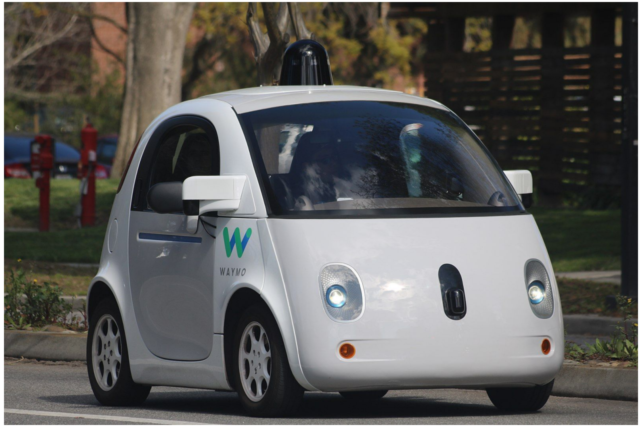
LeanLTL: A Unifying Framework for Linear Temporal Logics in Lean

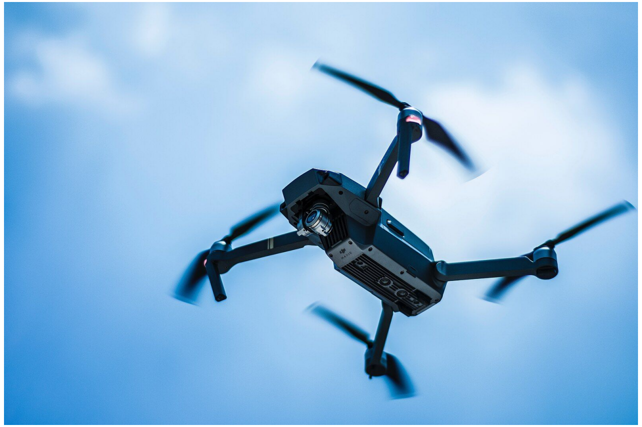
Eric Vin, Kyle A. Miller, Daniel J. Fremont

UNIVERSITY OF CALIFORNIA SANTA CRUZ

Motivations

- **Learning enabled cyber-physical systems** are prevalent and often **safety-critical**.
- Such systems can be **tested**, but are difficult to **verify**.
- **Linear temporal logic (LTL) is a modal temporal logic** that has long been used in the verification community to **reason about system properties over time**.





LeanLTL

- **LeanLTL** is a **unified framework** for reasoning about **linear temporal properties** of systems in **Lean 4** with **convenient syntax and automation.**
- Applied in upcoming work as part of a larger verification framework to **verify simplified but non-trivial automatic emergency braking system** (700+ line proofs, available in LeanLTL repo).

```
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
```

Example: A short proof in LeanLTL that for all infinite traces with a natural number variable n, the LTL-with-nonlinear-arithmetic formula $n = 5 \land \mathbf{G}((\mathbf{X}\ n) = n^2) \rightarrow \mathbf{G}(5 \leq n)$ holds.

Outline

1. Motivations
2. **Background**
3. Core Library
4. Embeddings and Applications
5. Future Work

Linear Temporal Logic

- **Linear Temporal Logic** [1]:
  - Finite set of propositional variables **P**
  - Includes the **standard logical operators** ($\neg$, $\vee$, $\wedge$, $\rightarrow$)
  - **Time is discrete** and over an **infinite horizon**.
  - Includes two **temporal operators**:
    - **X** Ψ **:** Ψ must hold in the next timestep
    - Ψ **U** Φ : Ψ must hold until Φ holds. If Φ never holds, Ψ must hold forever.
  - Additional operators can be defined using the above:
    - **G** Ψ: Ψ must hold in this and all future timesteps.
    - **F** Ψ: Ψ must eventually hold.
- **Examples**:
  - `LightYellow`$\rightarrow$ **X** `LightRed` : If the light is currently yellow, it will be red in the next timestep.
  - **G F** `LightGreen`: The light must *always eventually* turn green (i.e. the light always turns green at some point in the future).

[1] Pnueli, "The Temporal Logic of Programs."

Linear Temporal Logic Extensions

- **Finite Linear Temporal Logic (LTLf)** [2]:
  - Defined over a finite instead of infinite time horizon.
  - Next operator (**X**) split into two operators: **weak and strong next**.
    - **Weak Next:** If last timestep, vacuously true. Else **X**.
    - **Strong Next:** If last timestep, vacuously false. Else **X**.

- **Linear Temporal Logic Modulo Theories (LTLMT)** [3]:
  - Adds support for **SMT-style theories.**
  - Next operator extended to also apply to **values in theories**.
  - Also has a **finite** extension (LTLfMT).
  - **Example:**
    - $(a = 0) \land \mathbf{G}\,((\mathbf{X}\,a) = a + 1) \rightarrow \mathbf{F}\,(a > 10)$

[2] De Giacomo and Vardi, "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces."

[3] Geatti et al., "Linear Temporal Logic Modulo Theories over Finite Traces."

Why Interactive Theorem Provers?

- **LTL and LTLf are decidable** and there are known **efficient decision procedures**
- Adding **theories increases complexity**, but recent work [4] has shown decidability for **some theories**.
- Using **undecidable theories is often essential** to prove useful things, but cannot be solved automatically in all cases.
- **Example:** Nonlinear arithmetic

$$((\leftarrow n) = 5 \wedge \mathtt{G}\ ((\mathtt{X}\ (\leftarrow n)) = (\leftarrow n)\ \hat{}\ 2)) \to \mathtt{G}\ (5 \le (\leftarrow n))$$

- **Solution:** Interactive theorem provers!

[4] Geatti et al., “Decidable Fragments of LTLf Modulo Theories”

Project Overview

- **LeanLTL is a framework and Lean4 library that:**
  - Can to be used to **reason about linear temporal properties** of systems.
  - Has core types for modeling temporal properties across both **infinite and finite traces**.
  - Has support for **arbitrary Lean expressions inside formulas** (i.e. theories)
  - Includes **convenient macro syntax** for creating LeanLTL formulas.
  - Supports **automation** to simplify reasoning.

Outline

1. Motivations
2. Background
3. **Core Library**
4. Embeddings and Applications
5. Future Work

Traces

- **Traces** model properties across time:
  - Can have **finite or infinite length**.
  - **Next operator** enabled by **shift** operator which drop terms from the beginning of the sequence.
- **Example:**
  - `Light1Green : Trace Prop` := [True, True, False, True, …]
  - `Light1Queue : Trace Nat` := [1, 2, 3, 2, 2, …]

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

Traces Sets and Functions

- **Trace Sets** represent a formula by its **set of satisfying traces** extensionally
  - Aligns with the definition of `Set` in Lean's Mathlib
- **Trace Functions** are functions from a given trace domain to an `Option` type.
  - Represent operators over traces
  - **None values indicate exceptional behavior**, such as querying a value past the end of a trace. Extracting a value is done with **weak or strong get operator.**
- **Example:**
  - `TraceSet.or (f₁ f₂ : TraceSet σ) : TraceSet σ := TraceSet.map₂ (· ∨ ·) f₁ f₂`
  - `TraceFun.add [Add 𝕜] (f₁ f₂ : TraceFun σ 𝕜) : TraceFun σ 𝕜 := TraceFun.map₂ (· + ·) f₁ f₂`

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

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

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

Tools and Automation

- To aid in writing LeanLTL formulas, **we offer an `LLTL[...]` macro**.
  - **Example Macro Transformation:**

    `` `t |= LLTL[G ((←s f) < 10)]` `` **=>**

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

- Preliminary automation is centered on **simp sets**, which can be used by `simp` to transform LeanLTL formulas.
  - **Example:** `push_ltl` "pushes" the LTL "satisfies" operation as deep as possible, translating LTL operations into their first-order logic semantics
  - Transformed formulas can often be **directly solved by existing Lean tactics** like `linarith` or `omega`.

Outline

1. Motivations
2. Background
3. Core Library
4. **Embeddings and Applications**
5. Future Work

Worked Example (Definitions)

```
abbrev TL1ToTL2Green     := LLTL[G ((TL1Green ∧ ((← TL1Queue) = 0)) → (Xˢ (¬TL1Green ∧ TL2Green)))]
abbrev TL2ToTL1Green     := LLTL[G ((TL2Green ∧ ((← TL2Queue) = 0)) → (Xˢ (TL1Green ∧ ¬ TL2Green)))]
abbrev TL1StayGreen      := LLTL[G ((TL1Green ∧ ((← TL1Queue) ≠ 0)) → (Xˢ (TL1Green ∧ ¬ TL2Green)))]
abbrev TL2StayGreen      := LLTL[G ((TL2Green ∧ ((← TL2Queue) ≠ 0)) → (Xˢ (¬ TL1Green ∧ TL2Green)))]

abbrev TL1GreenDeparts   := LLTL[G (TL1Green → ((← TL1Departs) = max_departs))]
abbrev TL1RedDeparts     := LLTL[G (¬TL1Green → ((← TL1Departs) = 0))]
abbrev TL2GreenDeparts   := LLTL[G (TL2Green → ((← TL2Departs) = max_departs))]
abbrev TL2RedDeparts     := LLTL[G (¬TL2Green → ((← TL2Departs) = 0))]

abbrev TL1ArrivesBounds := LLTL[G (0 ≤ (← TL1Arrives) ∧ (← TL1Arrives) ≤ max_arrives)]
abbrev TL2ArrivesBounds := LLTL[G (0 ≤ (← TL2Arrives) ∧ (← TL2Arrives) ≤ max_arrives)]

-- Note: Queues are defined as naturals, and so won't go negative if departures exceed queue size + arrivals
abbrev TL1QueueNext      := LLTL[G ((X (← TL1Queue)) = (← TL1Queue) + (← TL1Arrives) - (← TL1Departs))]
abbrev TL2QueueNext      := LLTL[G ((X (← TL2Queue)) = (← TL2Queue) + (← TL2Arrives) - (← TL2Departs))]

-- Goal Properties
abbrev G_OneLightGreen    := LLTL[G (TL1Green ↔ ¬TL2Green)]
```

Worked Example (Proof)

```
theorem Satisfies_G_OneLightGreen : ⊨ⁱ LLTL[TLBaseProperties → G_OneLightGreen] := by
  simp [TLBaseProperties, TraceSet.sem_imp_inf_iff, TraceSet.sat_imp_iff]
  intro t h_t_inf h
  simp [TraceSet.sat_and_iff] at h
  rcases h with ⟨h1, h2, h3, h4, h5, h6, h7, h8, h9, h10, h11, h12, h13, h14⟩

  apply TraceSet.globally_induction
  . simp [push_ltl] at h1 h2 ⊢
    tauto
  . simp [push_ltl, h_t_inf, TraceFun.eval_of_eq] at h3 h4 h5 h6 ⊢
    intro n hn
    by_cases h : t.shift n (Trace.coe_lt_length_of_infinite h_t_inf n) ⊨ LLTL[TL1Green]
    · specialize h3 n h
      specialize h5 n h
      tauto
    · specialize h4 n
      specialize h6 n
      tauto
```

Embeddings and Applications

- We show that **LTL and LTLf can be directly embedded into LeanLTL**.
- We have applied LeanLTL as part of **example verifying a simplified Automatic Emergency Braking System**.
    - Many proofs, some quite complicated **(700+ lines)**.
    - Uses **undecidable theories**, so could not have been accomplished without **manual proving effort**.
- Actively working on **incorporating LeanLTL into other verification tools**, to hopefully be applied to aid in **verifying real world systems**.

LeanLTL

LeanLTL is a **unified framework** for reasoning about **linear temporal properties** of systems in **Lean 4** with **convenient syntax and automation.**

Future Work:

- More automation, including incorporating **best-effort solver for some decidable fragments** as Lean tactics.
- Show **embeddability of LTLMT and LTLfMT**.
- Support for other LTL variants, such as **past-time and bounded-time operators**

LeanLTL Repo:

https://github.com/UCSCFormalMethods/LeanLTL