Veil: A Framework for Automated and Interactive Verification of Transition Systems

George Pîrlea[1], Vladimir Gladshtein[1], Elad Kinsbruner[2⋆], Qiyuan Zhao[1], and Ilya Sergey[1]

[1] National University of Singapore, Singapore
verse-lab.github.io
[2] Technion, Haifa, Israel

Abstract. We present Veil, an open-source framework for automated and interactive verification of transition systems, aimed specifically at conducting machine-assisted proofs about concurrent and distributed algorithms. Veil is implemented on top of the Lean proof assistant. It allows one to describe a transition system and its specification in a simple imperative language, producing verification conditions in first-order logic, to be discharged automatically via a range of SMT solvers. In case automated verification fails or if the system's description requires statements in a higher-order logic, Veil provides an interactive verification mode, by virtue of being embedded in a general-purpose proof assistant. We have evaluated Veil on a large set of case studies from the distributed system verification literature, showing that its automated verification performance is acceptable for practical verification tasks, while it also allows for seamless automated/interactive verification of system specifications beyond the reach of existing automated provers.

1 Introduction

Over the years, the research community has developed a spectrum of tools for formal reasoning about transition systems, ranging from interactive verification frameworks to fully automated tools. For distributed protocols in particular, formal verification has traditionally been carried out in interactive proof assistants [15, 18, 25, 41, 43, 44, 46–48, 54, 60], as the expressivity of their logics allows specifying and proving arbitrary properties, and their foundational nature [4], in which proofs are given only in terms of well-accepted axioms and are machine-checked, provides a high degree of assurance. The downside of interactive proofs, however, is that large systems take months-to-years of effort to verify [55].

At the other end of the spectrum, tools such as Ivy [38], UPVerifier [31], and mypyvy [53] use decidable fragments of first-order logic and advanced automated theorem provers [5,33] to automatically verify properties of distributed protocols and thus reduce the manual proof effort to zero. Such tools, however, are not foundational and are limited in terms of the properties that can be naturally encoded, often requiring contorted specifications designed by experts in decidable reasoning to keep verification automated [37]. Whilst this has been shown to be

[⋆] Work done during a research visit to National University of Singapore.

a viable approach for many real-world distributed protocols, certain properties of interest are simply not expressible in a decidable logic. To work around this limitation, some tools of this nature provide an escape-hatch to allow the user to interactively prove difficult properties [23, 30, 50]. These escape-hatches tend to be much less usable than interactive proof assistants, however, often lacking visibility into the goals that need to be proven or advanced tactic support.

In this paper we present Veil, a verification framework embedded in the Lean 4 proof assistant [32] that delivers the best of both worlds—providing both push-button verification for decidable fragments of first-order logic and the full power of a modern higher-order proof assistant for when automation falls short.

Importantly, Veil is *foundational*: its verification condition (VC) generator is proven sound with respect to the semantics of its specification language. It is also *lightweight*: Veil is implemented as a library using Lean metaprogramming [40], its specification language and VC generation can be easily extended to support new constructs, along with their soundness proofs. Finally, Veil allows for *seamless* interaction between automated and human-assisted proofs, all done in Lean, allowing one to establish system specifications that are not expressible in first-order logic, and, thus, are beyond the reach of existing automated tools.

2 A Tour of Veil

This section gives an tour of Veil's features and its advantages over existing tools.

2.1 Case Study: Suzuki-Kasami Algorithm

As a running example, we consider an implementation of the Suzuki-Kazami protocol for ensuring mutual exclusion [49, 58]. An implementation of the protocol in Veil is shown in Fig. 1. Some boilerplate has been removed for brevity.

A Veil specification consists of three parts: a state definition and initialisation (lines 1–34), action definitions (lines 36–83) and invariants (lines 84–98).

State. The state is comprised of uninterpreted sorts, as well as constant, relation and function symbols. As Veil is embedded in Lean, it supports most Lean types, including structures and numbers. Veil supports both *mutable* and *immutable* constants, relations and functions. Mutable fields represent the state of the algorithm, while immutable fields cannot be modified and represent symbols from a background theory. To encode persistent facts about immutable values, Veil supports a notion of *assumptions*, whch behave like axioms in Ivy and can be used to define a background theory for the immutable symbols. In Fig. 1, all values are implicitly mutable except for `init_node` (line 21).

The state is initialised before every execution using the `after_init` block (lines 24–34). In Veil assignment statements, variables starting with a capital letter are implicitly ∀-quantified, *i.e.*, `n_RN N M := 0` means "set `n_RN N M` to `0` for all `N`, `M`".

Actions. Actions in Veil define the possible transitions between states of the protocol. They may take parameters and may have a return value. They can make assumptions using `require` statements (*e.g.*, line 58), which behave as preconditions for invoking the action. Actions can comprise of any Lean code, including

```
1   type node
2
3
4   ––– Requests
5   relation reqs : node → node → Nat → Prop
6
7   ––– Tokens
8   relation t_for : Nat → node → Prop
9   relation t_q : Nat → node → Prop
10  function t_LN : Nat → node → Nat
11
12  ––– Critical section
13  relation crit : node → Prop
14
15  ––– Nodes
16  relation n_privilege : node → Prop
17  relation n_req : node → Prop
18  function n_RN : node → node → Nat
19  –– seq num of the most recently granted req
20  function n_seq : node → Nat
21  immutable individual init_node : node
22
23
24  after_init {
25   n_privilege N := N = init_node
26   n_req N := False
27   n_RN N M := 0
28   n_seq N := if N = init_node then 1 else 0
29   reqs N M I := False
30   t_for I N := I = 1 ∧ N = init_node
31   t_LN I N := 0
32   t_q I N := False
33   crit N := False
34  }
35
36  action exit (n : node) = {
37   require crit n;
38   crit n := False;
39   n_req n := False;
40   let token : Nat := n_seq n
41   t_LN token n := n_RN n n;
42   t_q token N := n_RN n N = t_LN token N + 1;
43   if m : (t_q token m) then
44     t_q token m := False;
45     n_privilege n := False;
46     let k : Nat := token + 1
47     t_for k m := True;
48     t_LN k N := t_LN token N;
49     t_q k N := t_q token N
50  }
```

```
51  action enter (n : node) = {
52     require n_privilege n
53     require n_req n
54     –– enter critical section
55     crit n := True
56  }
57  action rcv_privilege (n: node) (t: Nat) = {
58   require t_for t n;
59   require (n_seq n) < t;
60   n_privilege n := True;
61   n_seq n := t
62  }
63  action request (n : node) = {
64   require ¬ n_req n;
65   n_req n := True;
66   if (¬ n_privilege n) then
67     let k := (n_RN n n) + 1
68     n_RN n n := k;
69     reqs N n (n_RN n n) := N ≠ n
70  }
71  –– node `m` requesting from `n` with seq. number `r`
72  action rcv_request (n : node) (m : node) (r : Nat) = {
73   require reqs n m r;
74   let token : Nat := (n_seq n)
75   n_RN n m := if r ≤ (n_RN n m) then n_RN n m else r;
76   if (n_privilege n ∧ ¬ n_req n ∧
77        (t_LN token m) + 1 = (n_RN n m)) then
78     n_privilege n := False;
79     let k : Nat := token + 1
80     t_for k m := True;
81     t_LN k N := t_LN token N;
82     t_q k N := t_q token N
83  }
84  safety [mutex] (crit N ∧ crit M) → N = M
85  invariant [not_request_self] (reqs N M I) → N ≠ M
86  invariant (n_privilege N ∧ n_privilege M)
87                  → N = M
88  invariant (crit N) → (n_privilege N ∧ n_req N)
89  invariant ((t_for I N) ∧ (t_for I M)) → N = M
90  invariant ((n_seq N) ≠ 0) → t_for (n_seq N) N
91  invariant ((n_privilege N) ∧ N ≠ M)
92                  → (n_seq M) < (n_seq N)
93  invariant ((n_privilege N) ∧ (t_for I M))
94                  → I ≤ (n_seq N)
95  invariant ((t_for I N) ∧ ((J + 1) = I) ∧ (t_for J M))
96    → N ≠ M
97  invariant ((t_for I N) ∧ (t_for J M) ∧ I < J)
98                  → I ≤ (n_seq N)
99
100 #check_invariants
```

Fig. 1: An implementation of the Suzuki-Kazami locking protocol in Veil.

constant (`let`) and variable (`let mut`) definitions and calling other actions. Actions can use *demonically non-deterministic values* using the `*` symbol.

Invariants and Safety. Invariants are defined using the `invariant` keyword, and may optionally be given a custom name (*e.g.*, line 85), which will be used in generated theorems and Veil output. It is also possible to explicitly document an invariant as a safety property using the `safety` keyword which has the same semantics as `invariant` (*e.g.*, line 84). In invariants (and safety properties), variables starting with a capital letter are also implicitly universally quantified at the beginning of the formula, *i.e.*, the invariant `(crit N ∧ crit M) → N = M` will be interpreted as the Lean proposition $\forall N\ M,\ (\text{crit}\ N \land \text{crit}\ M) \to N = M$.

2.2 Bounded Model Checking

Once a protocol specification is defined, one might wish to verify that it is not vacuous, *i.e.*, that it produces non-empty execution traces.

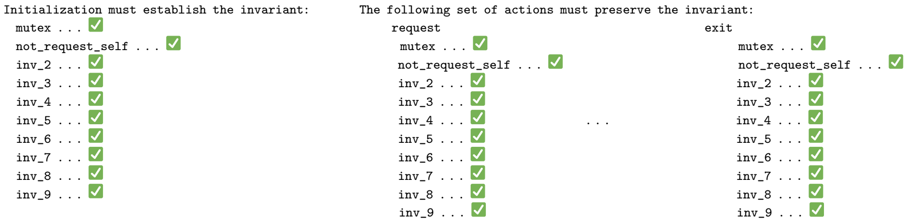
This is done with the `sat trace` command with a series of action calls, as seen in lines 1–8 in Fig. 2. This generates a Lean goal that asserts that there are parameter values and nondeterministic choices such that this trace could be a viable trace of the specification. The generated goal can be resolved interactively with standard Lean tactics, or automatically using Veil's `bmc_sat` tactic which searches for executions via SMT-based symbolic bounded model checking (BMC) [6, 17, 53]. If a satisfying trace is found, it is displayed to the user. One can also verify that specifications *do not* admit certain executions. This is done using the `unsat trace` command, as seen in lines 10–14 of Fig. 2. This, too, produces a goal that can be discharged either interactively or automatically using Veil's `bmc` tactic, which tries to prove that no such executions exist by invoking an SMT solver. `unsat trace` commands may involve `any action` or `any N actions` statements (*cf.* line 13) that will nondeterministically choose actions for a trace.

```
1  sat trace {
2    request
3    enter
4    exit
5    request
6    enter
7    exit
8  } by bmc_sat
9
10 unsat trace {
11   enter
12   enter
13   any 2 actions
14 } by bmc
```

Fig. 2: BMC in Veil

2.3 Automated Safety Proof

After specifying the protocol, the user can use the `#check_invariants` command (line 100 of Fig. 1) to try to automatically verify the protocol using SMT. Veil can use either Lean-auto [1] or Lean-SMT [3] to translate the Lean goal to SMT, and can use either cvc5 or Z3 to solve the goal automatically. The full details on Veil's SMT encoding can be found in Sec. 3.1. By default, if Veil cannot succeed with solving the goal automatically using one solver, it will try the other solver. After issuing the command, the user is met with the output shown below, where a result of either success, failure or unknown is reported for the initialisation of each invariant and for the preservation of each invariant under each action.



```
Initialization must establish the invariant:      The following set of actions must preserve the invariant:
  mutex ... ✓                                           request                                        exit
  not_request_self ... ✓                                  mutex ... ✓                                    mutex ... ✓
  inv_2 ... ✓                                            not_request_self ... ✓                          not_request_self ... ✓
  inv_3 ... ✓                                            inv_2 ... ✓                                     inv_2 ... ✓
  inv_4 ... ✓                                            inv_3 ... ✓                                     inv_3 ... ✓
  inv_5 ... ✓                                            inv_4 ... ✓            ...                      inv_4 ... ✓
  inv_6 ... ✓                                            inv_5 ... ✓                                     inv_5 ... ✓
  inv_7 ... ✓                                            inv_6 ... ✓                                     inv_6 ... ✓
  inv_8 ... ✓                                            inv_7 ... ✓                                     inv_7 ... ✓
  inv_9 ... ✓                                            inv_8 ... ✓                                     inv_8 ... ✓
                                                         inv_9 ... ✓                                     inv_9 ... ✓
```

With its default settings of a timeout of 5 seconds per SMT query and cvc5 as the solver, Veil can automatically verify that the invariant clauses are inductive in the specification in Fig. 1 and thus that the safety property holds.

2.4 Interactive Proof Mode

Sometimes, fully automated verification is not possible as SMT solvers time out and return `unknown` on the query. This can happen frequently when queries fall outside the decidable fragment. In these cases, because Veil is embedded in Lean, users can leverage Lean's theorem proving capabilities to interactively discharge these goals. In order to verify an invariant interactively, the user can ask Veil to generate the corresponding theorem statement. When one emits the #check_invariants? command, the IDE will automatically suggest to create a template (with proof to be filled in interactively) for *every* theorem needed to verify the safety of the system. Veil also supports #check_invariants!, which only suggests templates for the theorems that could not be proven automatically.

An example statement generated by #check_invariants? is shown in lines 1–3 of Fig. 3. The theorem expresses that given that the assumptions and invariants hold on a state st, all subsequent states st' reachable from st by taking the enter transition satisfy the invariant mutex, *i.e.*, mutex is preserved under enter. This statement relies on a relational semantics, which we discuss in Sec. 3.1. To discharge this goal, the user can use any native Lean tactics, as well as Veil-specific tactics (not shown in the example).

```
 1 theorem enter_mutex : ∀ (st st' : State),
 2     assumptions st → inv st → enter st st'
 3     → mutex st' := by
 4   intros st st' _ inv
 5   simp [enter, invSimp] at *
 6   rcases inv with ⟨allowed_crit, one_priv, _⟩
 7   rintro n priv req ⟨⟩ N M critN critM
 8   simp at *
 9   apply one_priv
10   . by_cases h : (N = n)
11     <;> simp [allowed_crit, h, priv, critN]
12   . by_cases h : (M = n)
13     <;> simp [allowed_crit, h, priv, critM]
```

Fig. 3: Interactive proof that mutex is preserved by the enter action.

In Fig. 3, the user uses the Lean intros tactic to bring the states and the invariant into the Lean context, and then simplifies the action definition (enter) and unfolds the invariant definition (using the invSimp lemma set introduced by Veil). The proof only uses two invariant clauses, allowed_crit and one_priv, which are extracted from the invariant conjunction via the rcases tactic. The proof is concluded using the by_cases tactic to case-split on whether the considered node is n, the argument to enter, with the subgoals discharged by using implications from the action definition (critN, critM, priv), an invariant clause (allowed_crit) and the case assumption (h).

Veil also supports semi-automatic verification: sometimes, for a query that falls outside of the decidable fragment and cannot be decided automatically, it is possible to decide essentially the same query automatically after changing its structure slightly using tactics. Therefore, Veil introduces the solve_clause tactic, which automatically tries to discharge the current goal using SMT in the same manner as #check_invariants. Running solve_clause can result in three possible outputs: (i) either the verification succeeded and the goal is admitted;[3] or (ii) the goal is found to be false and a minimised model is presented as a counterexample (*cf.* Sec. 3.4); or (iii) the query returns unknown and no verdict can be reached. This lends itself to a style of *solver-guided interactive verification*, in which users write interactive proofs and occasionally invoke solve_clause to see if the goal can be discharged automatically, if they're on a wrong path and the goal is false, or if they have to keep going (solver returned unknown).

[3] Currently, proof reconstruction (provided by Lean-SMT) is off by default in Veil.

3 Implementing Veil in Lean

Veil is implemented in Lean 4 [32], a dependently-typed programming language and theorem prover, which offers monad comprehensions with local mutation [51].

3.1 Language Embedding

At the core of Veil lies a domain-specific language (DSL) for writing and specifying transition systems. The DSL is inspired by Alloy [16], Ivy [38], and mypyvy [53]; it adopts standard first-order logic approach for specifying *properties* (*e.g.*, assumptions and safety), while the *transitions* (actions) are encoded as Lean's native monadic computations, embracing the full power of its `do`-notation [51].

Protocol states σ in Veil are represented by Lean structures with fields corresponding to the `relation`, `function`, and `individual` declarations in the specification. The type of the protocol's transitions is, thus, dependent on the type of its states. For a fixed type σ, each transition is encoded as an instance of a two-state relation of the form $\mathbf{BigStep}\,\sigma\,\rho \triangleq \sigma \to \rho \to \sigma \to \mathit{Prop}$, which relates an input state with the possible outcome result of type ρ and an output state.

A naïve approach to reuse Lean's `do`-notation for Veil actions would be to simply provide a monad instance for $\mathbf{BigStep}\ \sigma\,\rho$, defining the corresponding `bind` and `pure` operations. Unfortunately, a canonical definition of `bind` for $\mathbf{BigStep}\ \sigma\,\rho$ as the composition of transition relations is not well-suited for SMT-based proofs. For a relation $tr_1 : \mathbf{BigStep}\,\sigma\,\rho$ and a continuation $tr_2 : \rho \to \mathbf{BigStep}\,\sigma\,\rho'$, the composition is $\lambda s\ r'\ s',\ \exists (t : \sigma)(r : \rho'),\ tr_1\ s\ r\ t \wedge tr_2\ r\ t\ r'\ s'$. That is, an input state s is related to an output state s' and the return value r' if there *exist* an intermediate state t and a result r that serve as an output of tr_1 and an input to tr_2. This encoding introduces higher-order quantification over the elements of the structure σ (*i.e.*, the relations describing the protocol state) in the respective VCs, which makes it impossible to dischage them via SMT solvers. Although Veil implements heuristics for quantifier elimination, running them for each `bind` operation severely impacts its peformance.

To avoid the higher-order quantification introduced by such a definition of `bind`, Veil features an alternative encoding of transitions. We first define each atomic Veil command as an instance of type $\mathbf{WP}\,\sigma\,\rho \triangleq (\rho \to \sigma \to \mathit{Prop}) \to (\sigma \to \mathit{Prop})$. This type is a weakest-precondition predicate transformer [10] that takes an assertion over a transition result ρ and state σ and returns the weakest pre-condition on the pre-state σ which must be satisfied in order to guarantee that the assertion holds in all post-states. The `bind` operation for $\mathbf{WP}\,\sigma\,\rho$ can then be expressed simply via nesting: given $tr_1 : \mathbf{WP}\,\sigma\,\rho$ and $tr_2 : \rho \to \mathbf{WP}\,\sigma\,\rho'$, their composition is defined as $\lambda post,\ tr_1\ (\lambda r', tr_2\ r'\ post)$. Following this approach, the `enter` action from Fig. 1 is first expanded into `enter.wp` (lines 1–6 of Fig. 4), where `get` and `modify` operations are standard monadic operations for state reading and writing, and `WP.req` is a monadic operation for the `require` statement (lines 7–8 of Fig. 4).

Given the weakest precondition semantics of a transition $tr : \mathbf{WP}\,\sigma\,\rho$, our encoding of Veil DSL derives its more traditional relational counterpart as follows:

$$tr' : \mathbf{BigStep}\ \sigma\,\rho \triangleq \lambda st_0\ res_1\ st_1,\ \neg tr\ (\lambda res\ st,\ \neg(res = res_1 \wedge st = st_1))\ st_0$$

```
1  def enter.wp (n : ℕ) : WP σ Unit :=
2    get ▷.bind fun (st : σ) ⇒
3    WP.req (st.n_have_privilege n) ▷.bind fun _ ⇒
4    WP.req (st.n_requesting n) ▷.bind fun _ ⇒
5    modify fun st ⇒
6      { st with crit := fun x ⇒ st.crit x ∨ x = n }
7  def WP.req (P : Prop) : WP σ Unit := fun st post ⇒
8    P → post () st
9  def enter (n : ℕ) : BigStep σ Unit := fun st r st' ⇒
10     st.n_have_privilege n ∧
11     st.n_requesting n ∧ r = () ∧
12     st' = { st with crit := fun x ⇒ st.crit x ∨ x = n }
```

Fig. 4: Expansion steps for the `enter` action from Fig. 1.

Observe that $tr\ (\lambda res\ st,\ \neg(res = res_1 \land st = st_1))$ expresses the weakest precondition for the action tr under the postcondition that *excludes* result res_1 and state *post* as reachable outcomes. The definition of tr', thus, ensures that the state st_0 transitions to st_1 with result res_1 only when such weakest pre-condition *does not hold* for st_0, *i.e.*, when st_1 and res_1 are reachable from st_0. We formally prove that this way of deriving tr' from tr is equivalent to the definition of tr' as a relation using standard big-step semantics for language constructs (bindings, assertions, *etc.*) for all actions tr with no failing assertions.

By applying this transformation and unfolding and simplifying all $\mathbf{WP}\,\sigma\,\rho$ definitions, Veil generates a two-state formula for the `enter` action (Fig. 4, lines 9–12). The result is used in the VC that is passed as a query to SMT solvers.

3.2 Soundness of the Verification Condition Generator

The main Veil soundness theorem states the equivalence of the two transition semantics from Sec. 3.1: $\forall s\ post, tr\ post\ s\ \Leftrightarrow\ (\forall s' r',\ tr' s\ r' s'\ \Rightarrow\ post\ r' s')$ for all actions tr with no failing assertions. In other words, the weakest precondition of an action tr with a postcondition *post* holds on a state s *if and only if* the two-state formula tr' relates the input state s to s' and r' from the postcondition *post*. A proof of this theorem is generated by Veil for each action declaration using Lean's type class resolution mechanism.

3.3 Interaction with SMT

All proof obligations Veil generates are Lean proof goals, which users can choose to either prove interactively using Lean's native tactics, or discharge via Veil's built-in automation that leverages the cvc5 [5] and Z3 [33] SMT solvers.

Since Veil is embedded in a higher-order logic and our verification condition generator (VCG) can emit goals that employ higher-order quantification (*e.g.*, to model non-deterministic assignment to relations), the main challenge in discharging Veil-generated goals with SMT is to reduce them to first-order logic. To this end, we developed a suite of custom tactics and `simp` procedures to (a) automatically destruct higher-order structures into first-order components, (b) hoist higher-order quantification to the top-level of the goal (where SMT supports declaring relations and functions), and (c) change the structure of the goal to make it easier for SMT solvers to discharge (*e.g.*, by recursively case-splitting `if`-statements and invoking the SMT solver separately on each sub-branch).

To translate Lean goals to SMT-LIB queries, Veil employs the Lean-SMT [3] and Lean-auto [1] libraries: Smt generates small, readable queries, but has longer translation time, whereas Auto is fast but generates larger queries. Veil uses Auto by default, but users can configure which translator to use on a per-query basis.

3.4 Model Minimisation

When the SMT solver provides a counter-example (model), *e.g.*, if an invariant provided by the user is not inductive, Veil can minimise the counter-example by issuing further incremental SMT queries to first reduce the sort and then the relation cardinalities, similar to the approach taken by mypyvy [53]. The counter-example is then displayed to the user in a human-readable format. In our experience, minimisation was crucial to make protocol models understandable.

4 Evaluation and Case Studies

Veil is available online [42].[4] We evaluated it *w.r.t.* the research questions below:

RQ1: Can Veil automatically verify complex distributed protocol specifications?
RQ2: Can Veil automatically verify distributed protocol specifications that are encoded outside of Effectively Propositional Logic (EPR)?
RQ3: Is Veil expressive enough to supplement automation with interactive proofs?

The experiments in this section were ran on a 2024 MacBook Pro with the M4 chip, 32GB of RAM, with cvc5 version 1.2.1, Z3 version 4.14.0, Lean version 4.16.0, and Lean-auto revision 918a699. We use Ivy revision dbe45e7.

4.1 Automated Verification of Distributed Protocols

To test RQ1, we collected 16 case studies from the following sources:

- 9 case studies from IvyBench [13], with manually added invariants,
- 2 case studies from the work of Padon *et al.* [37] on verifying Paxos [20], and
- 5 case studies from various other sources [8, 28, 39, 58].

The case studies in our set total 1704 non-empty, non-comment lines of Lean code (average 100 lines per file), 85 actions and 185 invariants. All benchmarks have an equivalent formulation in Ivy. We attempted to prove each of them automatically using `#check_invariants`, without any interactive proof. We ran Veil on each of them separately and timed the execution by using Lean's profiler. For every benchmark, we also ran Ivy on the original formulation (with the `complete=fo` flag when required). Veil's timeout was its default of 5 seconds per SMT query for all benchmarks except Rabia, where it was set to 120 seconds. To the best of our knowledge, Ivy does not support per-query timeouts, so we set its overall timeout to 300 seconds. Times are an average over 5 runs.

Our results are summarised in Fig. 5. It splits the total time taken by Veil into simplification time (*cf.* Sec. 3.3), time taken for translation using Lean-auto, and time taken by SMT solver calls. The time taken by Ivy is also displayed. Veil successfully verified all benchmarks in the set, while Ivy has failed to verify several benchmarks, which are marked with * in Fig. 5.

Veil verifies all but 2 benchmarks in under 15 seconds (87.5%), and all but 4 benchmarks in under 10 seconds (75%). Ivy times out for 2 of the benchmarks. We conclude that Veil can verify a variety of distributed protocol specifications without the need for user effort, and its performance in doing so is acceptable.

[4] https://github.com/verse-lab/veil

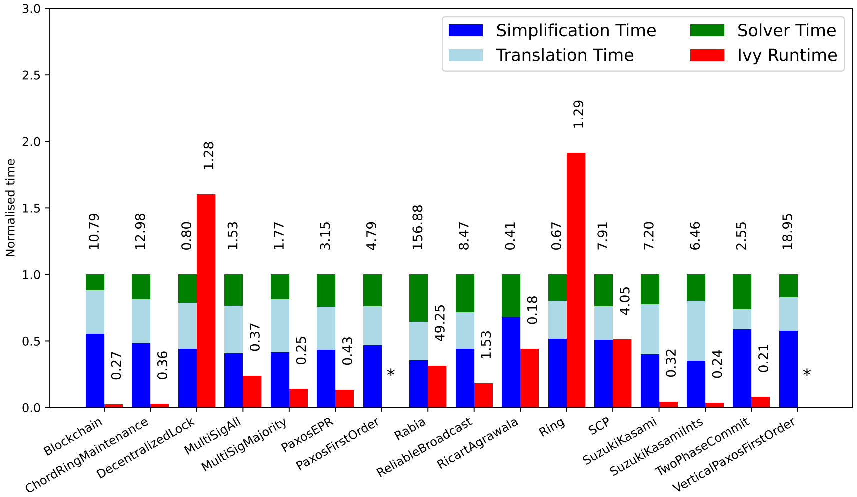


Fig. 5: Results of running Veil and Ivy on the benchmark set. The bar heights are normalised *w.r.t.* Veil verification times. All absolute times are in seconds.

4.2 Beyond EPR-Encoded Protocols

To test RQ2, we examine Veil's performance on benchmarks whose encoding to SMT is outside of the Effectively Propositional (EPR) fragment of first order logic, which is known to be decidable [26]. SMT solvers use heuristics and specialised techniques to try to decide first-order non-EPR queries, which may yield `unknown`. Nonetheless, it is often convenient to write protocols and algorithms in general FOL, as substantial work is often needed to restate them in EPR [37].

We will examine four case studies from the benchmark set of Sec. 4.1:

- two variants of the Paxos protocol that are discussed in Padon *et al.* [37]: the Single-Decree Paxos [20], as well as Vertical Paxos [22],
- the Suzuki-Kasami algorithm with positive integer indices (Sec. 2), and
- the Reliable Broadcast protocol [8].

These four benchmarks are expressed in unconstrained first-order logic, and their encoding falls outside of EPR. Veil manages to verify all of them.[5] When running Ivy on these benchmarks (with the `complete=fo` flag to force Ivy to ignore that they are outside of the decidable fragment), it succeeds in verifying Reliable Broadcast and Suzuki-Kasami and times out for Paxos and Vertical Paxos.

From our results, we conclude that Veil is successful in verifying protocols whose encodings are even outside of EPR, which increases its versatility.

4.3 Combining SMT Automation with Interactive Proofs

For the RQ3, we considered two case studies, which have theorems established in interactive theorem provers, in addition to Ivy specifications. We ported these theorems and their proofs into Veil, and report our findings below.

[5] For Suzuki-Kasami, we disable the `--finite-model-find` option of cvc5, which Veil enables by default. With it enabled, verification takes around 120 seconds.

Stellar Consensus Protocol (SCP) [27] features a formally defined model that includes several higher-order components, which, for example, involve quantification over sets. To encode SCP into Ivy, Losa and Dodds [28] abstracted these components by stating their required properties in FOL as assumptions; the soundness of the abstraction *w.r.t.* a concrete higher-order model is then manually proven in Isabelle/HOL [52]. In our case, we bundle the assumptions as a type class and assume the existence of its instance during automated verification, so that the assumptions can be utilised by SMT solvers. We further ported the proofs from Isabelle/HOL to Veil and validated the abstraction soundness by deriving an instance of the assumptions type class from the concrete model. Compared with the combination Ivy + Isabelle/HOL, our approach allows the abstract model, the concrete model, and their correspondence to be verified in the same framework, without relying on a trusted manual assumption translation.

Rabia protocol [39] comes with a formalisation in both Ivy and the Rocq Prover, where the Rocq one only differs in that it (1) interprets the type of *phases* (a notion in Rabia) as the natural number type, (2) admits all invariants to be checked by Ivy as axioms, and (3) proves several more properties to be invariants of Rabia. To manually prove an invariant P in Rocq, it suffices to show that the established invariants (either checked separately by Ivy and admitted in Rocq, or proven in Rocq) entail P. Those additional invariants are proven in Rocq since they cannot be proven in pure FOL, *e.g.*, some require induction on phases. Thanks to Veil's support for manual invariant reasoning, we successfully verified all additional invariants proven in the Rocq formulation, for the Veil encoding of Rabia. Notably, during the porting process, we spotted one discrepancy between an invariant in the Ivy formulation and the corresponding admitted one in the Rocq code, since our Veil formulation was faithfully translated from the Ivy one, and Lean complained when we attempted to use that invariant as it is used in the original Rocq proof. Such nitpicking would require careful examination for the original Ivy + Rocq combination, which, again, shows the benefits of having a unified framework for both automated and interactive proofs.

5 Related and Future Work

Alloy [16] and the TLA$^+$ toolbox [21] are de-facto the most popular tools to date for prototyping and modelling state-transition systems in general and distributed protocols in particular [2,36]. Alloy only allows for bounded verification, assuming that each sort is finite. TLA$^+$ is most commonly used for concrete-state model checking of protocol designs [57], but also offers a proof system for deductive verification, TLAPS [9]. Unlike Veil, TLAPS does not offer easy extensibility via metaprogramming and relies on a *trusted* translation to the languages of its backend provers, Zenon [7] and Isabelle/HOL [52], to discharge its proof obligations.

The language of Veil is almost a verbatim port of RML, the specification language of Ivy [38], while its bounded model checking capability is inspired by a similar feature of mypyvy [53]. Unlike these tools, Veil is a foundational verification framework with a formal soundness proof of its VC generator, offering the full power of interactive proofs in Lean, its extensibility, and libraries.

Veil is a spiritual successor of Jahob [19] and Why3 [11], tools that provide rich specification languages and rely on third-party provers to discharge verification conditions, automatically or interactively. More recent frameworks RefinedC [45], RefinedRust [12], and Diaframe [34] are foundational embeddings of mostly-automated verifiers into the Rocq Prover (formerly known as Coq). Those tools target general-purpose programming languages and rely on custom tactic-based automation rather than general-purpose first-order logic solvers, making them not immediately suitable for effective reasoning about transition systems.

We believe our initial prototype of Veil opens several avenues for exciting future work. In particular, we are planning to explore the integration of state-of-the-art approaches for inferring system invariants in first-order logic [14,29,56,59] into Veil. Given the available higher-order specification composition mechanisms of Lean, we are also planning to explore ways to compose properties of individually verified protocols, in the style of the recent Bythos framework [60]. Finally, going beyond simple transition systems, we are hopeful that our experience of implementing Veil will pave the way for embedding general-purpose SMT-based program verifiers [23, 24, 35] into a foundational proof assistant.

Acknowledgements. We thank Peter Müller for his feedback on a draft of this paper. We also thank the reviewers of CAV'25 for their insightful comments. Mark Yuen has implemented the initial version of Suzuki-Kasami algorithm in TLA$^+$ and Ivy as a part of his Capstone thesis [58]. This work has been supported by a Singapore Ministry of Education (MoE) Tier 1 grant T1 251RES2108 "Automated Proof Evolution for Verified Software Systems", MoE Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001, by Stellar Development Foundation Academic Research Grant, and by Sui Academic Research Award. The work of Elad Kinsbruner has been supported by the European Union (ERC, EXPLOSYN, 101117232). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

1. leanprover-community/lean-auto: Experiments in automation for Lean. Available at https://github.com/leanprover-community/lean-auto, last accessed on January 30, 2025.
2. TLA+ Examples. Available at https://github.com/tlaplus/Examples, last accessed on January 30, 2025.
3. ufmg-smite/lean-smt: Tactics for discharging Lean goals into SMT solvers. Available at https://github.com/ufmg-smite/lean-smt, last accessed on January 30, 2025.
4. Appel, A.W.: Foundational proof-carrying code. In: LICS. pp. 247–256. IEEE Computer Society (2001). https://doi.org/10.1109/LICS.2001.932501
5. 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

6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
7. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In: LPAR. LNCS, vol. 4790, pp. 151–165. Springer (2007). https://doi.org/10.1007/978-3-540-75560-9_13
8. Chang, J., Maxemchuk, N.F.: Reliable broadcast protocols. ACM Trans. Comput. Syst. **2**(3), 251–273 (1984). https://doi.org/10.1145/989.357400
9. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: A TLA+ Proof System. In: Proceedings of the LPAR 2008 Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008), https://ceur-ws.org/Vol-418/paper2.pdf
10. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
11. 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
12. Gäher, L., Sammler, M., Jung, R., Krebbers, R., Dreyer, D.: RefinedRust: A Type System for High-Assurance Verification of Rust Programs. Proc. ACM Program. Lang. **8**(PLDI), 1115–1139 (2024). https://doi.org/10.1145/3656422
13. Goel, A., Sakallah, K.A.: ivybench: Collection of Distributed Protocol Verification Problems. Available at https://github.com/aman-goel/ivybench, last accessed on January 29, 2025.
14. Hance, T., Heule, M., Martins, R., Parno, B.: Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In: NSDI. pp. 115–131. USENIX Association (2021), https://www.usenix.org/conference/nsdi21/presentation/hance
15. 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
16. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006), http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928
17. Konnov, I., Kukovec, J., Tran, T.: TLA+ model checking made symbolic. Proc. ACM Program. Lang. **3**(OOPSLA), 123:1–123:30 (2019). https://doi.org/10.1145/3360549
18. Krogh-Jespersen, M., Timany, A., Ohlenbusch, M.E., Gregersen, S.O., Birkedal, L.: Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In: ESOP. LNCS, vol. 12075, pp. 336–365. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_13
19. Kuncak, V.: Modular Data Structure Verification. Ph.D. thesis, Massachusetts Institute of Technology. (2007), available at https://dspace.mit.edu/handle/1721.1/38533.
20. Lamport, L.: Paxos Made Simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) pp. 51–58 (2001), https://www.microsoft.com/en-us/research/publication/paxos-made-simple/
21. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002), http://research.microsoft.com/users/lamport/tla/book.html
22. Lamport, L., Malkhi, D., Zhou, L.: Vertical paxos and primary-backup replication. In: PODC. pp. 312–313. ACM (2009). https://doi.org/10.1145/1582716.1582783

23. 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
24. 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
25. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: POPL. pp. 357–370. ACM (2016). https://doi.org/10.1145/2837614.2837622
26. Lewis, H.R.: Complexity results for classes of quantificational formulas. J. Comput. Syst. Sci. **21**(3), 317–353 (1980). https://doi.org/10.1016/0022-0000(80)90027-6
27. Lokhava, M., Losa, G., Mazières, D., Hoare, G., Barry, N., Gafni, E., Jove, J., Malinowsky, R., McCaleb, J.: Fast and secure global payments with Stellar. In: SOSP. pp. 80–96. ACM (2019). https://doi.org/10.1145/3341301.3359636
28. Losa, G., Dodds, M.: On the formal verification of the stellar consensus protocol. In: 2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020. OASIcs, vol. 84, pp. 9:1–9:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASICS.FMBC.2020.9
29. Ma, H., Goel, A., Jeannin, J., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: incremental inference of inductive invariants for verification of distributed protocols. In: SOSP. pp. 370–384. ACM (2019). https://doi.org/10.1145/3341301.3359651
30. McMillan, K.L., Padon, O.: Deductive Verification in Decidable Fragments with Ivy. In: SAS. LNCS, vol. 11002, pp. 43–55. Springer (2018). https://doi.org/10.1007/978-3-319-99725-4_4
31. Mora, F., Desai, A., Polgreen, E., Seshia, S.A.: Message chains for distributed system verification. Proc. ACM Program. Lang. **7**(OOPSLA2), 2224–2250 (2023). https://doi.org/10.1145/3622876
32. de Moura, L., Ullrich, S.: The Lean 4 Theorem Prover and Programming Language. In: CADE. LNCS, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
33. de Moura, L.M., Bjørner, N.S.: 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
34. Mulder, I., Krebbers, R., Geuvers, H.: Diaframe: automated verification of fine-grained concurrent programs in Iris. In: PLDI. pp. 809–824. ACM (2022). https://doi.org/10.1145/3519939.3523432
35. 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
36. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
37. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA), 108:1–108:31 (2017). https://doi.org/10.1145/3140568
38. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI. pp. 614–630. ACM (2016). https://doi.org/10.1145/2908080.2908118

39. Pan, H., Tuglu, J., Zhou, N., Wang, T., Shen, Y., Zheng, X., Tassarotti, J., Tseng, L., Palmieri, R.: Rabia: Simplifying State-Machine Replication Through Randomization. In: SOSP. pp. 472–487. ACM (2021). https://doi.org/10.1145/3477132.3483582
40. Paulino, A., Testa, D., Ayers, E., Karunus, E., Bövinga, H., Limperg, J., Gadgil, S., Bhat, S.: Metaprogramming in Lean 4 (2024), available at https://leanprover-community.github.io/lean4-metaprogramming-book/
41. Pîrlea, G., Sergey, I.: Mechanising blockchain consensus. In: CPP. pp. 78–90. ACM (2018). https://doi.org/10.1145/3167086
42. Pîrlea, G., Gladshtein, V., Kinsbruner, E., Zhao, Q., Sergey, I.: Veil: A Framework for Automated and Interactive Verification of Transition Systems. Software Artefact. (Apr 2025). https://doi.org/10.5281/zenodo.15208271
43. Qiu, L., Kim, Y., Shin, J., Kim, J., Honoré, W., Shao, Z.: Lido: Linearizable byzantine distributed objects with refinement-based liveness proofs. Proc. ACM Program. Lang. **8**(PLDI), 1140–1164 (2024). https://doi.org/10.1145/3656423
44. Rahli, V., Vukotic, I., Völp, M., Veríssimo, P.J.E.: Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In: ESOP. LNCS, vol. 10801, pp. 619–650. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_22
45. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: automating the foundational verification of C code with refined ownership types. In: PLDI. pp. 158–174. ACM (2021). https://doi.org/10.1145/3453483.3454036
46. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Program. Lang. **2**(POPL), 28:1–28:30 (2018). https://doi.org/10.1145/3158116
47. Sharma, U., Jung, R., Tassarotti, J., Kaashoek, M.F., Zeldovich, N.: Grove: a Separation-Logic Library for Verifying Distributed Systems. In: SOSP. pp. 113–129. ACM (2023). https://doi.org/10.1145/3600006.3613172
48. Sprenger, C., Klenze, T., Eilers, M., Wolf, F.A., Müller, P., Clochard, M., Basin, D.A.: Igloo: soundly linking compositional refinement and separation logic for distributed system verification. Proc. ACM Program. Lang. **4**(OOPSLA), 152:1–152:31 (2020). https://doi.org/10.1145/3428220
49. Suzuki, I., Kasami, T.: A distributed mutual exclusion algorithm. ACM Trans. Comput. Syst. **3**(4), 344–349 (1985). https://doi.org/10.1145/6110.214406
50. Taube, M., Losa, G., McMillan, K.L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J.R., Woos, D.: Modularity for decidability of deductive verification with applications to distributed systems. In: PLDI. pp. 662–677. ACM (2018). https://doi.org/10.1145/3192366.3192414
51. Ullrich, S., de Moura, L.: 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). Proc. ACM Program. Lang. **6**(ICFP), 512–539 (2022). https://doi.org/10.1145/3547640
52. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: TPHOLs. LNCS, vol. 5170, pp. 33–38. Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_7
53. Wilcox, J.R., Feldman, Y.M.Y., Padon, O., Shoham, S.: mypyvy: A research platform for verification of transition systems in first-order logic. In: CAV. LNCS, vol. 14682, pp. 71–85. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_4
54. 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

55. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: CPP. pp. 154–165. ACM (2016). https://doi.org/10.1145/2854065.2854081
56. Yao, J., Tao, R., Gu, R., Nieh, J.: DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In: OSDI. pp. 485–501. USENIX Association (2022), https://www.usenix.org/conference/osdi22/presentation/yao
57. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA$^+$ Specifications. In: CHARME. LNCS, vol. 1703, pp. 54–66. Springer (1999). https://doi.org/10.1007/3-540-48153-2_6
58. Yuen, M.: Verifying Distributed Protocols: from Executable to Decidable. Capstone thesis, Yale-NUS College, Singapore (2022), accompanying code available at https://github.com/markyuen/tlaplus-to-ivy/.
59. Zhang, T.N., Hance, T., Kapritsos, M., Chajed, T., Parno, B.: Inductive Invariants That Spark Joy: Using Invariant Taxonomies to Streamline Distributed Protocol Proofs. In: OSDI. pp. 837–853. USENIX Association (2024), https://www.usenix.org/conference/osdi24/presentation/zhang-nuda
60. Zhao, Q., Pîrlea, G., Grzeszkiewicz, K., Gilbert, S., Sergey, I.: Compositional Verification of Composite Byzantine Protocols. In: CCS. pp. 34–48. ACM (2024). https://doi.org/10.1145/3658644.3690355