Engineering a Certified Model Checker with Class

QIYUAN ZHAO*, National University of Singapore, Singapore
ZIYU MAO*†, University of Toronto, Canada
GEORGE PÎRLEA, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

Explicit-state model checking automatically checks safety properties of finite instances of concurrent and distributed protocols, without requiring an inductive invariant or a user-written proof. Such checks are valuable in their own right and support an iterative verification workflow in which counterexamples guide subsequent symbolic checks and deductive proofs. When separate tools independently implement the semantics of a modelling language, this workflow relies on their implementations agreeing. Semantic discrepancies can cause missed violations or spurious counterexamples, misleading later verification efforts. Connecting these stages through machine-checked proofs must also reconcile the logical abstractions suited to proof with the concrete representations needed for efficient state exploration.

We present Lace, a certified explicit-state safety-focused model checker built entirely in Lean and integrated into Veil, a foundational verifier for distributed protocols. Our approach retains proof-oriented abstractions and supplies efficient executable counterparts with proofs of semantic agreement. The central mechanism uses type classes to pair logical state fields with runtime data structures under a single lawfulness condition. Rewriting with this law and generalising the remaining reads recovers the verification conditions of the logical representation, preserving existing invariants and proofs. A modular multi-extraction procedure turns non-deterministic actions into successor functions, automatically proving that they enumerate exactly the outcomes permitted by the axiomatic semantics. A parallel breadth-first search completes the checker, with correctness invariants embedded in the types of its search state and worker results. We evaluate Lace on 28 independently developed TLA$^+$ specifications translated into Veil. On 20 benchmarks instances, Lace outperforms TLC at every measured thread count. Moreover, an ablation shows that concrete field representations improve performance by up to nearly two orders of magnitude, demonstrating that preserving proof-oriented abstractions is compatible with efficient exhaustive checking. Together, our techniques for lawful field representations, certified enumeration, and proof-carrying search offer reusable building blocks for constructing efficient certified model checkers and related exploration tools within a proof assistant.

1 Introduction

Explicit-state model checking [25, 40, 75] provides an automated way to analyse state-transition systems such as models of concurrent and distributed protocols. For safety checking, it systematically explores the reachable states of finite instances, without requiring an inductive invariant or a user-written proof. A failing check provides a concrete counterexample that might indicate a design error, while a successful exhaustive check establishes the checked safety for those instances. Such checks are valuable in their own right, even when no deductive proof is attempted [10, 23, 30, 47, 58, 63].

These checks are also helpful when users seek to establish correctness in the unbounded setting: a standard methodology is to perform them as a first step to avoid wasting effort on a flawed design. This is advocated in the documentation of TLAPS [5, 65], the proof system for TLA$^+$ [34], a well-established specification language for concurrent and distributed systems. In this setting, the checking step is performed by TLC [75], the explicit-state checker for TLA$^+$. This methodology shapes an iterative workflow [28]: users typically move from explicit-state checking to symbolic

*Both authors contributed equally to this research.
†Work done during an internship at National University of Singapore.

Authors' Contact Information: Qiyuan Zhao, zhaoqiyuan@u.nus.edu, National University of Singapore, Singapore, Singapore; Ziyu Mao, ziyu.mao@mail.utoronto.ca, University of Toronto, Toronto, Canada; George Pîrlea, gpirlea@u.nus.edu, National University of Singapore, Singapore, Singapore; Ilya Sergey, ilya@nus.edu.sg, National University of Singapore, Singapore, Singapore.

checking of candidate inductive invariants and then to deductive proof, but may return to model checking to test candidates even at that stage [35]. Counterexamples guide revisions, while successful checks build confidence, as illustrated by the verification experience of Schultz *et al.* [56]: they used TLC to search for counterexamples during invariant discovery, and the resulting invariant required no further revision in the subsequent TLAPS proof.

In the TLA$^+$ ecosystem, different stages of this workflow are supported by separate tools (*e.g.*, TLC and TLAPS) [28]. Since each tool independently implements the semantics of TLA$^+$, the workflow implicitly relies on these implementations faithfully realising the same semantics. However, reported bugs have caused model checkers to miss reachable states or admit impossible transitions [1, 66], showing that this assumption can fail in practice. Such errors undermine the workflow in both directions: a missed violation can lead users to attempt to prove an incorrect design, while a spurious counterexample can lead them to discard a correct design or a valid invariant. Although techniques (*e.g.*, via differential fuzzing [16]) can uncover such discrepancies, it cannot rule them out.

A recent approach leaves the methodology and the workflow unchanged and changes how the toolchain is built: in *foundational* verifiers, the modelling language is embedded in a proof assistant, and every tool the verifier provides is connected to a single semantics by machine-checked proofs [17, 18, 53]. What remains trusted is general-purpose infrastructure, such as the proof assistant's kernel and compiler (and, for automated checks, SMT solvers), rather than a separate implementation of the semantics in each tool. Veil [53] is such a verifier for distributed protocols, embedded in the Lean proof assistant, and supports *multi-modal verification* across the workflow, combining symbolic bounded model checking (BMC) and randomised simulation [18] for counterexample search with SMT-based checking of inductive invariants and interactive proof. For the first stage, however, both techniques have limitations: symbolic BMC explores only executions of bounded length and can time out even at shallow depths (Sec. 2), while randomised simulation is not exhaustive. Neither provides the exhaustive checking of finite instances offered by TLC.

In this work, we show that, inside a foundational verifier (here, Veil), exhaustive safety checking of finite instances can (a) provably agree with the semantics used by the later stages, and (b) outperform state-of-the-art model checkers. These two goals pull in opposite directions. Goal (a) ties the checker to the *abstractions* that suit proofs, which in Veil take two forms: an action may make a *non-deterministic choice* by stating a predicate the chosen value must satisfy, without saying how to compute it, and state fields are *logical* relations and functions over *uninterpreted sorts*, a representation well suited for SMT-based reasoning [48]. Goal (b) needs *concrete* counterparts: a function returning all successors of a pre-state by enumerating values satisfying each choice's predicate, and runtime field representations that make states cheap to hash and compare, since an explicit-state checker usually does so for millions of states. Abandoning these abstractions would complicate symbolic checking and proof, while separately maintained executable specifications could diverge from the original ones. Our insight is therefore to retain these abstractions and supply generic executable counterparts with machine-checked proofs of semantic agreement.

The central component of our solution concerns field representations. Using Lean's *type class* mechanism, we pair logical fields with runtime structures (arrays, hash maps, *etc.*) satisfying a single machine-checked *lawfulness* condition: updating a runtime structure and then reading off its logical field gives the same result as updating the logical field directly. Rewriting with this law eliminates all concrete updates from a verification condition (VC), after which we generalise the remaining concrete reads to fresh logical variables. This *rewrite-then-generalise* procedure yields exactly the VCs of the logical representation, preserving symbolic checks and proofs while enabling efficient hashing and membership tests without changing the specification or its invariants.

The second component concerns non-deterministic choices. In Veil's *axiomatic* semantics, actions are formulas relating a state to its successors; model checking instead requires enumerating them.

Exploiting Veil's encoding of choices as syntactically distinct nodes in a monadic program, we use Lean metaprogramming [49] to implement *multi-extraction*, a procedure that turns each action into the successor function required by Goal (b). The resulting function uses type class instances to enumerate exactly the values satisfying each choice's predicate, and comes with a proof, assembled automatically from reusable lemmas, that it returns exactly the successors allowed by the axiomatic semantics (Theorem 4.1). Our library supplies multi-extraction rules for common constructs and lawful representations for common field types, so users need proofs only for their extensions.

Finally, the search must also be verified: dropping states (*e.g.*, due to races between parallel workers) could hide violations. The search runs in Lean's `IO` monad so it can report progress and be interrupted. Its workers run in parallel as native `IO` tasks, whose results Lean's logic cannot relate to the computations that produced them. We therefore make the parallel search *proof-carrying*: its state and each worker's result carry Lean proof terms establishing the search's correctness.

We combine these components in Lace, a certified, efficient explicit-state model checker for Veil, built *entirely in* Lean. That is, the verdicts of its concrete executions are consistent with the axiomatic semantics that Veil's symbolic checks and proofs rely on to establish safety properties.[1]

A case study (Sec. 2) illustrates how Lace supports Veil's verification workflow: exhaustive checking of finite instances builds confidence in a model before proof development and exposes an error in a proposed optimisation. Beyond this case study, our evaluation (Sec. 6) assesses Lace's applicability and performance on existing TLA$^+$ specifications developed independently of our work and translated into Veil. The translations further let us compare Lace with TLC in the task they share, namely safety checking of finite instances: for every evaluated finite instance, both tools explore the same number of distinct states, providing a consistency check on the translations. In our performance comparison, Lace runs faster than TLC on all 20 benchmark instances at every tested thread count (1–14). In addition, an ablation on several protocols written directly in Veil shows that efficient field representations make Lace up to nearly *two orders of magnitude* faster than the logical ones, demonstrating the practical value of lawful field representations.

Contributions. To summarise, we make the following technical contributions:

- *Type class-based lawful field representations.* A mechanism pairing logical state machine fields with runtime data structures for efficient model checking, and a single lawfulness condition that removes runtime representation details from VCs via rewrite-then-generalise (Sec. 3).
- *Enumerative execution for non-determinism.* A procedure that extracts successor functions from actions by resolving their non-deterministic choices, with proofs that they enumerate exactly the successors allowed by Veil's axiomatic semantics (Sec. 4).
- *Lace: a certified, safety-focused explicit-state model checker.* A parallel breadth-first search whose search state and worker results carry proof terms of the search's correctness invariants, combining the two components above into an end-to-end verified checker (Sec. 5).
- *Case study and empirical evaluation.* An illustrative case study (Sec. 2), and an evaluation on protocol models translated from TLA$^+$, showing that Lace explores large state spaces with performance competitive with TLC, and an ablation of field representations (Sec. 6).

2 Lace in Veil's Verification Workflow

We illustrate Lace's role in Veil's verification workflow using the FloodSet synchronous consensus protocol [41]. After describing it, we introduce the relevant features of Veil through its specification (Sec. 2.1) and then walk through the use of Lace (Sec. 2.2). The walk-through focuses on the stage that Lace adds to this workflow, namely exhaustive checking of finite instances, showing both

[1] Lace does not yet check general temporal properties, which is an ongoing work.

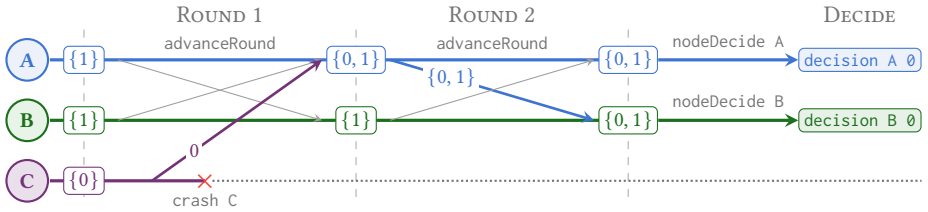


Fig. 1. Example FloodSet execution with three nodes, two values, and $f = 1$. Boxes show each node's seen set at round boundaries. Node C starts with the minimum value 0 but crashes during the first round after sending only to A. Because the second round is crash-free among the surviving nodes, A's knowledge $\{0, 1\}$ reaches B, so both live nodes decide $\min\{0, 1\} = 0$.

of its outcomes: a passing check builds confidence that the model faithfully captures the original protocol where symbolic BMC times out, while a failing check refutes a proposed optimisation with a concrete counterexample, thus avoiding fruitless proof attempts.

The FloodSet protocol. In FloodSet, each node starts with an initial value and records the set of values it has seen. In each round, every alive node broadcasts this set to all the other nodes; after $f + 1$ rounds, where f bounds the number of crashes, each node decides on the minimum value it has seen. A node may crash in the middle of a round, so its values may reach only some nodes. Despite this, the protocol guarantees *agreement* (all deciding nodes choose the same value) and *strong validity* (the decided value is some node's initial value). The $f + 1$ rounds are what make agreement hold: at least one of them is crash-free, and after it all surviving nodes have seen the same values. Fig. 1 illustrates how the second, crash-free round allows the surviving nodes to reach agreement despite a partial broadcast in the first round.

2.1 Modelling FloodSet in Veil

Fig. 2 shows the Veil specification of FloodSet, followed by the commands used in Sec. 2.2. It has three parts: the state and its initialisation (lines 1–25), the actions (lines 27–56), and the safety properties (lines 58–65). Several of these features are the abstractions discussed in Sec. 1: they suit symbolic checks and proofs, but Lace needs efficient, executable counterparts for them.

State. A state consists of *fields* declared as `individual`s, `relation`s, or `function`s, over uninterpreted sorts (*e.g.*, `node` and `value`) or concrete Lean types. Mutable fields record the protocol state, including the values each node has seen (`seen`) and whether it has crashed (`crashed`). Immutable fields, such as the fault bound `f`, form its background theory. The `instantiate` declaration (line 3) assumes a total order on `value`, which `nodeDecide` uses to choose the minimum. A `ghost relation`, such as `crashedPreviously`, is not a field itself, but abbreviates a formula over the fields (hence "ghost"). A field is logically a relation or function, while its executable counterpart in Lace is a runtime data structure, which Sec. 3 keeps consistent with the logical view.

Actions. Actions describe transitions imperatively as monadic programs. The state is initialised by executing the `after_init` block (lines 17–25). A precondition `require P` (line 43) restricts the action to states where `P` holds: elsewhere, the action is not enabled. A failing `assert` inside an action, in contrast, raises an exception. In assignments, variables starting with a capital letter are implicitly ∀-quantified, *i.e.*, `decision N V := false` updates all entries of `decision`. The *pick-such-that* operator `let x :| P` (line 29) [38] non-deterministically chooses a value satisfying the predicate `P` and assigns it to `x`. For example, in `advanceRound`, it chooses a delivery relation with the constraint that alive nodes deliver to all receivers while nodes that crashed in a previous round deliver to none; the delivery for nodes crashed in that round is unconstrained, meaning they can

```
1  type node
2  type value
3  instantiate val_ord : TotalOrder value
4
5  immutable individual f : Nat
6  individual round : Nat
7  relation seen (n : node) (v : value) : Bool
8  function initialValue : node → value
9  relation decision (n : node) (v : value) : Bool
10 function crashed (n : node) : Bool
11 function crashedInRound (n : node) : Nat
12 individual numCrashed : Nat
13
14 ghost relation crashedPreviously (n : node) :=
15   (crashed n ∧ crashedInRound n < round)
16
17 after_init {
18   round := 0
19   initialValue := *
20   seen N V := (V == initialValue N)
21   decision N V := false
22   crashed N := false
23   crashedInRound N := 0
24   numCrashed := 0
25 }
26
27 action advanceRound {
28   require round < f + 1
29   let delivery : (node → node → Bool) :|
30     (∀ (sender : node),
31       (crashedPreviously sender →
32         (∀ receiver, ¬ delivery sender receiver)) ∧
33       (¬crashed sender →
34         (∀ receiver, delivery sender receiver)))
35   -- receive according to delivery relation
36   seen N V := decide (seen N V ∨
37     (¬crashed N ∧ ∃ sender,
38       delivery sender N ∧ seen sender V))
39   round := round + 1
40 }
41 action crash (n : node) {
42   require numCrashed < f
43   require ¬crashed n
44   crashed n := true
45   crashedInRound n := round
46   numCrashed := numCrashed + 1
47 }
48
49 action nodeDecide (n : node) {
50   require round = f + 1
51   require ¬crashed n
52   require ¬(∃ v, decision n v)
53   let v :| seen n v ∧
54     (∀ v', seen n v' → val_ord.le v v')
55   decision n v := true
56 }
57
58 safety [agreement]
59   ∀ n1 n2 v1 v2,
60     decision n1 v1 ∧
61     decision n2 v2 → v1 = v2
62
63 safety [strong_validity]
64   ∀ n v, decision n v →
65     (∃ m, initialValue m = v)
66
67 sat trace {
68   any 4 actions
69   assert (∃ n1 n2 v1 v2, n1 ≠ n2 ∧
70     decision n1 v1 ∧ decision n2 v2)
71 }
72
73 unsat trace {
74   any 4 actions
75   assert ¬ (agreement ∧ strong_validity)
76 }
77
78 #model_check
79   { node := Fin 3, value := Fin 3 }
80   { f := 3 }
```

Fig. 2. Synchronous FloodSet consensus protocol in Veil.

deliver to any subset of nodes. Unconstrained non-deterministic assignment is indicated by `:= *` (line 19), here giving each node an arbitrary initial value. A choice is logically just its predicate, while its executable counterpart must enumerate exactly the values satisfying the predicate, which Sec. 4 guarantees with a machine-checked proof.

Properties. Safety properties are declared with the **`safety`** keyword, or equivalently **`invariant`** (lines 58–65). The later stages of the workflow prove them through inductive invariants, while Lace checks them on every state it explores (Sec. 5).

2.2 Checking FloodSet before Verifying It

It is common to make mistakes when modelling non-trivial protocols such as FloodSet, and it is hard to realise immediately how the model can violate a given property. The workflow originally presented for Veil [53] uses symbolic BMC to check that the specification admits expected executions and to search for safety violations before attempting a proof.

When symbolic BMC is inconclusive. Following this workflow, we issue the reachability and safety queries shown in Fig. 2 (lines 67–76). The reachability check (lines 67–71) asks the solver to find a trace of 4 actions in which two distinct nodes have both decided; the safety check (lines 73–76) asks it to confirm that no trace of 4 actions violates agreement or strong validity. However, for the FloodSet protocol—with its constraints on the delivery relation and the complex quantification of

the strong validity property—even these modest query depths push the SMT solver to its limits. The safety check with a bound of just 4 actions times out after 10 minutes. The timeout leaves us without an answer to the safety query; consequently, proceeding directly to invariant discovery risks spending effort on a specification that may still contain a bug.

Building confidence with Lace. To gain confidence in the specification despite the BMC timeout, we use Lace to exhaustively check a finite instance of FloodSet. Unlike symbolic BMC, Lace explicitly enumerates the reachable states of a finite instance without bounding execution length. It either confirms that the safety properties hold in every reachable state or returns a counterexample trace. The `#model_check` command (lines 78–80) fixes such an instance by instantiating both `node` and `value` with `Fin 3`, a type with three elements, and setting $f = 3$. On this instance, Lace explores all 175,869 reachable states in a few seconds on a commodity laptop and confirms that agreement and strong validity hold. Since FloodSet is a textbook protocol, a violation would have pointed to a mistake in the model; its absence builds confidence that the specification is faithful, at least on this instance. With this confidence, we can continue with the model towards invariant discovery and, eventually, a proof. The checked model also serves as a baseline for trying out changes to the protocol, such as the optimisation we consider next.

Testing an optimisation. Suppose a designer now wants nodes to decide earlier. Deciding after $f + 1$ rounds is necessary in the worst case [41], but in runs without crashes every node has seen all values after a single round. This suggests letting a node decide as soon as all values it has seen agree, as the action `fastDecide` in Fig. 3 does: unlike `nodeDecide`, it may fire after any round, provided that the node has seen a single value. Whether this is correct is not known, so the designer checks it on a small instance, with three nodes, two values, and $f = 1$, as in Fig. 1. Lace reports a violation of agreement within a few seconds, with the trace in Fig. 3. Each block represents a state, with a header recording the action, its arguments, and the non-deterministic choices that led to that state. Expanding a block reveals the values of the state fields. The trace agrees with the run in Fig. 1 up to the end of round 1, when B has seen only $\{1\}$. Instead of waiting for round 2, B now takes `fastDecide` and decides 1, while A later decides 0, thus violating agreement. For this $f = 1$ instance, an agreement violation requires a crash and two decisions. Since two `fastDecide` actions cannot yield different decisions[2], at least one node must use `nodeDecide` after round 2. Thus, a counterexample needs at least five actions: one crash, two round advances, and two decisions. A depth-4 BMC query (*e.g.*, lines 73–76 from Fig. 2) could therefore not find it, even if it finished. Without

```
1  action fastDecide (n : node) {
2    require round > 0
3    require ¬crashed n
4    require ¬(∃ v, decision n v)
5    require ∀ v₁ v₂,
6      seen n v₁ ∧ seen n v₂ → v₁ = v₂
7    let v :| seen n v ∧
8      (∀ v', seen n v' → val_ord.le v v')
9    decision n v := true
10 }
```

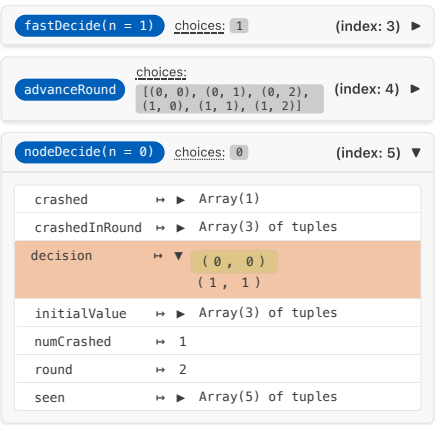


Fig. 3. Top: `fastDecide`, whose shaded guards differ from those of `nodeDecide` (Fig. 2). Bottom: the counterexample (excerpt) to agreement found by Lace after incorporating `fastDecide` into FloodSet.

[2]Both nodes must survive round 1 before executing `fastDecide`, so each receives the other's initial value, and retain both initial values thereafter. Their `seen` sets can therefore both be singletons only if they contain the same value.

the counterexample, the designer would find the flaw only through a failed proof, again unable to tell a wrong rule from a weak invariant; but with it, the rule can be dropped at once.

One semantics across the workflow. In the two examples, we used Lace to build confidence in the original specification and to reject a proposed optimisation. Both uses are meaningful within Veil's workflow only if Lace interprets the specification according to the same semantics as Veil's symbolic checks and proofs. Our correctness theorem (Theorem 5.1) guarantees this agreement, and the rest of the paper shows how this is achieved without giving up performance.

3 Deriving Efficient Field Representations with Type Classes

Veil's state fields are represented as first-order relations and functions—ideal for SMT solvers, but prohibitively inefficient for explicit-state exploration. To solve this issue, we introduce a type-class-based abstraction layer that decouples a state field's logical content from its runtime representation, and a machine-checked *lawfulness* condition that guarantees the abstraction layer is transparent, in the sense of all references to it being eliminated during verification condition generation. As a result, the model checker uses efficient concrete data structures without requiring users to adapt their specification proofs to the chosen lawful field representations.

3.1 Relations and Functions as State Fields

As described in Sec. 2, the state of a Veil specification is organised as a collection of *fields*, and in particular, relation or function symbols whose parameters and codomain may range over uninterpreted sorts (*e.g.*, `node` and `value` in our example). This first-order representation facilitates symbolic model checking and verification, since it aligns well with the capabilities of SMT solvers.

States evolve via assignments to fields. Veil supports both point updates, *e.g.*, `foo a b := 42` where `foo` is a function of type $\alpha \to \beta \to$ `Nat`, and bulk updates using *universally quantified variables* (variables starting with a capital letter), *e.g.*, `foo A b := foo A b + 1` increments `foo a b` by 1 for all values `a : α`. These bulk assignments implicitly encode loops in a compact way.

Efficiency vs symbolic amenability. The first-order representation creates problems for explicit-state model checking. In particular, explicit-state model checkers frequently hash and compare states—for instance, to detect already-visited states. When a field is stored as a function closure, computing its hash requires enumerating the entire domain. For relations, this means evaluating the closure on every domain element to decide membership, even though the relation's support may be small. Functional updates compound the problem: each assignment wraps the previous closure in a new one, building a chain whose lookup cost grows linearly with the number of past assignments—in contrast to the $O(1)$ or $O(\log n)$ access that hash maps or balanced trees provide. As an example, take the `foo a b := 42` assignment. When `foo` is represented as a function closure, it is updated by creating a new closure that wraps the old one: `fun a' b' => if a = a' ∧ b = b' then 42 else foo a' b'`. This is highly inefficient.

On the other hand, naïvely replacing closures with efficient data structures shifts the burden to symbolic tools. All field accesses must then go through data-structure-specific APIs, and SMT solvers cannot reduce these calls without additional axioms. For example, after executing `foo a b := 42` on a hash-map-backed field, reading the updated value involves `HashMap.get` and `HashMap.update`; unless properties like `∀ mp k v, HashMap.get k (HashMap.update mp k v) = v` are supplied, the solver cannot derive that the read returns 42. Even when such lemmas are available, they must be manually supplied during VC generation. Prioritising the logical representation thus paralyses the explicit-state checker, while prioritising efficient runtime paralyses the prover or burdens the user.

```
1 def UpdatePat (Dom : List Type) : Type := IProd (Dom.map Option)
2 def UpdateDescr (Dom : List Type) (Cod : Type) : Type := UpdatePat Dom × CanonicalTy Dom Cod
3 class FieldRepresentation (Dom : List Type) (Cod ConcTy : Type) where
4   get : ConcTy → CanonicalTy Dom Cod
5   set : UpdateDescr Dom Cod → ConcTy → ConcTy
```

Fig. 4. The interface for interacting with fields and some auxiliary definitions.

3.2 An Interface for Field Representations

Our solution is the type class shown in Fig. 4. A `FieldRepresentation` is parameterised by a domain `Dom`, a codomain `Cod`, and a concrete backing type `ConcTy`. `Dom` is a list of types corresponding to the field's parameter sorts. For example, the field `foo : α → β → Nat` has `Dom = [α, β]` and `Cod = Nat`. Relations have `Cod` fixed to `Bool`. The logical representation of a field, denoted as `CanonicalTy Dom Cod`, is defined such that `CanonicalTy [T₁, ⋯, Tₙ] Cod` reduces to the iterated arrow type $\mathtt{T}_1 \to \cdots \to \mathtt{T}_n \to \mathtt{Cod}$. Conveniently, since Lean is based on dependent type theory, `CanonicalTy` can be straightforwardly defined as a right fold over `Dom`. The class provides two operations: `get` (line 4) converts a `ConcTy` value into its logical representation, after which field values can be read by ordinary function application; `set` (line 5) applies an *update description* to a `ConcTy` value and produces a new one, operating directly on the concrete backing representation.

This design separates the logical view of a field—a pure function from `Dom` to `Cod`—from its runtime storage in `ConcTy`: the model checker reads values through `get` and writes them through `set`, never touching the concrete type directly. Specific instances of `FieldRepresentation` then replace pure functions and relations with efficient data structures: for example, a relation `node → value → Bool` is backed by a `HashSet (node × value)`, and a function `node → Nat` by a `HashMap node Nat`.

Update descriptions. Update descriptions are the runtime representation of Veil assignment statements: an assignment such as `seen N V := expr`, where some parameters may be universally quantified, is compiled into an update description and passed to `set`. Concretely, an update description (Fig. 4, line 2) is a pair of an *update pattern* and a right-hand side value. The pattern (line 1) of type `IProd (Dom.map Option)` encodes the left-hand side of the assignment:[3] one optional entry per parameter—`some v` pins that position to the specific value `v`, while `none` means "all values". For `seen N V := expr`, both positions are `none`, so `set` updates every entry; for a single-node update `seen n V := expr`, the first position is `some n` and only the entries for node `n` are touched.

Example. Suppose the field `foo` is backed by a `HashMap`, as indicated by a type class instance called `inst` having type `FieldRepresentation [α, β] Nat HashMap`. The runtime type of `foo` is `HashMap`, while its logical representation is $\alpha \to \beta \to \mathtt{Nat}$. Consider an assignment `foo A b := foo A b + 1`. The occurrence of `foo A b` on the right-hand side is interpreted via the field representation as `(inst.get foo) A b`. The corresponding update pattern (`up`) is `(none, some b)` since `A` is universally quantified. The resulting update description (`ud`) is `(up, fun A b => (inst.get foo) A b + 1)`. Finally, the assignment yields a new value `inst.set ud foo` of type `HashMap`.

3.3 Lawfulness of Concrete Field Representations

The `FieldRepresentation` interface alone does not guarantee soundness: an implementation could adhere to the type signatures while exhibiting behaviour that diverges from the intended semantics, such as, for instance, silently discarding updates. To rule this out, we require every instance to satisfy the `get_set` property defined in the `LawfulFieldRepresentation` type class, shown in Fig. 5 (line 3). It states that performing a concrete update and then lifting to the logical level gives the same result as first lifting and then performing the corresponding logical

[3] In Lean, `IProd [T₁, ⋯, Tₙ]` denotes the Cartesian product $\mathtt{T}_1 \times \cdots \times \mathtt{T}_n$; `IProd []` reduces to `Unit`.

```
class LawfulFieldRepresentation (Dom : List Type) (Cod ConcTy : Type)
  [inst : FieldRepresentation Dom Cod ConcTy] where
  get_set : ∀ (conc : ConcTy) (ud : UpdateDescr Dom),
    inst.get (inst.set ud conc) = CanonicalTy.set ud (inst.get conc)
```

Fig. 5. The type class stating the lawfulness constraint of field representation.

update via `CanonicalTy.set`. Put differently, `get` is a *homomorphism* with respect to `set`. Here, `CanonicalTy.set` is the functional update with respect to the update description. For example, when `Dom` has size 3, `CanonicalTy.set ((some a, none, some c), f) g` reduces to the function `fun a' b' c' => if a = a' ∧ c = c' then f a' b' c' else g a' b' c'`.

Why one law suffices. Notably, the `get_set` law is the *only* requirement needed to make the field representation transparent, eliminating it completely during verification condition (VC) generation. The key insight is that `get_set` can be read as a rewrite rule: any expression of the form `inst.get (inst.set upd conc)` can be replaced by `CanonicalTy.set upd (inst.get conc)`, pushing `inst.get` one layer inward and replacing the concrete `inst.set` with the logical `CanonicalTy.set`. Applying this rule repeatedly to an expression `inst.get (inst.set (inst.set ⋯))` drives `inst.get` all the way to the innermost concrete value and replaces every concrete write with a logical one. `CanonicalTy.set` expands into a simple conditional without any dependence on the concrete backing type, so after this rewriting all `inst.set` calls have been eliminated and the formula mentions the concrete backing type only through remaining `inst.get` reads.

Once only reads remain, we can go further. Consider any field `foo : ConcTy` universally quantified in the VC (fields are universally quantified because a VC must hold for all states, not a specific one): since `inst.get foo` has exactly the type `CanonicalTy Dom Cod`, we can replace `foo` with a fresh variable `foo' : CanonicalTy Dom Cod` and drop `inst.get` entirely. For example, a VC `P (inst.get foo)` with `foo : HashMap` becomes `P (foo')` with `foo'` ranging over the logical function type—exactly the VC one would obtain without any field representation. This rewrite-then-generalise procedure thus makes the entire field representation layer transparent to the prover.

3.4 A Library of Lawful Field Representations

We have implemented a collection of lawful field representation instances covering common modelling scenarios. Users can select which instance to use for each field in a specification, and the corresponding concrete backing is used automatically during model checking; if no instance is specified, the efficient instance based on Lean's `ExtTreeMap` is used.

Canonical instance. As a baseline, `CanonicalTy Dom Cod` admits a trivial instance where `get` is the identity and `set` is `CanonicalTy.set`. This preserves the purely logical view of fields and introduces no additional abstraction overhead. We use this instance in our ablation study (Sec. 6.3), to represent the fallback case where no specific field representation is used.

Individuals. When `Dom` is empty, a field degenerates to a single value and `Cod` itself serves as the backing type. The field representation instance and its lawfulness are similarly trivial.

Maps and sets. Relations and multi-argument functions are the most common field kinds in Veil, and they are naturally backed by map-like data structures such as hash maps or sorted trees. Rather than requiring users to re-prove a full `LawfulFieldRepresentation` from scratch for every such structure, we provide a generic derivation: it suffices to instantiate two lightweight type classes (Fig. 6), and the framework automatically promotes them to a certified field representation. `MapRep` is an interface for a map with keys of type α, values of type β, and backing store of type γ, supporting

lookup via `mapGet` and update via `mapSet`. Given a `MapRep` instance, we derive a `FieldRepresentation` by treating tuples of domain elements (`IProd Dom`) as keys and `Cod` as values. The `get` operation is `mapGet` applied to the uncurried key tuple. The `set` operation first computes the *footprint* of an update pattern—the set of keys that the assignment actually touches—and then calls `mapSet` once for each key in the footprint. The footprint is the Cartesian product induced by the pattern: position i contributes the full set of the i-th sort when marked `none`, or a singleton when marked `some v`.[4] For example, for `Dom = [α, β]` and pattern `(none, some b)`, the footprint is all pairs `(a, b)` for every `a : α`, so every row for that fixed `b` is updated. To lift this to a *lawful* field representation, the user additionally provides a `LawfulMapRep` instance, which requires only a single get-after-set law (Fig. 6). The framework then derives `LawfulFieldRepresentation` automatically by iterating this law over the footprint. The same derivation works for relation fields by fixing `Cod = Bool`.

```
class MapRep (α β γ : Type) where
  mapGet : γ → α → β
  mapSet : α → β → γ → γ
class LawfulMapRep
  [inst : MapRep α β γ] where
  map_set_get : ∀ a a' b mp,
    inst.mapGet (inst.mapSet a b mp) a'
    = if a = a' then b
                else inst.mapGet mp a'
```

Fig. 6. Classes for map-like data structures, used to derive lawful field representations.

Through this mechanism, we have registered common Lean structures as lawful field representations, including bit vectors, arrays, hash maps, and binary search trees. In practice, users rarely need to choose: the `ExtTreeMap` instance is registered as the default for all map and relation fields, and most specifications work well with it out of the box. Switching to a different structure is a one-line annotation on the field declaration and requires no changes to the specification or proofs.

4 Enumerative Execution via Multi-Extraction

Sec. 3 pairs logical state fields with efficient runtime representations while preserving their logical view for symbolic reasoning. We now address the other abstraction identified in Sec. 1: nondeterministic choices specified by predicates. Their executable counterparts must enumerate all admissible choices and their outcomes. In this section, we introduce *multi-extraction*, which turns each Veil action into an executable computation, together with a machine-checked proof that its enumeration is sound and complete with respect to the action's axiomatic semantics. The resulting successor functions are the input to the search algorithm in Sec. 5.

4.1 Mono-Extraction versus Multi-Extraction

Our work builds on the revised version of Veil [53] presented by Gladshtein et al. [18], where each action is in the `VeilM` monad (Fig. 7). It is built in two layers. The inner layer, `VeilExecM` (line 3), is a deterministic, executable monad: unfolding its definition, a `VeilExecM` computation, of type $\rho \to \sigma \to$ `DivM` (`Except ExId` $\alpha \times \sigma$), takes an immutable background theory ρ and a mutable state σ, and either diverges, raises an exception (paired with the state at failure), or returns a value (paired with the final state). The outer layer, `NonDetT` (lines 4–7), adds non-determinism.[5]

A `NonDetT` value is a syntax tree: `pure` injects values, `vis` invokes effects from the inner monad, and `pickCont` chooses a value of type τ satisfying the predicate `p` and passes it to the continuation `f`. Because `NonDetT` is syntax, a `VeilM` program is *not* directly executable—it must be *extracted* into

[4] This requires every sort in `Dom` to be finite; otherwise the universal set would be infinite and the footprint uncomputable. All sorts used in our benchmarks (Sec. 6) satisfy this condition. Fields over infinite sorts can still be represented using the canonical instance, which stores the field as a plain Lean function and requires no finiteness.

[5] In the code, both `VeilM` and `VeilExecM` carry an additional *mode* parameter, which we elide for clarity. Compared to [18], we swap the order of `StateT` and `ExceptT` in the monad stack, so that exceptions return the state when they are thrown.

```
inductive Except (ε α : Type) where | error (e : ε) | ok (a : α)
inductive DivM (α : Type) where | res (x : α) | div
def VeilExecM (ρ σ α : Type) := ReaderT ρ (ExceptT ExId (StateT σ DivM)) α
inductive NonDetT (m : Type → Type) : (α : Type) → Type where
  | pure (x : α) : NonDetT m α
  | vis {β} (x : m β) (f : β → NonDetT m α) : NonDetT m α
  | pickCont (τ : Type) (p : τ → Prop) (f : τ → NonDetT m α) : NonDetT m α
def VeilM (ρ σ α : Type) := NonDetT (VeilExecM ρ σ) α
```

Fig. 7. Definitions related to `VeilM`. Type universe level parameters are elided for simplicity.

a concrete monad by resolving every `pickCont` node. In other words, we must resolve the non-determinism by picking concrete values that satisfy each predicate, and then execute the resulting fully-determined continuations. If no value exists to satisfy the predicate, the computation diverges.

Extraction. Prior work proposed a technique that resolves each `pickCont` by committing to a single value, producing a single `VeilExecM` computation [18]. This technique is sound—it produces only computations allowed by the non-deterministic semantics—but not complete. To enable model checking of Veil specifications, we must instead enumerate *all* possible non-deterministic choices and *all* outcomes produced by these choices, rather than committing to a single execution result. For instance, if an action includes a non-deterministic choice like "pick an arbitrary x such that $0 \leq x \leq 100$", we would like its operational semantics to be returning the union of outcomes corresponding to all values of x in $[0, 100]$, instead of the outcome associated with one specific choice. To clarify this distinction, we refer to the previously implemented extraction [18] as *mono-extraction*, and the interpretation strategy required for model checking as *multi-extraction*. While mono-extraction is sound, multi-extraction must be both *sound and complete*: a post-state `s'` must appear among the results of executing the multi-extracted action from a starting state `s` *if and only if* `s` and `s'` are related by the transition relation induced by the action.

4.2 The Monad for Enumerative Execution

To realise the sound-and-complete multi-extraction semantics above, we need a concrete monad whose computations return all possible outcomes at once. The key observation to construct such a monad is that any extraction recursively maps a `NonDetT m α` program into a program of type `m' α`, where `m'` is another monad. Mono-extraction uses `VeilExecM` (Fig. 7) as `m'`, but `VeilExecM` can only return one outcome. For multi-extraction, we need a target monad that returns all outcomes at once, and a procedure to produce programs in it. Fortunately, when `m` is `VeilExecM`, such a target monad does exist. This monad, denoted as `LaceM`, equips `VeilExecM` with the ability to maintain *multiple* execution paths *simultaneously*. We define it as follows:

```
def LaceM (κ ρ σ α : Type) := ρ → σ → List (List κ × DivM (Except ExId α × σ))
```

The interpretation of `LaceM` closely mirrors that of `VeilExecM`: a `LaceM` computation takes a background theory (of type ρ) and an initial state (of type σ), and returns a *list* of outcomes. Each outcome pairs a value of type `DivM (Except ExId α × σ)` that represents the same three possible results as in `VeilExecM` with a `List κ`, which we call a *choice log*. A choice log records the sequence of non-deterministic choices made by the computation to arrive at the corresponding outcome, with each choice represented as a value of type κ. The type κ is a parameter left abstract in `LaceM`; in Lace it is instantiated to a type that encodes the witnesses resolving each `pickCont` node, so that a counterexample trace can be reconstructed from the log of a failing execution. The overall structure of `LaceM`—replacing a single outcome with a list of all outcomes—instantiates the classical idea of replacing failure by a list of successes [70] in the setting of a stateful, effectful monad.

```
instance : Monad (LaceM κ ρ σ) where
  pure a := fun _ st => [([], DivM.res (Except.ok a, st))]
  bind m f := fun th st => (m th st).flatMap fun (log, oc) =>
      match oc with
      | Div.div                      => [(log, DivM.div)]
      | Div.res (Except.error e, st') => [(log, DivM.res (Except.error e, st'))]
      | Div.res (Except.ok a   , st') => (f a th st').map fun (log', oc') => (log ++ log', oc')
```

Fig. 8. The pure and bind operations of LaceM.

LaceM forms a monad, as in Fig. 8. The pure constructor lifts a plain value into a single-outcome, successful computation that leaves the state unchanged and records no choice; bind sequences computations via flatMap: diverging or erroneous outcomes propagate unchanged, while each successful outcome feeds its result and intermediate state to the continuation, concatenating the choice logs. In effect, LaceM extends VeilExecM with two capabilities realised by List: executable non-determinism (multiple outcomes) and a writer effect (accumulating choice logs).

4.3 Multi-Extraction Made Single-Pass and Modular

With LaceM established as the target monad, it remains to show how to systematically transform VeilM programs into LaceM programs; our starting point is mono-extraction, shown in Fig. 9.

4.3.1 Two-pass mono-extraction. Mono-extraction cannot serve as multi-extraction for two independent reasons. First, and fundamentally, it resolves each pickCont node to *one* witness value, producing a single outcome; to enumerate all reachable states we need it to resolve each choice to *all* satisfying values. Second, its two-pass design introduces performance problems that make it impractical for model checking even setting aside completeness. We describe both problems in turn, then show how our single-pass multi-extraction design addresses them together.

Mono-extraction operates in two passes. In the first pass, a meta-level tactic constructs a value of the inductive type Extract (lines 1–13) from the input NonDetT program. Extract mirrors the structure of NonDetT, but pairs every pickCont node with a *witness* that resolves the predicate p. The witness type is a parameter; for mono-extraction, it is instantiated to SingleChoice (lines 14–16), which supplies an Option value together with a soundness proof. Since the original NonDetT term carries no witnesses, the tactic resolves them via Lean's type class instance inference during elaboration. In the second pass, the recursive function monoExtract (lines 17–26) traverses the Extract structure and produces a computation in the target monad m: it threads through pure and

```
1  inductive Extract {m}
2    {Witness :
3      {τ : Type} → (p : τ → Prop) → Type}
4    : {α} → NonDetT m α → Type where
5    | pure (x : α) : Extract (NonDetT.pure x)
6    | vis (x : m β) (f : β → NonDetT m α)
7      (ex : (y : β) → Extract (f y)) :
8      Extract (NonDetT.vis x f)
9    | pickCont (τ) (p : τ → Prop)
10     (f : τ → NonDetT m α)
11     (w : Witness τ p)
12     (ex : (t : τ) → Extract (f t)) :
13     Extract (NonDetT.pickCont τ p f)
14 class SingleChoice {τ} (p : τ → Prop) where
15   val : Option τ
16   sound : ∀ v, val = some v → p v
17 def monoExtract (s : NonDetT m α)
18   (ex : Extract (Witness := SingleChoice) s)
19   : m α := match ex with
20   | Extract.pure a => pure a
21   | Extract.vis x f ex => bind x fun y =>
22     monoExtract (f y) (ex y)
23   | Extract.pickCont τ p f w ex =>
24     match w.val with
25     | some v => monoExtract (f v) (ex v)
26     | none   => ⊥
```

Fig. 9. Implementation of mono-extraction [18]. Here ⊥ denotes a m α value that represents divergence. For m being VeilExecM, ⊥ is fun _ _ => DivM.div.

vis nodes directly, and at each pickCont node, it inspects the witness to determine whether a value was chosen (the some case) or not (the none case, which produces divergence).

This design, however, incurs runtime costs that are difficult to eliminate, making it unsuitable for model checking. First, executing large Extract terms leads to excessive closure allocation at runtime, which was severe enough to crash both Lean's interpreter and their compiled binaries.[6] Second, it is difficult to partially evaluate monoExtract on a given Extract value, so as to eliminate the Extract layer and produce a direct computation in m. To see why, we note that NonDetT, as a monad transformer, supports a bind operation shown in Fig. 10, which recurses down the structure of its first NonDetT argument until it reaches a leaf (NonDetT.pure) node, where it applies the continuation f. As mentioned above, a NonDetT.bind in the source program is mapped to a similar recursive function Extract.bind. Partially evaluating monoExtract over such a term would require unfolding these recursive definitions, which is difficult to control.

```
def NonDetT.bind
  (x : NonDetT m α)
  (f : α → NonDetT m β)
  : NonDetT m β :=
  match x with
  | NonDetT.pure a => f a
  | NonDetT.vis x f' =>
    NonDetT.vis x fun y =>
      NonDetT.bind (f' y) f
  | NonDetT.pickCont τ p f' =>
    NonDetT.pickCont τ p fun t =>
      NonDetT.bind (f' t) f
```

Fig. 10. Definition of NonDetT.bind

4.3.2 Multi-extraction made single-pass. Our design collapses the two passes into one by bypassing the intermediate Extract structure and producing a LaceM value directly.

The extraction relation. To define what a valid extraction is, we introduce the inductive proposition ExtractsTo, shown in Fig. 11, which relates a source VeilM program to its extracted LaceM counterpart. Its constructors specify, for each syntactic form of VeilM, what the extracted program must look like. The pure case is the same as mono-extraction. The vis case (lines 27–30) specifies that the LaceM counterpart of NonDetT.vis x f is a bind, where x is lifted into a singleton-outcome computation with an empty choice log by liftEffect (lines 12–14), and the continuation f' is the pointwise extraction of f. The pickCont case (lines 31–36) specifies that the LaceM counterpart of NonDetT.pickCont τ p f is pickAll p f', where f' is again the pointwise extraction of f.

Enumerating all choices. The key challenge at a pickCont node is to produce one outcome per satisfying value of the predicate p, while recording each choice in the log so that a failing trace can later be reconstructed. The helper pickAll (lines 15–21) relies on two type classes: MultiChoices (lines 1–4) and ChoiceLog (lines 5–6). MultiChoices κ p provides a list vals of all values satisfying p, a proof that vals is sound and *complete* with respect to p, and a function toLogEntry that converts each chosen value into a log entry of type κ. Since model checking operates over finite instances, requiring a sound and complete enumeration is not overly restrictive. ChoiceLog κ m provides an operation addEntry that records a log entry in the underlying monad; its LaceM instance implements addEntry k analogously to pure (), except that its choice log is the singleton [k], with log concatenation handled by bind. The function pickAll combines these: it iterates over vals, records each choice via addEntry, and concatenates the results using flatMap.

A multi-extraction result for a given VeilM term s is bundled as a MultiExtractResult (lines 37–39), pairing a LaceM value res with a proof that ExtractsTo s res holds. The top-level extraction function (lines 40–42) simply returns h.res, where h is a MultiExtractResult that Lean synthesises automatically using the multi_extract tactic (described below).

4.3.3 Multi-extraction made modular. Once we prove that ExtractsTo s s' holds for a certain s, s' pair, we can package this theorem as a new MultiExtractResult entry. As an example, consider the monadic bind operation on VeilM (*i.e.*, NonDetT.bind from Fig. 10). Concretely, the theorem ExtractsTo.bind in Fig. 12 states that if s extracts to s' and f extracts pointwise to f', then

[6] This was a bug in Lean, which we reported and which has been confirmed and fixed. We omit the reference for anonymity.

```
class MultiChoices {τ} (κ p) where
  vals : List τ
  toLogEntry : τ → κ
  sound_complete : ∀ v, v ∈ vals ↔ p v
class ChoiceLog (κ m) where
  addEntry : κ → m Unit
instance : ChoiceLog κ
    (LaceM κ ρ σ) where
  addEntry k := fun th st =>
    [([k], DivM.res (Except.ok (), st))]

def liftEffect (x : VeilExecM ρ σ α)
  : LaceM κ ρ σ α :=
  fun th st => [([], x th st)]
def pickAll (p) [mc : MultiChoices κ p]
  [cl : ChoiceLog κ (LaceM κ ρ σ)]
  (f' : τ → LaceM κ ρ σ α)
  : LaceM κ ρ σ α :=
  fun th st => mc.vals.flatMap fun t =>
    let logAct := cl.addEntry (mc.toLogEntry t)
    (bind logAct fun _ => f' t) th st
```

```
inductive ExtractsTo {κ ρ σ}
  : {α} → VeilM ρ σ α →
    LaceM κ ρ σ α → Prop where
  | pure :
    ExtractsTo (NonDetT.pure x) (pure x)
  | vis (x f f') :
    (∀ y, ExtractsTo (f y) (f' y)) →
    ExtractsTo (NonDetT.vis x f)
               (bind (liftEffect x) f')
  | pickCont (τ p f f')
    [MultiChoices p]
    [ChoiceLog κ (LaceM κ ρ σ)] :
    (∀ t, ExtractsTo (f t) (f' t)) →
    ExtractsTo (NonDetT.pickCont τ p f)
               (pickAll p f')
structure MultiExtractResult (κ s) where
  res   : LaceM κ ρ σ α
  proof : ExtractsTo s res
def multiExtract (s : VeilM ρ σ α) (h :
  MultiExtractResult κ s := by multi_extract)
  : LaceM κ ρ σ α := h.res
```

Fig. 11. Multi-extraction. Unqualified `pure` and `bind` refer to the `Monad` instance of `LaceM` (*cf.* Fig. 8).

`NonDetT.bind s f` extracts to `LaceM.bind s' f'`. The proof proceeds by induction on `hs`, which is straightforward since `ExtractsTo` is inductively defined. This theorem can then be packaged into a `MultiExtractResult` for `bind`, as the one in Fig. 12 demonstrates, which can in turn be applied elsewhere. The same pattern, namely proving an `ExtractsTo` theorem and then packaging it, applies to any composite construct, thereby enabling modular multi-extraction. For example, to extract a `bind` expression, it is not necessary to reduce it to a normal form; instead, it suffices to extract the two operands independently and compose the results with `MultiExtractResult.bind`.

The `multi_extract` tactic automates the construction of `MultiExtractResult` terms. It maintains a database of `MultiExtractResult` entries—initially just those for the three constructors of `ExtractsTo`—and works like a top-down proof search: given a goal `MultiExtractResult s`, it matches the outermost constructor of `s` against the database, applies the matching entry, and recurses on the resulting subgoals. Users can register their own entries to handle domain-specific constructs or to cache results for sub-programs shared across multiple actions. Together, the inductive relation `ExtractsTo`, the certified combinator `MultiExtractResult.bind`, and the `multi_extract` tactic give a modular extraction pipeline: large programs are extracted by composing independently certified pieces, with no global traversal and no trust placed in the tactic—correctness is guaranteed by the proof carried in each `MultiExtractResult`.

```
theorem ExtractsTo.bind {s : VeilM ρ σ α}
  {s' : LaceM κ ρ σ α}
  {f : α → VeilM ρ σ β}
  {f' : α → LaceM κ ρ σ β}
  (hs : ExtractsTo s s')
  (hf : ∀ x, ExtractsTo (f x) (f' x))
  : ExtractsTo (NonDetT.bind s f) (bind s' f')
def MultiExtractResult.bind
  (hs : MultiExtractResult κ s)
  (hf : ∀ x, MultiExtractResult κ (f x))
  : MultiExtractResult κ (NonDetT.bind s f) :=
  res := bind hs.val fun x => (hf x).val
  proof := (ExtractsTo.bind hs.proof
    (fun x => (hf x).proof)).proof
```

Fig. 12. Modular multi-extraction for `NonDetT.bind`.

4.3.4 Post-extraction simplification. The modular approach described above implies that the `LaceM` value returned by `multiExtract` may not yet be fully explicit: it can still contain references to

the `res` fields of intermediate `MultiExtractResult` terms. However, each such reference should be either a concrete `LaceM` expression (from the constructors of `ExtractsTo`) or a composition of sub-results. Therefore, unfolding all these definitions should yield a fully expanded `LaceM` term with no residual abstractions. To perform this unfolding in a controlled manner, we annotate every `MultiExtractResult` definition in the database with a custom `simp` attribute `multiExtractSimp`. After the `multi_extract` tactic produces a `MultiExtractResult`, we invoke Lean's simplifier restricted to this attribute, simplifying all intermediate definitions to obtain a fully explicit `LaceM` computation.

4.4 Soundness and Completeness of Multi-Extraction

The proof carried by `MultiExtractResult` formally guarantees that the extracted result preserves the semantics from the source `VeilM` term. Concretely, we have the following theorem:

THEOREM 4.1. *Let* `act` *be a* `VeilM` *action and let* `act'` *be a* `LaceM` *computation such that the Lean relation* `ExtractsTo act act'` *holds. Then for any background theory* `th` *and initial state* `st`*:*

(1) (Normal state transitions.) *A final state* `st'` *appears among the outcomes of* `act' th st` *if and only if* `st` *and* `st'` *are related by the two-state transition relation induced by* `act`*.*

(2) (Exceptions.) *An exception identifier* `e` *appears among the outcomes of* `act' th st` *if and only if the triple describing that "starting from* `th` *and* `st`*,* `e` *can be triggered under angelic non-determinism" holds true under the total correctness semantics [18].*

Both properties have been proved by induction on `ExtractsTo`. Together, they establish that the outcomes of a multi-extracted `LaceM` computation are *exactly* the states reachable by the original `VeilM` action, giving a Lean proof that applying `multiExtract` to each action computes exactly its successor states—the formal justification on which the model-checking algorithm is built.

5 Proof-Carrying Explicit-State Model Checking

Lean's logical core is designed for reasoning about pure, total functions: propositions and proofs inhabit a fragment without effects such as I/O or non-termination. As a practical programming language, however, Lean also provides a full `IO` monad for effectful code and a task-parallel runtime. A practical model checker needs both: it must report progress, measure elapsed time, handle user interruption, and exploit parallelism for large state spaces. Embedding the search loop in `IO` creates a verification tension: the `IO` primitives it relies on, such as `IO.asTask` for spawning parallel workers, are implemented natively and are opaque to Lean's logic. Once a worker is spawned, the logic cannot relate the result it returns to the computation that produced it, so the standard *extrinsic* approach—proving correctness as separate theorems after the fact—cannot establish any property of that result. In this section, we outline the parallel search algorithm of Lace, and describe how we verify its correctness despite the `IO` barrier by adopting a *proof-carrying* design [46]: the runtime search context and the return types of effectful functions carry Lean proof terms as first-class fields, so invariants are established by construction rather than proved after the fact.[7]

5.1 Parallel Search Algorithm

To explore large state spaces efficiently, Lace uses a parallel level-synchronised breadth-first search (BFS) following the MapReduce paradigm [8], as shown in Algorithm 1. The search maintains a *search context* $\mathcal{S}$ consisting of the *seen set* of all states discovered so far, the *frontier* $\mathcal{F}$ of discovered states that have not yet been expanded (*i.e.*, had their successors computed), a *log* mapping each discovered state to its predecessor, from which counterexample traces are reconstructed, and the termination status. To save memory, the seen set and the log store *fingerprints* of states (typically

[7] In an extrinsic approach, data structures carry only data, and correctness properties are stated and proved as separate theorems about the functions that manipulate them. In the intrinsic approach, invariants are embedded directly in the fields of the data structures, so any function that constructs or modifies a value must discharge them by construction.

Algorithm 1 ParallelBFS(C, $\mathcal{T}_{\text{sys}}$): Parallel level-synchronised breadth-first search

```
1:  Input: Transition system T_sys and search configuration C
2:  Output: Search result S
3:  S ← Initialize(T_sys)
4:  while ¬ S.hasFinished do
5:      F ← S.frontier                          ▷ Discovered states not yet expanded
6:      if F = ∅ then
7:          S ← {S with hasFinished := true}
8:          return S                            ▷ All reachable states explored
9:      k ← if |F| < C.threshold then 1 else C.nTasks   ▷ Adaptive parallelism
10:     F_1, ..., F_k ← SplitList(k, F)
11:     for all i ∈ {1, ..., k} in parallel do  ▷ Map step
12:         L_i ← ProcessChunk(T_sys, F_i, S.seen)
13:     S ← MergeAndReduce(S, L_1, ..., L_k)    ▷ Parallel reduce step
14: return S
```

fixed-size hashes, as in TLC) rather than the states themselves. The *depth* of a state is its distance from the initial states. Each iteration expands all states of one depth, so at the start of iteration d the frontier consists exactly of the states at depth d (hence *level-synchronised*). If $\mathcal{F}$ is smaller than a configurable threshold, the workload is processed by a single worker ($k = 1$); otherwise, SplitList partitions it into k chunks $\mathcal{F}_1, \ldots, \mathcal{F}_k$. In the map step, threads concurrently expand the states in their chunks and check safety properties, using the seen set to skip post-states discovered in earlier iterations. Note that different threads might discover the same post-state. The MergeAndReduce step aggregates the per-thread results $\mathcal{L}_i$ into $\mathcal{S}$. Concretely, it de-duplicates post-states discovered by the threads (a sequential operation) to obtain the frontier for the next iteration, and adds them to the seen set (a parallel operation: the seen set is split by fingerprint into $C.nTasks$ separate *shards*, which can be updated in parallel).

This level-synchronised BFS design has two appealing properties. First, the map step is embarrassingly parallel: each thread operates on a disjoint chunk of the frontier and requires no coordination. Second, every reported counterexample trace has minimal length, since states are discovered depth by depth. An alternative is depth-first search (DFS), which can find counterexamples far from the initial states sooner; however, it yields longer traces, and its search order is inherently sequential [55], leaving no natural synchronisation point for parallel workers. We chose level-synchronised BFS as the simplest design that is both practically effective and amenable to intrinsic verification: the synchronisation barrier between iterations provides a single point at which the global invariant must hold, making the correctness argument inductive over iterations.

5.2 Intrinsic Verification of Effectful Code

The property we want to establish is that ParallelBFS is both *sound*—every state it explores is reachable—and *complete*—every reachable state is eventually explored. We reduce both to an inductive invariant ψ of the search context, which holds for the initial context and is preserved by every iteration. Soundness is one of its conjuncts: every frontier state is reachable. Since safety properties are checked on frontier states as they are expanded, every reported violation is a true violation. Another conjunct states that every expanded state has all its successors in the seen set; once the frontier is empty, this makes the seen set closed under successors, which yields

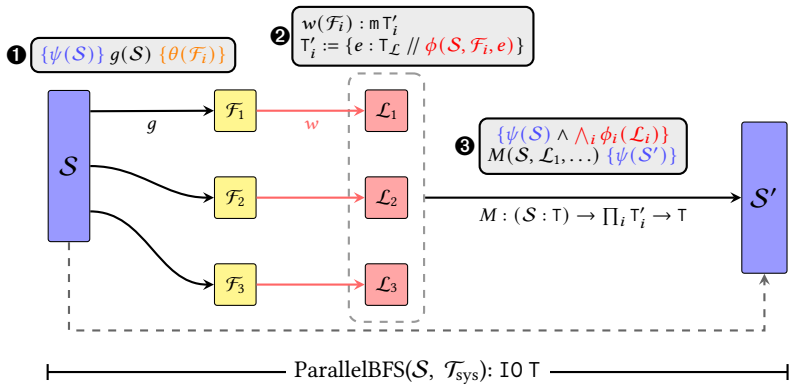


Fig. 13. Intrinsic proof outline for one iteration of ParallelBFS with 3 threads. The search contexts $\mathcal{S}$ and $\mathcal{S}'$ before and after the iteration have type T; $\mathcal{F}_i$ are the chunks of the frontier of $\mathcal{S}$, and $\mathcal{L}_i$ are the workers' results. ψ is the conjunction of the invariants from which Theorem 5.1 follows, θ is the local precondition of the workers, and ϕ_i abbreviates $\phi(\mathcal{S}, \mathcal{F}_i, \cdot)$. Since ϕ_i refers to $\mathcal{S}$ and $\mathcal{F}_i$, each result type T'_i depends on them, and M takes $\mathcal{S}$ as an input (dashed arrow).

completeness. The difficulty is that the loop obtains each worker's result through `IO.asTask`, so we cannot show extrinsically that an iteration preserves ψ.

Our approach has two ingredients. The first is to *embed* proof obligations directly into the search context $\mathcal{S}$ as structure fields: $\mathcal{S}$ combines the runtime data described above with *proof terms* that certify ψ. Because these proofs are ordinary Lean fields, the type checker forces every function that constructs or modifies $\mathcal{S}$ to discharge them at the call site, making it impossible to silently break an invariant. The second ingredient handles functions that run inside `IO`: when such a function returns a value of type B that we cannot observe from the logic, we make its return type the Lean *subtype* $\{e : \mathsf{B} \mid\!/\!/ \phi(e)\}$, where ϕ may mention the function's arguments, so any returned value is paired with a proof of ϕ irrespective of the monadic wrapper. Fig. 13 shows how these ingredients interact across one BFS iteration. ❶ At the start of the map step, a splitting function g (SplitList in Algorithm 1) partitions the frontier of $\mathcal{S}$ into chunks $\mathcal{F}_i$, with a proof, derived from ψ, that each chunk satisfies the local precondition θ of the workers: $\{\psi(\mathcal{S})\}\ g(\mathcal{S})\ \{\theta(\mathcal{F}_i)\}$. ❷ Each thread runs a monadic function w (ProcessChunk) on its chunk, assuming $\theta(\mathcal{F}_i)$. The return type T'_i is a subtype carrying $\phi_i(e) = \phi(\mathcal{S}, \mathcal{F}_i, e)$, which refers to the seen set of $\mathcal{S}$ and to the chunk, so T'_i is a dependent type and differs from thread to thread, and the result is certified independently of the monad. ❸ The reduce step merges the local results into $\mathcal{S}$ via a pure function M (MergeAndReduce). Since M constructs a new search context, the first ingredient requires it to re-establish ψ, which it does from $\psi(\mathcal{S})$ together with the ϕ_i that each $\mathcal{L}_i$ carries in its type: $\{\psi(\mathcal{S}) \wedge \bigwedge_i \phi_i(\mathcal{L}_i)\}\ M(\mathcal{S}, \mathcal{L}_1, \ldots)\ \{\psi(\mathcal{S}')\}$.

Concretely, ProcessChunk runs in `IO` and returns a local result $\mathcal{L}_i$ whose type carries ϕ_i. Beyond what $\psi(\mathcal{S})$ already guarantees, ϕ_i records in particular that every successor of a state in $\mathcal{F}_i$ is either already seen or discovered by $\mathcal{L}_i$: exactly what M needs to re-establish the conjunct of ψ that yields completeness. This proof is discharged when the return value is built, so the reduce step can use ϕ_i directly: it is a claim about the returned *value* only, so `IO` cannot invalidate it.

Together, these steps let us verify each component independently and compose the proofs into a correctness argument for a program that runs in `IO`, even though every worker's result passes through the opaque `IO.asTask`. Combined with Theorem 4.1, which certifies that multi-extraction computes exactly the set of successor states, and the lawfulness conditions on the concrete field representations (Sec. 3), this delivers the following end-to-end guarantee:

Theorem 5.1 (Correctness of Lace). *The correctness of Lace consists of:*

- ***Soundness.*** *Every state explored by Lace is reachable in the transition system induced by the Veil specification. Any property violation reported is therefore a true violation.*
- ***Completeness.*** *If no two distinct reachable states share a fingerprint, then whenever Lace terminates because the frontier is empty, rather than stopping early (*e.g.*, on user interruption, on finding a violation, or on reaching the user-configured depth bound), every reachable state in the transition system has been explored.*

Put differently, whenever the premises of completeness hold, Lace explores a state if and only if that state is reachable under the *transition relation* induced by the Veil specification (Sec. 4.1).

Fingerprints and hash collisions. The fingerprint function is supplied through a type class and defaults to a hash of the state. Soundness holds for any fingerprint function: every explored state is an initial state or a successor of an explored state, deduplication can only discard states, and every reported counterexample trace is recovered from the log and then re-executed against the transition system. Completeness needs the fingerprint function to be injective only on *reachable* states: if two of them collide, the one discovered later is pruned, together with any states reachable only through it. A collision can thus make Lace miss a violation but never report a spurious one—the same trade-off as TLC's fingerprinting [75]. A fingerprint function that keeps the whole state meets this requirement at the cost of memory.

6 Evaluation

The preceding sections establish Lace's semantic guarantees. We now evaluate its practicality and performance for exhaustive safety checking of finite instances within Veil. We use existing TLA$^+$ specifications as benchmarks developed independently of our work. Translating these specifications into Veil lets us assess the range of models that Lace can check and compare its performance with TLC [75], a well-established explicit-state model checker for TLA$^+$, on the same task of exhaustive safety checking. We also use protocols written directly in Veil to measure the performance benefits of concrete field representations. Specifically, we address the following research questions:

RQ1: Can Lace exhaustively check finite instances of protocol models written independently of this work, and what must users supply to do so?
RQ2: How does Lace's performance compare with that of TLC on the same finite instances?
RQ3: To what extent do concrete field representations improve the performance of explicit-state model checking, compared to the logical ones?

All experiments described in this section were conducted using a commodity laptop: MacBook Pro with an Apple M4 Pro chip and 24 GB of RAM, running macOS 26.5.2. Lace used Lean v4.32.0. We used TLC at Git revision `8f4bc8b`[8] running on Java SE 17 with a fixed heap size of 20 GB (`-Xms20G -Xmx20G`) and the enabled parallel garbage collector (`-XX:+UseParallelGC`).

6.1 RQ1: Applicability and User Requirements

We translated 28 TLA$^+$ specifications written independently of this work into Veil. As summarised in Tab. 1, they cover concurrent algorithms, storage systems, and distributed protocols, together with smaller transition-system examples. The suite includes both models used to check safety properties and models with known violations. The table also lists RingFin and FloodSet, which we implemented directly in Veil; these are separate from the translated models.

[8] https://github.com/tlaplus/tlaplus/commit/8f4bc8b73ad1202774a6bf70143436f8ba50aab0, dated 25 September 2026.

Table 1. Statistics for the benchmark cases translated from TLA⁺. From the perspective of the application domain, we divide them into four categories. For each case, we provide its description, the sizes of the TLA⁺ and Veil models respectively, along with their ratio. To highlight the specific Veil features utilised in each model, we indicate the presence of four distinct constructs: Set for sets and multisets, TM for termination conditions, BI for built-in Lean syntax, and ND for non-deterministic choices. The symbol 🐞 indicates models designed specifically to illustrate known design bugs.

#	Domain	Name	Description	Size (LOC)		Ratio	Constructs				Bug
				TLA⁺	Veil		Set	TM	BI	ND	
1	Concurrent Algorithm	Bakery	Lamport's bakery mutual exclusion for n processes [36]	174	161	0.93				✓	
2		Mutex	Mutex used in modern Rust OS kernel [64]	205	150	0.73		✓	✓		🐞
3		Peterson	Peterson's 2-process lock algorithm [51]	72	57	0.79					
4		FileLease	DFINITY's lease via atomic file operations [43]	213	141	0.66	✓	✓	✓	✓	
5		TicketSem	MongoDB's ordered ticket semaphore with resizing [15]	142	105	0.74	✓	✓	✓	✓	
6	Storage	KeyValueStore	Key-value store with snapshot transactions [24]	86	78	0.91					
7		KVSnap	Key-value store with snapshot isolation [9]	100	131	1.31	✓	✓	✓	✓	🐞
8		German	German's cache coherence protocol [54]	176	141	0.80					
9		Cosmos	Azure Cosmos DB model for 5 consistency levels [23]	165	213	1.29			✓	✓	🐞
10		LeaderlessLog	Leaderless log with concurrent appends and compaction [22]	280	188	0.67		✓	✓	✓	
11		Oswald	Write-ahead log on object storage with GC [69]	235	179	0.76	✓		✓	✓	
12		Percolator	Percolator transactions with snapshot isolation [50, 52]	284	183	0.64	✓		✓	✓	
13		ParCommits	CockroachDB's parallel commits and recovery [68]	657	269	0.41	✓		✓	✓	
14	Transition Systems	RiverCrossing	River crossing puzzle	45	65	1.44					🐞
15		Prisoners	Prisoners puzzle	66	42	0.64					
16		Room	Hotel room key-card access control [26]	64	87	1.36	✓		✓		
17		DieHard	Die Hard water jugs puzzle [37]	31	43	1.39					🐞
18		AWSDNS	AWS outage DNS race condition [10]	84	62	0.74			✓		🐞
19		SimpAllocator	Simple resource allocator with mutual exclusion	63	45	0.71	✓		✓	✓	
20	Distributed Algorithm	2PC	Two-phase commit with a transaction manager [20]	66	63	0.95					
21		bcastByz	Broadcast distributed algorithm with Byzantine faults [61]	91	84	0.92	✓		✓	✓	
22		EWD840	Dijkstra's token-ring termination detection [13]	68	61	0.90	✓	✓		✓	
23		Paxos	Paxos consensus for n-acceptors with crash faults [33]	98	125	1.28	✓		✓	✓	
24		MultiPaxos	Multi-slot Paxos consensus [32]	66	158	2.39	✓		✓	✓	
25		NOPaxos	Network-ordered consensus with gap commits [39]	426	480	1.13	✓		✓	✓	
26		PAStatic	Neon's WAL consensus with static membership [59]	253	165	0.65	✓		✓	✓	
27		LeaseGuard	Raft leader leases for consistent local reads [6, 7]	224	182	0.81	✓		✓	✓	
28		TaskClaiming	Task claiming with ephemeral locks and retries [21]	176	126	0.72			✓		
29		RingFin	Token ring leader election [48]	N/A	42	–				✓	
30		FloodSet	FloodSet consensus algorithm [41]	N/A	68	–			✓		

Model translation. Common TLA⁺ data constructs can be directly mapped to Lean's library types. For instance, TLA⁺ records can be encoded using Lean's `structure`s, and TLA⁺ sequences can be represented using Lean's `List` type. Therefore, all functions of `List` (`List.append`, `List.filter`, *etc*) can be used in Veil specifications. Veil relations and functions can also be defined over these concrete Lean types. For supported concrete data types, we provide derive handlers that automatically generate instances such as `Hashable` for state hashing. To support set and multiset declarations over uninterpreted sorts, we also provide type classes like `TSet` and `TMultiSet`, which can be used to instantiate uninterpreted sets and multisets supplied with suitable theories.

In TLA⁺, an initial-state predicate `Init` specifies the allowed starting states, while a next-state relation `Next` describes the possible transitions. The latter is typically a disjunction `\/` of actions, and an action is typically a conjunction `/\` of formulas, where each conjunct describes either the guard condition or the variable updates. We translate `Init` into an `after_init` block and the component actions of `Next` into Veil actions. Guards are typically translated into `require` preconditions in Veil. In TLA⁺, variable updates are commonly expressed by equations such as `x' = e`, where unprimed variables denote current-state values and primed variables denote next-state values. We encode these updates as imperative assignments, preserving the simultaneous nature of the updates. Variables listed in `UNCHANGED` are left unassigned in the corresponding Veil action. Action-level non-deterministic choice in TLA⁺ is modelled using the existential quantifier `\E` with constraints,

```
1  Phase1a(b) ==
2  /\ \E m \IN msgs :
3  (m.type = "1a") /\ (m.bal = b)
4  /\ Send([type ↦ "1a", bal ↦ b])
5  (* unchanged variables *)
6  /\ UNCHANGED <<maxVBal, maxBal, maxVal>>
7
8  Phase1b(a) ==
9  \E m \IN msgs :
10 /\ m.type = "1a" /\ m.bal > maxBal[a]
11   /\ Send([type ↦ "1b",
12      bal ↦ m.bal, maxVBal ↦ maxVBal[a],
13      maxVal ↦ maxVal[a], acc ↦ a])
14   /\ maxBal' = [maxBal EXCEPT![a] = m.bal]
15   (* unchanged variables *)
16   /\ UNCHANGED <<maxVBal, maxVal>>
```

(a) TLA+ model of Paxos' Phase1a and Phase1b.

```
1  action Phase1a (b : BallotTy maxBallot) {
2    require b ≠ noBallot
3    require ∄ msg : { m // m ∈ msgs},
4      (match msg.val with
5      | .phase1a bal => bal == b
6      | _ => false) = true
7    Send (.phase1a b)
8  }
9
10 action Phase1b (a : acceptor) {
11   let m : { m // m ∈ msgs } :|
12     (match m.val with
13     | .phase1a bal => bal > maxBal a
14     | _ => false) = true
15   let b := m.val.ballot
16   Send (.phase1b a b (maxVBal a) (maxVal a))
17   maxBal a := b
18 }
```

(b) Translation to Veil specification.

Fig. 14. Comparison of TLA+ and Veil specifications.

which maps directly to the non-deterministic **pick** operator `:|` in Veil. The bounding set `S` can be encoded as a subtype, with additional constraints on the chosen value supplied after the operator, as illustrated by the `Phase1b` action in Fig. 14.

We translate the checked invariants into Veil's **safety** or **invariant** declarations. For deadlock checking, we extend Veil's syntax with a **termination** keyword to allow users to specify termination conditions. Lace reports a deadlock when a state has no successors and does not satisfy the specified termination condition.

An illustrative example. Fig. 14 illustrates the translation of two Paxos actions, `Phase1a` and `Phase1b`, highlighting the distinction between guards and non-deterministic choices. In the TLA+ specification Fig. 14a, both actions (transitions) use the same existential quantification `\E m \IN msgs` (lines 2–3 and lines 9–10), but they are for totally different purposes: in `Phase1a`, it serves as a *guard condition* that checks whether a certain type message has already been received, while in `Phase1b` it performs a non-deterministic choice over the set of messages. In Veil specification Fig. 14b, this distinction is made syntactically explicit. The guard in `Phase1a` is translated into a **require** precondition (lines 3–6 in Fig. 14b), whereas the non-deterministic choice in `Phase1b` is translated into a **let** statement with selection operator `:|` (lines 11–14 in Fig. 14b). Besides, the function update `maxBal' = [maxBal EXCEPT ![a] = m.bal]` in TLA+ (line 14 in Fig. 14a) is simplified to a direct assignment `maxBal a := b` (line 17 in Fig. 14b), also eliminating the need for explicit `UNCHANGED` frame conditions as in TLA+.

Library support and user requirements. All evaluated models are supported by the supplied extraction rules and lawful field representations. No additional model-specific extraction rules or lawfulness proofs were required. Users extending the library must, however, provide instances and lawfulness proofs for new field representations, or extraction rules and correctness proofs for unsupported language constructs.

Checking outcomes and translation consistency. We run the translated models and their TLA+ counterparts under corresponding finite configurations, aligning the checked properties and deadlock settings. Distinct-state counts matched whenever both tools completed exhaustive exploration,

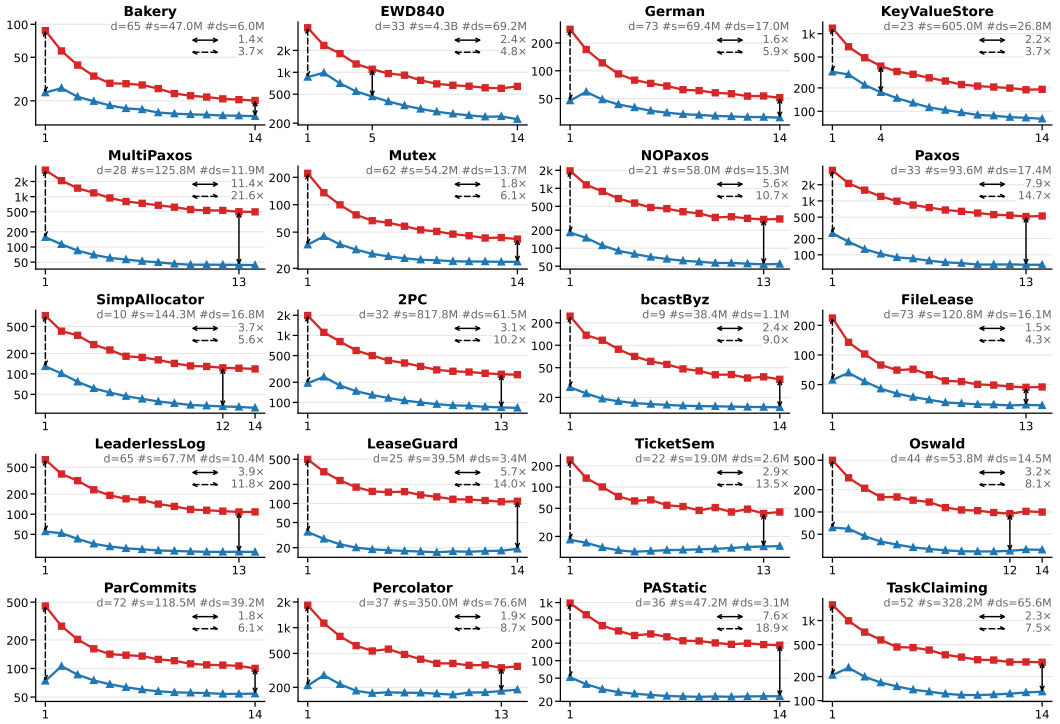


Fig. 15. Wall-clock time comparison of Lace and TLC on 20 benchmark instances using 1–14 worker threads. Each panel plots time in seconds on a logarithmic vertical axis: Lace is shown in blue (▲), and TLC in red (■). Solid and dashed double-headed arrows mark the thread counts yielding the minimum and maximum speedup of Lace over TLC, respectively, where speedup is defined as $T_{\text{TLC}}/T_{\text{Lace}}$. The corresponding ratios appear in the upper-right corner of each panel. The annotations d, #s, and #ds denote the search depth, total number of generated states, and number of distinct states, respectively.

and both tools reported an invariant violation on every evaluated instance with known violations. These results thus provide empirical evidence supporting the consistency of the translations.

6.2 RQ2: A Comparative Performance Evaluation of Lace and TLC

Fig. 15 compares the wall-clock execution time of Lace and TLC across thread counts from 1 to 14 on 20 representative benchmarks from Tab. 1. Overall, Lace outperforms TLC on all 20 benchmarks at every thread count. Nevertheless, the figure reveals several noteworthy overall trends, which we examine in turn below.

From one to two threads. Across all benchmarks shown in Fig. 15, Lace achieves its maximal speedup against TLC on a single thread. Moreover, on many benchmarks, increasing the thread count from one to two actually increases Lace's execution time. This pattern may be partly explained by the different data structures used in its sequential and parallel searches. The sequential search keeps the seen set and the log in a single array-based hash table, which it updates in place. In the parallel search, the seen set is shared by all workers, and Lean marks objects shared with a task as *multi-threaded*: they are never considered exclusively owned, so they cannot be updated in place, and their reference counts are updated atomically [67]. A shared array-based hash table would

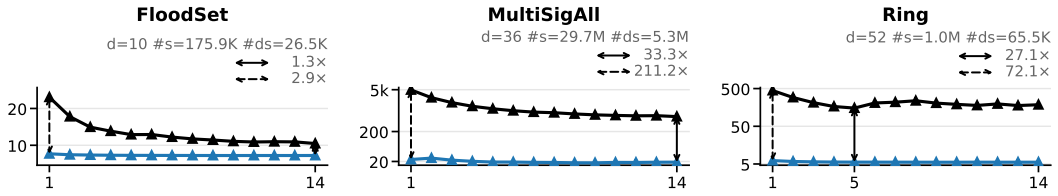


Fig. 16. Total time for two field representations in Lace: array-based (blue line, ▲) and canonical (black line, ▲; Sec. 3.4). Horizontal axes show thread counts (1–14); vertical axes show seconds on a logarithmic scale. Speedup is the canonical time divided by the array-based time at the same thread count. Solid and dashed double-headed arrows mark the minimum and maximum speedups across thread counts, respectively. Annotations give search depth (d), generated states (#s), and distinct states (#ds).

thus be copied on every insertion, so the parallel search uses persistent data structures. This design avoids excessive copying but might sacrifice some efficiency compared to hash-based structures.

Parallel scalability. As the number of threads increases, Lace generally achieves smaller speedups over its own single-threaded execution than TLC does. This difference may partly stem from the overhead of the persistent data structures discussed above. Another possible contributor is redundant work across MapReduce workers: different workers may generate the same successor states, incurring additional duplicate-handling and merging costs that limit the benefits of parallelism.

Non-monotonic behaviour. We also observe an intriguing non-monotonic behaviour in benchmarks including TaskClaiming and TicketSem: execution time decreases as the number of threads increases up to a point, but exhibits sudden step-wise increases when the thread count reaches certain thresholds (*i.e.*, around 6 and 11). We hypothesise that this effect is related to the underlying hardware topology of our evaluation platform. The Apple M4 Pro chip features 10 performance cores and 4 efficiency cores. According to system-level profiling tools, the performance cores appear to be organised into two clusters of 5, each sharing an L2 cache, while the efficiency cores form a separate cluster. When the number of worker threads exceeds 5, threads spill into the second performance-core cluster, and cross-cluster communication introduces additional cache-coherence traffic and interconnect latency. At 11 threads, workers may be scheduled on the efficiency cores, whose throughput is substantially lower, creating a straggler effect in our bulk-synchronous parallel BFS. Given that Lean is primarily intended for interactive use, these observations are practically relevant: they suggest that hardware topology can have a noticeable impact on performance, and may need to be taken into account when running Lace on commodity multi-core machines.

6.3 RQ3: Impact of Field Representation

Fig. 16 compares the performance of Lace on three Veil specifications under two different field representations: the array-based representation and the canonical instance that serves as a baseline (Sec. 3.4). The FloodSet benchmark is taken directly from Sec. 2 using the same configuration, while Ring and MultiSigAll are drawn from previously verified Veil protocols [53]. These benchmarks exhibit different structural characteristics: Ring involves no non-deterministic choice and uses only relations; MultiSigAll also relies solely on relations but involves relations with up to five parameters; FloodSet, as shown in Sec. 2, uses both functions and relations.

The results show that the array-based representation yields substantial performance improvements over the baseline across almost all thread counts. In most cases, the speedup exceeds an order of magnitude, and in extreme cases such as MultiSigAll, approaches almost two orders of magnitude. This clearly demonstrates the significance of efficient field representations on the performance of

explicit-state model checking. Regardless of each representation is used for model checking, all three protocols remain verifiable without requiring any modifications to their specifications.

7 Related Work

Symbolic and explicit-state model checking. Practitioners routinely use both symbolic and explicit-state model checkers, since each excels where the other struggles. For TLA$^+$, TLC [75] enumerates concrete states to find deep bugs in bounded instances, while Apalache [27] encodes transitions as SMT constraints to reason about unbounded state spaces. Having both is valuable, but requires two independent implementations of the same language semantics, which is error-prone: bugs from such inconsistencies are well-documented in production model checkers [76]. Ivy [44] includes both a symbolic model checker and support for exporting to the external finite-state model checker ABC [2], via eager abstraction. No formal proof establishes that the exported specification matches the semantics of the original. mypyvy [71] is a related research platform for verifying transition systems in first-order logic that similarly supports symbolic model checking. Quint [62] is a modern TLA$^+$-inspired language whose specifications can be both symbolically checked by Apalache and randomly executed (in Quint's terms—*simulated*) to find bugs; unlike Lace, it makes no formal guarantee that the two interpretations of a specification agree. P [12] takes a different angle: specifications are written as collections of communicating state machines that serve a dual purpose—they can be executed by a model checker that systematically explores message interleavings to find bugs, and they also define the safety properties against which the system is checked; like Quint, P provides no formal proof that execution and verification interpret the same model consistently.

The closest predecessor to Lace within Veil's own lineage is the randomised simulator introduced in the Loom paper [18], which executes Veil actions concretely by resolving each non-deterministic choice to a single candidate via Lean's `Findable` type class. That simulator is proved *sound*: any trace it produces is a valid execution of the specification. Lace improves on it in two fundamental respects. First, it provides an *exhaustive* enumeration of all reachable states: the extraction algorithm is proved both sound and *complete*, so no reachable state is ever missed. Second, Lace introduces a certified efficient state representation layer, equipping each state component with a runtime data structure (*e.g.*, hash sets and sorted maps) that is guaranteed to behave identically to its logical counterpart, enabling practical performance on large state spaces. Together, these contributions turn a single-trace random simulator into a fully verified explicit-state model checker.

Model checking for invariant inference. Finding inductive invariants is often the most laborious part of a deductive proof, and many tools automate it by learning from the reachable states of finite instances. DistAI [74] and DuoAI [73] learn candidate invariants from states sampled by simulating small instances and weaken them with SMT checks until they are inductive. Endive [57] uses TLC to check candidate invariants of TLA$^+$ specifications on finite instances, with correctness then established in TLAPS. More recently, I3DP [3] combines an IC3-style search with a large language model, using TLC to screen candidates on a finite instance, Apalache [27] to check their inductiveness there, and TLAPS for the unbounded proof. Because the final invariant is checked by an SMT solver or a proof, a buggy model checker or simulator cannot make these tools unsound, but it can mislead them: a missed reachable state lets a non-invariant survive filtering, while a spurious state eliminates a genuine invariant, possibly causing inference to fail. Lace could complement these tools by guaranteeing that its explored states are exactly the reachable states of the instance, under the same semantics that Veil's inductiveness checks and proofs use. Moreover, Lace is embedded in Veil's ecosystem, where calling different backends is seamless: as Fig. 2 shows, each takes only a few lines of commands. These capabilities suggest that integrating such invariant-inference algorithms into Veil is a promising direction for future work.

Certifying and certified model checkers. Namjoshi [45] introduced *certifying* model checkers, which supplement every outcome—safety proof or counterexample—with an independently checkable certificate: a Hoare-style proof tree for a positive result, or an explicit witness trace for a bug, each dischargeable by a dedicated proof checker that is itself simple enough to verify independently. Our approach differs in scope: rather than generating a per-run certificate, we prove the model checker correct *once and for all* inside Lean, so every run is trustworthy by construction without any certificate being generated or checked at runtime. The closest predecessor to Lace is the fully verified LTL model checker by Esparza *et al.* [14], mechanised in Isabelle/HOL with over 4,000 lines of extracted, executable ML code. Like Lace, it embeds a verified exploration algorithm in a proof assistant and extracts a runnable implementation; the key differences are domain and technique: it targets LTL properties over finite automata and uses Isabelle/HOL's Refinement Framework [31] to separate an abstract correctness proof from a refined efficient implementation in a two-stage process, while Lace achieves the same separation via type class instances, unifying the two stages into a single definition. Because Lace lives inside Veil, it also inherits Veil's deductive and symbolic verification modes at no extra cost, making explicit-state model checking one component of a broader foundational verification ecosystem rather than a standalone tool. Wimmer and Lammich [72] extend the same approach to verified timed automata model checking in Isabelle/HOL, targeting real-time systems rather than reachability properties of distributed protocols.

Type class-based program derivation in proof assistants. Two prior works in Rocq establish the pattern that our field representation layer follows. Krebbers and Spitters [29] use type classes to pair an abstract specification of exact real arithmetic with an efficient machine-integer implementation, proved lawful by a condition guaranteeing the two agree on all operations. The CoqEAL library [11] generalises this into a refinement framework for computational algebra, linking abstract algebraic structures to efficient implementations of matrix and polynomial operations via Rocq's canonical constructions [42]. Our contribution applies this pattern to a new domain—model checking state representation—and exploits the fact that Lean's type class instance generation mechanism discharges the lawfulness obligation automatically, requiring no manual proofs. Palamedes [19] is a Lean library that uses type class instance generation to automatically derive constrained random generators à la QuickCheck [4] from specifications, a different application of type class inference.

The type class-based approach is not the only way to decouple specifications from implementations in a proof assistant. The Refinement Framework of Isabelle/HOL [31] provides general infrastructure for stepwise refinement from abstract programs to efficient ones, used in the mentioned work on LTL model checker [14]. Alternatively, the lawfulness condition can be baked into the runtime representation's type as a dependent pair of a data structure and a proof, as supported by Lean's and Rocq's [60] subtype mechanisms, thus, making instantiation impossible without supplying the proof, but pushing the obligation to each definition site rather than resolving it once via instance synthesis. We chose type classes because Lean's instance resolution discharges the lawfulness obligation automatically, requiring no manual proof threading.

8 Conclusion and Future Work

> *"No model checker can handle all the specifications that we can write in a language as expressive as TLA⁺. However, TLC seems able to handle most TLA⁺ specifications that people actually write."* — Leslie Lamport [34, §14.2]

Based on the evaluation results presented in Sec. 6, we argue that this work yields a new explicit-state model checker that is (a) certified, (b) practical, and (c) implemented entirely inside a proof assistant. Although Lace remains far from matching the feature set and maturity of a long-standing

industrial tool such as TLC, it already attains competitive performance on a wide family of benchmarks after roughly two person-years of development. Notably, by combining a diverse set of programming language techniques (monads, type classes, meta-programming, and dependent types), we demonstrate that a proof-carrying implementation can be both expressive and efficient, a meaningful win given the modest development time and the high bar set by existing tools.

Several avenues remain open. Our current implementation does not yet support liveness checking, lacks the TLA$^+$ module instantiation mechanism (*e.g.*, its `INSTANCE`-style reuse), and its parallel state-space exploration leaves room for improvement. Addressing these gaps will require both engineering effort and new verification insights. We therefore view the current Lace implementation as a solid foundation rather than a finished product. Promising directions for future work include generalising our approach to enumerative semantics to other monads or into a reusable methodology, extending both the enumerative semantics and field representations to support infinite domains, supporting liveness reasoning through both explicit-state [25] and symbolic techniques [77], and exploring alternative multi-threading designs such as lock-based concurrency in the style of TLC.

We note that at the time of this writing, Lean itself is an actively evolving language and ecosystem. This project represents an early but encouraging attempt to achieve high performance, high expressiveness, and good ergonomics simultaneously for a tool implemented entirely within a proof assistant. The experience so far suggests that Lean is a promising implementation language for verified tooling of this kind. We believe that further work along this direction will continue to demonstrate the viability of building certified high-performance systems directly within Lean.

Data Availability Statement

Benchmark materials for the evaluation in Sec. 6 are available in the Veil repository:

https://github.com/verse-lab/veil

Some third-party TLA$^+$ benchmark sources are not included because their redistribution permissions are unclear. For these sources, we provide references to fixed upstream revisions and scripts to download the required files and verify their checksums. The software artefact accompanying this paper contains the source code of Veil and Lace, benchmark configurations, evaluation data, and build and evaluation scripts. Reproducing the evaluation requires retrieving these external benchmark sources using the provided scripts.

References

[1] Apalache 2022. Spurious Reachable State in Apalache. Apalache issue tracker, issue #1903. https://github.com/apalache-mc/apalache/issues/1903 Reported 3 July 2022; fixed by pull request #1906. Accessed 11 October 2026.

[2] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*. Springer, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

[3] Weining Cao, Guangyuan Wu, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2026. I3DP: Neuro-Symbolic Inductive Invariant Inference for Distributed Protocols. In *SOSP*. ACM, 620–635. https://doi.org/10.1145/3830418.3843876

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

[5] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. 2012. TLA$^+$ Proofs. In *FM (LNCS, Vol. 7436)*. Springer, 147–154. https://doi.org/10.1007/978-3-642-32759-9_14

[6] A. Jesse Jiryu Davis and Murat Demirbas. 2024. LeaseGuard in TLA$^+$. https://github.com/muratdem/RaftLeaderLeases/blob/c773c123bdbbf9aeb18031feddd6b291317faf5e/TLA/leaseGuard.tla

[7] A. Jesse Jiryu Davis, Murat Demirbas, and Lingzhi Deng. 2025. LeaseGuard: Raft Leases Done Right. *CoRR* abs/2512.15659 (2025). https://doi.org/10.48550/arXiv.2512.15659 arXiv:2512.15659

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* (2008), 107–113. https://doi.org/10.1145/1327452.1327492

[9] Murat Demirbas. 2023. Client-Centric Verification of Snapshot Isolation in PlusCal in TLA$^+$. https://github.com/tlaplus/Examples/blob/master/specifications/KeyValueStore/KVsnap.tla

[10] Murat Demirbas. 2025. TLA+ modeling of AWS outage DNS race condition. https://github.com/muratdem/TLA-seminar/blob/main/AwsDNSrace/AwsDNSRace.tla

[11] Maxime Dénès, Anders Mörtberg, and Vincent Siles. 2012. A Refinement-Based Approach to Computational Algebra in Coq. In *ITP (LNCS, Vol. 7406)*. Springer, 83–98. https://doi.org/10.1007/978-3-642-32347-8_7

[12] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 159:1–159:30. https://doi.org/10.1145/3276529

[13] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. 1983. Derivation of a Termination Detection Algorithm for Distributed Computations. Vol. 16. 217–219. https://doi.org/10.1016/0020-0190(83)90092-3

[14] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. 2013. A Fully Verified Executable LTL Model Checker. In *CAV (LNCS, Vol. 8044)*. Springer, 463–478. https://doi.org/10.1007/978-3-642-39799-8_31

[15] Daniel Gómez Ferro. 2026. OrderedTicketSemaphore in TLA+. https://github.com/mongodb/mongo/blob/43180330a36009ecd18a08325fd4fa2a043810a9/src/mongo/tla_plus/Concurrency/OrderedTicketSemaphore/OrderedTicketSemaphore.tla

[16] FuzzTLA 2026. FuzzTLA: Hardened Testing of TLA+ Model Checkers. Software repository and testing reports. https://github.com/tlaplus/model-checker-hardening Supported by the TLA+ Foundation. Accessed 11 October 2026.

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

[18] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. 2026. Foundational Multi-Modal Program Verifiers. *Proc. ACM Program. Lang.* 10, POPL (2026). https://doi.org/10.1145/3776719 Code available at https://github.com/verse-lab/loom.

[19] Harrison Goldstein, Hila Peleg, Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin C. Pierce. 2026. The Search for Constrained Random Generators. *Proc. ACM Program. Lang.* 10, PLDI (2026), 2060–2084. https://doi.org/10.1145/3808329

[20] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 133–160. https://doi.org/10.1145/1132863.1132867

[21] Sijie Guo. 2026. Coordination-Delegated Task Claiming Protocol in TLA+. https://github.com/openlakestream/leaderless-log-protocol/blob/df7a398718dc3b6f726514ba4b99b0e5ae6aa521/tlaplus/TaskClaiming.tla

[22] Sijie Guo. 2026. Leaderless Log Protocol in TLA+. https://github.com/openlakestream/leaderless-log-protocol/blob/df7a398718dc3b6f726514ba4b99b0e5ae6aa521/tlaplus/LeaderlessLog.tla

[23] A. Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. 2022. Understanding Inconsistency in Azure Cosmos DB with TLA+. (2022). https://doi.org/10.48550/ARXIV.2210.13661 arXiv:2210.13661

[24] Andrew Helwer. 2023. A snapshot isolated Key-Value Store in TLA+. http://github.com/tlaplus/Examples/blob/master/specifications/KeyValueStore/KeyValueStore.tla

[25] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[26] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[27] 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. https://doi.org/10.1145/3360549

[28] Igor Konnov, Markus Kuppe, and Stephan Merz. 2022. Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles (ISoLA) (Lecture Notes in Computer Science, Vol. 13701)*. Springer, 88–105. https://doi.org/10.1007/978-3-031-19849-6_6 arXiv:2211.07216

[29] Robbert Krebbers and Bas Spitters. 2011. Type classes for efficient exact real arithmetic in Coq. *Log. Methods Comput. Sci.* 9, 1 (2011). https://doi.org/10.2168/LMCS-9(1:01)2013

[30] Sanjeev Kumar and Kai Li. 2002. Using Model Checking to Debug Device Firmware. In *OSDI*. USENIX Association. https://www.usenix.org/conference/osdi-02/using-model-checking-debug-device-firmware

[31] Peter Lammich. 2013. Automatic Data Refinement. In *ITP (LNCS, Vol. 7998)*. Springer, 84–99. https://doi.org/10.1007/978-3-642-39634-2_9

[32] Leslie Lamport. 1998. The part-time parliament. In *ACM Transactions on Computer Systems (TOCS)*. 277–317. https://doi.org/10.1145/279227.279229

[33] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[34] 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

[35] Leslie Lamport. 2018. Using TLC to Check Inductive Invariance. https://lamport.azurewebsites.net/tla/inductive-invariant.pdf 23 August 2018.

[36] Leslie Lamport. 2019. A new solution of Dijkstra's concurrent programming problem. In *Concurrency: the works of Leslie Lamport*. 171–178. https://doi.org/10.1145/361082.361093

[37] Leslie Lamport. 2021. Teaching examples for TLA$^+$. https://github.com/tlaplus/Examples/blob/master/specifications/DieHard/DieHard.tla

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

[39] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*. USENIX Association, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[40] Yang Liu, Jun Sun, and Jin Song Dong. 2010. Developing Model Checkers Using PAT. In *ATVA (LNCS, Vol. 6252)*. Springer, 371–377. https://doi.org/10.1007/978-3-642-15643-4_30

[41] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.

[42] Assia Mahboubi and Enrico Tassi. 2013. Canonical Structures for the Working Coq User. In *ITP (LNCS, Vol. 7998)*. Springer, 19–34. https://doi.org/10.1007/978-3-642-39634-2_5

[43] Ognjen Marić. 2025. A File-Based Lease in TLA$^+$. https://github.com/dfinity/formal-models/blob/09151cf5ea555afe62c5055b4188cfc54c7f98bf/tla/file_based_lease/Lease.tla

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

[45] Kedar S. Namjoshi. 2001. Certifying Model Checkers. In *CAV (LNCS, Vol. 2102)*. Springer, 2–13. https://doi.org/10.1007/3-540-44585-4_2

[46] George C. Necula. 1997. Proof-Carrying Code. In *POPL*. ACM Press, 106–119. https://doi.org/10.1145/263699.263712

[47] Lingzhi Ouyang, Xudong Sun, Ruize Tang, Yu Huang, Madhav Jivrajani, Xiaoxing Ma, and Tianyin Xu. 2025. Multi-Grained Specifications for Distributed System Model Checking and Verification. In *EuroSys*. ACM, 379–395. https://doi.org/10.1145/3689031.3696069

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

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

[50] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*. USENIX Association, 251–264. https://www.usenix.org/conference/osdi10/large-scale-incremental-processing-using-distributed-transactions-and

[51] Gary L. Peterson. 1981. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.* 12, 3 (1981), 115–116. https://doi.org/10.1016/0020-0190(81)90106-X

[52] PingCAP. 2018. Percolator in TLA$^+$. https://github.com/pingcap/tla-plus/blob/8ae629ddf9f5ad8efc763bff93e3e43ce0a3585d/Percolator/Percolator.tla

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

[54] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. 2001. Automatic Deductive Verification with Invisible Invariants. In *TACAS (LNCS, Vol. 2031)*. Springer, 82–97. https://doi.org/10.1007/3-540-45319-9_7

[55] John H. Reif. 1985. Depth-First Search is Inherently Sequential. *Inf. Process. Lett.* 20, 5 (1985), 229–234. https://doi.org/10.1016/0020-0190(85)90024-9

[56] William Schultz, Ian Dardik, and Stavros Tripakis. 2022. Formal Verification of a Distributed Dynamic Reconfiguration Protocol. In *CPP*. ACM, 143–152. https://doi.org/10.1145/3497775.3503688 arXiv:2109.11987

[57] William Schultz, Ian Dardik, and Stavros Tripakis. 2022. Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In *FMCAD*. IEEE, 273–283. https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_34

[58] William Schultz and Murat Demirbas. 2025. Design and Modular Verification of Distributed Transactions in MongoDB. *Proc. VLDB Endow.* 18, 12 (2025), 5045–5058. https://doi.org/10.14778/3750601.3750626

[59] Arseny Sher. 2024. ProposerAcceptorStatic: The Neon Safekeeper Protocol in TLA$^+$. https://github.com/neondatabase/neon/blob/fa504217c61bbcaf5c512d75830564541f917f8f/safekeeper/spec/ProposerAcceptorStatic.tla

[60] Matthieu Sozeau. 2006. Subset Coercions in Coq. In *TYPES (LNCS, Vol. 4502)*. Springer, 237–252. https://doi.org/10.1007/978-3-540-74464-1_16

[61] T. K. Srikanth and Sam Toueg. 1987. Simulating Authenticated Broadcasts to Derive Simple Fault-Tolerant Algorithms. *Distributed Comput.* 2, 2 (1987), 80–94. https://doi.org/10.1007/BF01667080

[62] Informal Systems. 2024. Quint Specification Language. https://quint-lang.org.

[63] Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. 2024. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *EuroSys*. ACM, 736–753. https://doi.org/10.1145/3627703.3650077

[64] Ruize Tang, Minghua Wang, Xudong Sun, Lin Huang, Yu Huang, and Xiaoxing Ma. 2025. Converos: Practical Model Checking for Verifying Rust OS Kernel Concurrency. In *ATC*. USENIX Association, 1143–1159. https://www.usenix.org/conference/atc25/presentation/tang

[65] TLAPS 2026. Hints on Using TLAPS Effectively. TLA+ Proof System documentation. https://proofs.tlapl.us/doc/web/content/Documentation/Tutorial/Practical_hints.html Accessed 11 October 2026.

[66] TLC 2026. Equality of Function Sets Is Unsound When a Domain or Co-domain Is Empty. TLA+ tools issue tracker, issue #1407. https://github.com/tlaplus/tlaplus/issues/1407 Reported 29 August 2026; fixed by pull request #1408 on 9 September 2026. Accessed 11 October 2026.

[67] Sebastian Ullrich and Leonardo de Moura. 2019. Counting immutable beans: reference counting optimized for purely functional programming. In *IFL*. ACM, 1–12. https://doi.org/10.1145/3412932.3412935

[68] Nathan VanBenschoten. 2019. CockroachDB Parallel Commits in TLA$^+$. https://github.com/cockroachdb/cockroach/blob/d30c905fff79ef825adc96bcc647f1872a90f2ff/docs/tla-plus/ParallelCommits/ParallelCommits.tla

[69] Jack Vanlightly. 2026. OSWALD: Object Storage Write-Ahead Log Device in TLA$^+$. https://github.com/Vanlightly/s3-wal-collection/blob/0a08df719f7512a85c15cd36b4dfdcc193e9ac0e/oswald/Oswald.tla

[70] Philip Wadler. 1985. How to Replace Failure by a List of Successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA (LNCS, Vol. 201)*. Springer, 113–128. https://doi.org/10.1007/3-540-15975-4_33

[71] James R. Wilcox, Yotam M. Y. Feldman, Oded Padon, and Sharon Shoham. 2024. mypyvy: A Research Platform for Verification of Transition Systems in First-Order Logic. In *CAV (LNCS, Vol. 14682)*. Springer, 71–85. https://doi.org/10.1007/978-3-031-65630-9_4

[72] Simon Wimmer and Peter Lammich. 2018. Verified Model Checking of Timed Automata. In *TACAS (LNCS, Vol. 10805)*. Springer, 61–78. https://doi.org/10.1007/978-3-319-89960-2_4

[73] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *OSDI*. USENIX Association, 485–501. https://www.usenix.org/conference/osdi22/presentation/yao

[74] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *OSDI*. USENIX Association, 405–421. https://www.usenix.org/conference/osdi21/presentation/yao

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

[76] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and understanding bugs in software model checkers. In *ESEC/FSE*. ACM, 763–773. https://doi.org/10.1145/3338906.3338932

[77] Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. Compositional Verification of Composite Byzantine Protocols. In *CCS*. ACM, 34–48. https://doi.org/10.1145/3658644.3690355