Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean

Vladimir Gladshtein[1], Vitaly Kurin[2*], Yueyang Feng[1], Dipesh Kafle[1], George Pîrlea[1], Qiyuan Zhao[1], and Ilya Sergey[1] (✉)

[1] VERSE lab, National University of Singapore, Singapore
verse-lab.org
[2] Neapolis University Pafos, Cyprus

Abstract. We present Velvet—a Dafny-style verifier for imperative programs embedded in the 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 the interactive proof mode of the Lean proof assistant, in which it is embedded, thus enabling *multi-modal* proofs. Implemented as a Lean library, Velvet enjoys interaction with the rest of the Lean ecosystem, and in particular, with its automation tactics and rich library of mathematical theories. In this paper, we give a tour of Velvet's features, outline the techniques underlying its implementation, and evaluate its performance and expressivity in comparison with Dafny.

1 Introduction

Modern SMT-based automated program verifiers, such as Dafny [18], Viper [25], and Verus [15] 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. The SMT-based automation comes at a price: because 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 [15], 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 [29] or Lean [24], 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 shortcomings of existing automated program verifiers listed above. Velvet is implemented as a library on top of the Lean proof assistant as an instance of

* Work done during a research internship to National University of Singapore.

Loom [10], a new general framework for implementing foundational multi-modal program verifiers. Velvet programs are Lean programs, with semantics 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. It 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. By building on a library of monadic effects, Velvet supports mutable variables, arrays, loops (including non-terminating ones), and the random choice operator, in addition to all of Lean's native datatypes. This makes it ideal for reasoning about the correctness of textbook algorithms and competitive programming tasks. Velvet's specification language allows one to separately verify the functional correctness and termination of programs, and combine those proofs to show total correctness. As a Lean library, Velvet inherits access to the entire Lean ecosystem, including mathlib [22], the world's largest library of formalised mathematics. Finally, Velvet is *foundational*, *i.e.*, it is itself formally verified: we have proven that any program verified in it satisfies its specification at runtime.

2 Velvet by Example

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

2.1 Auto-Active Program Verification via SMT

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

Fig. 1: Discrete square root in Velvet

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 no greater than its argument `x`, while the second states that `res` is the largest number to under-approximate $\sqrt{\texttt{x}}$. The body of the program at lines 4–12 computes the result. Line 9 provides an explicit loop invariant for the **`while`**-loop at lines 8–11, which is required to prove the postcondition. Line 14 constitutes the proof of the program *wrt* 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 tries to discharge

them with the `grind` automation tactic [17]. Any remaining goals are then discharged with the `loom_smt` tactic using an external SMT solver. The interaction between Lean and the SMT solver is done by using the lean-auto library [28] that compiles Lean statements to SMT-LIB [2] queries and sends them to an SMT solver; Velvet currently uses cvc5 [1] by default, but can be configured to use Z3 [7]. In this example, all VCs produced by Velvet can be proven by either `grind` or an SMT solver automatically. Currently, the results of SMT solvers are trusted, yet proof reconstruction is possible by using the Lean-SMT [23] library.

2.2 Combining Automated and Interactive Proofs

The VCs produced by Velvet are just Lean theorems, and one can inspect the remaining goals after running `loom_solve`. One of them is shown in Fig. 2. As is 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 + 1, j * j ≤ x`) is displayed right after it at the last line. Velvet automatically assigns names to hypotheses based on their source (**`invariant`**, **`if`**-branch, **`require`**, **`ensures`**). One can discharge this goal interactively, by first introducing the variable `j` and then performing case analysis on `j = i`. 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 means for automating proofs of the generated verification conditions. That is, if an SMT solver fails to discharge a certain VC, *e.g.*, due to it being outside its 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 [20]. Furthermore, one can also simplify an unsolved VC (as it is 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` [28]. 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.

```
x : ℕ
if_neg : ¬x = 0
i : ℕ
invariant_1 : ∀ j < i, j * j ≤ x
if_neg_1 : ¬i * i ≤ x
⊢ ∀ j < i + 1, j * j ≤ x
```

Fig. 2: A proof goal in Velvet

2.3 Property-Based Testing against Specifications

In addition to proof automation, Velvet provides facilities for randomised testing of programs against their specifications. This approach, known as *property-based testing* (PBT) [5], has proven to be effective in practice for exposing issues either in the program code or in the specification [11].

The first step required to perform PBT for a Velvet program is to extract executable code from its definition. Velvet provides a command that automatically generates a pure function that takes the same arguments as its Velvet counterpart

and returns the same result. If the original program has no non-deterministic choices, the extraction guarantees that its determinised version runs exactly the same code. To enable PBT, the preconditions and postconditions must be shown to be decidable propositions, so they can be evaluated against generated inputs and resulting outputs. Velvet automates this step, and lets users supply a manual proof for non-trivial `Decidable` instances which are not inferred automatically. Velvet puts together these components 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 use the tester with any existing PBT library. In our experiments, we use Plausible [16].

2.4 Program Proofs and (Non-)Termination

In its default mode, Velvet allows one to verify *partial* (*a.k.a.* functional) correctness of programs: unlike Dafny, it is not required for Velvet programs to terminate to satisfy an ascribed specification. In fact, removing or commenting out line 11 in Fig. 1 will not prevent the proof at line 14 from succeeding: this way, the `while`-loop at lines 8–11 will never terminate, hence the program will trivially satisfy any postcondition, even `False`. Luckily, Velvet allows for different treatments of non-termination: as a success or as a failure, thus, enabling *total* correctness reasoning. Switching to the latter mode is as easy as providing a single-line option. The theoretical foundations needed to support both kinds of correctness reasoning, total and partial, are provided in Sec. 5 of the paper detailing the design of the Loom framework, which Velvet builds upon [10].

In the case of total correctness, the user is expected to provide explicit termination measures by supplying the respective `decreasing`-clauses to each `while`-loop, as highlighted by the grey box at line 10 of Fig. 1. With these annotations provided and the code at line 11 commented out, the VC generator will produce, amongst others, a Lean goal which cannot be proven automatically (or even interactively), but provides a useful insight into why the proof has failed. In particular, it indicates that the termination measure `x + 1 - i` provided at line 10 does not, in fact, decrease with each iteration of the outer `while`-loop.

2.5 Reasoning about Non-Determinism

Velvet supports Dafny-style *let-such-that* operator `:|` (*a.k.a.* Hilbert's epsilon operator) [19], which is useful for modelling non-deterministic choice. Fig. 3 features a program which takes a predicate `s` and returns an array with all the values that satisfy `s`. The postcondition states that any element in the result should satisfy the initial predicate and any value satisfying the predicate should be present in the result. The code itself is straightforward: to produce such an array, the program non-deterministically takes a value

```
1  method Predicate.toArray (mut s: α → Bool)
2    return (res: Array α)
3  ensures ∀ x, sOld x ↔ x ∈ res do
4    let mut res := #[]
5    while ∃ x, s x
6    invariant ∀ x, sOld x ↔ x ∈ res ∨ s x do
7      let x :| s x
8      res := res.push x
9      s := fun y => if y = x then false else s y
10   return res
```

Fig. 3: A program with a choice operator

that satisfies `s` (line 7), adds it to the answer (line 8) and excludes it from the predicate (line 9), repeating this process until there are none left (the condition on line 5). The operator `:|` is used to model such choice inside the loop.

For the program in Fig. 3, the choice of concrete values via `:|` does not affect the program's outcome or correctness, yet, it is easy to imagine a scenario in which the choice of a non-deterministic value leads to an undesired outcome, which the user might be interested to identify. To support these two kinds of reasoning, traditional-style correctness (*a.k.a.* over-approximation) and the one in the style of Incorrectness Logic (*a.k.a.* under-approximation) [26, 34], Velvet provides two different ways to treat non-determinism symbolically: as *Angelic* or *Demonic* [3]. Angelic semantics results in existentially quantifying over the outcome of non-determinism in the VCs necessary to establish the postcondition, while Demonic semantics uses universal quantification, requiring one to prove that the postcondition holds for *any* outcome of the choice operator.

2.6 Using Mathematical Libraries in Program Specifications

Velvet comes with an ability to combine program verification with reasoning about pure Lean data types and mathematical theories from the mathlib library [22]. Let us illustrate this feature with a program in Fig. 4 that computes an approximation of $\int_0^1 x^2 dx$ with a left Riemann sum and a partition into n equal segments. The `while` loop calculates the left Riemann sum by adding up values at the left endpoints of partition iteratively. Velvet automatically verifies the program in Fig. 4 after adding hints to the `grind` tactic about `Finset.range`. It remains to show that the computed sum indeed approximates the value of the integral, which is done by proving that $\lim_{n\to\infty} \sum_{j=0}^{n-1} j^2/n^3 = \frac{1}{3}$. The proof of this fact is done interactively and does not involve any reasoning about code. In principle, such proofs can be delegated to AI systems optimised for mathematical reasoning in Lean.

```
noncomputable method oneThird (n: Nat) return (res: ℝ)
require n > 0
ensures res = ∑ j ∈ range n, (j^2/n^3 : ℝ) do
  let mut res : ℝ := 0
  let mut i := 0
  while i < n
  invariant i ≤ n
  invariant res = ∑ j ∈ range i, (j^2/n^3 : ℝ)
  decreasing n - i do
    let x : ℝ := i / n
    res := res + (x * x) / n
    i := i + 1
  return res
```

Fig. 4: Approximating $\int_0^1 x^2 dx = \frac{1}{3}$ in Velvet

3 Elements of Velvet Implementation

3.1 Semantic Foundations

Velvet is built as an instance of Loom [10]—a general Lean library for implementing provably correct multi-modal verifiers on top of Lean by means of a monadic *shallow* embedding (*i.e.*, Velvet programs are Lean *programs*, not data instances). Although Velvet's syntax resembles regular imperative code, under the hood, its programs are encoded as monadic computations. For example, after a series of macro-expansions, the code from Fig. 3 is translated into its monadic representation shown in Fig. 5. This is a computation in the `VelvetM` monad, which supports effects such as non-deterministic choice and divergence.

```
def Predicate.toArray {α} s :
  VelvetM ((Array α) × (α → Bool)) := do
  let sOld := s;
  let res := #[];
  VelvetM.repeat (res, s) fun (res, s) => do
    invGadget ∀ x, sOld x ↔ x ∈ res ∨ s x
    if ∃ x, s x = true then do
      let x <- pickSuchThat α fun x => s x = true
      let s := fun y => if y = x then false else s y
      pure (ForInStep.yield (res.push x, s))
    else pure (ForInStep.done (res, s))
```

Fig. 5: Monadic form of the code from Fig. 3

Mutable variables in Velvet are supported by a specialised macro-expansion that "threads" values associated with them through the monadic computations. For instance, to mimic the mutation of variable `s` in Fig. 3, Velvet adds an additional value $\alpha \to$ `Bool` to the return type of the computation. This value is then threaded (together with the mutated result array `res`) throughout the **while** loop. The loop itself is modelled by the monadic `repeat` function, which iterates the loop body, passing through the intermediate values of affected variables (in this case, `res` and `s`):

```
repeat (init : β) (body : β → VelvetM (ForInStep β)) : VelvetM β
```

As per `repeat`'s signature, at each iteration, the `body` function produces a `ForInStep` value, which can be either `ForInStep.yield x` to continue the loop with a new value `x`, or `ForInStep.done x` to terminate the loop with a final value `x`. Here, if the loop condition is satisfied, we resume the loop, modifying the result array `res` with a non-deterministically chosen `x` (`:|` is represented with `pickSuchThat`); otherwise, the loop terminates. Finally, in addition to the computational payload, the code in Fig. 5—like that in Fig. 3—features multiple annotations useful for VC generation procedure, which we discuss next.

3.2 Verification Conditions Generation

A verification condition generator for Velvet is obtained "for free", thanks to its underlying Loom framework, which automatically derives VC generators for computations represented by monads expressing common semantic effects. In particular, `VelvetM` is defined as `NonDetT DivM`, where `NonDetT` is a monad transformer for non-determinism and `DivM` is a monad for divergence. This combination provides exactly the set of operations we need: (a) `NonDetT` supports non-deterministic choice (such as `pickSuchThat` from Fig. 5), and (b) `DivM` supports divergence (such as the potentially non-terminating `repeat` loop from Fig. 5). Both `NonDetT` and `DivM` are supported by Loom out of the box, which means that Loom automatically derives a VC generator for `VelvetM` computations. In particular, it derives the weakest precondition transformer `wp` for `VelvetM` of type `wp : VelvetM β → (β → Prop) → Prop`. The `loom_solve` tactic then uses this transformer to convert a Velvet program into a formula and split it into separate goals—one for each verification condition.

3.3 Annotations for Intrinsic Program Verification

In the examples above, we have seen various program 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. Velvet's VC generator collects the conjunction of all **`require`** and **`ensures`** clauses into the program's Hoare-style correctness statement. Likewise,

loop **invariant** annotations are compiled into additional hypotheses for the respective VCs, thanks to the no-op combinator `invGadget` (*cf.* Fig. 5):

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

3.4 Non-Deterministic Choice Operator

The general form of Velvet's non-deterministic choice operator is:

```
def pickSuchThat {τ} (p : τ → Prop) [Findable p]: VelvetM τ := ...
```

This operator chooses an arbitrary value of type τ that satisfies the predicate `p`. Following Dafny-style notation, in program code, it is written as `let x :| p x`. To ensure that we can execute code featuring it, we require the predicate `p` to come with an instance of the `Findable` type class, which provides a value `find : τ` and a proof that, if the predicate is satisfiable, then `find`'s result satisfies `p` (otherwise, the value can be anything). In practice, constructing a `Findable` instance is rarely a problem. For example, if `p` is a decidable predicate and τ is countable and non-empty, such instance will be inferred automatically.

The weakest precondition transformer for this operator depends on the verification style chosen by the user and can be configured accordingly. For example, if the user wants to prove that the specification holds for *all* possible choices of `pickSuchThat`, the weakest precondition should ensure that the predicate is satisfiable and that the postcondition holds for all values satisfying `p`:

$$wp\ (pickSuchThat\ p)\ post = (\forall x \in p, post\ x) \wedge (\exists x, x \in p) \tag{1}$$

The last conjunct in (1) ensures that the value returned by `find` always satisfies the predicate `p`, and hence, that the execution of the Velvet program is guaranteed to satisfy the specification. Alternatively, the user might be interested in proving a reachability property of the program, *i.e.*, proving that the specification holds for *some* choice of `pickSuchThat`. In this case, the weakest precondition should assert that there *exists* a value satisfying both the predicate and the postcondition. Velvet supports both styles of VCs via an option flag.

4 Evaluation

Velvet is available online.[3] We have evaluated it *wrt* these research questions:

RQ1: How does Velvet's effectiveness and performance as an automated program verifier compare to that of Dafny?

RQ2: How much manual proof effort is required to complete Velvet proofs when its automation fails?

RQ3: What are characteristic examples where Velvet's capabilities as an interactive prover offer a better verification user experience than Dafny?

All the experiments described below were run on a 2026 MacBook Pro with 64GB of RAM and an Apple M4 chip, with cvc5 version 1.3.1, Z3 version 4.15.4, Lean version 4.24.0, and lean-auto revision `f62266d`.

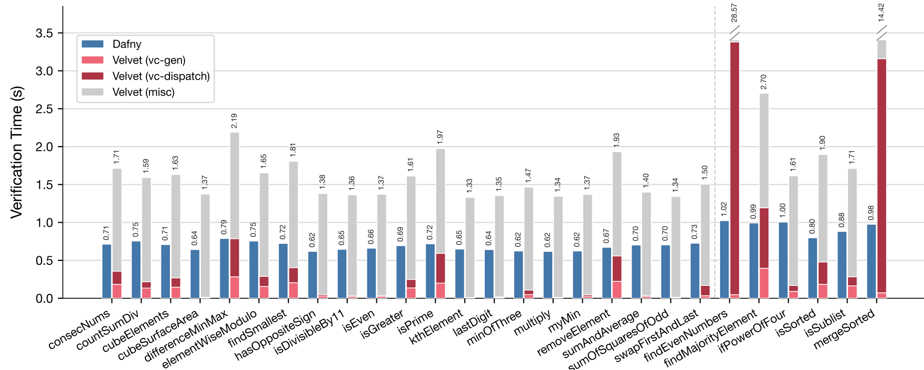

Fig. 6: Comparison of verification times between Dafny and Velvet, with Velvet's time broken down into VC generation (`vc-gen`), VC solving (`vc-dispatch`), and remaining overhead (`misc`).

RQ1: Automated Program Verification. To test RQ1, we manually selected 27 medium-complexity case studies from the VERINA benchmark suite [33]: each benchmark is a single program containing at least one loop requiring non-trivial invariant annotations. The case studies total 618 non-empty, non-comment lines of Lean code (average 23 lines per program), of which 303 lines are specifications, invariant annotations, and proofs. Each benchmark has an equivalent Dafny implementation. We aimed to automate verification as much as possible, inserting similar manual proofs for both Velvet and Dafny when necessary. For all programs except one, we used Lean's `grind` tactic to discharge the VCs. One program, namely `isPrime`, required an additional call to the cvc5 SMT solver (via lean-auto bindings) with a timeout of 1 second, as it required proofs about non-linear arithmetic. All reported times are averaged over 10 runs.

Our results are shown in Fig. 6. The case studies fall into two groups: 21 that were automatically discharged by Velvet, and 6 that required some manual proof effort (split by a dotted line). Notably, Velvet and Dafny required manual proofs for the same case studies, suggesting comparable effectiveness. In terms of performance, Velvet is on average 3.67x slower than Dafny, yet for all but 2 case studies it discharges the verification conditions in under 3 seconds, and in less than 45 seconds for all of them. The breakdown in Fig. 6 further shows that, for most benchmarks, Velvet's runtime is dominated by a roughly constant `misc` overhead (Lean elaboration, file start-up time and kernel checking), while VC generation and dispatch each take well under a second; the two outliers, `findMajorityElement` and `mergeSorted`, are instead dominated by VC dispatch, reflecting harder proof obligations rather than higher infrastructure cost. We conclude that Velvet can verify a variety of programs with the same level of user effort as Dafny while maintaining acceptable performance.

[3] https://github.com/verse-lab/velvet

RQ2: Combining Automated and Interactive Proofs. To answer RQ2, we selected six case studies from the VERINA benchmark suite that required manual proof effort in Dafny. We manually inserted additional assertions and proved the required lemmas in Dafny to help the SMT solver discharge VCs. Notably, Velvet was unable to discharge the same set of VCs automatically and required similar additional lemmas.

Unlike Dafny, when VCs cannot be discharged automatically, Velvet shows the proof goal context (as mentioned in Sec. 2.2), allowing users to inspect available facts and guide the proof process. Furthermore, Velvet leverages datatypes from the rich Lean standard library, which contains an extensive set of lemmas useful to discharge residual VCs without additional manual effortthis is why Velvet's proofs of the required lemmas were shorter than Dafny's. Fig. 7 breaks down the manual effort for these six benchmarks into two categories: *annotations*, which comprise function pre- and postconditions (`requires`/`ensures`) and loop invariants, and *proof*, which counts auxiliary definitions, lemmas, and `assert` statements interleaved with the program to guide the solver. Across all six benchmarks Velvet requires on average 3.3× fewer total lines than Dafny, and the gap is driven almost entirely by the proof component: Velvet's proofs are 4.9× shorter on average. Annotation counts, in contrast, are nearly identical between the two systems, indicating that the savings come from Lean's standard library and tactic ecosystem rather than from a less verbose specification language.

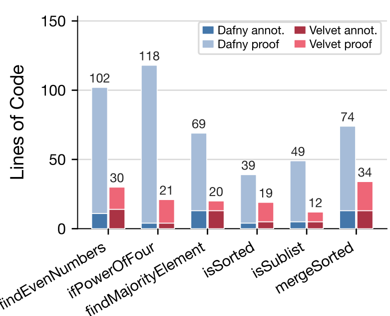

Fig. 7: Lines of code for the six case studies that required manual proof effort, split into specification and invariant annotations and proof lines, in Dafny vs. Velvet.

For example, one of the benchmarks is the `findMajorityElement` function, which computes the majority element of an array. Dafny's proof requires defining an element-counting function `count` on lists and proving that the count of any element in a concatenation equals the sum of counts in the individual lists. Velvet instead allows one to use `List.count` from the standard library, for which this lemma is already proved. Adding this and other such lemmas to the Velvet automation database allows Velvet to discharge the verification conditions fully automatically. From our results, we conclude that Velvet provides a better user experience than Dafny for programs with complex mathematical specifications that cannot be discharged automatically, demonstrating its versatility.

RQ3: Increased Expressivity beyond Auto-Active Verifiers. For RQ3, we considered three case studies:

- `memAlloc` implements a classic first-fit memory allocator with a free list;
- `isNonPrime` efficiently checks if a number is prime;
- `oneThird` computes an approximation of $\int_0^1 x^2 dx$ with a left Riemann sum.

Each of these case studies requires substantial manual proof effort. We manually verified those case studies in Velvet and report our results below.

Memory allocation. This procedure implements a classic first-fit memory allocator that searches a linked list of free memory blocks in a memory graph to find one that is large enough to satisfy an allocation request. In this case, the memory graph is encoded functionally via the function `next : addr → addr` returning the next address and the function `block_size : addr → Nat` mapping addresses to their sizes. Verification of this program in Velvet requires developing a custom theory reasoning about paths in the graph amounting to 15 lemmas (250 lines of interactive proofs in total) such as `path_append` (about path concatenation), `path_split_at_p` (about path splitting at a given position) *etc.* As Velvet is built on top of Lean, all such lemmas can be proven using Lean's native interactive proof mode, which gives access to the proof context at each step of the proof, as well as a rich set of automation tactics like `aesop` [20], `simp`, and `grind` [17]. Then, these lemmas can be seamlessly used to discharge the remaining VCs in Velvet. This example illustrates that Velvet can be used to verify programs where proofs require developing custom theory libraries.

Prime number checking. The prime number checking procedure implements an efficient algorithm to determine whether a number is prime by verifying that it is not divisible by any number up to its square root. Verifying this program requires extensive reasoning about multiplication and division of integers, which is known to be hard for SMT solvers and is therefore difficult in SMT-based program verifiers such as Dafny. In Velvet, however, the proof amounts to just 9 LOC, as we can simply use the `Nat.prime_def_le_sqrt` lemma from the mathlib library.[4] This example illustrates that, by leveraging the rich library ecosystem of Lean, even programs whose correctness relies on complex mathematical reasoning can be verified with minimal manual effort in Velvet.

Integral approximation. This procedure was shown in Sec. 2.6. It computes an approximation of $\int_0^1 x^2 dx$ using a left Riemann sum. Specifying and verifying `oneThird` requires working with complex mathematical objects such as limits, summations, and integrals. Many such objects cannot be encoded from the first principles in SMT-based program verifiers such as Dafny. Thus, even providing a specification for such programs goes beyond the reach of state-of-the-art automated verifiers. As a Lean library, Velvet can accommodate arbitrarily complex mathematical specifications while still providing Dafny-level automation for program-relevant verification conditions. This gives the user the opportunity to focus on the problem-specific mathematical parts of the proof.

5 Discussion

Expressivity. Velvet is a shallow verifier on top of Lean: programs are encoded directly using Lean's own datatypes and language features. As a result, Velvet natively supports reasoning about Lean's functional features, including local mu-

[4] This lemma states that a number is prime if and only if it is greater than one and not divisible by any number from 2 up to its square root.

table state, type classes, and pattern matching. This is in contrast to auto-active verifiers like Dafny and Why3 [8], which model imperative programs in an OOP-flavoured setting built around a global heap and reference-based classes. The flip side of Velvet's design is that it does not currently support global heap state or OOP-style classes; programs that essentially rely on shared mutable references (*e.g.*, doubly-linked lists with aliasing) are outside its present scope.

Early adoption. Even though the first version of Velvet has been released only in October 2025, it has already been used in several verification projects outside the core development team. In November 2025, Velvet was used to win the first place in one of the categories of the regional VeHa verification competition:[5] a one-person team implemented and verified in Velvet an efficient procedure for searching, in an arbitrary string, the longest substring consisting of digits only. A verification problem stated in Velvet (`memAlloc` from RQ3 of Sec. 4) was offered as one of the tasks in the 2025 CCF ChinaSoft Theorem Proving Contest.[6]

6 Related and Future Work

Conceptually, the approach of Velvet to multi-modal program verification is similar to Why3 [8], which provides WhyML—a programming language with support for verification, and which uses SMT solvers and interactive provers (*e.g.*, Rocq) to discharge VCs. Unlike Velvet, Why3 is not foundational: while its logic fragment has been mechanised in Rocq [6], the mechanisation does not yet cover the semantics of WhyML and the VC generation. Both Dafny [18] and F* [31], while positioned as auto-active verifiers, offer experimental support for tactic-based interactive proofs [4, 21], but those proof modes are not easily extensible with user-defined tactics and do not provide access to math libraries, such as mathlib. Daenerys [30] is a recent attempt to supplement interactive proofs in the Rocq-based foundational program logic Iris [13, 14] with SMT-enabled automation. Unlike Velvet, Daenerys does not generate SMT-friendly VCs automatically and requires manually translating them from Rocq to SMT-LIB. Lean's own VC generator for monadic computations of the Std.Do library provides facilities for reasoning about imperative computations, but does not support programs with non-terminating loops or non-determinism.

Veil [27] is another multi-modal verifier built on top of Loom [10]. Unlike Velvet, Veil does not offer a general-purpose imperative language, as it targets verification of distributed protocols expressed in a high-level DSL that is optimised for emitting SMT-friendly VCs in first-order logic. In the future, we plan to tackle multi-layer verification efforts of large-scale systems in the style of IronFleet [12, 15] or Verdi [32] by making use of both Velvet and Veil in tandem.

7 Conclusion

We have presented Velvet, a foundational multi-modal verifier for imperative programs built on Loom, integrating SMT-style automation, interactive proofs, and property-based testing in a single Lean environment with seamless access

[5] https://sites.google.com/view/veha2025/ (in Russian)
[6] https://chinasoft.ccf.org.cn/#competition/theorem-proving (in Chinese)

to mathlib. Our evaluation on VERINA and three case studies shows this design matches state-of-the-art auto-active verifiers, while enabling proofs whose specifications rely on mathematical theories beyond the reach of SMT solvers. Velvet is a step toward verification environments in which push-button automation and interactive foundational reasoning coexist within a single unified workflow.

Acknowledgements. We thank the CAV'26 reviewers for their insightful comments. This work has been partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001 and by Stellar Development Foundation Academic Research Grant.

Data Availability Statement. The software artefact accompanying this paper is available online [9]. The artefact contains the source code and build scripts for the Lean-embedded implementation of Velvet, along with a collection of programs verified in it.

Disclosure of Interests. The authors have no further competing interests to declare that are relevant to the content of this article.

References

1. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: TACAS. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
2. Barrett, C., Fontaine, P., Tinelli, C.: SMT-LIB: The Satisfiability Modulo Theories Library. Available at `https://smt-lib.org/` (2025)
3. Broy, M., Wirsing, M.: On the algebraic specification of nondeterministic programming languages. In: CAAP. LNCS, vol. 112, pp. 162–179. Springer (1981). https://doi.org/10.1007/3-540-10828-9_61
4. Ciobâc, S., Leino, K.R.M., Merca, S., Timon, R.M.: The Design of an Interactive Proof Mode for Dafny. In: Proceedings of the Third Workshop on Dafny (2026)
5. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP. pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266
6. Cohen, J.M., Johnson-Freyd, P.: A Formalization of Core Why3 in Coq. Proc. ACM Program. Lang. **8**(POPL), 1789–1818 (2024). https://doi.org/10.1145/3632902
7. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Filliâtre, J., Paskevich, A.: Why3 - Where Programs Meet Provers. In: ESOP. LNCS, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
9. Gladshtein, V., Kurin, V., Feng, Y., Kafle, D., Pîrlea, G., Zhao, Q., Sergey, I.: Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean. Software Artefact. (Apr 2026). https://doi.org/10.5281/zenodo.19801401, v1
10. Gladshtein, V., Pîrlea, G., Zhao, Q., Kurin, V., Sergey, I.: Foundational Multi-Modal Program Verifiers. Proc. ACM Program. Lang. **10**(POPL) (2026). https://doi.org/10.1145/3776719

11. Goldstein, H., Cutler, J.W., Dickstein, D., Pierce, B.C., Head, A.: Property-based testing in practice. In: ICSE. pp. 187:1–187:13. ACM (2024). https://doi.org/10.1145/3597503.3639581
12. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: SOSP. pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428
13. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
14. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: POPL. pp. 205–217. ACM (2017). https://doi.org/10.1145/3009837.3009855
15. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A Practical Foundation for Systems Verification. In: SOSP. pp. 438–454. ACM (2024). https://doi.org/10.1145/3694715.3695952
16. leanprover-community: Plausible: A property testing framework for Lean 4. `https://github.com/leanprover-community/plausible` (2025), accessed on 28 Jan 2026
17. leanprover-community: Lean language reference: The grind tactic. `https://lean-lang.org/doc/reference/latest/The--grind--tactic/` (2026), last accessed on 26 January 2026
18. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: LPAR. LNCS, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
19. Leino, K.R.M.: Compiling Hilbert's epsilon operator. In: LPAR. EPiC Series in Computing, vol. 35, pp. 106–118. EasyChair (2015). https://doi.org/10.29007/RKXM
20. Limperg, J., From, A.H.: Aesop: White-Box Best-First Proof Search for Lean. In: CPP. pp. 253–266. ACM (2023). https://doi.org/10.1145/3573105.3575671
21. Martínez, G., Ahman, D., Dumitrescu, V., Giannarakis, N., Hawblitzel, C., Hritcu, C., Narasimhamurthy, M., Paraskevopoulou, Z., Pit-Claudel, C., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N.: Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In: ESOP. LNCS, vol. 11423, pp. 30–59. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_2
22. mathlib Community, T.: The Lean mathematical library. In: CPP. pp. 367–381. ACM (2020). https://doi.org/10.1145/3372885.3373824, `https://github.com/leanprover-community/mathlib4`
23. Mohamed, A., Mascarenhas, T., Khan, M.H.A., Barbosa, H., Reynolds, A., Qian, Y., Tinelli, C., Barrett, C.W.: lean-smt: An SMT Tactic for Discharging Proof Goals in Lean. In: CAV. LNCS, vol. 15933, pp. 197–212. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_11
24. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: CADE. LNCS, vol. 9195, pp. 378–388. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_26
25. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: VMCAI. LNCS, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
26. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL), 10:1–10:32 (2020). https://doi.org/10.1145/3371078

27. Pîrlea, G., Gladshtein, V., Kinsbruner, E., Zhao, Q., Sergey, I.: Veil: A Framework for Automated and Interactive Verification of Transition Systems. In: CAV. LNCS, vol. 15933, pp. 26–41. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_2
28. Qian, Y., Clune, J., Barrett, C.W., Avigad, J.: Lean-Auto: An Interface Between Lean 4 and Automated Theorem Provers. In: CAV. LNCS, vol. 15933, pp. 175–196. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_10
29. Rocq Development Team: The rocq prover. https://rocq-prover.org (2025), version 9.0.0, released March 12, 2025
30. Spies, S., Mück, N., Zeng, H., Sammler, M., Lattuada, A., Müller, P., Dreyer, D.: Destabilizing Iris. Proc. ACM Program. Lang. **9**(PLDI) (2025). https://doi.org/10.1145/3729284, https://doi.org/10.1145/3729284
31. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: POPL. pp. 256–270. ACM (2016). https://doi.org/10.1145/2837614.2837655
32. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI. pp. 357–368. ACM (2015). https://doi.org/10.1145/2737924.2737958
33. Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., Song, D.: VERINA: Benchmarking Verifiable Code Generation (2025), https://arxiv.org/abs/2505.23135
34. Zilberstein, N., Dreyer, D., Silva, A.: Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. Proc. ACM Program. Lang. **7**(OOPSLA1), 522–550 (2023). https://doi.org/10.1145/3586045