Lessons from Building an Auto-Active Verifier in Lean

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

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

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

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

Abstract

For the past 18 months, we have been continuously developing Veil, an open-source framework for automated and interactive verification of transition systems, entirely embedded in the Lean proof assistant. Unlike traditional verifiers, which are implemented as standalone tools and (optionally) use proof assistants as backend certificate checkers, Veil is a Lean library, with all its functionality entirely embedded in the proof assistant. We believe this approach has significant advantages over traditional verifiers and that, in the near future, more verifiers will be built in this way. Nonetheless, Lean's extraordinary meta-programming facilities—like all powerful tools—can prove to be both a blessing and a curse, and it is not at all difficult to create an unmaintainable mess. In this talk, we describe our vision for building verifiers in Lean, and share our experience of writing (and rewriting) Veil, and the lessons we have learned along the way.

1 Introduction

Auto-active verifiers like Dafny [14], F⋆ [27], and Viper [18] have demonstrated that automated verification scales to realistic software systems. These tools strike a pragmatic balance: they provide substantial automation through SMT solvers, while allowing users to guide the proof search process when automation fails by providing additional assertions. However, existing auto-active verifiers face three persistent challenges. First, their trusted computing bases (TCBs) are large, encompassing, at the very least, the language semantics axiomatisation, the VC generator, and the SMT solver(s) used. Second, when the automation fails, the user experience degrades significantly—often, the user has to *guess* which additional assertions might guide the SMT solver's proof search process along productive paths, without having any visibility into the the solver's internal state. Third, because the language of assertions is limited to first-order logic, certain properties of interest (*e.g.*, relational properties like refinement) are not directly expressible in the logic and must instead be verified on artificially constructed programs (*e.g.*, product programs).

Dafny'26, Rennes, France
2026. ACM ISBN 978-1-4503-XXXX-X/2018/06

The Lean embedding thesis. We claim that by building the *entire* verifier inside a proof assistant such as Lean, all of these challenges are addressed. The key is to use Lean not just as a backend certificate checker, as done in prior work [26], but to have Lean "own" the entire user experience. Thanks to its powerful meta-programming facilities [22], Lean is a *customisable* proof assistant. As such, it can be used to build an entire verifier, which piggy-backs on top of Lean's existing functionality for parsing, its rich logic and proof automation, as well as, importantly, its IDE integration for syntax highlighting, highlighting of failed assertions, displaying counter-examples via the Lean infoview, etc. With this framing, auto-active verifiers become simply domain-specific extensions of general-purpose proof assistants like Lean. The philosophy is simple:

> *All goals are Lean goals.*
> *All proofs are Lean proofs.*
> *All UI is Lean UI.*

The benefits of this approach are manifold:

1. **No boilerplate:** there is no need to implement a custom parser, LSP integration, IDE extensions, *etc.* Lean provides all these amenities out of the box.
2. **Powerful automation ecosystem:** Lean provides a rich ecosystem of proof automation tools, including tactics like native `simp` and `omega`, customisable proof search tactics such as `aesop` [15], SMT integration (with proof reconstruction) via Lean-Auto [24] and Lean-SMT [17], and the emerging and increasingly powerful `grind` tactic.
3. **Seamless fallback to interactive proofs:** if all VCs are Lean proof goals, users can fall back to interactive proofs using Lean tactics (including domain-specific tactics) whenever SMT-based proof automation fails.
4. **Expressive logic:** one can use Lean's rich logic to *directly* encode complex properties, including refinement, or properties involving higher-order quantification. Such properties can then be *reduced* to first-order formulas via tactics, to be automatically solved via SMT.

5. **Reduced TCB:** it is possible and often easy to *prove* the soundness of different components of the verifier, reducing the TCB to that of Lean itself.

2 Unified Multi-Modal Verification in Lean

Veil is a framework for multi-modal verification of transition systems, embedded in the Lean proof assistant [23]. We developed it out of frustration with existing tools used for modelling and verifying distributed protocols. In particular, we were unsatisfied with both TLA$^+$ [12] and Ivy [16, 20], the two most popular tools in this space. Both of these tools are great *in some respects*: TLA$^+$ is amazing for modelling protocols and quickly testing them via concrete state model checking using TLC [30]. In principle, one *can* also symbolically model check TLA$^+$ specifications using Apalache [10, 19] and semi-automatically prove properties using TLAPS [4], but the tooling for these tasks is less user-friendly and *outside the default path*, and thus, in practice, many people treat TLA$^+$ as a fancy DSL for breadth-first search [9]. Ivy is extraordinarily powerful for proving decidable properties of distributed protocols using SMT. It also supports symbolic model checking and even tactic-based proofs [16], but again, these feel like bolt-ons rather than core features of the tool.

Zawinski's Law humorously states that "every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can."[1] Unfortuantely, something similar is happening with program verifiers. We half-jokingly postulate the following law:

> *Every verifier expands until it contains an ad hoc, bug-ridden, unusable implementation of half of an interactive theorem prover. Those verifiers which cannot so expand are replaced by ones which can.*

There is a good reason for this: testing and automated verification is what people want. The more you can automate, the better. But you *cannot* automate everything, and eventually, you will need to resort to interactive proofs.

2.1 A Unified Approach

Veil was built to avoid this trap. We still prioritise automation, like existing verifiers, but by embedding our tool entirely in Lean, we also provide a seamless fallback to a state-of-the-art interactive theorem prover when required. Put another way, Veil can be thought of as a domain-specific extension of Lean. It provides the best of *all* worlds. The user is encouraged, but not required, to model their protocol in an Ivy-style domain-specific language which ensures verification conditions fall in a decidable fragment of first-order logic. This is the *all-inclusive path*, where Veil offers push-button verification and symbolic model checking, both powered by SMT. But users who eschew the all-inclusive path because their specification is not easily expressible in decidable FOL are not left in the lurch. They still get TLC-style concrete state model checking to test their specifications, as well as the full power of Lean to interactively prove properties when required. Moreover, Veil's architecture is such that, as proof automation (*e.g.*, the grind tactic) becomes more powerful, users benefit without having to change their workflow: Veil transparently invokes all the automation tools at its disposal for every goal.

Fig. 1 gives a taste of Veil's capabilities. The protocol model along with its safety specification are defined in Fig. 1a, in a DSL heavily inspired by Ivy's Relational Modelling Language (RML) [20]. Within the same Lean file, users can invoke bounded model checking to test that the protocol admits non-trivial executions (lines 33–36) and does not trivially violate the safety specification (lines 39–42). Moreover, since the specification comes with invariant annotations, users can invoke Veil's push-button verification to automatically verify that the protocol preserves the invariants (line 48). The `#check_invariants` command checks that every invariant clause is preserved by every transition—in this case, by invoking an external SMT solver. The results are displayed directly in the editor InfoView. If an invariant is not preserved, a concrete counter-example to induction (CTI) is displayed, allowing the user to refine their specification with an additional invariant clause that eliminates the CTI, repeating the process until an inductive invariant is discovered. Moreover, external SMT solvers do not have to be trusted: Veil supports proof reconstruction (at a 3-5x performance penalty) using Lean-SMT [17] (line 14). Finally, if the user issues the `#check_invariants?` command (not shown), Veil prints a theorem template like the one on lines 51–56, which the user can prove manually using arbitrary Lean tactics, as shown on lines 58–62. (This is one of the theorems that was proven automatically at line 48.) All this occurs within a unified tool, fully integrated with the Lean ecosystem, with a seamless transition between different verification styles.

2.2 A Principled Approach

Besides the user experience benefits, embedding Veil in Lean also allows us to greatly reduce the trusted computing base of our tool. The different verification styles offered by Veil are supported by a unified semantic framework, fully formalised and proven sound in Lean. Importantly, the mechanised meta-theory serves as the *actual implementation* of our verification condition generators. Practically, this means that users of Veil only have to trust the Veil frontend (*i.e.*, syntax elaboration), the Lean compiler (which is used to execute the frontend code), and the Lean kernel (which is used to check the proofs). Moreover, for the more paranoid, Veil allows transparent desugaring to the underlying Lean definitions and theorem statements, which can be manually inspected for correctness—thus reducing the TCB to merely the Lean kernel. This is conceptually similar to *translation validation* modes of existing verifiers like Viper [5, 21], but with a much

[1] Greenspun's Tenth Rule is another take on a similar phenomenon.

```
type node
instantiate tot : TotalOrder node -- FOL theory of total order
instantiate btwn : Between node -- FOL theory of ring

relation leader (n : node) -- protocol state as FOL structure
relation pending (id : node) (dest : node)

after_init {
 leader N := False
 pending M N := False
}

action send (n next : node) {
 require ∀ Z, n ≠ next ∧ ((Z ≠ n ∧ Z ≠ next) → btw n next Z)
 pending n next := True
}

action recv (id n next : node) {
 require ∀ Z, n ≠ next ∧ ((Z ≠ n ∧ Z ≠ next) → btw n next Z)
 require pending id n
 if (id = n) then
  leader n := True
 else
  if (le n id) then
   pending id next := True
}

safety [single_leader] leader L1 ∧ leader L2 → L1 = L2
invariant [leader_greatest] leader L → le N L
invariant pending S D ∧ btw S N D → le N S
invariant pending L L → le N L
```

(a) Protocol model and specification

```
-- bounded model checking: checking non-vacuousness
sat trace [can_elect_leader] {
 any 4 actions
 assert (∃ l, leader l)
} by { bmc_sat }

-- checking bounded safety
unsat trace [bounded_safety] {
 any 4 actions
 assert (¬ single_leader)
} by { bmc }

-- proof reconstruction removes SMT solvers from the TCB
set_option veil.smt.reconstructProofs true

-- push-button safety verification
#check_invariants -- prints ✅ for every preserved invariant clause

-- interactive proof of safety property
theorem send_tr_single_leader' :
 ∀ (n : node) (next : node),
  TwoState.meetsSpecificationIfSuccessful
   (Ring.send.ext.twoState n next)
   (fun th st => (Ring.assumptions th) ∧ (Ring.inv th st))
   (fun th st' => Ring.single_leader th st') :=
 by
 intro n next th st st' ⟨has, hinv⟩ htr
 simp only [initSimp, actSimp, invSimp] at *
 rcases htr with ⟨hIsNext, _, htr⟩
 concretize_state; sdestruct_all; simp_all
 assumption
```

(b) Bounded checking, automated verification, and interactive proof

Fig. 1. Leader election in a ring topology, implemented, tested, and verified in Veil.

tighter integration—Veil is designed to be proof-producing by default, rather than as a later design addition, and having Lean itself be the only "intermediate" representation.

3 Lessons and Reflections

While others have attempted to embed automated verifiers inside proof assistants (notably, in Isabelle/HOL [2]), we believe we are the first that sought to push this idea as far as possible. This is partly because, prior to Lean, meta-programming in ITPs was difficult and error-prone: everything was possible, but nothing was easy. Lean is in a completely different class of usability in this respect: writing complex meta-programs is now *relatively* easy, and the system scales to large meta-programs, as evidenced by the fact that most of Lean is written in Lean itself [6]. That said, we are constantly pushing the boundaries of what is possible with Lean's meta-programming facilities,[2] and we have had to learn a lot along the way. We are hoping that, in sharing some of our experience, we can help other verifier implementers avoid some of the mistakes we made at first, and anticipate some of the challenges we are now facing.

3.1 Good Software Engineering Practices

The most obvious lesson, which is often forgotten in research projects, is that good software engineering practices pay off big time in the long run. As one develops a prototype, it is normal to have less than ideal code—part of the process lies exactly in discovering what are the right interfaces and abstractions to use. In our case, we were also learning meta-programming in Lean in the process of developing the original version of Veil, and our learning was directly driven by our actual needs. This made for quick progress, but also meant that the original version of Veil was a bit of a mess. (More on this in the next section.) For the past few months, we have been rewriting Veil from scratch to address the tech debt we had accumulated, and make Veil 2.0 a stable foundation for further development, with a sound architecture and clean interfaces between components. Some lessons:

[2]When he first saw Veil, Leo de Moura could not believe it was built in Lean.

- Test early and often: verifiers are complex beasts, and both Lean and your project's dependencies will change at a very rapid pace; regression testing is crucial for building a reliable tool. In particular, meta-programs can still produce ill-typed terms and need to be tested.
- Modularise your implementation as early as possible: separation of concerns is paramount for maintainability, and especially important when it comes to meta-programming. Keep syntax and semantics separate, and use clean interfaces between components.
- Lean's module system is primitive: make sure all your definitions are namespaced to your project and do not use the global namespace to avoid conflicts with other libraries.

3.2 Metaprogramming is a Double-Edged Sword

Lean's meta-programming capabilities are extraordinarily powerful, but this power must be wielded carefully. The Veil 2.0 rewrite was largely motivated by the need to impose proper architecture on the "spaghetti meta-programs" of the original implementation. The problem, as we found out the hard way, is that *when everything is possible, there are no guard rails*. You can shoot yourself in the foot really badly, and in particular, you can very tightly couple functionality that really should be separate.

Separation of verification concerns. An example of multiple verification-related sub-routines tightly tangled together is the original implementation of the `#check_invariants` command: it both (1) generated the theorems to be checked, (2) ran the candidate proof scripts and identified whether they were successful, (3) added the successfully verified theorems to the environment, and (4) displayed the verification results, including counter-examples. This makes sense if you think of `#check_invariants` as an elaborator for a command—the verifier does in fact have to do all these things. The problem is that this is a lot of responsibility for a single elaborator to carry, and it is completely unmaintainable. Instead, as we discovered by experimentation, it is better to think of `#check_invariants` as a *viewer* of verification results. Separately, there must a *controller* to ensure the verification conditions are produced based on the user-provided specification and sent to the solvers. Finally, in order to generate the VCs, the verifier needs to have a *internal representation* of the user-provided specification.

This, as you likely have guessed, is the *model-view-controller* (MVC) architecture, and our experience tells us that it makes a lot of sense for verifiers embedded in Lean. The idea is simple: keep your own representations of objects of interest as Lean environment extensions. Then, some elaborators modify these representations (to record user-provided information), and others read them, either to trigger some back-end computation (*e.g.*, sending VCs to a solver), or to generate Lean declarations based on them, or to display information to the user. The key is to *keep your own representations*, rather than piggy-backing on the Lean environment directly.

Avoid being too shallow. Another mistake we made—and which we think others may fall into as well—was driven by the desire to have the Veil DSL extensively interoperate with Lean. For example, the original implementation of Veil reused Lean's section variable mechanism: user-defined `type` parameters and `instantiate` commands were elaborated as Lean section variables. Conversely, section variables affected definitions produced by Veil. This seemed elegant—users could employ familiar Lean syntax and the parameters would "just work", including in definitions not produced by Veil—but it was a disaster. As Veil's complexity increased, we needed intricate control over which definitions take which parameters. Lean's section variables do not easily provide this granular control, and attempting to work around the limitations led to brittle, unmaintainable code. The solution: we now implement our own module system with parameter tracking. The general lesson is that *you should control your representations*. It is fine to reuse Lean's execution semantics, but do not rely on its compile-time environment representations (*e.g.*, section variables). Rather, maintain your own representations as Lean environment extensions, and ensure all changes to these are done via elaborators *you* defined.

3.3 Spectrum from Shallow to Deep Embedding

Whilst, as described in Sec. 3.2, we avoid being shallow at the meta level (*i.e.*, we *do not* conflate definition-level constructs of Lean, such as `type` and `section`, with similar ones of Veil), we fully embrace shallowness at the object level. By this we mean that all definitions produced by Veil are regular Lean definitions, which *could have been* written by the user directly. In particular, Veil actions are normal Lean monadic computations, rather than a deeply-embedded representation. Concretely, Veil's action syntax is an extension of Lean's existing *do* notation [28], which supports local mutation and which we extend with Veil-specific constructs like `require`, `assert`, and `pick`. This has the massive benefit of Veil being completely compatible with the entire Lean ecosystem. It also gives us access to efficient execution semantics, which proves useful for implementing concrete state model checking by directly leveraging Lean's compiler.

One of the big open questions, in our view, is how far one can push this shallow embedding approach. For Veil, which is focused on modelling abstract transition systems, it seems to be a good fit. If one tries to model more realistic programming languages with the runtime semantics and memory model significantly different from Lean's, however, the shallow embedding approach may not be enough, and it is not clear to us at this stage how good the verification user experience would be for a deeply-embedded language. If you are interested, we would encourage you to experiment.

3.4 The Problem of Generality and Encodings

A problem that is just starting to become apparent in Veil 2.0 is that of generality. To support Veil's multiple verification modes—concrete execution, symbolic model checking, and deductive verification—we need a very general representation of protocol actions. In particular, in order to be *efficient*, concrete execution requires a different encoding of the state than symbolic model checking, which uses an encoding that is effective for SMT-based proofs. But we want our proofs to be agnostic to which encoding is used. Moreover, there is a need for modularity of specifications, *e.g.*, that actions defined in one specification can be reused in another (which might operate on a different state space). But, again, we want proofs of invariant preservation to be agnostic to which environment actions are run in. These desiderata require the *definition* of actions to be highly generic, in the style of datatype-generic programming [7]. We achieve this using Lean's type classes, but the issue is that, as more and more layers of generality get added to the framework, the definitions that users have to manipulate in order to prove theorems interactively become increasingly unwieldy. We offer tactics to help users "strip away" the generality in proofs, but the statements themselves remain more complex than we would want. This is due to the proof assistant keeping us honest (*i.e.*, we cannot "drop" certain proof obligations from the context), which we appreciate, but it would be great to find a way to hide the full complexity away from the user.

The message in this section is that there is no free lunch. If you want to build a powerful, general verification tool and want to keep it fully verified itself, you will need to pay the price *somewhere*. In the future, however, we hope to develop better "interfaces" to the full generality, which would only exist internally within the tool, rather than being exposed to the user in theorem statements.

3.5 Performance is a Huge Issue

Finally, performance has been our most persistent problem. When it comes to SMT-based proof automation, Veil is more than 10x slower than Ivy, the most closely related tool, although Veil is still *reasonable*, with verification of small specifications taking on the order of a few seconds, and our largest case study taking 160 seconds, compared to Ivy's 50 seconds. In effect, when the SMT solver is the bottleneck, we are not doing too badly, but when the goals are easy, the overhead imposed by our verified translation logic becomes prohibitive, especially, during the early phase of protocol prototyping. After all, verified translation of Lean's higher-order logic statements to SMT-LIB is (unsurprisingly) much slower than pretty-printing SMT-LIB queries, as done, *e.g.*, by Ivy.

We can summarise our experience as follows:

> If you plan to build an auto-active verifier in Lean, expect to spend *most* of your time on performance engineering.

We use Lean's simp tactic machinery extensively in Veil: for generating verification conditions, creating derived definitions (*e.g.*, transition relations given imperative actions), hoisting quantifiers, and reducing higher-order goals to first-order logic. The appeal is correctness by construction—every rewrite performed by simp is justfied by a proven theorem and checked by Lean, so translations are inherently sound. As a bonus, simplification-based translations are easy to implement and debug: you can inspect each rewrite step and add new simplification rules incrementally. The downside is that simplification has poor performance characteristics, and the generated proof terms are large and slow to check. This is not unique to Veil: the bottleneck in the bv_decide verified bitblasting tool in Lean [3] is the preprocessing step, which applies rewrite-based simplification; all other steps are roughly as performant as the state-of-the-art bitblaster.

Partial solutions to this problem exist (*e.g.*, not checking the produced proof terms), and the Lean FRO is planning to write a less general, but more efficient simplifier, but how to do verified translation *efficiently* is still an open question.

4 Conclusion

The lessons we extracted from implementing Veil in Lean suggest exciting directions for future research.

First, we believe, more research should be done one structuring and maintaining large formal developments, containing a large body of meta-programs. Some issues of this sort can be mitigated by developing a more powerful module system in Lean, similar to that of Rocq proof assistant. While the challenges of repairing programs [13] and mechanised formal proofs [8, 25] has been studied in the past, to some extent, less work has been done on doing so for meta-programs. Even documenting the good practices of structuring such projects [29] would be a significant contribution to the body of knowledge on developing verified software.

Second, as our experience with Lean meta-programming outlined in Sec. 3.2 demonstrates, further research is needed to identify robust design patterns for implementing domain-specific verifiers via meta-programming within an extensible proof assistant. Although the ITP community already features impressive examples of what can be achieved through extensible notations and user-level automation [11], very little has been written about how to keep such developments intellectually manageable and future-proof. In particular, the folklore knowledge on building domain-specific verifiers is mostly limited to implementing bespoke translations from the domain-specific language to one of the well-established verification IRs, such as Boogie [1] or Viper [18]. While this

approach takes care of emitting and discharging verification conditions, it does little to address the challenges of offering informative and domain-relevant feedback to the user of the verifier, or of extending the prover's functionality in principled ways—issues that are typically treated in an ad-hoc manner. We believe, identifying patterns similar to MVC, as we discussed above, will help to make construction of program verifiers less of a dark art.

Finally, we are looking forward for the verification community to reconcile multiple views on implementing language-specific verifiers by comparing their experiences of doing so and documenting the practices that did or did not work. For example, when explaining the internal workings of Veil to other researchers, we frequently had to justify one of our main design decisions: making it shallow at the object level (*cf.* Sec. 3.3) rather than following a more traditional approach of a "deep embedding", when both the syntax and the semantics of the domain-specific language are implemented as data types within the language of the meta-verifier, in our case, Lean.[3] This has prompted us to outline the advantages of our approach (*e.g.*, immediate compatibility with the rest of Lean ecosystem) but also acknowledge possible shortcomings of the shallow embedding methodology, which might make it unsuitable for implementing verifiers for languages that are too different from Lean due to either being untyped (*e.g.*, TLA$^+$) or because their memory model and a type system are too different from the one of Lean (*e.g.*, C).

References

[1] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS, Vol. 4111)*. Springer, 364–387. doi:10.1007/11804192_17

[2] Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. 2008. HOL-Boogie - An Interactive Prover for the Boogie Program-Verifier. In *TPHOLs (LNCS, Vol. 5170)*. Springer, 150–166. doi:10.1007/978-3-540-71067-7_15

[3] Henrik Böving, Siddharth Bhat, Luisa Cicolini, Alex Keizer, Léon Frenot, Abdalrhman Mohamed, Léo Stefanesco, Harun Khan, Joshua Clune, Clark Barrett, and Tobias Grosser. 2025. Interactive Bitvector Reasoning using Verified Bit-Blasting. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025). doi:10.1145/3763167

[4] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2008. A TLA+ Proof System. In *Proceedings of the LPAR 2008 Workshops (CEUR Workshop Proceedings, Vol. 418)*. CEUR-WS.org. https://ceur-ws.org/Vol-418/paper2.pdf

[5] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 2025. Formal Foundations for Translational Separation Logic Verifiers. *Proc. ACM Program. Lang.* 9, POPL (2025). doi:10.1145/3704856

[6] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.* 1, ICFP (2017). doi:10.1145/3110278

[7] Jeremy Gibbons. 2006. Datatype-Generic Programming. In *Datatype-Generic Programming - International Spring School (SSDGP) (Lecture Notes in Computer Science, Vol. 4719)*. Springer, 1–71. doi:10.1007/978-3-540-76786-2_1

[8] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. doi:10.1145/3591221

[9] Andrew Helwer. 2024. TLA$^+$ is more than a DSL for breadth-first search. https://ahelwer.ca/post/2024-09-18-tla-bfs-dsl/

[10] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 123:1–123:30. doi:10.1145/3360549

[11] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. ACM, 205–217. doi:10.1145/3009837.3009855

[12] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. http://research.microsoft.com/users/lamport/tla/book.html

[13] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. doi:10.1145/3318162

[14] 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

[15] 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

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

[17] 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

[18] 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

[19] Rodrigo Otoni, Igor Konnov, Jure Kukovec, Patrick Eugster, and Natasha Sharygina. 2023. Symbolic Model Checking for TLA+ Made Faster. In *TACAS*. Springer, 126–144. doi:10.1007/978-3-031-30823-9_7

[20] 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

[21] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 2024. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language. *Proc. ACM Program. Lang.* 8, PLDI (2024). doi:10.1145/3656438

[22] 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/.

[23] 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 Code available at https://github.com/verse-lab/veil.

[24] 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. doi:10.1007/978-3-031-98682-6_10

[25] Talia Ringer. 2021. *Proof Repair*. Ph. D. Dissertation. University of Washington, USA. https://hdl.handle.net/1773/47429

[3] The "deep embedding" approach is taken, for instance, by the currently actively developed Strata framework: https://github.com/strata-org/Strata.

[26] Wojciech Różowski, Georges-Axel Jaloyan, and Sean McLaughlin. 2024. Lean on Dafny: Exploring Interactive Verification of Dafny Programs in Lean. In *Second Workshop on Dafny*.

[27] 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

[28] Sebastian Ullrich and Leonardo de Moura. 2022. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang.* 6, ICFP (2022), 512–539. doi:10.1145/3547640

[29] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *CPP*. ACM, 154–165. doi:10.1145/2854065.2854081

[30] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA$^+$ Specifications. In *CHARME (LNCS, Vol. 1703)*. Springer, 54–66. doi:10.1007/3-540-48153-2_6