LeanLTL: A Unifying Framework for Linear Temporal Logics in Lean

Eric Vin
University of California, Santa Cruz, CA, USA

Kyle A. Miller
University of California, Santa Cruz, CA, USA

Daniel J. Fremont
University of California, Santa Cruz, CA, USA

Abstract

We propose LeanLTL, a unifying framework for linear temporal logics in Lean 4. LeanLTL supports reasoning about traces that represent either infinite or finite linear time. The library allows traditional LTL syntax to be combined with arbitrary Lean expressions, making it straightforward to define properties involving numerical or other types. We prove that standard flavors of LTL can be embedded in our framework. The library also provides automation for reasoning about LeanLTL formulas in a way that facilitates using Lean's existing tactics. Finally, we provide examples illustrating the utility of the library in reasoning about systems that come from applications.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Modal and temporal logics; Theory of computation → Type theory

Keywords and phrases Linear Temporal Logic, Interactive Theorem Proving, Lean 4

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.37

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/UCSCFormalMethods/LeanLTL/releases/tag/vITP25`, archived at `swh:1:dir:5302fde5440711a7f847eb04552761bca6acc2b2`

Funding This material is based upon work supported by the National Science Foundation under Award No. 2303564.

1 Introduction

Linear temporal logic (LTL) [25, 24] has long been used in the verification community to specify the behaviors of systems over time. LTL has achieved such longevity by striking a balance between expressivity – for example being able to express convenient properties such as "`CarAtDestination` will eventually be true forever" and "`GateOpen` is always eventually true" – and simplicity, being a decidable logic [24] with practical tools for satisfiability checking, model checking, and synthesis (e.g. [16, 12, 13, 18]). LTL has been repeatedly used to great effect in real-world applications such as communication protocols [5], hardware synthesis [3], security [21], and planning [8].

Applications of LTL often require variations on or extensions to the core logic. For example, in runtime monitoring [2] one needs to describe system behaviors over time periods of *finite* extent, rather than the infinite traces used in LTL. The logic LTLf [6] accommodates this need; like LTL, it is decidable [6] and efficiently solvable with existing tools [12].

Basic LTL only supports propositional (boolean) variables, but, more generally, system specifications often involve predicates over integers, real numbers, arrays, and other data types. For example, in LTL the property "Eventually, `DistanceToCar` < 100" would be encoded as $\mathrm{F}\, p$ with p an atomic proposition defined outside of LTL itself to be true at exactly those timesteps where `DistanceToCar` < 100. The LTL encoding does not represent the relationship between the proposition p and the underlying attribute of the system `DistanceToCar`, meaning that some valid deductions cannot be carried out within LTL. For example, if we know that `DistanceToCar` is either zero or decreases by at least 1 in each timestep of an infinite trace, then the property above would hold; but LTL cannot encode the conditions on `DistanceToCar` more precisely than as independent, opaque propositions. LTL Modulo Theories (LTLMT) [27] and LTLf Modulo Theories (LTLfMT) [10] address this shortcoming by replacing propositions with atoms from a given *theory* in the vein of Satisfiability Modulo Theories (SMT) [1]. Commonly-used theories include linear integer arithmetic (LIA, a.k.a. Presburger arithmetic), bitvectors (BV), and nonlinear real arithmetic (NRA). When the underlying theory is decidable, there is a semi-decision procedure for LTLMT [10], with certain theories supporting a full decision procedure [11]. Returning to our previous example, we could represent and solve the implication "(`DistanceToCar` is 0 or decreases by at least 1) implies (Eventually, `DistanceToCar` < 100)" by encoding it as an LTLMT problem that utilizes the decidable theory of Linear Real Arithmetic [1].

Prior work on increasing the expressivity of linear temporal logics has largely attempted to stay within the realm of (semi-)decidability, to ensure that tools for the new logics can be fully automatic. However, such approaches rule out the use of many useful undecidable theories, such as nonlinear real arithmetic with exponents or trigonometric functions (often needed for modeling cyber-physical systems). Another limitation is that decision procedures for LTLMT are not available for all decidable theories [11]. Finally, even in decidable cases, the complexity of the decision procedure can render realistic problems intractable in practice. The above problems arise primarily in the context of *automated* tools, but with interactive theorem provers one may permit undecidable theories, as users may provide proofs for what is not in the purview of decision procedures.

With this goal in mind, we propose LeanLTL[1], a unified framework for reasoning about linear temporal properties of systems in Lean 4 [19]. Its key features include:

- Core types for modeling temporal properties across both infinite and finite traces, with support for arbitrary Lean expressions inside these formulas. Proofs using these types can make use of the full power of existing Lean tactics, as illustrated in Figure 1.
- Convenient syntax for creating LeanLTL formulas with the appropriate semantics represented directly in Lean, with embedded Lean propositions.
- Basic automation and tools to simplify reasoning about LeanLTL formulas.
- Formalized proofs that LTL and LTLf can be embedded into LeanLTL.

In the rest of the paper, we describe these features and present real-world-inspired examples that illustrate the utility of LeanLTL. Our library, examples, and proofs are included in the project repository.

Related Work. Prior work on temporal logics in interactive theorem provers includes published work [26, 28, 15, 7] and open-source repositories [14, 20, 22, 9]. To our knowledge, our work is the first framework in Lean to support discrete-time linear temporal logics over both finite and infinite traces while allowing full utilization of arbitrary underlying theories. The most similar work to ours is the TLPVS system [26], a library for reasoning about LTL properties in PVS [23] with support for theories, but it does not consider finite traces.

[1] Available at: `https://github.com/UCSCFormalMethods/LeanLTL`. The exact version presented in this paper is tagged `vITP25`.

```
example : ⊨ⁱ LLTL[((←n) = 5 ∧ G ((X (←n)) = (←n) ^ 2)) → G (5 ≤ (←n))] := by
    rw [TraceSet.sem_entail_inf_iff]
    rintro t hinf ⟨h1, h2⟩
    apply TraceSet.globally_induction <;> simp_all [push_ltl, hinf]
    intros; nlinarith
```

Figure 1 A proof in LeanLTL that for all infinite traces with a natural number variable n, the LTL-with-nonlinear-arithmetic formula $n = 5 \wedge \mathrm{G}((\mathrm{X}\, n) = n^2) \to \mathrm{G}(n \geq 5)$ holds.

2 Background

In this section we summarize the syntax and semantics of LTL and some of its extensions.

A full definition of the syntax and semantics of LTL can be found in [24]. LTL includes the standard propositional operators NOT ($\neg$), AND ($\wedge$), and OR ($\vee$), and the constant True ($\top$), in addition to several *temporal operators*:

- Next: $\mathrm{X}\, p$ (or $\bigcirc p$) is true if and only if p is true in the next timestep.
- Until: $p\,\mathcal{U}\,q$ is true iff p is true at least until q is true, and q must eventually be true.
- Release: $p\,\mathcal{R}\,q$ is true iff p is true up until and including the point where q is true. If q never becomes true, p must be true forever.
- Finally: $\mathrm{F}\, p$ (or $\Diamond p$) is true iff p is true in some future timestep, equivalent to $\top\,\mathcal{U}\,p$.
- Globally: $\mathrm{G}\, p$ (or $\Box p$) is true iff p is true in all future timesteps, equivalent to $\neg\,\mathrm{F}\,\neg p$.

LTLf, whose full syntax and semantics can be found in [6], modifies the semantics of X to account for the possibility that the next value does not exist due to being at the end of a finite trace. The two resulting operators are *strong next*, with $\mathrm{X}^{\mathrm{s}}\, p$ ($\bigcirc p$ in [6]) evaluating to false at the end of a finite trace, and *weak next*, with $\mathrm{X}^{\mathrm{w}}\, p$ (originally $\circledcirc p$) evaluating to true at the end of a finite trace; it is equivalent to $(\mathrm{X}^{\mathrm{s}}\, p) \vee \neg\,\mathrm{X}^{\mathrm{s}}\,\top$.

LTLfMT expands these semantics further by replacing propositions with atomic formulas from an underlying theory, allowing X^{s} and X^{w} to be applied to terms inside those formulas. A full definition of its syntax and semantics can be found in [10].

3 Library Details and Implementation

In this section we describe the key components of LeanLTL: types for traces and formulas, a macro enabling the use of traditional LTL syntax, and automation for proofs involving LTL.

3.1 Core Types

Traces. To model properties across linear time, we use sequences of values called *traces*. Often LTL formulas are about traces over a *state* type, a structure containing fields for all the time-varying propositions and values under consideration, and we model these time-varying variables as projections of a trace of states (see Section 4 for an example).

Concretely, we define traces over a type σ in the following way:

```
structure Trace (σ : Type*) where
  toFun? : ℕ → Option σ
  length : ℕ∞
  nempty : 0 < length
  defined : ∀ i : ℕ, i < length ↔ (toFun? i).isSome
```

This models both finite and infinite traces by using an option-valued sequence `toFun?`, which has the property that if it ever becomes `none` it remains `none` for all time (enforced by the `defined` proof). The extended natural number `length` records the length of the trace for convenience; all of our `Trace` constructions yield concrete values for the `length` field.

In order to ensure propositions always have a defined truth value, we require traces to be nonempty, as in LTLf. Reasoning about possibly-undefined truth values only appears in conjunction with the strong/weak next operators. We remark that if LTLf had notions of "strong/weak evaluation" of propositional variables, one could relax the nonempty constraint and use the LTL next operator semantics.

We model moving forward in time by shift operators on traces, which drop some number of terms from the beginning of the sequence. These operators require proof that the shift results in a nonempty trace. We also provide a number of operations to support constructions for the `TraceSet` and `TraceFun` types, as we will see below.

Trace Sets. We take a semantic view of LTL formulas, representing a formula by its set of satisfying traces, which we call a *trace set*.

```
structure TraceSet (σ : Type*) where
  sat : Trace σ → Prop

notation t " ⊨ " p => TraceSet.sat p t
```

This type is equivalent to `Set (Trace σ)`, using `Set` from Lean's Mathlib, but a custom structure gives us some advantages: (1) we can use `t ⊨ p` notation for satisfaction without it being pretty-printed as `t ∈ p`, (2) we can ensure it is never confused for `Set` in any Lean automation, and (3) we can make heavy use of *generalized field notation* by putting declarations in the `TraceSet` namespace: for example, we can write `p.implies (q.and r)` for `TraceSet.implies p (TraceSet.and q r)`. As we will describe in Section 3.2, we can further simplify such notation with a macro enabling the use of Lean's own logical connectives.

Trace Functions. To enable encoding logical formulas embedding Lean propositions which refer to the current trace, we introduce *trace functions*, which are simply functions from traces on a given domain to an option type:

```
structure TraceFun (σ α : Type*) where
  eval : Trace σ → Option α
```

The `none` value is intended to indicate exceptional behavior, such as needing to access a value past the end of a finite trace. Every `TraceSet σ` can be cast as a `TraceFun σ Prop`, but going in the reverse direction needs a choice of whether to interpret `none` as `True` or `False`.

The main interface between trace functions and trace sets are the `TraceFun.sget` and `TraceFun.wget` functions (*strong get* and *weak get*), which bind the value of a trace function `f : TraceFun σ α` to a variable that can then be used inside a trace set, where the first evaluates to `False` on `none` and the second to `True`. For example, with $\alpha = \mathbb{N}$, we may write

```
t ⊨ TraceFun.sget f (fun x => TraceSet.const (x < 10))
```

and, if the value of `f` exists, it is compared to 10, and otherwise the proposition evaluates to false as this is a strong get. We will see a convenient notation for this in Section 3.2.

The main trace functions are `TraceFun.of`, for projecting a value from the current state, and `TraceFun.shift`, for precomposing trace functions with a trace shift operator, which models having the trace function look forward in time.

3.2 LeanLTL Macro Syntax

To aid in writing LeanLTL formulas, we offer an `LLTL[...]` macro that reinterprets Lean's logical connectives as LeanLTL formulas, giving a seamless embedding of Lean's logic while exposing the ability to use trace functions embedded in Lean expressions. The macro can be compared to idiom brackets in [17], and the embedded trace functions can be compared to the `!x` notation in Idris [4] and the `(← x)` notation in Lean for embedded monad expressions.

The macro has two main components. First, we have a mechanism to install interpretations of host connectives as LTL connectives. For example,

```
declare_lltl_notation p q : p → q => TraceSet.imp p q
```

causes `LLTL[x → y]` to expand as `TraceSet.imp x y`, while also creating a pretty printer making `TraceSet.imp` display using the `LLTL` macro. This macro supports Lean's editor integrations, and hovering over → in the Infoview gives type information, as if it were written as `TraceSet.imp`. Second, all Lean expressions that do not have a registered interpretation are assumed to be Lean propositions, possibly with embedded `TraceFun` expressions. Each `(`$\leftarrow^s$` f)` (synonym: `(← f)`) and `(`$\leftarrow^w$` f)` is lifted out using the `TraceFun.sget` and `TraceFun.wget` functions mentioned in Section 3.1. For example, `t ⊨ LLTL[G ((`$\leftarrow^s$` f) < 10)]` expands to

```
t ⊨ TraceSet.globally (TraceFun.sget f fun x => TraceSet.const (x < 10))
```

The left arrows should be compared to Lean's monad arrow expressions, which in monadic contexts are similarly lifted out of expressions using monadic binds. If there are multiple strong gets and weak gets, we use the convention that strong gets are bound first, which reliably causes the surrounding LTL formula to evaluate to false at the end of a trace, no matter the order in which the arrow expressions appear inside the embedded Lean formula.

The macro expansion of `LLTL[...]` notation is carried out using recursively applied rules. First, the `declare_lltl_notation` macros are applied; for example, `LLTL[G ((`$\leftarrow^s$` f) < 10)]` is first expanded to `TraceSet.globally LLTL[(`$\leftarrow^s$` f) < 10]`. Second, `X` notation is pushed into arrow notations, so for example `LLTL[X ((← a) < (← b))]` becomes `LLTL[(← X a) < (← X b))]`, which implements X for compound values without needing to use `TraceFun` analogues of Lean operators. Third, arrow notations are lifted out, for example `LLTL[(`$\leftarrow^s$` f) < 10]` becomes `TraceFun.sget LLTLV[f] fun x => LLTL[x < 10]`, where `LLTLV[...]` is a similar macro for `TraceFun`. Finally, whatever remains is coerced to a `TraceSet`. Pure propositions use `TraceSet.const`, and we also coerce predicates σ `→ Prop` to `TraceSet` σ using `TraceSet.of`, which enables the direct use of structure projections in the formulas in Figure 3. The `LLTLV[...]` macro expands similarly and implements similar coercions.

3.3 Automation

We provide preliminary automation for reasoning about LeanLTL formulas, with the intention of expanding on this as we continue development. Our initial automation is centered on *simp sets*, curated sets of theorems that can be provided to the `simp` tactic. The primary simp set is `push_ltl`, which "pushes" the LTL "satisfies" operation as deep as possible into an expression, translating LTL operations into their first-order logic semantics.

This simp set can often reduce problems to a form that other tactics such as `linarith` or `omega` can finish, providing a halfway point between the semi/full decision procedures supported by solvers of LTLfMT. Figure 1 is an example, where after applying the `TraceSet.globally_induction` principle (provided by LeanLTL) for proving formulas of the form G p by induction on time, the `push_ltl` simp set discharges the base case and leaves the following goal:

```
def Var (σ : Type*) := σ -> Prop

structure Trace (σ : Type*) where
  trace : LeanLTL.Trace σ
  infinite : trace.Infinite

inductive Formula (σ : Type*) where
  | var (v : (Var σ))
  | not (f : Formula σ)
  | or (f₁ f₂ : Formula σ)
  | next (f : Formula σ)
  | until (f₁ f₂ : Formula σ)

def sat {σ : Type*} (t : Trace σ) (f : Formula σ) : Prop := by ...
def toLeanLTL {σ : Type*} (f : Formula σ) : LeanLTL.TraceSet σ := ...

theorem equisat {σ : Type*} (f : Formula σ) (t : Trace σ) :
    sat t f ↔ (t.trace ⊨ toLeanLTL f) := by ...
```

Figure 2 A high-level skeleton of the LTL embedding into `TraceSet`.

```
h1 : n (t.toFun 0 ···) = 5
h2 : ∀ (n : ℕ), n (t.toFun (1 + n) ···) = n (t.toFun n ···) ^ 2
⊢ ∀ (n : ℕ), 5 ≤ n (t.toFun n ···) → 5 ≤ n (t.toFun n ···) ^ 2
```

These nonlinear inequalities are then solved using the `nlinarith` tactic.

In future work, we aim to integrate LTL and LTLMT decision procedures directly into our library.

4 Logic Embeddings and Examples

In this section, we illustrate the expressivity of LeanLTL and its ability to help prove useful properties about real-world systems. For the former, we provide Lean proofs that LTL and LTLf can be embedded into LeanLTL. The embedding proof first provides an inductive syntax for each logic and an evaluation function giving its semantics. We then define a function translating the logic to LeanLTL. Finally, we show that the evaluation function applied to the original syntax is equisatisfiable with our LeanLTL translation and the LeanLTL semantics. An abbreviated version of the LTL embedding proof can be found in Figure 2, and the complete Lean proofs are included in the project repository.

To illustrate the application of LeanLTL to non-trivial real-world scenarios, we provide an abbreviated example applying the library to a system consisting of two traffic lights at an intersection, with which we demonstrate how LeanLTL can be used to model a system, its environment, and its specifications, and prove that a specification is satisfied for any given trace. The full example can be found in the supplemental materials.

To begin, we declare the state of the system at each timestep, shown in the first part of Figure 3. The state consists of whether each of the two lights is green, as well as the numbers of cars arriving, leaving, or waiting at each light.

Next we define properties that we assume about the system. In our example these properties are assumed to be derived from external specifications, but they could also include a Lean function representing a part of the system used directly in a LeanLTL formula. The

```
-- State structure, defining the data available at each timestep
structure ExState where
  (TL1Green TL2Green : Prop)
  (TL1Arrives TL1Departs TL1Queue : ℕ)
  (TL2Arrives TL2Departs TL2Queue : ℕ)
open ExState
-- Assumed properties, describing the behavior of the system
abbrev TL1StartGreen    := LLTL[TL1Green]
abbrev TL1ToTL2Green    :=
  LLTL[G ((TL1Green ∧ (← TL1Queue) = 0) → (Xˢ (¬TL1Green ∧ TL2Green)))]
abbrev TL1StayGreen     :=
  LLTL[G ((TL1Green ∧ (← TL1Queue) ≠ 0) → (Xˢ (TL1Green ∧ ¬ TL2Green)))]
abbrev TL1GreenDeparts  := LLTL[G (TL1Green → (← TL1Departs) = max_departs)]
abbrev TL1RedDeparts    := LLTL[G (¬TL1Green → (← TL1Departs) = 0)]
abbrev TL1ArrivesBounds :=
  LLTL[G (0 ≤ (← TL1Arrives) ∧ (← TL1Arrives) ≤ max_arrives)]
abbrev TL1QueueNext     :=
  LLTL[G ((X (← TL1Queue)) = (← TL1Queue) + (← TL1Arrives) - (← TL1Departs))]
-- Desired properties, which we prove
abbrev G_OneLightGreen  := LLTL[G (TL1Green ↔ ¬TL2Green)]
abbrev G_F_Green        := LLTL[(G (F TL1Green)) ∧ (G (F TL2Green))]
```

Figure 3 The state structure and some assumed/desired properties in the traffic light example.

second part of Figure 3 shows a selection of these properties: for example, `TL1ToTL2Green` states that when light 1 is green and has no cars waiting, in the next timestep it turns red and light 2 turns green. Note the use of arithmetic here and in `TL1QueueNext`.

In the last part of the figure, we state the properties that we wish to prove our system satisfies: `G_OneLightGreen` states that exactly one of the traffic lights is green at any given time, and `G_F_Green` states that each light will turn green infinitely often.

Finally, we prove that our desired properties hold for all traces satisfying the assumed properties. Complete proofs can be found in the project repository.

5 Conclusion

In this paper we presented our initial prototype of LeanLTL, a unifying framework for linear temporal logics in Lean. We described the core features of the library and illustrated how they can be used to encode and prove temporal properties of systems in Lean. In future work we plan to develop further automation, including integrations with decision procedures, as well as embedding proofs for LTLMT and LTLfMT. We also plan to consider adding support for other LTL variants, including past-time and bounded-time operators.

References

1 Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.

2 Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, June 2010. doi:10.1093/logcom/exn075.

3 Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, Compile, Run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, November 2007. doi:10.1016/j.entcs.2007.09.004.

4 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013. doi:10.1017/S095679681300018X.

5 Conrado Daws, Marta Kwiatkowska, and Gethin Norman. Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. *International Journal on Software Tools for Technology Transfer*, 5(2):221–236, March 2004. doi:10.1007/s10009-003-0118-5.

6 Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, IJCAI '13, pages 854–860, Beijing, China, August 2013. AAAI Press. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997.

7 Christian Doczkal and Gert Smolka. Completeness and Decidability Results for CTL in Constructive Type Theory. *Journal of Automated Reasoning*, 56(3):343–365, March 2016. doi:10.1007/s10817-016-9361-9.

8 Lu Feng, Clemens Wiltsche, Laura Humphrey, and Ufuk Topcu. Controller synthesis for autonomous systems interacting with human operators. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ICCPS '15, pages 70–79, New York, NY, USA, April 2015. Association for Computing Machinery. doi:10.1145/2735960.2735973.

9 Galois, Inc. lean-protocol-support/galois/temporal at master · GaloisInc/lean-protocol-support. URL: https://github.com/GaloisInc/lean-protocol-support/tree/master/galois/temporal.

10 Luca Geatti, Alessandro Gianola, and Nicola Gigante. Linear Temporal Logic Modulo Theories over Finite Traces. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, volume 3, pages 2641–2647, July 2022. doi:10.24963/ijcai.2022/366.

11 Luca Geatti, Alessandro Gianola, Nicola Gigante, and Sarah Winkler. Decidable Fragments of LTLf Modulo Theories. In *ECAI 2023*, pages 811–818. IOS Press, 2023. doi:10.3233/FAIA230348.

12 Luca Geatti, Nicola Gigante, and Angelo Montanari. BLACK: A Fast, Flexible and Reliable LTL Satisfiability Checker. *CEUR Workshop Proceedings*, September 2021.

13 G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. doi:10.1109/32.588521.

14 Simon Hudon. unitb/temporal-logic, November 2024. URL: https://github.com/unitb/temporal-logic.

15 Katherine Kosaian, Zili Wang, Elizabeth Sloan, and Kristin Rozier. Formalizing MLTL Formula Progression in Isabelle/HOL, February 2025. doi:10.48550/arXiv.2410.03465.

16 Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 585–591, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-22110-1_47.

17 Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. doi:10.1017/S0956796807006326.

18 Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. Strix: Explicit Reactive Synthesis Strikes Back! In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 578–586, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96145-3_31.

19 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_37.

20 Logan Murphy. loganrjmurphy/lean-temporal, November 2024. URL: `https://github.com/loganrjmurphy/lean-temporal`.

21 Gethin Norman and Vitaly Shmatikov. Analysis of Probabilistic Contract Signing. In Ali E. Abdallah, Peter Ryan, and Steve Schneider, editors, *Formal Aspects of Security*, pages 81–96, Berlin, Heidelberg, 2003. Springer. `doi:10.1007/978-3-540-40981-6_9`.

22 James Oswald. James-Oswald/linear-temporal-logic, November 2024. URL: `https://github.com/James-Oswald/linear-temporal-logic`.

23 Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, Berlin, Heidelberg, June 1992. Springer-Verlag. `doi:10.1007/3-540-55602-8_217`.

24 Nir Piterman and Amir Pnueli. Temporal Logic and Fair Discrete Systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 27–73. Springer International Publishing, Cham, 2018. `doi:10.1007/978-3-319-10575-8_2`.

25 Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, October 1977. `doi:10.1109/SFCS.1977.32`.

26 Amir Pnueli and Tamarah Arons. TLPVS: A PVS-Based ltl Verification System. In Nachum Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pages 598–625. Springer, Berlin, Heidelberg, 2003. `doi:10.1007/978-3-540-39910-0_26`.

27 Andoni Rodríguez and César Sánchez. Boolean Abstractions for Realizability Modulo Theories. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 305–328, Cham, 2023. Springer Nature Switzerland. `doi:10.1007/978-3-031-37709-9_15`.

28 Dante Zanarini, Carlos Luna, and Luis Sierra. Alternating-Time Temporal Logic in the Calculus of (Co)Inductive Constructions. In Rohit Gheyi and David Naumann, editors, *Formal Methods: Foundations and Applications*, pages 210–225, Berlin, Heidelberg, 2012. Springer. `doi:10.1007/978-3-642-33296-8_16`.