Lasso: Fast Verified LTL Model Checking with Fairness Assumptions in Lean

Arbenit Kamberi[1⋆], Qiyuan Zhao[2], and Ilya Sergey[2]

[1] ETH Zürich, Switzerland
[2] National University of Singapore, Singapore

Abstract. We present Lasso, a model checker for Linear Temporal Logic (LTL) that is implemented and fully verified in Lean. Given a transition system, described by its initial states and a successor function, and an LTL formula ϕ, Lasso decides on the fly whether the system satisfies ϕ. We prove in Lean that Lasso is sound, and complete for systems with finitely many reachable states. Lasso targets liveness properties under fairness assumptions, which make standard automata-based checking expensive. It detects the fairness assumptions in ϕ, hoists them out before translating the rest of the formula into a generalised Büchi automaton, and handles them separately through a recursive pruning of strongly connected components, so the automaton does not grow with the number of fairness assumptions. Lasso is integrated into Veil, a Lean framework for specifying and verifying distributed protocols, which provides it with a front end. We evaluate Lasso against seven model checkers, including TLC, the explicit-state model checker for TLA$^+$. On the official TLA$^+$ examples, its running times are comparable to those of TLC, and on benchmarks where fairness dominates the cost, it is substantially faster.

1 Introduction

Model checking [3] provides an automated way to analyse models of concurrent and distributed protocols, whose nodes act independently, communicate over an unreliable network and may fail at any point. The properties of such protocols are commonly stated in Linear Temporal Logic (LTL), which covers both *safety*, "nothing bad ever happens", and *liveness*, "something good eventually happens" [23]. On models with finitely many reachable states, an LTL model checker decides such properties without a user-written proof, and the approach has proved useful in industrial practice. Amazon Web Services reported using the formal specification language TLA$^+$ [24] to specify complex distributed systems, and TLC [43], the explicit-state model checker for TLA$^+$, to find subtle design bugs that had escaped extensive design reviews, code reviews, and testing [30]. For one component in their internal distributed lock manager, the outcome they list is "Found one bug and verified an aggressive optimization".

Many liveness properties, however, hold only under *fairness* assumptions that exclude executions in which some component is starved forever. Accounting for these assumptions can significantly increase the computational cost of model checking [25]. Reducing this cost motivates sophisticated algorithms and aggressive optimisations, which can make model checkers difficult to implement

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

correctly, and bugs in these tools can undermine the reliability of their verdicts [40, 44]. Such bugs can lead to spurious counterexamples and may cause users to spend time investigating nonexistent violations or make unnecessary changes to their designs. For example, a bug in TLC caused it to report a property violation when a double negation appeared under a temporal operator.[3] We also encountered an incorrect verdict in LTSmin [21] in our evaluation (Sec. 5).

The formal methods community has recognised the need to improve the reliability of model checkers. For example, the TLA$^+$ Foundation supports a project to harden TLA$^+$ model checkers, including TLC, through differential testing [16]. Such testing can uncover defects, but passing these tests does not guarantee that a checker returns the correct verdict on every input. A stronger guarantee comes from formally verifying the checker itself. The CAVA checker demonstrated this approach [13], but in its authors' experiments it ran about an order of magnitude slower than the unverified SPIN checker [19]. Its authors proposed it as a trusted reference implementation for testing more advanced implementations.

We aim to combine verified correctness with practical performance. We take TLC as our performance baseline, motivated by its industrial use. We present Lasso, a verified LTL model checker designed around three goals: performance, trust, and usability. *Performance* comes from handling fairness assumptions separately from the rest of the temporal property and combining this treatment with on-the-fly exploration. *Trust* comes from implementing and verifying all components of Lasso in the Lean proof assistant [26]. For models with finitely many reachable states, our machine-checked theorems guarantee that Lasso's verdict correctly reflects whether the input model satisfies the checked property under the specified fairness assumptions. *Usability* comes from integration with Veil [31], a foundational verifier embedded in Lean for specifying and verifying distributed protocols. Veil provides the front end for specifying systems, and our integration extends this front end with support for specifying LTL properties and checking them with Lasso, especially those involving TLA$^+$-style action *enabled* and *taken* primitives, which none of the other compared tools, apart from TLC, supports natively.

We benchmark the performance of Lasso against a selection of widely used LTL model checkers: TLC [43], Spot [11], PAT [38], SPIN [19], NuSMV [7], LTSmin [21] and CAVA [13], the last being the only other fully verified LTL model checker we are aware of. On benchmarks originally specified in TLA$^+$, Lasso achieves performance comparable to TLC and is faster at the median. Where fairness drives the cost, Lasso is faster by a wide margin, and it decides every benchmark that TLC decides as well as several that TLC does not. On the TLA$^+$ benchmarks, Lasso is orders of magnitude faster than CAVA.

To summarise, in this work we make the following contributions: (1) Lasso itself, combining a verified LTL3BA translation [1, 17] of LTL into transition-based generalised Büchi automata (TGBA) with a fully on-the-fly emptiness check refining Tarjan's SCC procedure [32, 39] (Sec. 3); (2) a verified yet fast fairness-aware emptiness check that hoists the fairness out of the formula entirely,

[3] https://github.com/tlaplus/model-checker-hardening/issues/183, closed.

normalises it and prunes recursively [12,38] (Sec. 3); (3) the integration of Lasso into Veil, giving the model checker a specification language with intuitive syntax (Sec. 4); (4) an extensive evaluation against seven tools, plus an ablation study comparing each stage of our development in Lasso (Sec. 5).

2 A Tour of Lasso with Veil

One of the main obstacles to good performance in LTL model checking is the handling of fairness. Most interesting properties of distributed systems hold only under some notion of fairness. This section gives a tour of Lasso, via its integration with Veil, using a simple example with two strong-fairness assumptions.

For now, our goal is to model check whether a system *Sys* satisfies a property of interest ϕ_p under a conjunction of fairness assumptions $\phi_{fair,i}$:

$$Sys \models \phi_{fair} \to \phi_p, \qquad \text{where } \phi_{fair} := \bigwedge_i \phi_{fair,i}.$$

2.1 Case Study: A Strongly Fair Hub

As a stylised example, we consider sfHub, a state machine whose locations form a star. Fig. 1 shows the case $n = 2$. Its only state variable, `loc`, initially denotes the central location, or *hub*, at 0. From the hub, the system can move to any other location, called a *leaf*; from a leaf, it must return to the hub. For n clients, the locations are numbered $0, \ldots, 2n-1$. We label location $2i$ as client i's request and location $2i+1$ as its acknowledgement. For client 0, the hub itself is the request location. For every other client, both the request and acknowledgement locations are leaves.

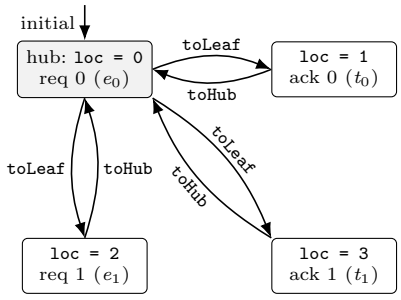


Fig. 1: sfHub ($n = 2$): actions on edges; req/ack labels on states.

Let $e_i \equiv (\texttt{loc} = 2i)$ and $t_i \equiv (\texttt{loc} = 2i+1)$ denote client i's request and acknowledgement locations, respectively. Our target property is $\phi_p = \Box(e_0 \to \Diamond\, t_0)$. Without fairness, the execution $\texttt{loc} = 0, 2, 0, 2, \ldots$ is a counterexample for $n = 2$. We then check the property under strong fairness SF for every client:

$$\phi_{fair} = \bigwedge_{i<n} \texttt{SF}(e_i, t_i) \equiv \bigwedge_{i<n} \left(\Box\Diamond\, e_i \to \Box\Diamond\, t_i \right) \equiv \bigwedge_{i<n} \left(\Diamond\Box\, \neg e_i \vee \Box\Diamond\, t_i \right)$$

Each conjunct requires that if a client's request location is visited infinitely often, its acknowledgement location must also be visited infinitely often. The model checker's task is therefore to rule out executions satisfying $\phi_{fair} \wedge \neg\phi_p$: a counterexample must both obey the fairness assumptions and violate the property.

This example separates the size of the system from the cost of handling fairness. The system has only $2n$ states, and the fairness assumption contains only n conjuncts. Yet translating the fairness together with the property can produce a large Büchi automaton [25]. Even when fairness is handled separately,

```
1  veil module SFHub
2
3  param n : Nat
4  individual loc : Nat
5
6  after_init { loc := 0 }
7
8  action toLeaf {
9    require loc = 0
10   let newLoc :|
11     newLoc < 2 * n ∧ newLoc ≠ 0
12   loc := newLoc
13 }
14
15 action toHub {
16   require loc ≠ 0
17   loc := 0
18 }
19
20 ghost relation reqAt (m : Nat) :=
21   loc = 2 * m
22 ghost relation ackAt (m : Nat) :=
23   loc = 2 * m + 1
24
25 temporal [noFair]
26   □ (⌜ reqAt 0 ⌝ → ◇ ⌜ ackAt 0 ⌝)
27
28 -- goal2: the n = 2 assumption written out
29 temporal [goal2]
30   ((□◇ ⌜ reqAt 0 ⌝ → □◇ ⌜ ackAt 0 ⌝)
31     ∧ (□◇ ⌜ reqAt 1 ⌝ → □◇ ⌜ ackAt 1 ⌝))
32   → □ (⌜ reqAt 0 ⌝ → ◇ ⌜ ackAt 0 ⌝)
33
34 -- goalN: the same assumption for any n
35 -- with SF written out explicitly
36 temporal [goalN]
37   (∀ i : Fin n,
38     (□◇ ⌜ reqAt i.val ⌝ →
39       □◇ ⌜ ackAt i.val ⌝))
40   → □ (⌜ reqAt 0 ⌝ → ◇ ⌜ ackAt 0 ⌝)
41
42 #temporal_check [goal2] [goalN] { n := 2 }
```

Fig. 2: The parameterised sfHub model and its temporal properties in Veil.

distributing the conjunction produces 2^n disjuncts: each combines one of the two alternatives from every fairness conjunct. A checker that searches separately under each combination must rule out every combination before concluding that the property holds. At $n = 22$, this approach requires over four million searches on a system with only 44 states.

Lasso avoids both sources of overhead on this example. It translates only $\neg\phi_p$ into an automaton and checks the fairness conjuncts directly during its verified search, without expanding them into all 2^n combinations. With this approach, Lasso checks the property at $n = 22$ in under 50 ms, whereas most other tools time out after 600 s and TLC raises a `java.lang.OutOfMemoryError` with the default 4 GB heap. We describe this fairness-aware search in Sec. 3.

2.2 Modelling in Veil

To illustrate Lasso's integration with Veil, we show how to express sfHub and its temporal properties in Veil. A Veil specification consists of three parts: a state definition and initialisation (lines 4–6), action definitions (lines 8–18) and temporal properties (lines 25–40).

State. A Veil specification declares module parameters and uninterpreted sorts, and represents its state using constant, relation, and function symbols. Since Veil is embedded in Lean, these declarations can use Lean types, including structures and numbers. In sfHub, the module parameter `n` specifies the number of clients, and `individual loc : Nat` (line 4) introduces the mutable state variable `loc`. The state is initialised at the start of each execution using the `after_init` block (line 6), here setting `loc` to 0.

Actions. The possible transitions between states of the protocol are defined as actions, which may take parameters, return a value, and may state preconditions with `require` (*e.g.* line 9). Actions can update state variables through assignments such as `loc := newLoc` (line 12). The syntax `let x :| p x` nondeter-

ministically chooses a value for `x` satisfying `p x`. In `toLeaf` (line 10), this selects a non-hub location, which is then assigned to `loc`.

Temporal properties. The `temporal` keyword declares a property, optionally given a name in square brackets (*e.g.* line 29). A temporal property is an LTL formula whose atomic propositions are state predicates. It may also contain quantifiers. In `goalN`, quantification over `Fin n` (a finite type containing $0, \ldots, n-1$) expresses the conjunction of the strong-fairness assumptions for all n clients.

Enabled and Taken. In addition to state predicates, Veil supports another two kinds of atoms: `E(a)`, which holds when action `a` is enabled in the current state, and `T(a)`, which holds when the current transition executes `a`. These predicates can be used to express fairness assumptions about action execution: in Veil, `WF(a)` abbreviates $\Diamond\Box$ `E(a)` $\rightarrow \Box\Diamond$ `T(a)` and `SF(a)` abbreviates $\Box\Diamond$ `E(a)` $\rightarrow \Box\Diamond$ `T(a)`.

2.3 Model Checking Temporal Formulas in Veil with Lasso

The `#temporal_check` command (line 42) passes the temporal properties to Lasso for checking, one at a time.

Instantiation. The instantiation supplies concrete values for module parameters and sorts. Here, { `n := 2` } selects the instance with two clients.

Properties. The bracketed names `[goal2]` and `[goalN]` select the properties to check. If no names are supplied, the checker checks all declared temporal properties in declaration order, stopping at the first violation.

Checking semantics. LTL model checking has no depth bound, and a deadlocked state is extended with a stuttering step so that every run is infinite. Safety invariants and deadlocks remain the job of Veil's `#model_check`.

Counterexample display. Veil reports progress and results through Lean's InfoView, using its user widgets [29]. During the search, a widget shows the property currently being checked, the number of properties checked, the elapsed time, and a button allowing the user to stop the check. When it finishes, the same panel shows the verdict. If every selected property holds, it states how many properties were checked and how many states of the product of system and automaton were visited. Otherwise it renders the counterexample as a lasso: a finite stem into a loop that repeats infinitely and violates the property. For each transition, it highlights the state variables that changed. Fig. 3 shows a counterexample to `noFair`. The last state of the stem is also the first state of the loop, giving the execution `loc` $= 0, 2, 0, 2, \ldots$: the system alternates between the hub and client 1's request leaf forever. Client 0 thus requests infinitely of-

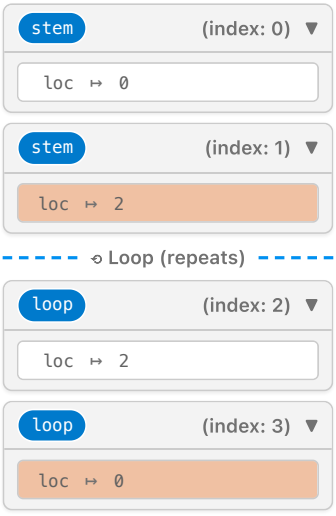


Fig. 3: A counterexample found by Lasso rendered in Lean InfoView.

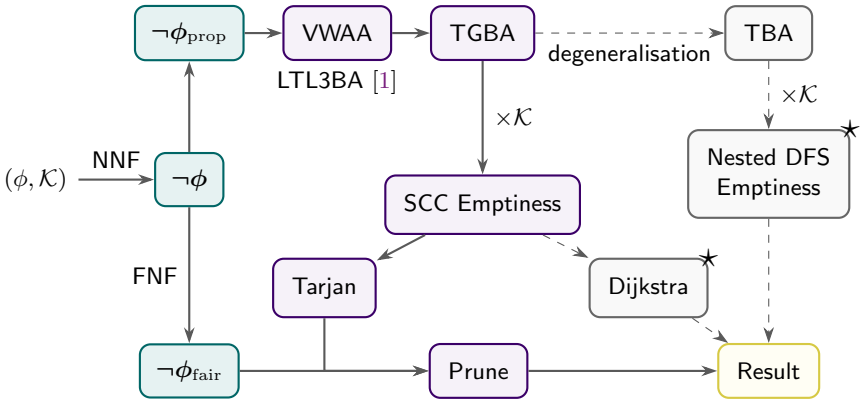


Fig. 4: Overview of the model checking pipeline. Thick arrows are the default route; the others are earlier designs that are measured in the ablation (Sec. 5). A $\star$ marks a route with no specialised fairness handling; in general, the crossing can route through Dijkstra.

ten and is never acknowledged at `loc = 1`, which is the run the first strong-fairness conjunct of `goal2` excludes.

3 The Design of Lasso

The execution pipeline of Lasso is depicted in Fig. 4. Our model checker takes an LTL formula ϕ and a Kripke structure $\mathcal{K}$, the user-specified system under test, and follows the automata-theoretic approach [3, 8] with two deviations. Rather than translating $\neg\phi$ as a whole, Lasso first splits it into a fairness part $\neg\phi_{\text{fair}}$ and a remaining property $\neg\phi_{\text{prop}}$. Only $\neg\phi_{\text{prop}}$ reaches the translator, which builds a transition-based generalised Büchi automaton (TGBA) through a very weak alternating automaton (VWAA) [1, 17]. The TGBA's product with $\mathcal{K}$ is then searched for an accepting strongly connected component (SCC), an SCC-based emptiness check [32, 35] rather than the nested depth-first search [9] that the classical presentations use. The fairness part never becomes automaton states at all: it is handed to the search as a condition that an SCC must additionally satisfy before it is reported. Fairness assumptions are what make the translation, and therefore the subsequent search, blow up, while as a per-SCC condition they cost comparatively little. This is the subject of Sec. 3.1.

3.1 Fairness Handling

Translating a property together with its fairness assumptions is where the complexity often sits. A conjunction of n strong-fairness assumptions is a conjunction of n disjunctions, so a translator given the whole formula faces 2^n combinations under which a property might hold. On sfHub itself, handing $\text{nnf}(\neg\phi)$ to LTL3BA [1] gives 519 states at $n = 6$, 4 503 at $n = 8$, and 39 879 states with 807 651 transitions at $n = 10$, that last built in a minute. At $n = 12$ it produces 356 343 states and 12.8 million transitions and takes 3 h 14 m on an Apple M1

Table 1: Hoisting the fairness conditions of sfHub ($n = 2$).

ϕ	$\big((\Box\Diamond\, loc{=}0 \to \Box\Diamond\, loc{=}1) \wedge (\Box\Diamond\, loc{=}2 \to \Box\Diamond\, loc{=}3)\big)$ $\to\ \Box(loc{=}0 \to \Diamond\, loc{=}1)$	
$\mathrm{nnf}(\neg\phi)$	$c_0 \wedge c_1 \wedge c_2$	
	conjunct c_i	$\mathrm{fnf}(c_i)$
c_0	$\Diamond\Box\, \neg loc{=}0 \vee \Box\Diamond\, loc{=}1$	$\Diamond\Box\, \neg loc{=}0 \ \vee\ (\Diamond\Box\, \top \wedge \Box\Diamond\, loc{=}1)$
c_1	$\Diamond\Box\, \neg loc{=}2 \vee \Box\Diamond\, loc{=}3$	$\Diamond\Box\, \neg loc{=}2 \ \vee\ (\Diamond\Box\, \top \wedge \Box\Diamond\, loc{=}3)$
c_2	$\Diamond(loc{=}0 \wedge \Box\, \neg loc{=}1)$	*not a fairness*
hoisted	c_0, c_1 (2 conjuncts, 2 marks on the product)	
kept	c_2 (translator builds 2 states, against 19 for $\neg\phi$)	

MacBook Pro. We did not try translating at $n = 22$, which Lasso still finishes in milliseconds by avoiding said translation entirely.

Our aim is thus to take as much of the fairness out of the formula as possible and hand the translator only what is left. The price is that an accepting SCC of the product is no longer enough: it must also satisfy the conditions we took out.

Hoisting fairness conditions. Tab. 1 is sfHub at $n = 2$ (Fig. 2) put through that split. We search for a counterexample, so the formula is negated and put in negation normal form first, and there it is a conjunction of three parts: one per fairness assumption, plus c_2, the negated property of interest. The classifier recognises c_0 and c_1 as fairnesses; c_2 is not one and stays in the property.

A detected fairness is then put into *fair normal form* [25, Theorem 1]:

$$\bigvee_i \Big(\Diamond\Box\, e_i \ \wedge \bigwedge_j \Box\Diamond\, e_{ij} \Big), \qquad \text{each } e \text{ propositional.}$$

Each disjunct is one *option*, and its two parts are what an SCC is asked about:

- The $\Diamond\Box\, e_i$ part is the option's *region*, and it asks that e_i hold *everywhere* in the accepting SCC.
- Each $\Box\Diamond\, e_{ij}$ part is a *mark*, and it asks that e_{ij} hold *somewhere* in the SCC.

The translator therefore sees c_2 alone and builds two states, against nineteen for the whole of $\neg\phi$ at just $n = 2$, where the exponential effect has barely started.

Crossing. The hoisted conjuncts are a conjunction of disjunctions. A direct approach is to distribute [25]: pick one option from c_0 and one from c_1, and each of the $2^2 = 4$ choices gives a conjunction of regions and marks that a single search can decide. The product is then searched four times, once per choice, and the property holds only if every search comes back empty. This is already a large improvement: one runs many tiny searches instead of a large one. An exponential blow-up remains: at $n = 22$, sfHub with the crossing performs 4M searches.

Algorithm 1 The prune, on a region r and a candidate C.

```
1: procedure CHECK(r, C)
2:     for all non-trivial maximal SCCs B of C restricted to r do
3:         if B has all property marks and an available option per conjunct then
4:             report B and stop
5:         if B misses a property mark then
6:             discard B                          ▷ marks are monotone downward
7:         else
8:             let i be the first conjunct with no available option
9:             live ← { opt ∈ i | opt's marks are all witnessed inside B }
10:            if live = ∅ then
11:                discard B                      ▷ no subgraph of B satisfies i either
12:            else if an enclosing call already committed to an option of i then
13:                discard B                      ▷ i was already branched on
14:            else
15:                for all opt ∈ live do
16:                    CHECK(r ∧ opt.region, B)   ▷ opt is new to r
```

Pruning (Algorithm 1). The key intuition is that the fairness assumptions are additional restrictions on an SCC, so any SCC we find for $\neg\phi_{\text{prop}}$ is either fair itself, contains a smaller fair subgraph, or no fair counterexample can be constructed from it at all. This leads to a decision procedure where for any candidate accepting SCC we find for $\neg\phi_{\text{prop}}$, we recursively restrict it until it fulfils all fairness assumptions or nothing is left.

The crossing pays 2^n because it commits to one option per conjunct before searching. The prune commits only when an SCC violates a conjunct, and only to the options that could still repair it, restricting the candidate accordingly and recursing. For weak and strong fairness that is at most one option, so the prune never branches. For general fairness it may, as the problem is NP-complete [12].

More complicated fairness. Most tools recognise a fairness only in a pre-specified shape: TLC accepts $\Box\Diamond e$ and $\Diamond\Box e$, the two forms weak and strong fairness expand into, and NuSMV its built-in JUSTICE and COMPASSION. The reason to hoist an *arbitrary* fairness instead is that the assumptions that would be natural for a user to write are often in neither shape. We have three natural examples in our benchmarks.

A *nested cycle* fairness says that a whole sequence completes infinitely often, rather than that each of its steps does. For a server that accepts, processes and replies, the natural statement is $\Box\Diamond(\mathit{recv} \wedge \Diamond(\mathit{work} \wedge \Diamond\, \mathit{emit}))$. Assuming only $\Box\Diamond\, \mathit{recv}$ is too weak: a server that accepts and then aborts satisfies it without ever replying. This is not of the form $\Box\Diamond\, p$ for a state predicate p, so almost none of the tools we compare against recognise it as a fairness at all.

A *conditional* fairness makes the obligation depend on something that is not itself a fairness. A fault-tolerant scheduler has to keep serving a process only while that process is alive, so the assumption one writes per process is $\Box\neg \mathit{crash}_i \rightarrow \Box\Diamond\, \mathit{tick}_i$, that is $\Diamond\, \mathit{crash}_i \vee \Box\Diamond\, \mathit{tick}_i$. The right-hand side is a fair-

ness, but the left-hand side is a plain eventuality, so the whole is not a fairness [25, Definition 1]. Lasso splits the disjunction instead, hoisting the fairness side and keeping the other; this gives one prune run per combination of sides, *i.e.* a crossing over a prune. Our condFair family is n such conjuncts.

The nested form can be rewritten by hand into a shape a given tool recognises. The conditional form cannot.

Finally, a fairness can appear in the *property* as well as in the assumption: checking that a process enters its critical section infinitely often is itself a $\square\lozenge$ condition. Because the split runs on the negated formula, such a property is hoisted alongside the assumption, and the translator can be left with nothing to build but the formula $\top$.

3.2 Enabled and Taken Actions

Writing a fairness about an *action* rather than a state is the idiom TLA$^+$ users reach for, and Lasso supports it directly: `E(a)` holds where action `a` can fire and `T(a)` where the step being taken is an `a`-step (Sec. 2). `E(a)` is a property of a state and needs nothing new. `T(a)` is a property of a *step*. For it, each successor carries the step as an atomic proposition on the state. Under $\square\lozenge$, such an atom simply becomes a mark, which is the shape an *assumed* fairness takes. Under $\lozenge\square$ it becomes a hoisted region, and we switch to the crossing (Sec. 6). This arises, for example, when strong fairness is itself the *property*.

3.3 Where Does Lasso Stand Compared to Others?

The inspiration for the pruning approach comes from PAT [38], but the general form on which we base ours goes back to Emerson and Lei [12]. PAT offers only three built-in fairness notions (weak, strong local and strong global) and applies the chosen one uniformly to every action of the system.[4]

We instead use a more generic form of the same idea and we read the fairness out of the formula directly. A user can thus state the fairness per action: one client can be assumed strongly fair and another not at all, and weak and strong fairness can be mixed in the same assumption together with further assumptions. We give an overview in Tab. 2.

Checking $\mathit{Sys} \models \mathit{fair} \rightarrow \phi$ has three independent costs: translating the assumption into the property automaton, the number n of fairness conditions, and recognising a fairness at all. Tab. 2 places the tools we compare against along those three axes and we also compare along these axes in our evaluation (Sec. 5).

4 Implementing Lasso in Lean

In this section we discuss some details about implementing Lasso in Lean and the statements of the main theorems ensuring its correcntess.

4.1 Integrating Lasso into Veil

Lasso has a single centralised interface for integration into tools that allow for system specification. What it requires is the formula in LTL form with its atomic

[4] In PAT 3.5.1 this is a command-line setting with four values.

Table 2: How each tool takes a fairness assumption. Lasso, TLC [24,43], PAT [38], NuSMV [7], Spot [11], SPIN [19], LTSmin [21] and CAVA [13].

	supplied as	shapes recognised	n strong pairs	nested	verified
Lasso	the LTL formula	LTL($\Diamond, \Box$) fairness, rewritten first	prune: poly. in $\|S\|$, lin. in n	hoisted, as marks	Lean
TLC	spec: $\mathrm{WF}_v(A)$, $\mathrm{SF}_v(A)$	syntactic $\Box\Diamond p$ / $\Diamond\Box p$	crossing: 2^n passes	tableau	
PAT	run mode, 3 fixed notions	not read from a formula	prune	translated	
NuSMV	model: JUSTICE/ COMPASSION	$\Box\Diamond p$ pairs	symbolic, polynomial	translated	
Spot	the LTL formula [a]	fairness subformulas	Emerson–Lei check, full product	as marks	
SPIN	`-f`: weak process	weak process only	translated	translated	
LTSmin	none, Büchi only	—	translated	translated	
CAVA	none, Büchi only	—	translated	translated	Isabelle

[a] With Emerson–Lei acceptance (`-G`), Spot also splits the formula and translates fairness subformulas separately before recombining. The Spot row reports this mode.
translated means the assumption ends up in the property automaton, either because the tool has no fairness mechanism or because the shape does not fit the one it has. TLC is the exception that builds a tableau instead of an automaton.

propositions, a set of initial states, a successor function, a state labelling function to check which atoms hold on a state and an edge labelling function to check which action was taken for a transition. Veil already has a built-in model checker for safety invariants, so it provides all of these. All that Veil needs in order to integrate Lasso is an extension of its syntax with LTL formulas.

4.2 Key Theorems

Lasso exposes one entry point per search route and states a pair of correctness theorems (Soundness, Completeness) for each route. These theorems *prove* that our checker never returns a wrong answer for any system and LTL formula ϕ. In Fig. 5, we show a simplified pair for the default route now used in Veil that does all of the special handling.

4.3 Assumptions and Trusted Code Base

Lasso is fully verified in Lean and `sorry`-free. The trusted computing base is therefore reduced to Lean's only three axioms and its kernel implementation.

```
-- Soundness: an `.empty` result means every run satisfies φ
theorem checkFair_empty
    -- Given the checker returned nothing
    (h : checkFair sys φ = .empty) :
    -- Any execution of the system satisfies the formula φ
    ∀ ρ, (toSpec sys).isRun ρ → directSatisfies ρ φ

-- Completeness: a `.nonEmpty` is a real run violating φ
theorem checkFair_nonEmpty {stem loop}
    -- Given the checker returned a counterexample
    (h : checkFair sys φ = .nonEmpty stem loop) :
    -- The returned lasso is an execution of the system
    -- that does not satisfy φ
    ∃ ρ, (toSpec sys).isRun ρ ∧ ¬ directSatisfies ρ φ
         ∧ isEquiv (stem, loop) ρ
```

Fig. 5: Correctness theorems of the default route (simplified). A counterexample is a lasso: a finite `stem` followed by a `loop` that repeats forever.

5 Evaluation and Case Studies

We compare Lasso with seven model checkers on two benchmark suites, the official TLA$^+$ examples (Suite A) and families built to vary the fairness cost (Suite B), and measure the impact of each stage of Lasso in an ablation study.

5.1 Methodology

Suite A shows realistic specifications copied from TLA$^+$'s official examples repository [42]. Suite B isolates how different forms of fairness are handled. We also use Suite B for the ablation study of six routes in our tool.

Our claim is the absence of an exponential blow-up in certain cases rather than a millisecond ranking. What follows is our best attempt at a level playing field. We measure end-to-end, from invocation until the tool reports, since the tools divide the work differently: Lasso explores the system graph on the fly, Spot takes the full system already as input, and PAT reports only the time for its emptiness check after performing the graph construction, so graph construction has to be counted for all three.

PAT's fairness is a global command-line setting that is a strictly stronger hypothesis than the other tools are given, so we use it only where it coincides with the assumption (**Goal(SF)**) and give PAT the fairness inside the formula elsewhere. NuSMV gets three files per property: the implication as one `LTLSPEC`, the property with JUSTICE/COMPASSION declared on the model, and the property alone as a control. NuSMV reads the native JUSTICE/COMPASSION constraint form wherever the constraint says exactly what the implication says in other tools, which is the full Suite A and in Suite B the families that need no hand flattening before they can be declared (work Lasso handles automatically); elsewhere it reads `LTLSPEC`. SPIN is measured using **SPIN-Spot**, which uses Spot's newer and superior LTL-to-Büchi translator. For Spot we measure using its *generic*

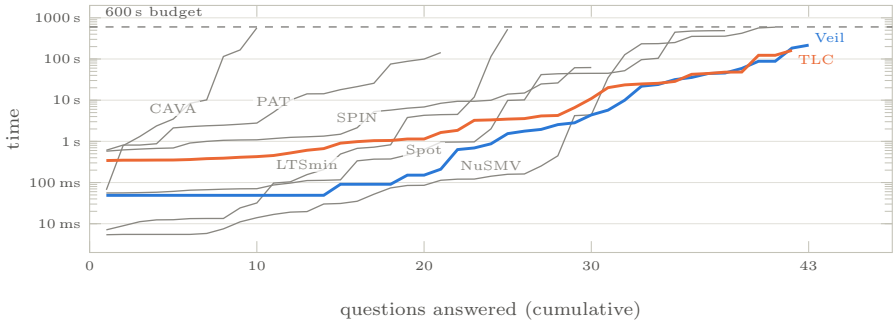


Fig. 6: Suite A: each tool's answered questions, sorted by that tool's own time, against their cumulative count. Lower and further right is better. Timeout set at 600 s. A curve stops where the tool stops being able to answer questions, either due to timeout or run failure. Failures are LTSmin segfaulting and SPIN's `spin -a` failing to translate.

mode, designed to handle fairness better via an Emerson–Lei automaton [12]. The same model set serves SPIN, LTSmin, and CAVA.

We translated every model from TLA$^+$ ourselves, so we checked each translation against TLC: for every reported model and size, each port's reachable state graph, projected onto the TLA$^+$ variables, equals TLC's. The projection matters because a purely state-based tool cannot name an action: which one was just taken must be stored in the state, an inflation inherent to translating TLA$^+$'s action-based fairness. Suite B states its fairness over states directly and avoids it, except in phil's **Goal(SF)**. PAT's ports were compared through our own interpreter of the generated CSP#.

All tools but PAT run on an Apple M1 MacBook Pro, 16 GB RAM, macOS 27.0.1. All tools run using their default configurations, except as listed below. LTSmin ships for x86-64, so we ported it to ARM64 and built from source with `-O2`. PAT depends on .NET and is Windows-only, measured separately on a different Windows machine (Intel Core i5-8265U, 1.6 GHz, 16 GB RAM). A verdict is reported only where it agrees with the other tools that decided the same question. The appendix (Tables 3, 4 and 5) holds the full tables with the results. In the tables, >600 s is a timeout, `oom` memory exhausted, and `err` other errors, which are LTSmin segfaulting (cross-validated on the Windows machine) or SPIN's `spin -a` failing to translate.

Versions. Veil on Lean 4.32.0; TLC 2026.10.06 (git `94d0c50`), on Java 25.0.1 with the JVM's default heap (4 GB, a quarter of the RAM); Spot 2.15.1; SPIN 6.5.2, its verifier compiled with Apple clang 21 at `-O2` and run to depth 10^6 with a 2^{26}-slot hash table; NuSMV 2.7.1; LTSmin 3.0.2; CAVA from the AFP of 2026-08-07, built with Isabelle 2025-2 and MLton 20241230 and capped at an 8 GB heap; PAT 3.5.1 on Windows 11.

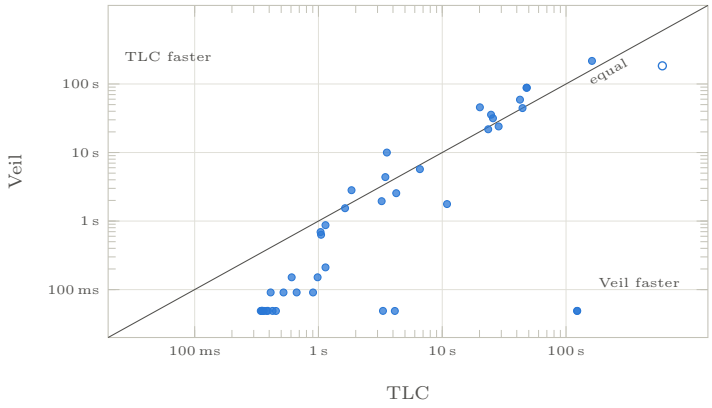


Fig. 7: Suite A question by question: TLC on x, Veil on y, both log. A point below the diagonal is a question Veil answers faster. Veil is faster on 32 of the 42 tests both tools answer. Hollow markers are questions only one of the two answered inside the 600 s timeout. TLC's single miss is a timeout.

5.2 Suite A: Official TLA$^+$ Examples

Fig. 6 and Fig. 7 show our results for a sample of tests from the official TLA$^+$ examples repository. The repository was scoured for models with interesting liveness properties, of which a sample was taken and translated for all tools. We highlight the comparison with TLC directly. Our comparison shows that even for larger tests, Lasso follows TLC closely on TLC's own examples, whereas smaller tests may be dominated by start-up.

5.3 Suite B: Comparisons across Cost Dimensions

This suite is specifically designed to measure across the three different axes of handling fairness in Sec. 3.1. We compare against other tools first in Fig. 8 and highlight the comparison with TLC directly, while deferring the exact numbers for all tools to the appendix.

Each family in Suite B is asked some of the following questions: **Goal** is $\mathit{fair} \rightarrow \phi$, **NoFair** drops the assumption and is expected to fail, and **GoalFlat** rewrites a nested assumption by hand into the flat form other checkers recognise, so the gap between **Goal** and **GoalFlat** is a tool's sensitivity to syntax. **Goal(SF)** states phil's assumption as strong fairness of each philosopher's action, in the style of TLA$^+$, and **WeakGoal** is a goal that follows from the assumption alone.

The families are small parameterised systems, each built around one kind of fairness. phil is the dining philosophers with a strong-fairness pair per philosopher, and sfHub an arbiter serving n strongly fair clients with a request–response goal. In servers, n workers cycle through receive, work and emit under a nested fairness assumption; as written, the property is a tautology. serversAck adds a bit the model sets when a worker completes a cycle, so the goal no longer follows from the assumption alone. mixed combines strong, weak and nested fairness, deepFair fixes two workers and varies the nesting depth instead, condFair as-

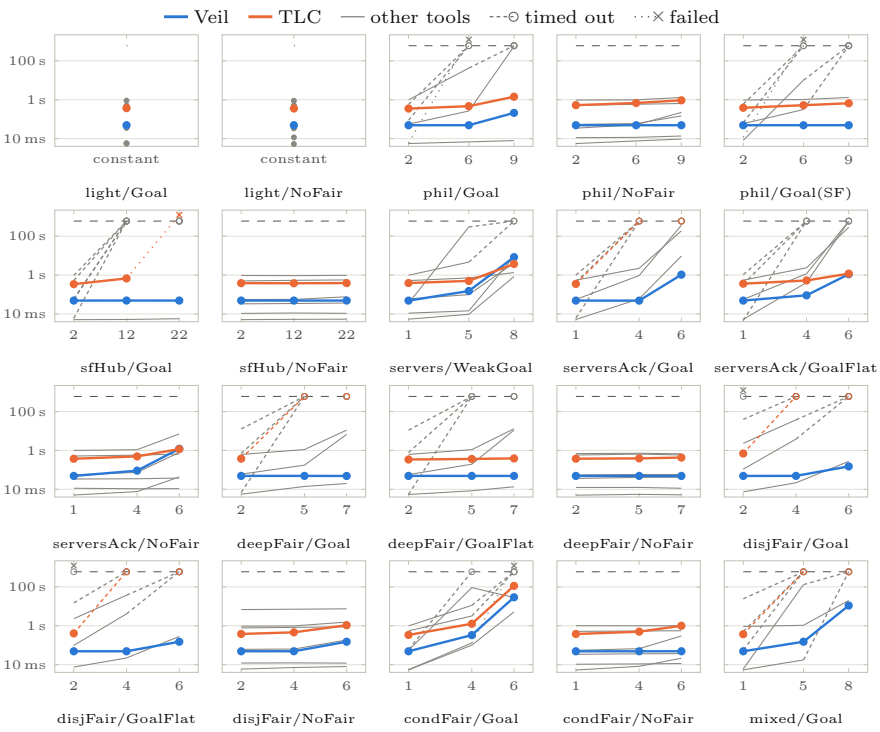


Fig. 8: Suite B as growth curves: one panel per family and property, model size on x, wall clock on a shared log y. Veil and TLC are highlighted. Timeout at 600 s. A hollow marker on the ceiling is a run that timed out. A cross above it is a run that failed instead. Failures are LTSmin segfaulting, SPIN's `spin -a` failing to translate, and TLC exhausting the heap.

sumes fairness only for processes that never crash, and disjFair has a disjunctive goal. The state space of most families grows exponentially in n. sfHub's state space grows only linearly ($2n$ states) and deepFair's state space grows quadratically while the depth of its nested fairness grows with n. light, a four-state ring with one strong pair, is so small that every tool's time on it is start-up alone, so it marks each tool's floor.

A wrong verdict in LTSmin. LTSmin 3.0.2 gave us the wrong answer for serversAck $n = 2$ **Goal** and deepFair $n = 3$ **Goal** (sizes not shown). We validated this on two different machines with x86-64 (official pre-built release on Windows Subsystem for Linux (WSL)) and ARM64 (self-compiled port). Therefore, we rule out that it is due to our port. **GoalFlat**, being logically equivalent, delivers the correct result for both. We did not investigate further ourselves, but opened a GitHub issue on LTSmin's repository.

Ablation study. Fig. 9 shows the results of our ablation study on Suite B. The ablation runs the same question through six routes to the same answer, so each consecutive pair isolates one design decision with everything else held fixed. The

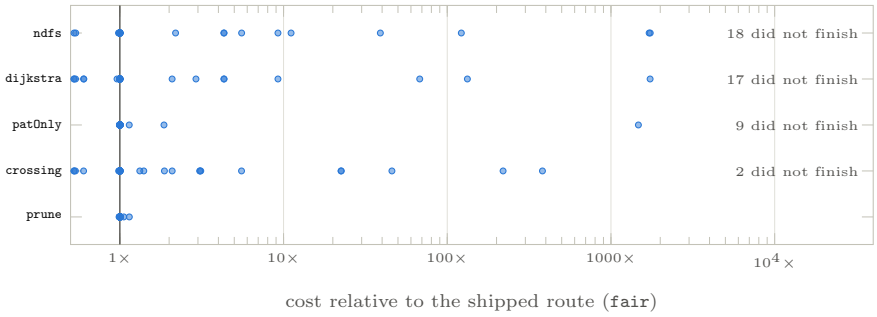


Fig. 9: The ablation as a ratio: for every Suite B question, each alternative route's wall clock divided by the shipped route's with one dot per question. The vertical rule at 1× is the default route. Timeout set at 600 s. Fair, compared to prune, attempts additional splitting on disjuncts after hoisting fairness.

`patOnly` route hoists only the conjuncts that are *syntactically* exactly $\Box\Diamond p$ or $\Diamond\Box p$ over state predicates, so its gap to the full `prune` route measures what the generalisation of Algorithm 1 with formula pre-processing improves. The cross-tool comparison cannot separate that gain from differences in language and engineering effort. `fair` is the shipped entry point, which differs from `prune` in choosing between the prune and the crossing (Sec. 3.1), and in crossing then pruning on a disjunction where only one side can be hoisted.

Where another route beats `fair`, the property fails (servers, serversAck, disjFair). `dijkstra` reports as soon as a partial SCC carries every acceptance mark [32], `ndfs` as soon as its nested search closes an accepting cycle [9, 35], whereas the prune decides an SCC only once it is complete, so it explores a violating SCC in full first. `crossing` runs `dijkstra`'s check once per combination of fairness options and can therefore stop at the first combination that results in finding a counterexample.

6 Discussion

When searching for strongly connected components (SCCs), there are two popular algorithms: Dijkstra's and Tarjan's. Both can be extended to act as a procedure for finding accepting SCCs [32]. Dijkstra's procedure may exit early when it finds an accepting SCC that is not yet maximal. We used Dijkstra's approach first, but switched to Tarjan's when implementing the prune, as the prune requires a maximal SCC as its starting point. Independent formalisations of Tarjan's [6] and Gabow's algorithms [22] already exist and have served as references.

The prune itself has one limitation, which shows when fairness is the property rather than an assumption. It reads an option's region and its marks off the state it has reached. For `E(a)` that is always sound, and for `T(a)` it is sound whenever the fairness is *assumed*, since `T(a)` then sits under $\Box\Diamond$ and becomes a mark, read off an edge the candidate SCC already contains. It fails for a strong fairness written as the *property* instead of the *assumption*. The search negates the property, so `T(a)` lands under $\Diamond\Box$ ($\neg\Box\Diamond$ `T(a)` $\equiv \Diamond\Box\neg$`T(a)`), inside a region,

and a region over a step atom cannot be decided by looking at any single state: options chosen for different conjuncts must hold on one and the same step. The choice no longer decomposes, and the options have to be enumerated at a cost exponential in the number of conjuncts, a problem that is NP-complete for general fairness [12]. The prune could be extended to read its regions off edges instead, but this would slow down every common case significantly, while this case would remain exponential, so we route this one property shape to the crossing instead. This is the same exponential cost that TLC already pays for every fairness it is given (Tab. 2), and none of the models in Suite A uses this shape. Most `T(a)` formulas can also be rewritten to check for a specific value that the action sets in the state, for which the prune applies again, and a user who writes such a formula can be told so.

7 Related and Future Work

Lasso connects two lines of research: verified LTL model checking, and algorithms that keep fairness from multiplying the cost of checking liveness.

7.1 Verified LTL Model Checking

The CAVA model checker [13] is fully formalised in Isabelle/HOL. It does no fairness handling, instead translating the entire formula into a Büchi automaton and performing the emptiness check by the standard automata-theoretic approach. CAVA uses an older tableau construction [18, 34] for its LTL-to-Büchi translation. We instead went with the LTL3BA [1, 17, 20] construction, which produces smaller automata faster.

7.2 Fairness Normal Forms and Generic Acceptance

We take the fair normal form (FNF) from [25], whose Algorithm 1 is also our crossing route. Others even call the FNF rewriting folklore [27,28] and also credit it to [14, 37]. The characterisation of the fairness class goes back to [36]. Li et al. [25] also have a separate method for covering fairness with Next (X) and Until (U) operators more efficiently, whereas ours focuses on Eventually ($\Diamond$) and Always ($\Box$) as these are more relevant for our application.

The prune was initially motivated by PAT [38], although the generic version of it we refine goes back to Emerson and Lei's recursive restriction of a component by the pairs it violates [12]. This same recursion runs underneath much of this area: NuSMV [7] runs it for symbolic model checking over BDDs, and Baier et al. [2] report that their generic check behaves like Emerson and Lei's on Streett conditions, which includes the fairness conditions we described.

A second line of work keeps the fairness in the formula and handles it on the automata level. This happens by translating into other types of automata (Emerson–Lei, Rabin or Streett automata) than the Büchi type we use. Fairness conditions we hoist out entirely can then be read off the automata transitions instead. Duret-Lutz et al. [10] introduce an emptiness check for Streett automata, also escaping the exponential blow-up caused by strong fairness on Büchi automata. Bloemen et al. [5] bring generalised Rabin acceptance into explicit-state model checking, naming Streett conditions as future work. Baier et al. [2] extend

it to Emerson–Lei conditions without first converting them to disjunctive normal form, staying polynomial for Streett and generalised Rabin. The uses they report are automaton cross-comparison, stutter invariance, obligation and inherent weakness checks and probabilistic model checking. Even for LTL-to-Büchi translation there exist techniques that attempt to control the state space blow-up caused by fairness on Büchi automata, for example suspension [1]. Although it reduces the state space, it cannot avoid the exponential blow-up entirely.

This line removes the same cost we do, but one stage later. We hoist the fairness out of the formula such that it never reaches the translation at all, then use state-of-the-art LTL-to-Büchi translation mechanisms which are capable of handling the rest and are very well studied already. That line instead extends the translation itself into automata types that can better represent such conditions, and a recursive pruning approach then underpins all of them.

What we have not found assembled is the combination: on-the-fly explicit-state model checking under a fairness the user wrote as part of the property, with detection and hoisting of the fairness plus pruning. NuSMV [7] takes declared COMPASSION pairs and is symbolic on BDDs, PAT [38] takes one of three whole-system modes, and the automata line translates the fairness into the automata. Baier et al. [2] report it can run on the fly but Spot, where the generic check is implemented, does not. In its generic mode for LTL translation (`-G`) Spot also splits the LTL formula into multiple parts and then recombines the resulting automata of each part, using different types of automata per part, whereas we do not translate the hoisted fairness at all. None of these tools has built-in support for actions being enabled or taken *à la* TLA$^+$ as we do, and TLC itself does not follow a pruning approach. Ours recognises fairness inside the formula, hoists it, builds the product with the system and decides it with the prune, with each step and their combination verified in Lean.

7.3 Parallelisation

The emptiness check currently runs sequentially. Parallelisation of our approach could happen along two separate axes: (1) running (independent) crossed checks in parallel, and (2) parallel emptiness checks. The first could very easily be achieved thanks to Lean's built-in `Task`. It takes a pure function and evaluates it in a different thread chosen from a thread pool whose size can be configured. This applies only to the crossing. The second is the much harder case, namely multiple threads collaborating to find an accepting SCC in a single given graph. How to do this efficiently is still an open research question in general. Many interesting approaches have been put forward [4, 15, 33]. How to formally verify the more complicated concurrent data structures required to achieve this in Lean would certainly be an interesting future endeavour.

8 Conclusion

We have presented Lasso, an LTL model checker whose soundness and completeness are proved in Lean. Being verified does not make Lasso slow: it matches TLC on the official TLA$^+$ examples, is substantially faster where fairness dominates the cost, and is orders of magnitude faster than CAVA, the only other verified

LTL model checker. Because Lasso is implemented in Lean, it plugs directly into Veil and the wider Lean ecosystem, including Mathlib [41], which opens the way to *proving* the properties it checks for the same specification in Lean.

9 Data Availability

The software artefact accompanying this manuscript will be submitted separately for artefact evaluation and made public after the paper's acceptance. It will contain the source code of Lasso, the version of Veil used in the evaluation (Veil itself is already open source[5]), and all benchmark data and build scripts for reproducing the evaluation results.

References

1. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi automata translation: Fast and more deterministic. In: TACAS. LNCS, vol. 7214, pp. 95–109. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_8
2. Baier, C., Blahoudek, F., Duret-Lutz, A., Klein, J., Müller, D., Strejček, J.: Generic emptiness check for fun and profit. In: ATVA. LNCS, vol. 11781, pp. 445–461. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_26
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
4. Barnat, J., Brim, L., Ročkai, P.: DiVinE multi-core – A parallel LTL model-checker. In: ATVA. LNCS, vol. 5311, pp. 234–239. Springer (2008). https://doi.org/10.1007/978-3-540-88387-6_20
5. Bloemen, V., Duret-Lutz, A., van de Pol, J.: Model checking with generalized Rabin and Fin-less automata. Int. J. Softw. Tools Technol. Transf. **21**(3), 307–324 (2019). https://doi.org/10.1007/s10009-019-00508-4
6. Chen, R., Cohen, C., Lévy, J.J., Merz, S., Théry, L.: Formal proofs of Tarjan's strongly connected components algorithm in Why3, Coq and Isabelle. In: ITP. LIPIcs, vol. 141, pp. 13:1–13:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.13
7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: CAV. LNCS, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
9. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods Syst. Des. **1**(2/3), 275–288 (1992). https://doi.org/10.1007/BF00121128
10. Duret-Lutz, A., Poitrenaud, D., Couvreur, J.M.: On-the-fly emptiness check of transition-based Streett automata. In: ATVA. LNCS, vol. 5799, pp. 213–227. Springer (2009). https://doi.org/10.1007/978-3-642-04761-9_17
11. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: CAV. LNCS, vol. 13372, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9
12. Emerson, E.A., Lei, C.L.: Modalities for model checking: Branching time logic strikes back. Sci. Comput. Program. **8**(3), 275–306 (1987). https://doi.org/10.1016/0167-6423(87)90036-0

[5] veil.dev

13. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV. LNCS, vol. 8044, pp. 463–478. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_31
14. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: CONCUR. LNCS, vol. 1877, pp. 153–167. Springer (2000). https://doi.org/10.1007/3-540-44618-4_13
15. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved multi-core nested depth-first search. In: ATVA. LNCS, vol. 7561, pp. 269–283. Springer (2012). https://doi.org/10.1007/978-3-642-33386-6_22
16. FuzzTLA: Hardening TLC and Apalache through systematic, differential, fuzz, and regression testing. Work in progress on the grant "Hardened Testing of TLA$^+$ Model Checkers", supported by the TLA$^+$ Foundation (2026), `https://github.com/tlaplus/model-checker-hardening`, commit `bf5678c`. Accessed 15 October 2026.
17. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: CAV. LNCS, vol. 2102, pp. 53–65. Springer (2001). https://doi.org/10.1007/3-540-44585-4_6
18. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall (1995). https://doi.org/10.1007/978-0-387-34892-6_1
19. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
20. Jantsch, S., Norrish, M.: Verifying the LTL to Büchi automata translation via very weak alternating automata. In: ITP. LNCS, vol. 10895, pp. 306–323. Springer (2018). https://doi.org/10.1007/978-3-319-94821-8_18
21. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-performance language-independent model checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_61
22. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: ITP. LNCS, vol. 8558, pp. 325–340. Springer (2014). https://doi.org/10.1007/978-3-319-08970-6_21
23. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. **SE-3**(2), 125–143 (1977). https://doi.org/10.1109/TSE.1977.229904
24. Lamport, L.: Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002), `https://lamport.azurewebsites.net/tla/book.html`
25. Li, Y., Song, L., Feng, Y., Zhang, L.: Verify LTL with fairness assumptions efficiently. In: TIME. pp. 41–50. IEEE (2016). https://doi.org/10.1109/TIME.2016.12, proof appendices appear only in the extended version, arXiv:1606.08116
26. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: CADE. LNCS, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
27. Müller, D.: Alternative Automata-based Approaches to Probabilistic Model Checking. Ph.D. thesis, Technische Universität Dresden (2019), `https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-361005`
28. Müller, D., Sickert, S.: LTL to deterministic Emerson–Lei automata. In: GandALF. EPTCS, vol. 256, pp. 180–194. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.256.13
29. Nawrocki, W., Ayers, E.W., Ebner, G.: An extensible user interface for Lean 4. In: ITP. LIPIcs, vol. 268, pp. 24:1–24:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.ITP.2023.24

30. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM **58**(4), 66–73 (2015). https://doi.org/10.1145/2699417
31. Pîrlea, G., Gladshtein, V., Kinsbruner, E., Zhao, Q., Sergey, I.: Veil: A framework for automated and interactive verification of transition systems. In: CAV. LNCS, vol. 15933, pp. 26–41. Springer (2025). https://doi.org/10.1007/978-3-031-98682-6_2
32. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Three SCC-based emptiness checks for generalized Büchi automata. In: LPAR. LNCS, vol. 8312, pp. 668–682. Springer (2013). https://doi.org/10.1007/978-3-642-45221-5_44
33. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Variations on parallel explicit emptiness checks for generalized Büchi automata. Int. J. Softw. Tools Technol. Transf. **19**(6), 653–673 (2017). https://doi.org/10.1007/s10009-016-0422-5
34. Schimpf, A., Merz, S., Smaus, J.G.: Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In: TPHOLs. LNCS, vol. 5674, pp. 424–439. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_29
35. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: TACAS. LNCS, vol. 3440, pp. 174–190. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_12
36. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects Comput. **6**(5), 495–511 (1994). https://doi.org/10.1007/BF01211865
37. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: CAV. LNCS, vol. 1855, pp. 248–263. Springer (2000). https://doi.org/10.1007/10722167_21
38. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: CAV. LNCS, vol. 5643, pp. 709–714. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_59
39. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010
40. Tauriainen, H., Heljanko, K.: Testing LTL formula translation into Büchi automata. Int. J. Softw. Tools Technol. Transf. **4**(1), 57–70 (2002). https://doi.org/10.1007/s100090200070
41. The mathlib Community: The Lean mathematical library. In: CPP. pp. 367–381. ACM (2020). https://doi.org/10.1145/3372885.3373824
42. TLA$^+$ Examples. GitHub repository (2026), `https://github.com/tlaplus/Examples`, commit `52c4c65`. Accessed 13 August 2026.
43. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: CHARME. LNCS, vol. 1703, pp. 54–66. Springer (1999). https://doi.org/10.1007/3-540-48153-2_6
44. Zhang, C., Su, T., Yan, Y., Zhang, F., Pu, G., Su, Z.: Finding and understanding bugs in software model checkers. In: ESEC/FSE. pp. 763–773. ACM (2019). https://doi.org/10.1145/3338906.3338932

A Comparison Results Table

Table 3: Suite A: model checking each tool against TLC's own

Model / Property	Veil	TLC	NuSMV	SPOT	SPIN	LTSmin	PAT	CAVA
CoffeeCan								
EventuallyTerminates	90.8 ms	521.9 ms	14.0 ms	113.1 ms	1.09 s	491.6 ms	14.31 s	570 s
TerminationHypothesis	90.9 ms	666.6 ms	19.1 ms	111.7 ms	1.25 s	722.7 ms	21.34 s	>600 s
Prisoners	49.0 ms	390.3 ms	7.5 ms	65.2 ms	648.9 ms	155.8 ms	2.12 s	>600 s
PrisonersSingleSwitch	48.9 ms	351.3 ms	5.8 ms	58.1 ms	674.8 ms	12.5 ms	875.0 ms	10.15 s
MovingCatPuzzle	48.9 ms	352.4 ms	5.5 ms	56.9 ms	690.7 ms	12.4 ms	609.0 ms	1.31 s
DiningPhilosophers	90.9 ms	411.0 ms	251.9 ms	87.2 ms	2.14 s	11.90 s	75.44 s	>600 s
SimpleAllocator								
ClientsWillObtain	48.9 ms	361.7 ms	5.5 ms	69.0 ms	14.91 s	>600 s	>600 s	>600 s
ClientsWillReturn	48.9 ms	342.1 ms	5.5 ms	70.8 ms	14.02 s	err	>600 s	>600 s
InfOftenSatisfied	49.0 ms	348.7 ms	5.5 ms	66.9 ms	1.29 s	err	>600 s	>600 s
EWD840	48.8 ms	453.5 ms	11.1 ms	70.4 ms	911.1 ms	24.2 ms	2.28 s	166 s
TokenRing	1.54 s	1.64 s	142.3 ms	633.2 ms	999.1 ms	4.45 s	87.78 s	>600 s
ChangRoberts	48.9 ms	425.8 ms	16.8 ms	61.0 ms	1.17 s	31.9 ms	2.61 s	>600 s
SpanTree	151.1 ms	607.7 ms	445.3 ms	97.0 ms	1.34 s	97.0 ms	5.11 s	115 s
Reachable	48.9 ms	3.32 s	5.5 ms	56.0 ms	619.7 ms	8.9 ms	812.0 ms	66.5 ms
EWD687a	1.95 s	3.24 s	4.21 s	336.2 ms	1.49 s	3.79 s	143 s	>600 s
HourClock								
AlwaysTick	48.9 ms	380.2 ms	5.4 ms	55.8 ms	577.0 ms	11.2 ms	813.0 ms	786.8 ms
AllTimes	48.9 ms	349.2 ms	74.0 ms	467.4 ms	5.28 s	13.4 ms	>600 s	>600 s
BcastFolklore								
UnforgLtl	48.9 ms	123 s	120.8 ms	44.96 s	26.22 s	13.3 ms	18.00 s	3.48 s
CorrLtl	48.9 ms	123 s	121.3 ms	43.77 s	24.73 s	7.1 ms	14.23 s	2.37 s
RelayLtl	217 s	162 s	4.40 s	44.85 s	61.35 s	529 s	>600 s	>600 s
ReliableChan	184 s	>600 s	35.04 s	44.66 s	62.01 s	>600 s	>600 s	>600 s
TCP	151.2 ms	984.2 ms	113.6 ms	116.2 ms	6.39 s	104.0 ms	2.47 s	>600 s
FlashWithMutex								
ReqProgress	88.12 s	48.24 s	567 s	>600 s	err	>600 s	>600 s	>600 s
DirProgress	35.62 s	24.79 s	356 s	488 s	err	>600 s	>600 s	>600 s
UniProgress	88.04 s	47.92 s	593 s	>600 s	err	>600 s	>600 s	>600 s
InvProgress	44.47 s	44.51 s	358 s	>600 s	err	>600 s	>600 s	>600 s
RpProgress	59.03 s	42.46 s	354 s	>600 s	err	>600 s	>600 s	>600 s
WbProgress	31.61 s	25.69 s	251 s	477 s	err	>600 s	>600 s	>600 s
ShWbProgress	21.83 s	23.51 s	236 s	447 s	err	>600 s	>600 s	>600 s
NakcProgress	24.01 s	28.55 s	234 s	485 s	err	>600 s	>600 s	>600 s
ReqSuccess	9.99 s	3.57 s	>600 s	41.74 s	>600 s	>600 s	>600 s	>600 s
GermanData								
ReqServed	871.3 ms	1.14 s	31.0 ms	51.99 s	>600 s	>600 s	>600 s	>600 s
ShrReqGranted	631.1 ms	1.05 s	30.1 ms	96.22 s	>600 s	>600 s	>600 s	>600 s
ExclReqGranted	691.4 ms	1.04 s	35.1 ms	104 s	>600 s	>600 s	>600 s	>600 s
DijkstraMutex								
DeadlockFreedom	45.68 s	20.14 s	127 s	10.15 s	6.84 s	115 s	>601 s	>600 s
StarvationFree	4.38 s	3.47 s	422 s	9.87 s	8.41 s	4.51 s	>600 s	>600 s
DisruptorMPMC	2.82 s	1.85 s	>600 s	2.01 s	5.74 s	>600 s	>600 s	>600 s
ACPNB								
AC3_2	90.8 ms	904.1 ms	84.9 ms	369.5 ms	1.06 s	206.7 ms	2.78 s	>600 s
AC5	210.9 ms	1.14 s	161.4 ms	373.2 ms	1.08 s	676.0 ms	9.98 s	>600 s
BcastByz								
UnforgLtl	48.8 ms	4.14 s	19.7 ms	955.5 ms	9.39 s	13.4 ms	2.39 s	8.38 s
CorrLtl	2.55 s	4.25 s	52.0 ms	962.6 ms	9.40 s	824.9 ms	25.75 s	>600 s
RelayLtl	5.72 s	6.60 s	85.6 ms	971.0 ms	9.96 s	4.29 s	99.62 s	>600 s
MultiCarElevator	1.77 s	10.93 s	158.7 ms	>600 s	>600 s	err	>603 s	>600 s

Table 4: Suite B: the fairness families, every tool

Family	n Property	Veil	TLC	NuSMV	SPOT	SPIN	LTSmin	PAT	CAVA
light	0 Goal	48.9 ms	371.6 ms	5.8 ms	54.4 ms	515.6 ms	5.6 ms	906.0 ms	35.3 ms
light	0 NoFair	48.8 ms	356.3 ms	5.3 ms	54.0 ms	521.1 ms	11.6 ms	891.0 ms	33.7 ms
phil	2 Goal	48.9 ms	351.5 ms	5.6 ms	55.7 ms	570.3 ms	5.9 ms	984.0 ms	87.0 ms
phil	6 Goal	48.9 ms	466.2 ms	6.8 ms	263.6 ms	>600 s	err	42.31 s	>600 s
phil	9 Goal	211.0 ms	1.43 s	7.9 ms	556 s	>600 s	>600 s	>600 s	>600 s
phil	2 NoFair	48.9 ms	527.1 ms	5.6 ms	54.1 ms	517.3 ms	11.2 ms	969.0 ms	34.5 ms
phil	6 NoFair	48.9 ms	680.0 ms	7.6 ms	59.4 ms	585.1 ms	11.6 ms	985.0 ms	51.5 ms
phil	9 NoFair	48.9 ms	936.9 ms	9.5 ms	147.1 ms	641.8 ms	13.6 ms	1.31 s	222.0 ms
phil	2 Goal(SF)	48.9 ms	386.1 ms	8.1 ms	59.1 ms	597.9 ms	11.6 ms	969.0 ms	66.6 ms
phil	6 Goal(SF)	48.9 ms	523.6 ms	10.20 s	313.6 ms	>600 s	err	1.03 s	>600 s
phil	9 Goal(SF)	48.9 ms	659.6 ms	>600 s	>600 s	>600 s	>600 s	1.31 s	>600 s
sfHub	2 Goal	48.9 ms	341.9 ms	5.1 ms	55.9 ms	529.7 ms	6.0 ms	954.0 ms	60.2 ms
sfHub	12 Goal	49.0 ms	671.9 ms	5.2 ms	>600 s	>600 s	>600 s	>600 s	>600 s
sfHub	22 Goal	48.9 ms	oom	5.7 ms	>600 s	>600 s	>600 s	>600 s	>600 s
sfHub	2 NoFair	48.9 ms	385.1 ms	5.1 ms	55.5 ms	501.4 ms	10.7 ms	953.0 ms	33.2 ms
sfHub	12 NoFair	48.9 ms	379.4 ms	5.3 ms	56.0 ms	537.2 ms	11.2 ms	937.0 ms	34.5 ms
sfHub	22 NoFair	48.9 ms	387.7 ms	5.4 ms	76.4 ms	562.9 ms	10.8 ms	969.0 ms	36.6 ms
servers	1 WeakGoal	48.6 ms	389.9 ms	5.4 ms	59.3 ms	522.0 ms	11.0 ms	984.0 ms	35.4 ms
servers	5 WeakGoal	151.2 ms	502.6 ms	9.6 ms	99.0 ms	712.4 ms	14.5 ms	4.64 s	299 s
servers	8 WeakGoal	8.36 s	3.74 s	841.5 ms	4.23 s	1.34 s	9.16 s	>602 s	>600 s
serversAck	1 Goal	48.8 ms	353.1 ms	5.2 ms	55.5 ms	549.8 ms	6.2 ms	1.00 s	431.6 ms
serversAck	4 Goal	48.5 ms	>600 s	60.0 ms	952.5 ms	2.23 s	>600 s	>600 s	>600 s
serversAck	6 Goal	1.05 s	>600 s	9.51 s	364 s	188 s	>600 s	>600 s	>600 s
serversAck	1 GoalFlat	48.8 ms	361.2 ms	5.3 ms	55.6 ms	547.7 ms	4.6 ms	1.06 s	340.4 ms
serversAck	4 GoalFlat	90.9 ms	527.2 ms	413.7 ms	1.17 s	2.37 s	>600 s	>600 s	>600 s
serversAck	6 GoalFlat	1.11 s	1.19 s	581 s	594 s	287 s	>600 s	>600 s	>600 s
serversAck	1 NoFair	48.8 ms	372.8 ms	5.1 ms	55.2 ms	515.3 ms	11.3 ms	1.00 s	33.9 ms
serversAck	4 NoFair	90.8 ms	492.2 ms	7.6 ms	74.4 ms	573.6 ms	10.7 ms	1.09 s	34.9 ms
serversAck	6 NoFair	1.22 s	1.17 s	43.2 ms	787.0 ms	646.9 ms	10.8 ms	7.05 s	37.9 ms
deepFair	2 Goal	48.9 ms	377.3 ms	5.6 ms	60.2 ms	623.5 ms	6.8 ms	704.0 ms	12.69 s
deepFair	5 Goal	49.0 ms	>600 s	14.0 ms	174.5 ms	1.11 s	>600 s	>600 s	>600 s
deepFair	7 Goal	48.6 ms	>600 s	19.9 ms	6.67 s	11.12 s	>600 s	>600 s	>600 s
deepFair	2 GoalFlat	48.9 ms	341.6 ms	5.4 ms	57.0 ms	621.7 ms	5.5 ms	781.0 ms	11.01 s
deepFair	5 GoalFlat	48.9 ms	364.7 ms	8.4 ms	196.0 ms	1.10 s	>600 s	>600 s	>600 s
deepFair	7 GoalFlat	48.8 ms	388.9 ms	13.5 ms	11.13 s	12.97 s	>600 s	>600 s	>600 s
deepFair	2 NoFair	48.9 ms	375.2 ms	5.0 ms	54.8 ms	580.1 ms	12.4 ms	688.0 ms	36.1 ms
deepFair	5 NoFair	48.9 ms	389.7 ms	5.5 ms	57.6 ms	645.2 ms	12.5 ms	718.0 ms	41.2 ms
deepFair	7 NoFair	48.7 ms	429.4 ms	5.2 ms	57.2 ms	582.8 ms	11.3 ms	672.0 ms	41.6 ms
disjFair	2 Goal	48.8 ms	695.1 ms	7.4 ms	108.3 ms	2.32 s	err	39.42 s	>600 s
disjFair	4 Goal	49.0 ms	>600 s	21.6 ms	3.82 s	37.44 s	>600 s	>600 s	>600 s
disjFair	6 Goal	150.6 ms	>600 s	273.5 ms	>600 s	>600 s	>600 s	>600 s	>600 s
disjFair	2 GoalFlat	48.8 ms	404.7 ms	7.5 ms	97.8 ms	2.32 s	err	14.94 s	>600 s
disjFair	4 GoalFlat	48.8 ms	>600 s	22.1 ms	4.11 s	37.12 s	>600 s	>600 s	>600 s
disjFair	6 GoalFlat	150.6 ms	>600 s	283.0 ms	>600 s	>600 s	>600 s	>600 s	>600 s
disjFair	2 NoFair	48.8 ms	379.1 ms	5.9 ms	61.2 ms	775.7 ms	12.1 ms	953.0 ms	6.81 s
disjFair	4 NoFair	48.9 ms	459.3 ms	7.1 ms	63.5 ms	827.5 ms	12.3 ms	984.0 ms	7.11 s
disjFair	6 NoFair	151.2 ms	1.07 s	8.0 ms	185.8 ms	882.1 ms	12.0 ms	1.55 s	7.44 s
condFair	1 Goal	48.7 ms	336.9 ms	5.4 ms	54.8 ms	545.3 ms	5.5 ms	985.0 ms	60.0 ms
condFair	4 Goal	331.1 ms	1.27 s	100.7 ms	91.66 s	3.19 s	130.6 ms	11.22 s	>600 s
condFair	6 Goal	29.19 s	113 s	5.16 s	28.56 s	>600 s	err	>602 s	>600 s
condFair	1 NoFair	48.7 ms	377.5 ms	5.3 ms	54.5 ms	516.3 ms	10.5 ms	1.01 s	34.0 ms
condFair	4 NoFair	48.7 ms	496.7 ms	8.2 ms	66.3 ms	538.2 ms	10.9 ms	1.00 s	36.2 ms
condFair	6 NoFair	48.9 ms	1.00 s	21.4 ms	298.6 ms	555.2 ms	11.4 ms	1.00 s	36.8 ms
mixed	1 Goal	48.8 ms	368.4 ms	6.0 ms	56.7 ms	510.2 ms	5.4 ms	922.0 ms	24.44 s
mixed	5 Goal	151.2 ms	>600 s	131 s	>600 s	>600 s	17.5 ms	1.06 s	>600 s
mixed	8 Goal	10.95 s	>600 s	>600 s	>600 s	>600 s	>600 s	20.03 s	>600 s

B Ablation Results Table

Table 5: The Veil/Lasso-only ablation: six search routes

Family	n Property	fair	prune	crossing	patOnly	dijkstra	ndfs
light	0 Goal	48.9 ms	48.9 ms	48.9 ms	48.9 ms	48.9 ms	48.9 ms
light	0 NoFair	48.8 ms	48.8 ms	48.9 ms	48.9 ms	48.8 ms	48.9 ms
phil	2 Goal	48.9 ms	48.8 ms	49.0 ms	48.8 ms	48.9 ms	48.9 ms
phil	6 Goal	48.9 ms	48.8 ms	270.8 ms	48.8 ms	6.49 s	83.67 s
phil	9 Goal	211.0 ms	211.2 ms	46.27 s	211.2 ms	>600 s	>600 s
phil	2 NoFair	48.9 ms	48.8 ms	48.9 ms	48.8 ms	48.9 ms	48.9 ms
phil	6 NoFair	48.9 ms	48.9 ms	48.6 ms	48.8 ms	48.8 ms	48.8 ms
phil	9 NoFair	48.9 ms	48.9 ms	48.9 ms	48.9 ms	49.1 ms	49.1 ms
phil	2 Goal(SF)	48.9 ms	48.8 ms	48.9 ms	48.9 ms	48.9 ms	48.8 ms
phil	6 Goal(SF)	48.9 ms	48.8 ms	48.8 ms	48.9 ms	3.32 s	5.96 s
phil	9 Goal(SF)	48.9 ms	48.9 ms	48.9 ms	48.9 ms	>600 s	>600 s
sfHub	2 Goal	48.9 ms	49.0 ms	48.9 ms	48.9 ms	48.9 ms	48.9 ms
sfHub	12 Goal	49.0 ms	48.9 ms	2.25 s	48.9 ms	>600 s	>600 s
sfHub	22 Goal	48.9 ms	48.9 ms	>600 s	48.8 ms	>600 s	>600 s
sfHub	2 NoFair	48.9 ms	48.9 ms	48.9 ms	49.0 ms	49.0 ms	48.9 ms
sfHub	12 NoFair	48.9 ms	48.9 ms	49.0 ms	48.9 ms	48.9 ms	48.0 ms
sfHub	22 NoFair	48.9 ms	48.7 ms	49.0 ms	48.8 ms	48.9 ms	48.9 ms
servers	1 WeakGoal	48.6 ms	48.9 ms	48.8 ms	48.8 ms	48.6 ms	48.8 ms
servers	5 WeakGoal	151.2 ms	151.3 ms	90.8 ms	151.3 ms	91.0 ms	331.1 ms
servers	8 WeakGoal	8.36 s	8.35 s	2.96 s	8.40 s	3.32 s	46.36 s
serversAck	1 Goal	48.8 ms	48.8 ms	48.8 ms	48.8 ms	48.8 ms	48.8 ms
serversAck	4 Goal	48.5 ms	48.5 ms	90.8 ms	>600 s	>600 s	>600 s
serversAck	6 Goal	1.05 s	1.11 s	1.47 s	>600 s	>600 s	>600 s
serversAck	1 GoalFlat	48.8 ms	48.7 ms	48.8 ms	48.7 ms	48.8 ms	48.7 ms
serversAck	4 GoalFlat	90.9 ms	91.0 ms	90.9 ms	90.9 ms	>600 s	>600 s
serversAck	6 GoalFlat	1.11 s	1.11 s	1.47 s	1.11 s	>600 s	>600 s
serversAck	1 NoFair	48.8 ms	48.8 ms	48.7 ms	48.7 ms	48.8 ms	48.8 ms
serversAck	4 NoFair	90.8 ms	89.8 ms	48.6 ms	90.8 ms	48.7 ms	48.8 ms
serversAck	6 NoFair	1.22 s	1.23 s	511.2 ms	1.23 s	48.9 ms	48.8 ms
deepFair	2 Goal	48.9 ms	48.8 ms	48.8 ms	48.9 ms	48.7 ms	48.8 ms
deepFair	5 Goal	49.0 ms	48.8 ms	48.9 ms	>600 s	>600 s	>600 s
deepFair	7 Goal	48.6 ms	48.7 ms	48.7 ms	>600 s	>600 s	>600 s
deepFair	2 GoalFlat	48.9 ms	48.9 ms	48.7 ms	48.9 ms	48.9 ms	48.8 ms
deepFair	5 GoalFlat	48.9 ms	48.8 ms	48.9 ms	48.7 ms	84.81 s	84.97 s
deepFair	7 GoalFlat	48.8 ms	48.6 ms	48.6 ms	48.7 ms	>600 s	>600 s
deepFair	2 NoFair	48.9 ms	48.9 ms	48.9 ms	48.9 ms	48.9 ms	48.9 ms
deepFair	5 NoFair	48.9 ms	48.8 ms	48.9 ms	48.6 ms	48.8 ms	48.6 ms
deepFair	7 NoFair	48.7 ms	48.7 ms	48.7 ms	48.7 ms	48.7 ms	48.7 ms
disjFair	2 Goal	48.8 ms	48.9 ms	48.8 ms	90.8 ms	451.1 ms	451.1 ms
disjFair	4 Goal	49.0 ms	48.8 ms	150.8 ms	>600 s	>600 s	>600 s
disjFair	6 Goal	150.6 ms	151.3 ms	3.39 s	>600 s	>600 s	>600 s
disjFair	2 GoalFlat	48.8 ms	48.9 ms	48.0 ms	48.8 ms	211.0 ms	210.8 ms
disjFair	4 GoalFlat	48.8 ms	48.9 ms	150.7 ms	71.78 s	>600 s	>600 s
disjFair	6 GoalFlat	150.6 ms	150.6 ms	3.38 s	>600 s	>600 s	>600 s
disjFair	2 NoFair	48.8 ms	48.9 ms	48.8 ms	48.9 ms	48.8 ms	48.8 ms
disjFair	4 NoFair	48.9 ms	48.7 ms	48.8 ms	48.9 ms	48.7 ms	48.5 ms
disjFair	6 NoFair	151.2 ms	151.0 ms	150.7 ms	151.2 ms	91.0 ms	5.90 s
condFair	1 Goal	48.7 ms	48.7 ms	48.8 ms	48.7 ms	48.7 ms	48.7 ms
condFair	4 Goal	331.1 ms	331.0 ms	691.2 ms	331.0 ms	690.7 ms	3.68 s
condFair	6 Goal	29.19 s	33.35 s	91.11 s	33.31 s	85.15 s	>600 s
condFair	1 NoFair	48.7 ms	48.7 ms	48.7 ms	48.7 ms	48.7 ms	48.7 ms
condFair	4 NoFair	48.7 ms	48.7 ms	48.7 ms	48.7 ms	48.7 ms	48.7 ms
condFair	6 NoFair	48.9 ms	48.8 ms	48.7 ms	48.8 ms	47.0 ms	48.8 ms
mixed	1 Goal	48.8 ms	48.8 ms	48.8 ms	48.8 ms	211.3 ms	211.3 ms

continued on next page

Table 5 – continued

Family	n Property	fair	prune	crossing	patOnly	dijkstra	ndfs
mixed	5 Goal	151.2 ms	151.2 ms	57.69 s	>600 s	>600 s	>600 s
mixed	8 Goal	10.95 s	10.95 s	>600 s	>600 s	>600 s	>600 s