter characters, whereas the imperative two-pointer implementation in Velvet requires maintaining a complex invariant, significantly complicating the proof. Depending on the problem, one style may yield a more concise implementation whose correctness is easier to establish. Since Velvet is embedded in Lean, users can freely fall back to single-mode synthesis.

Answer to RQ2. At the same USD $5 budget, the multi-modal LeetProof pipeline produces 44% more fully certified solutions on the evaluation set (23 vs. 16) and 65% more overall (28 vs. 17) compared to the single-mode Lean baseline using GPT-5.2. The advantage stems from staging: invariant inference and PBT filtering resolve issues early, leaving the prover with fewer obligations.

6.3 Last-Mile Interactive Proof Effort

The "partially proven" category in Fig. 7 contains programs whose proofs are incomplete: some verification conditions remain open after GPT-5.2 exhausts its $5 budget. This does not necessarily mean the program is wrong: it may simply need more proving effort. We therefore ask the following research question.

> **RQ3**: Are partially proven LeetProof-synthesised programs actually correct, *i.e.*, can their remaining verification conditions be discharged with additional proof effort, and does the multi-modal decomposition make them easier to close?

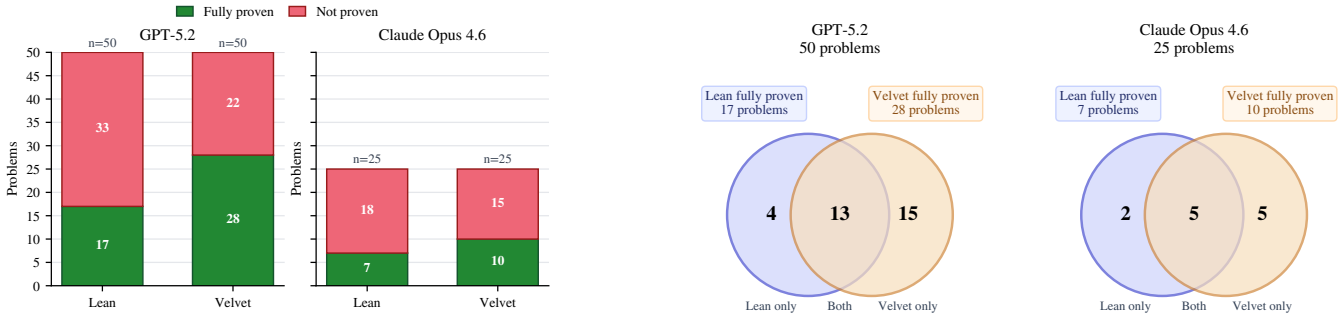

Fig. 8: LLM robustness. Left: fully proven vs. failure for single-mode Lean and Velvet. Right: overlap of fully proven problems.

To answer this, we attempted to close remaining verification conditions using Aristotle [1], a powerful AI prover. For Velvet, all 18 partially proven programs are fully discharged by Aristotle. This confirms that the LeetProof pipeline synthesises correct implementations and invariants in every case; the incomplete proofs reflect budget limitations, not errors in the synthesised artefacts.

For single-mode Lean, Aristotle successfully discharges 23 of the 30 partial solutions but fails on the remaining 7. Notably, 1 of these problems is fully solved by Velvet, and for the other six the corresponding Velvet verification conditions are fully discharged by Aristotle. This suggests that the staged pipeline generates proof obligations that are structurally easier to close.

Answer to RQ3. Every partially proven Velvet program from our benchmark suite is provably correct: the Aristotle prover successfully closes all residual obligations. The single-mode Lean baseline leaves seven programs whose VCs resist even Aristotle, whereas the equivalent Velvet VCs are discharged, confirming that multi-modal decomposition yields more tractable proof obligations.

6.4 Robustness to LLM Choice

RQ4: Are the advantages of synthesis with a multi-modal verifier over the single-mode one we discussed in RQ2 consistent across different frontier LLM backends?

We randomly sample 25 problems from our benchmark suite and replicate the experiment using Claude Opus 4.6 in place of GPT-5.2, keeping all other settings identical. Fig. 8 (left) confirms that Velvet outperforms single-mode Lean under both backends: 28/50 vs. 17/50 for GPT-5.2, and 10/25 vs. 7/25 for Opus 4.6. The lower absolute numbers for Opus 4.6 reflect its higher per-token cost, which leaves less room within the fixed $5 budget. Fig. 8 (right) shows the overlap of fully proven sets. Under GPT-5.2, 13 problems are solved by both pipelines, 4 only by single-mode Lean, and 15 only by Velvet. Under Opus 4.6, the split is 5/2/5. Changing the backend affects which individual problems are solved but does not change the overall ranking: Velvet remains the stronger pipeline.

Answer to RQ4. The results are qualitatively consistent across LLMs: the multi-modal pipeline yields more fully proven solutions than single-mode Lean under both GPT-5.2 and Claude Opus 4.6, even though the margin narrows on the Opus subset.

7 Threats to Validity

Benchmark scale and selection. Our benchmark comprises 50 algorithmic LeetCode problems (Sec. 6.1), which do not cover all software classes (*e.g.*, concurrent or I/O-heavy systems). Of these, 15 informed pipeline design, though no hyperparameters were tuned on them. To mitigate selection bias, we additionally test our specification inference against two external suites (VERINA, CLEVER) and observe consistent results across LLMs, suggesting structural rather than dataset-specific advantages.

Fixed budget and LLM evolution. The RQ2 comparison (Sec. 6.2) uses a fixed $5 budget per problem. At substantially higher budgets single-mode synthesis might close the gap; conversely, at lower budgets the gap may widen. As frontier models improve, single-mode approaches will also get stronger. However, multi-modality reduces cost by letting cheap validation modes (testing, SMT) filter candidates before expensive interactive proving is invoked—a cost-reduction principle that holds regardless of model capability.

Implementation and external dependencies. Although the staged decomposition of vericoding is a conceptual contribution not tied to any particular tool, our implementation relies on Velvet, Lean, and Aristotle. The RQ3 claim that all partial Velvet proofs are dischargeable (Sec. 6.3) depends on Aristotle's current capabilities. Porting LeetProof to another multi-modal verifier would require engineering effort, but the pipeline architecture transfers directly.

Validity of Natural Language Formalisation. Since no method to rigorously prove that a formal specification perfectly captures a natural language description exists, a small risk of semantic misalignment remains even when our outputs are equivalent to the benchmark (Sec. 5.1). While our pipeline empirically produces fewer errors than the human-written baseline, establishing a more rigorous logical connection between informal language and formal specifications remains an open challenge for future work.

8 Related Work

In this section we position our contributions with regard to the relevant prior efforts on AI-assisted vericoding, certified program synthesis, specification validation, and multi-modal verification.

AI-assisted vericoding. A growing body of work uses LLMs to generate verified code for specific verifiers: auto-active approaches target Dafny [4, 5, 35, 41, 43, 58], Verus [2, 10, 65], and F* [9], while

interactive-side efforts focus on Lean [57] and Rocq [25, 36, 61]. Several of these systems employ staged pipelines. Misu *et al.* [41] first generate a specification, then synthesise a verified Dafny method; Clover [58] generates code, docstrings, and annotations separately before cross-validating them; ATLAS [4] uses an explicit two-stage spec-then-implementation loop; and Laurel [43] decomposes invariant inference from proof search in Dafny. However, all these pipelines rely on a *single verification mode* (SMT) throughout every stage, leaving no fallback when the solver times out or cannot handle a particular obligation. LeetProof differs by building on a multi-modal verifier, matching each stage to the most cost-effective reasoning mode: PBT for specification validation, SMT + PBT for invariant inference, and interactive proofs for last-mile obligations. The reusable insight is this *mode-aware decomposition*, rather than a prompting strategy tied to one tool's annotation language.

The most closely related agentic effort is AutoRocq [62], which uses an LLM agent with an iterative feedback loop to generate tactic proofs in Rocq. However, AutoRocq addresses only the *proof generation* sub-problem: specifications and loop invariants are produced externally (*e.g.*, by Frama-C verifier [26]).

Mukherjee *et al.* [44] use an LLM in a feedback loop with VST [3] to synthesise verified C programs, but start from a formal specification rather than natural language and target a single verification mode: Rocq proofs in Separation Logic [46, 54]. By contrast, LeetProof is an end-to-end pipeline covering the full vericoding task from natural language to certified code, using multi-modal verification to match each sub-task to the most effective reasoning mode.

Non-LLM certified program synthesis. Before LLMs, several systems produced programs with machine-checkable correctness proofs using deductive synthesis or verified compilation. Fiat [15] synthesises correct-by-construction implementations of abstract data types in Rocq via tactic-driven stepwise refinement. Rupicola [51] compiles idiomatic Rocq functions to efficient imperative code, preserving correctness through the compilation pipeline. The work by Watanabe *et al.* [64] certifies the output of a separation logic-based program synthesiser [52] in a post-hoc way by translating deductive derivations into Rocq proofs. More recently, Goldstein *et al.* [21] used deductive synthesis in Lean to produce certified constrained random generators for property-based testing [11]. These approaches require the specification to be written manually in the verifier's logic; LeetProof complements them by using LLMs to bridge natural language and formal specifications.

Specification quality and validation. Ensuring generated specifications are neither too weak nor too strong is a recurring challenge. Existing approaches rely on manual tagging [41], user studies [37], LLM-as-judge with manual inspection [8], design disciplines such as non-computable specifications [60], or multi-stage evaluators mixing theorem proving with testing [67]. Notably, VERINA [67] also uses PBT (via Lean's Plausible library [29]) to check spec soundness and completeness, but as a *fallback* when theorem proving is inconclusive and only for retrospective benchmark evaluation—not as an online filter during synthesis. Moreover, VERINA compares generated specs against *ground-truth* reference specs, whereas our checks require only test cases and the generated spec itself. ATLAS [4] attempts a similar uniqueness check by generating Dafny lemmas that derive a contradiction from alternative outputs, but reports that this often exceeds SMT capabilities. Using randomised testing to validate formal specs has a long history [7, 24, 48]; however, these works target hand-written specs in stand-alone proof assistants, not LLM-generated specs inside a synthesis pipeline. Lahiri [27] proposes *symbolic specification testing*, checking LLM-generated Dafny specs against concrete input/outputs via the Dafny verifier. Our approach builds on these ideas but (1) uses PBT as the *primary* method and as a filtering stage in the synthesis pipeline (not a fallback or post-hoc evaluation), (2) replaces SMT-based checking with randomised testing, which we found more cost-effective in Lean, and (3) includes a uniqueness check that PBT handles naturally but SMT often cannot.

Multi-modal verifiers. 𝕂 [56] derives multiple reasoning modes (execution, model checking, deductive verification) from a single semantics but lacks interactive proofs. Ivy [39, 47] and Veil [50] combine SMT verification with model checking and manual proofs, but target distributed systems. Velvet [18], which LeetProof builds on, unifies SMT automation, interactive Lean proofs, and PBT—making all three modes available to an LLM-driven pipeline. To our knowledge, no prior multi-modal verifier has been used for implementing end-to-end LLM-assisted certified synthesis.

9 Conclusion

We have presented LeetProof, an agentic pipeline for certified program synthesis that leverages multi-modal verification—combining testing, SMT-based automation, and interactive proof scripting—to decompose vericoding into stages, each matched to the most effective reasoning mode. Our evaluation shows that this decomposition yields more fully certified solutions than single-mode baselines at the same cost, while randomised specification testing catches defects that existing benchmarks miss. While a sufficiently powerful AI prover could in principle subsume every stage, our results demonstrate that reserving interactive proving for the final stage—and delegating earlier filtering to cheaper modes such as PBT and SMT—reduces costs by orders of magnitude without sacrificing correctness. Multi-modality is not merely a convenience but a practical necessity: principled composition of complementary reasoning modes within a foundational framework provides a robust foundation for future vericoding systems.

Acknowledgments

We thank the ASE'26 reviewers for their feedback. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001 and the grant "Neuro-Symbolic Proof Automation for Multi-Modal Verifiers" awarded to Ilya Sergey by Beneficial AI Foundation.

Data Availability Statement. An artefact containing the implementation of LeetProof, the benchmark suite of 50 LeetCode problems, and the evaluation harness for reproducing the results from Sec. 5 and Sec. 6 is publicly available [16].

Generative AI Disclosure. We used Anthropic Claude Opus 4.6 to assist with harness development, diagram rendering, typesetting of code listings, and spell-checking the human-written English text. The authors take full responsibility for any resulting errors.

References

[1] Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladimir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. 2025. Aristotle: IMO-level Automated Theorem Proving. *CoRR* abs/2510.01346 (2025). doi:10.48550/ARXIV.2510.01346

[2] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. 2025. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement. In *ICML (Proceedings of Machine Learning Research)*. PMLR / OpenReview.net. https://proceedings.mlr.press/v267/aggarwal25a.html

[3] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press. doi:10.1017/CBO9781107256552

[4] Mantas Baksys, Stefan Zetzsche, Olivier Bouissou, Remi Delmas, Soonho Kong, and Sean B. Holden. 2025. ATLAS: Automated Toolkit for Large-Scale Verified Code Synthesis. *CoRR* abs/2512.10173 (2025). arXiv:2512.10173 doi:10.48550/ARXIV.2512.10173

[5] Debangshu Banerjee, Olivier Bouissou, and Stefan Zetzsche. 2026. DafnyPro: LLM-Assisted Automated Verification for Dafny Programs. *CoRR* abs/2601.05385 (2026). doi:10.48550/ARXIV.2601.05385

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS (LNCS, Vol. 13243)*. Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[7] Lukas Bulwahn. 2012. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *CPP (LNCS)*. Springer, 92–108. doi:10.1007/978-3-642-35308-6_10

[8] Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. 2025. A Benchmark for Vericoding: Formally Verified Program Synthesis. *CoRR* abs/2509.22908 (2025). doi:10.48550/ARXIV.2509.22908

[9] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu K. Lahiri, and Nikhil Swamy. 2025. Towards Neural Synthesis for SMT-Assisted Proof-Oriented Programming. In *ICSE*. IEEE, 1755–1767. doi:10.1109/ICSE55347.2025.00002

[10] Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K Lahiri, Tao Xie, and Lidong Zhou. 2025. Automated Proof Generation for Rust Code via Self-Evolution. *CoRR* abs/2410.15756 (2025). ICLR 2025. doi:10.48550/ARXIV.2410.15756

[11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*. ACM, 268–279. doi:10.1145/351240.351266

[12] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE (LNCS, Vol. 12699)*. Springer, 625–635. doi:10.1007/978-3-030-79876-5_37

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[14] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE (LNCS, Vol. 9195)*. Springer, 378–388. doi:10.1007/978-3-319-21401-6_26

[15] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL*. ACM, 689–700. doi:10.1145/2676726.2677006

[16] Yueyang Feng, Dipesh Kafle, Vladimir Gladshtein, Vitaly Kurin, George Pîrlea, Qiyuan Zhao, Peter Müller, and Ilya Sergey. 2026. LeetProof: Artefact for "Certified Synthesis via Multi-Modal Verification". Zenodo. Version v2. Available at https://zenodo.org/records/19624966; includes implementation, benchmark suite, and evaluation harness. doi:10.5281/zenodo.19624966

[17] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME (LNCS, Vol. 2021)*. Springer, 500–517. doi:10.1007/3-540-45251-6_29

[18] Vladimir Gladshtein, Vitaly Kurin, Yueyang Feng, Dipesh Kafle, George Pîrlea, Qiyuan Zhao, and Ilya Sergey. 2026. Velvet: A Foundational Multi-modal Verifier for Imperative Programs in Lean. In *CAV (LNCS, Vol. 16683)*. Springer, 228–242. doi:10.1007/978-3-032-32526-6_11

[19] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2026. Foundational Multi-Modal Program Verifiers. *Proc. ACM Program. Lang.* 10, POPL (2026), 1–28. doi:10.1145/3776719

[20] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *ICSE*. ACM, 187:1–187:13. doi:10.1145/3597503.3639581

[21] Harrison Goldstein, Hila Peleg, Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin C. Pierce. 2026. The Search for Constrained Random Generators. *Proc. ACM Program. Lang.* 10, PLDI (2026). doi:10.1145/3808329

[22] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. doi:10.1145/363235.363259

[23] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. 2026. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature* 651, 8106 (2026), 607–613. doi:10.1038/s41586-025-09833-y

[24] John Hughes. 2011. Specification based testing with QuickCheck: *tutorial talk*. In *FMCAD*. FMCAD Inc., 17. http://dl.acm.org/citation.cfm?id=2157659

[25] Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. 2024. Cobblestone: A Divide-and-Conquer Approach for Automating Formal Verification. *CoRR* abs/2410.19940 (2024). arXiv:2410.19940 http://arxiv.org/abs/2410.19940

[26] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects Comput.* 27, 3, 573–609. doi:10.1007/S00165-014-0326-7

[27] Shuvendu K. Lahiri. 2024. Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages. In *FMCAD*. IEEE, 142–147. doi:10.34727/2024/isbn.978-3-85448-065-5_19

[28] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *SOSP*. ACM, 438–454. doi:10.1145/3694715.3695952

[29] leanprover-community. 2025. Plausible: A property testing framework for Lean 4. https://github.com/leanprover-community/plausible. Accessed on 9 July 2025.

[30] leanprover-community. 2026. Lean Language Reference: The grind tactic. https://lean-lang.org/doc/reference/latest/The--grind--tactic/.

[31] leanprover-community. 2026. Lean Language Reference: Type Classes. https://lean-lang.org/doc/reference/latest/Type-Classes/.

[32] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[33] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL*. ACM Press, 333–343. doi:10.1145/199448.199528

[34] Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *CPP*. ACM, 253–266. doi:10.1145/3573105.3575671

[35] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2025. DafnyBench: A Benchmark for Formal Software Verification. *Trans. Mach. Learn. Res.* 2025 (2025). https://openreview.net/forum?id=yBgTVWccIx

[36] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. (2024), 1509–1520. doi:10.1145/3691620.3695521

[37] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *ICSE*. IEEE, 16–28. doi:10.1109/ICSE55347.2025.00129

[38] The mathlib Community. 2020. The Lean mathematical library. In *CPP*. ACM, 367–381. https://github.com/leanprover-community/mathlib4. doi:10.1145/3372885.3373824

[39] Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. In *CAV (LNCS, Vol. 12225)*. Springer, 190–202. doi:10.1007/978-3-030-53291-8_12

[40] Brando Miranda, Zhanke Zhou, Allen Nie, Elyas Obbad, Leni Aniva, Kai Fronsdal, Weston Kirk, Dilara Soylu, Andrea Yu, Ying Li, and Sanmi Koyejo. 2025. VeriBench: End-to-End Formal Verification Benchmark for AI Code Generation in Lean 4. (2025). https://openreview.net/forum?id=rWkGFmnSNl

[41] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* 1, FSE (2024), 812–835. doi:10.1145/3643763

[42] Abdalrhman Mohamed, Tomaz Mascarenhas, Muhammad Harun Ali Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark W. Barrett. 2025. lean-smt: An SMT Tactic for Discharging Proof Goals in Lean. In *CAV (LNCS, Vol. 15933)*. Springer, 197–212. doi:10.1007/978-3-031-98682-6_11

[43] Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025). doi:10.1145/3720499

[44] Prasita Mukherjee, Minghai Lu, and Benjamin Delaware. 2025. LLM-Assisted Synthesis of High-Assurance C Programs. In *ASE*. IEEE, 3108. doi:10.1109/ASE63991.2025.00255

[45] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62. doi:10.1007/978-3-662-49122-5_2

[46] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. doi:10.1007/3-540-44802-0_1

[47] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. ACM, 614–630. doi:10.1145/2908080.2908118

[48] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *ITP (LNCS, Vol. 9236)*. Springer, 325–343. doi:10.1007/978-3-319-22102-1_22

[49] Arthur Paulino, Damiano Testa, Edward Ayers, Evgenia Karunus, Henrik Bövinga, Jannis Limperg, Siddhartha Gadgil, and Siddharth Bhat. 2024. Metaprogramming in Lean 4. Available at https://leanprover-community.github.io/lean4-metaprogramming-book/.

[50] George Pîrlea, Vladimir Gladshtein, Elad Kinsbruner, Qiyuan Zhao, and Ilya Sergey. 2025. Veil: A Framework for Automated and Interactive Verification of Transition Systems. In *CAV (LNCS, Vol. 15933)*. Springer, 26–41. doi:10.1007/978-3-031-98682-6_2

[51] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational Compilation for Performance-Critical Applications. In *PLDI*. ACM, 918–932. doi:10.1145/3519939.3523706

[52] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *PACMPL* 3, POPL (2019), 72:1–72:30. doi:10.1145/3290385

[53] Yicheng Qian, Joshua Clune, Clark W. Barrett, and Jeremy Avigad. 2025. LeanAuto: An Interface Between Lean 4 and Automated Theorem Provers. In *CAV (LNCS, Vol. 15933)*. Springer, 175–196. doi:10.1007/978-3-031-98682-6_10

[54] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. doi:10.1109/LICS.2002.1029817

[55] Rocq Development Team. 2025. The Rocq Prover. https://rocq-prover.org. Version 9.0.0, released March 12, 2025.

[56] Grigore Rosu. 2017. K: A Semantic Framework for Programming Languages and Formal Analysis Tools. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 50. IOS Press, 186–206. doi:10.3233/978-1-61499-810-5-186

[57] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2025. Lean Copilot: Large Language Models as Copilots for Theorem Proving in Lean. (2025), 144–169. https://proceedings.mlr.press/v288/song25a.html

[58] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *SAIV (LNCS, Vol. 14846)*. Springer, 134–155. doi:10.1007/978-3-031-65112-0_7

[59] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *POPL*. ACM, 256–270. doi:10.1145/2837614.2837655

[60] Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. 2025. CLEVER: A Curated Benchmark for Formally Verified Code Generation. *CoRR* abs/2505.13938 (2025). NeurIPS 2025 Datasets and Benchmarks. doi:10.48550/ARXIV.2505.13938

[61] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *ICSE*. IEEE, 347–359. doi:10.1109/ICSE55347.2025.00161

[62] Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. 2026. Agentic Verification of Software Systems. *Proc. ACM Softw. Eng.* 3, FSE (2026). doi:10.1145/3808164

[63] Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. 2025. Hilbert: Recursively Building Formal Proofs with Informal Reasoning. *CoRR* abs/2509.22819 (2025). arXiv:2509.22819 doi:10.48550/ARXIV.2509.22819

[64] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473589

[65] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025). doi:10.1145/3763174

[66] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *NeurIPS*. https://openreview.net/forum?id=mXpq6ut8J3

[67] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. 2025. VERINA: Benchmarking Verifiable Code Generation. *CoRR* abs/2505.23135 (2025). ICLR 2026. doi:10.48550/ARXIV.2505.23135

[68] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *ISSTA*. ACM, 1592–1604. doi:10.1145/3650212.3680384

Received 2026-03-26; accepted 2026-06-18