Velvet: A Multi-Modal Verifier for Effectful Programs

Vladimir Gladshtein
National University of Singapore
Singapore
vgladsht@comp.nus.edu.sg

George Pîrlea
National University of Singapore
Singapore
gpirlea@comp.nus.edu.sg

Qiyuan Zhao
National University of Singapore
Singapore
qiyuanz@comp.nus.edu.sg

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

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

Abstract

We present Velvet—a Dafny-style verifier for imperative programs embedded into Lean proof assistant. Like Dafny, Velvet supports reasoning about effectful programs featuring mutable state, loops, and non-determinism. Unlike Dafny, Velvet seamlessly combines automated SMT-based proofs with interactive proof mode of Lean proof assistant, to which it is embedded, thus, allowing for *multi-modal* proofs. By virtue of being implemented as a Lean library, Velvet enjoys interaction with the rest of Lean ecosystem, in particular, its rich library of mathematical theories. In this presentation, we will give a tour of Velvet's main features and outline the techniques underlying its embedding into Lean.

1 Introduction

Modern SMT-based automated program verifiers, such as Dafny [14], Viper [20], and Verus [12] allow their users to enjoy "push-button" machine-checked correctness proofs of imperative programs that are ascribed a suitable specification and are annotated with loop invariants.

Unfortunately, SMT-based automation comes at a price: because the modern-day solvers frequently struggle with complex statements outside decidable fragments of first-order logic, users often need to supply additional annotations, in the form of assertions, lemmas, or even custom trigger instantiation strategies [12], to enable the solver to discharge the corresponding verification conditions. When a proof fails, such automated verifiers typically offer little help in pinpointing the precise issue in the specification, in contrast to interactive foundational proof assistants such as Rocq [23] or Lean [6], which allow the user to inspect the *proof context*, explore the available facts, and either guide the user toward a successful proof or conclude that the desired statement is false. Finally, none of the existing automated verifiers offer an easy or principled way to interact with the rich body of formalised mathematical theories available in the standard libraries of interactive theorem provers. As a result, verifying programs against specifications that rely on complex mathematical definitions becomes difficult, often forcing users to resort to ad hoc encodings instead.

In this work, we present Velvet—a verifier for imperative programs with effects such as mutable state, non-determinism, and non-termination, which addresses the listed above shortcomings of existing automated program provers, built as an instance of Loom [7], a new general framework for implementing foundational multi-modal program verifiers.

Velvet is implemented as a library on top of Lean proof assistant, so Velvet programs are Lean programs, whose semantics is given by composing a number of computational effects implemented as monad instances. Velvet is designed to support *multi-modal* verification: programs in its language can be compiled, executed, validated using property-based testing, and formally verified within one unified environment. Velvet seamlessly integrates SMT-based automation (as in Dafny and Verus) with interactive Lean proofs. When automation fails, the user can complete the proof interactively using standard Lean tactics. Velvet's specification language allows one to separately verify functional correctness of programs and their termination. By building on a library of monadic effects, Velvet supports mutable variables, arrays, loops (including non-terminating ones), and even Dafny-style non-deterministic choice operator, in addition to all native data types of Lean. This makes it ideal for reasoning about correctness of textbook algorithms and competitive programming tasks. As a Lean library, Velvet inherits the entire Lean ecosystem, including mathlib [18], the world's largest library of formalised mathematics. Our examples include verified implementations of sparse matrix operations [11], whose proofs rely on definitions from mathlib and theorems about them. Finally, Velvet is itself formally verified: we have proved, in Lean, that any program verified in Velvet satisfies its specification at runtime.

In this talk, we will provide a demonstration of Velvet's key features following a series of characteristic examples, outline implementation its Lean, and conclude with a list of possible future directions to extend Velvet for reasoning about a richer class of program properties and specifications.

*Work done during an internship at National University of Singapore.

Dafny'26, Rennes, France
2026.

```
1  method sqrt (x: ℕ) return (res: ℕ)
2    ensures res * res ≤ x
3    ensures ∀ i, i * i ≤ x → i ≤ res
4    do
5      if x = 0 then
6        return 0
7      else
8        let mut i := 0
9        while i * i ≤ x
10       invariant ∀ j, j < i → j * j ≤ x
11       do
12         i := i + 1
13       return i - 1
14
15 prove_correct sqrt by loom_solve
```

Fig. 1. Fully automated proof of a discrete square root

```
1  method cbrt (x: ℕ) return (res: ℕ)
2    ensures res * res * res ≤ x
3    ensures ∀ i, i * i * i ≤ x → i ≤ res
4    do
5      if x = 0 then
6        return 0
7      else
8        let mut i := 0
9        while i * i * i ≤ x
10       invariant  ∀ j, j < i → j * j * j ≤ x
11       do
12         i := i + 1
13       return i - 1
14
15 prove_correct cbrt by
16   loom_solve -- SMT left one unsolved goal
17   assumption
```

Fig. 2. Multi-modal proof of a discrete cube root

2 Velvet by Example

In this section, we provide a tutorial-style overview of Velvet's key features: support for SMT-based proof automation, multi-modal proofs, runtime verification via property-based testing, and reasoning about program termination.

2.1 Automated Program Verification via SMT

Fig. 1 displays a simple program written in Velvet, which computes an integer under-approximation of the square root of a natural number. The program is missing a precondition, which is trivially `True`, but features two conjuncts in its postcondition, each given by an individual `ensures`-clause. The first states that the result of the program `res`, squared, is smaller than its argument `x`, while the second one states that `res` is the largest number to under-approximate $\sqrt{x}$.

The body of the program at lines 5-13 computes the result. Line 10 provides an explicit loop invariant for the `while`-loop at lines 9-12, which is required to prove the postcondition. Line 15 constitutes the proof of the program *w.r.t.* the ascribed specification. It is achieved by Velvet computing the verification conditions (VC) for the program and the specification by implementing a version of the weakest-precondition calculus, producing a Lean statement, whose validity must be proven in order to show that the program is correct. The proof is done by the tactic `loom_solve` that produces the required VCs and discharges them using an external SMT solver. The interaction between Lean and the SMT solver is done by using the lean-auto library [22] that compiles Lean statements to SMT-LIB queries and sends them to one of the supported SMT solvers; Velvet currently uses cvc5 [2] as the default one. In this example, all VCs produced by Velvet could be proven by an SMT solver fully automatically. As of now, the outcome of SMT is trusted, yet proof term reconstruction is possible in principle by using LeanSMT library as an alternative to LeanAuto, as done in some other Lean-based project that rely on SMT solvers [19].

2.2 Combining Automated and Interactive Proofs

Fig. 2 shows a modified version of the program from Fig. 1, which computes an integer cube root of a natural number. Even though it is very similar to the previous example, cvc5 fails to discharge all its VCs automatically, leaving one unsolved, which Velvet displays as a Lean proof goal:

```
x : ℕ
if_neg : ¬x = 0
i : ℕ
invariant_6 : ∀ j < i, j * j * j ≤ x
if_neg_1 : ¬i * i * i ≤ x
⊢ ∀ j < i, j * j * j ≤ x
```

As common in Lean, the proof context with the available variables and assumptions is shown above the ⊢ symbol, while the remaining goal to prove (`∀ j < i, j * j * j ≤ x`) is displayed right after it at the last line. An observant reader might find it quite surprising that the goal could not be proven automatically, as its statement coincides with one of the assumptions inferred from the program annotations, namely `invariant_6` (Velvet assigns names to hypotheses automatically based on their location and designation). The proof can be now finished interactively by using a standard Lean tactic `assumption` (line 17), which simply looks through the proof context, identifies a hypothesis that matches the conclusion of the goal, and uses it to complete the proof.

The demonstrated example highlights a crucial difference between Velvet and automated verification tools such as Dafny and Verus. While the latter tools put the main automation burden on the SMT solvers, reducing the user's involvement into the proof process to proving helper lemmas and providing additional assertions, in Velvet, an SMT solver is *just one of the available ways* to automate proofs of the generated verification conditions. That is, if an SMT solver fails to discharge a certain VC, *e.g.*, due to the VC being outside of its

```
1  method insertionSort (mut a: arrInt) return (u: Unit)
2  require 1 ≤ size a
3  ensures ∀ i j, i ≤ j < size a → aNew[i] ≤ aNew[j]
4  ensures toMultiset a = toMultiset aNewdo
5    let a₀ := a
6    let mut n := 1
7    while n ≠ size a
8    invariant n ≤ size a
9    invariant ∀ i j, i < j < n → a[i] ≤ a[j]
10   invariant toMultiset a = toMultiset a₀
11   decreasing size a₀ - n do
12     let mut mind := n
13     while mind ≠ 0
14     invariant mind ≤ n
15     invariant ∀ i j, i < j ≤ n ∧ j ≠ mind → a[i] ≤ a[j]
16     invariant toMultiset a = toMultiset a₀
17     decreasing mind do
18       if a[mind] < a[mind - 1] then
19           swap a (mind - 1) mind
20       mind := mind - 1
21     n := n + 1
22 prove_correct insertionSort by loom_solve
```

Fig. 3. Velvet code and proof of insertion sort

supported theories, the user can proceed to prove it in an interactive mode or by using any means of automation available in Lean, such as the `Aesop` tactic [17]. Furthermore, one can also simplify an unsolved VC (as it's just a Lean goal) to the point it fits into one of SMT-supported first-order theories, at which point an SMT solver can be invoked via a dedicated Lean tactic, such as `auto` [22].

Finally, Velvet allows for a more traditional SMT-centered verification mode, in which the user can include additional facts proven as ordinary Lean theorems, into the database of theories available to SMT, by marking the respective theorems with the `@[solverHint]` annotation, so they can be used when performing an automated proof.

2.3 Testing Programs against Their Specifications

In addition to proof automation, Velvet provides facilities that facilitate testing programs against their specifications. This process, known as *property-based testing* (PBT) [5], has proven to be useful for exposing issues, either in the program or in the specification, in practice [8].

We illustrate how PBT is done in Velvet through an implementation of insertion sort shown in Fig. 3.[1] The input `a` is ascribed the type `arrInt`, which can be considered as an abstraction of integer arrays, instantiated at runtime with the concrete type of integer arrays in Lean, `Array Int`. Notably, `a` is marked as **`mut`** since its content is mutated during the program execution. The value of `a` at the end of the execution is referred to as `aNew` in the **`ensures`** clauses.

To test `insertionSort` and its specification, the user may first run the following commands in sequence:

[1]For now, let us ignore the **`decreases`** annotations in grey boxes at lines 11 and 17; they are necessary for proving program termination, but not safety.

```
def testWrapper (rounds : Nat) : IO Unit := do
  let g : Gen (Array Int × Bool) := do
    -- Sample an integer array
    let arr ← SampleableExt.interpSample (Array Int)
    -- Run the tester with the sampled array
    let passed? := insertionSortTester arr
    pure (arr, passed?)
  -- Control the size of sampled array and integer
  let exampleSize := 10
  -- Test for multiple rounds
  for _ in [1: rounds] do
    let (arr, passed?) ← Gen.run g exampleSize
    if !passed? then
      -- Report the counterexample
      IO.println s!"postcondition violated, input: {arr}"
      break
```

Fig. 4. A harness for randomised testing of `insertionSort`

```
extract_program_for insertionSort
prove_precondition_decidable_for insertionSort
prove_postcondition_decidable_for insertionSort by
  (exact (decidable_by_nat [(size arr), (size arr)]))
derive_tester_for insertionSort
```

The command **`extract_program_for`** invokes Loom's extraction mechanism to determinise a Velvet program, producing a directly executable functional program. In this case, since `insertionSort` has no non-deterministic choice, the extraction guarantees that its determinised version runs exactly the same code. The **`prove_precondition_decidable_for`** command attempts to construct a `Decidable` instance of the precondition (*i.e.*, the conjunction of **`require`** clauses). This instance, when given a concrete input (*e.g.*, `a`), can be used for deciding whether the precondition holds. The command **`prove_postcondition_decidable_for`** serves a similar purpose for the postcondition (*i.e.*, the conjunction of **`ensures`** clauses) and takes both the input and the post-state (*e.g.*, `aNew`) as arguments. For simple pre- and postconditions, Lean can automatically infer the corresponding `Decidable` instance. In other cases, the user can assist by applying tactics in a trailing **`by`** block. For example, here the `decidable_by_nat` tactic provided by Velvet is used to help produce the `Decidable` instance for the post-condition of `insertionSort`. Finally, the **`derive_tester_for`** command puts together the definitions produced by the previous commands to construct an end-to-end tester function. Given a specific input, it returns a `Bool` indicating whether the implication from "the input meets the precondition" to "the post-state of the determinised target Velvet program satisfies the postcondition" holds.

The user can then freely use the tester with any existing PBT library. The code in Fig. 4 illustrates one such usage, where Gen is a monad and `SampleableExt.interpSample` is a Gen monadic computation that samples a value of a given type. Both are from Lean's Plausible PBT framework [13].

```
▼ case «size a₀ - n»
arrInt : Type
arr_inst_int : TArray ℤ arrInt
a : arrInt
require_13 : size a ≥ 1
a_1 : arrInt
invariant_1 : size a_1 = size a
invariant_2 : 1 ≤ size a_1
invariant_3 : ∀ (i j : ℕ), i < j → j < 1 → a_1[i] ≤ a_1[j]
invariant_4 : toMultiset a_1 = toMultiset a
if_pos : ¬1 = size a_1
a_2 : arrInt
mind : ℕ
invariant_7 : size a_2 = size a
invariant_8 : mind ≤ 1
invariant_9 : ∀ (i j : ℕ), i < j → j ≤ 1 → ¬j = mind →
a_2[i] ≤ a_2[j]
invariant_10 : toMultiset a_2 = toMultiset a
done_11 : ¬¬mind = 0
⊢ size a - 1 < size a - 1
```

Fig. 5. Lean InfoView for a failed termination proof

Property-based testing comes useful, for instance, in the following possible scenario: the user changes the postcondition at the line 3 of Fig. 3 into `∀ i j, i < j < size a → aNew[i] < aNew[j]`. By running `testWrapper` with a sufficiently large `rounds` (*e.g.*, 500), the user will obtain a counterexample such as `[1, -3, 1]`, and then know something has gone wrong.

2.4 Reasoning about Termination

As a default mode, Velvet allows on to verify partial correctness of programs: unlike Dafny, it is not required for Velvet programs to terminate in order to satisfy an ascribed specification. In fact, removing or commenting out line 21 at Fig. 3 will not prevent the proof at line 22 from successding: indeed, this way, the `while`-loop at lines 7-21 will never terminate, hence the program will trivially satisfy any postcondition, even `False`. Luckily, Velvet allows for different treatment of non-termination: as a success or as a failure. Switching to the latter mode is as easy as adding a line

```
open TotalCorrectness
```

before the code of the program in question. We omit the discussion of the theoretical foundations needed to support both kinds of correctness reasoning, total and partial, as they are described in Sec. 5 of the work [7].

In the latter case, the user is expected to provide explicit termination measures by supplying the respective `decrease`-clauses to each `while`-loop, as highlighted by the grey boxes at lines 11 and 17 of Fig. 3. With these annotations provided and the code at line 21 commented out, the VC generator will produce, amongst others, the Lean goal shown in Fig. 5, which cannot be proven automatically (or even interactively), but provides a useful insight into why the proof has failed. In particular, in indicates that the termination measure `size` a_0 `- n` provided at line 11 does not, in fact, decrease with each iteration of the outer `while`-loop.

2.5 Other Features

Amongst other features of Velvet are (a) the support for Dafny-style *let-such-that* operator `:|` (also known as Hilbert's epsilon operator) [15], which is useful for modelling nondeterministic choice, and (b) an ability to combine reasoning about pure Lean data types and mathematical theories with program verification.

Our case studies include a large-scale verification effort that makes use of both these features: proving correctness of a *parallel* sparse tensor multiplication procedure [11], which manipulates several compressed matrices implemented as collections of arrays with bespoke encodings. The proof is completed following a so-called *two-layered paradigm* [1]. That is, the actual implementation is (1) first shown to *refine* a pure Lean function f, which performs manipulation with the algebraic representation of compressed matrices, and then (2) f is proven to correctly implement matrix multiplication. The first part of the proof is achieved in Velvet via an SMT solver, provided a number of facts about f, which are proven as ordinary Lean theorems, so the proof can treat f as an uninterpreted function. The second part of the proof is done interactively in Lean; pleasantly, it does not involve any reasoning about effectful code whatsoever.

3 Implementing Velvet in Loom

Velvet is built as an instance of Loom [7]—a general and versatile Lean library for implementing provably correct multimodal verifiers on top of Lean by means of a monadic shallow embedding. In this section, we briefly outline the specific components of Velvet that allow for its characteristic non-trivial features: intrinsic verification, executable nondeterministic choice operator, an ability to switch between reasoning about partial and total correctness, and first-order abstractions of common data structures.

3.1 Annotations for Intrinsic Program Verification

Under the hood, Velvet programs are ordinary Lean computations in the `VelvetM` monad (see the next subsection), enriched with specification annotations: `require`/`ensures` clauses, assertions, loop invariants, and termination measures. These annotations are verification-only: they do not change the executable behavior of the code. At the top level, Velvet collects the conjunction of all `require` and `ensures` clauses into the pre/post pair of a Hoare-style correctness statement for the program. Likewise, loop `invariant` annotations are compiled into hints that the verification condition generator (VCGen) can recognise. Concretely, a loop invariant is inserted via a no-op combinator `invGadget`:

```
def invGadget (inv : Prop) : VelvetM Unit := pure ()
```

At runtime, this expands to an idle Lean computation, but the VCGen works over the program's encoding and, hence, can take advantage of the first argument of `invGadget` to generate appropriate verification conditions for the loop.

3.2 Non-Deterministic Choice Operator

As we have mentioned before, Velvet programs are computations inside the `VelvetM` monad. This monad supports a non-deterministic choice operator and divergent computations, allowing one to reason about partial correctness. In this subsection, we will discuss the first component. The general form of non-deterministic choice operator is:

```
def pickSuchThat {τ} (p : τ -> Prop): VelvetM τ := ...
```

This operator chooses an arbitrary value of the type τ that satisfies the predicate `p`. Following Dafny-style notation, in program code, it is written as `let x :| p x`.

The weakest precondition semantics for this operator depends on the verification style user is interested in and can be controlled. For example, if user wants to prove that specification holds for *all* possible choices of `pickSuchThat`, the weakest precondition should be a conjunction of all postconditions for the values that satisfy the predicate `p`:

$$wp\ (pickSuchThat\ p)\ post = \bigwedge_{x \in p} post\ x \tag{1}$$

Such choice operator is called "demonic" [3] and can be enabled by simply opening a corresponding semantics namespace `open DemonicChoice`. At the same time, the user might be interested in proving a reachability property for the program, *i.e.*, proving that the specification holds for some choice of `pickSuchThat`. In this case, the weakest precondition should be a disjunction of all post-conditions for the values that satisfy the predicate and can be enabled by `open AngelicChoice`.

Executing a non-deterministic choice operator is more subtle. The problem is that `pickSuchThat` on its own does not provide a way to actually choose a value from the type τ that satisfies `p`. To execute computation `x` of type `VelvetM` α, Velvet implements a `VelvetM.run` function which automatically tries to infer a witness of a `Findable` type class for each predicate `p` in each non-deterministic choice in `x`. Such type class provides a recipe how to find value `t : τ`, satisfying `p`. In practice, instantiating `Findable` is rarely a problem. For example, if `p` is a decidable predicate and τ is `Encodable` (*i.e.*, countable) such instance will be inferred automatically.

3.3 Proofs about Non-Terminating Programs

Unlike vanilla Lean, where every recursive definition must carry an explicit termination argument, Velvet decouples functional correctness from the termination argument: a termination measure (the `decreasing` annotation) is required only when establishing total correctness. At the same time, for partial correctness, no such measure is needed—the program may diverge, and proofs assert only that, if it terminates, the postcondition holds. The user selects the intended proof mode and the semantics of divergence—by opening the corresponding semantic namespace, choosing between `open TotalCorrectness` to work with total correctness and `open PartialCorrectness` for partial correctness.

Velvet makes it possible to decouple proofs of partial and total correctness. This can be done by defining two identical versions of a program, one without termination related annotations (highlighted in grey in Fig. 3) and another without functional correctness-specific ones. The proof of the former constitutes partial correctness, and the latter proves termination. One can then use the following Velvet theorem to prove the program's total correctness:

```
lemma partial_total_split {α} : ∀ (c₁ c₂ : VelvetM α)
  triplePartial P c₁ Q →
  (P : Prop) (Q : α → Prop), eraseEq c₁ c₂ →
  tripleTotal P c₂ (λ _, True) → tripleTotal P c₁ Q
```

3.4 First-Order Abstractions of Data Structures

To make VCs amendable to SMT solvers, Velvet uses first-order (FO) representations of Lean data structures (such as Lean arrays). The reason is that SMT solvers are more effective on quantifier-light, algebraic interfaces than on rich, higher-order APIs. For example, in the insertion sort implementation from Fig. 3 we model arrays by the array FO theory rather than by Lean 's concrete implementation. Such an array theory is encoded via the `TArray` type class:

```
class TArray (α : outParam Type) (κ: Type) where
  get : Nat → κ → α
  set : Nat → α → κ → κ
  get_set (idx₁ idx₂ val arr) :
     idx₁ < size arr -> get idx₁ (set idx₂ val arr) =
      if idx₁ = idx₂ then val else get idx₁ arr
  ... -- other array theory operations and axioms
```

That is, instead of implementing insertion sort taking an array `a` of type `Array Int`, it receives `a` of type `arrInt`, for which we assume an instance of `TArray Int arrInt`. This way, we hide the concrete implementation details from the SMT solver: it only gets to see the abstract interface `TArray` where `set` and `get` will be translated into uninterpreted functions satisfying the array theory axioms such as `get_set`.

Finally, to execute programs manipulating with such array abstractions Velvet provides a `TArray` α `(Array` α`)` instance for arbitrary α. It means that user can pass regular Lean integer arrays anywhere `arrInt` is expected.

4 Future Directions

We believe, our development of Velvet—the first foundational multi-modal Dafny-style verifier on top of Lean—opens avenues for several lines of exciting future work. In particular, we envision its following extensions and application, which we are planning to explore in the future.

First, we are going to apply Velvet for specifying and verifying computational complexity results for textbook algorithms. We believe, in many cases, this can be achieved by

enhancing the `VelvetM` monad with a layer of the state monad transformer `StateT`, which is supported natively by Loom [7], and using it to updated a counter for "computational credits", treating it as ghost state. We expect that in some cases, *e.g.*, when reasoning about amortised complexity, we will have to extend Velvet with reasoning principles similar to that of Separation Logic with time credits [4]. We are confident that this can be achieved by implementing and specifying a set of custom operations on top of the state monad.

Another direction we are going to explore is introducing a type system with linear capabilities on top of Velvet, thus, allowing it to generate more SMT-friendly verification conditions, similarly to what is achieved in tools such as Linear Dafny [16] and Verus [12] by means of relying on a bespoke linear type system or by exploiting the guarantees provided by the rustc Rust compiler. An advantage of implementing this approach on top of Velvet is that the soundness claims similar to those of Linear Dafny and Verus can be mechanically verified within the proof assistant, thus reducing the trusted code base of the verifier to that of Lean.

Finally, we are planning to tackle large-scale systems verification efforts in the style of IronFleet [10] by making use of *multiple* foundational multi-modal verifiers implemented on top of Loom. For instance, by verifying in Velvet that an implementation of a distributed system refines its model defined in Veil [21], another verifier built on top of Loom, we can establish that all properties of the model—safety and liveness—provably transfer to the executable implementation. Furthermore, conducting such verification efforts in Lean will make it possible to extend the state-of-the-art in trustworthy systems to a larger class of their specifications, including information-theoretic and security properties [9].

References

[1] Andrew W. Appel. 2022. Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk). In *CPP*. ACM, 2–11. https://doi.org/10.1145/3497775.3503951

[2] 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. https://doi.org/10.1007/978-3-030-99524-9_24

[3] Manfred Broy and Martin Wirsing. 1981. On the Algebraic Specification of Nondeterministic Programming Languages. In *CAAP (LNCS, Vol. 112)*. Springer, 162–179. https://doi.org/10.1007/3-540-10828-9_61

[4] Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reason.* 62, 3 (2019), 331–365. https://doi.org/10.1007/S10817-017-9431-7

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

[6] 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. https://doi.org/10.1007/978-3-319-21401-6_26

[7] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2025. Foundational Multi-Modal Program Verifiers. Code available at https://github.com/verse-lab/loom.

[8] 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. https://doi.org/10.1145/3597503.3639581

[9] Kiran Gopinathan and Ilya Sergey. 2019. Towards Mechanising Probabilistic Properties of a Blockchain. In *CoqPL*. Informal Proceedings.

[10] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM, 1–17. https://doi.org/10.1145/2815400.2815428

[11] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. 2023. LAProof: A Library of Formal Proofs of Accuracy and Correctness for Linear Algebra Programs. In *ARITH*. IEEE, 36–43. https://doi.org/10.1109/ARITH58626.2023.00021

[12] 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. https://doi.org/10.1145/3694715.3695952

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

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

[15] K. Rustan M. Leino. 2015. Compiling Hilbert's epsilon operator. In *LPAR (EPiC Series in Computing, Vol. 35)*. EasyChair, 106–118. https://doi.org/10.29007/RKXM

[16] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–28. https://doi.org/10.1145/3527313

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

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

[19] 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. https://doi.org/10.1007/978-3-031-98682-6_11

[20] 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. https://doi.org/10.1007/978-3-662-49122-5_2

[21] 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. https://doi.org/10.1007/978-3-031-98682-6_2

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

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