15-414: Bug Catching: Automated Program Verification

Lecture Notes on LTL Model Checking

Matt Fredrikson

Carnegie Mellon University
Lecture 17
October 30, 2025

1 Introduction

We've seen how to check Computation Tree Logic (CTL) formulas against computation structures. The algorithm for doing so directly computes the semantics of formulas, and makes use of the fixpoint properties of monotone functions to derive the set of states in a transition structure that satisfy the formula. We saw in a previous lecture that LTL formulas are defined over traces, of where there are infinitely many in a computation structure, so a similar approach will not work for LTL.

In this lecture, we will see how to check LTL formulas against computation structures by reducing the problem to checking whether the language defined by a finite automaton is empty. However, because the traces of a computation structure are infinite, we cannot use the familiar tools available for nondeterministic finite automata (NFAs), and instead need to define a new type of automata that can recognize infinite words. These are called Büchi automata, and we will see that they have useful properties that can be used to construct effective model checking algorithms for LTL.

2 Kripke structures & LTL

In previous lectures we introduced Kripke and computation structures, which capture the transition behavior of a computation.

Definition 1 (Kripke structure)**.** A *Kripke frame* $(W, \curvearrowright)$ consists of a set W with a transition relation $\curvearrowright \subseteq W \times W$ where $s \curvearrowright t$ indicates that there is a direct transition from s to t in the Kripke frame $(W, \curvearrowright)$. The elements $s \in W$ are also called states. A *Kripke structure* $K = (W, \curvearrowright, v, I)$ is a Kripke frame $(W, \curvearrowright)$ with a mapping $v : W \to 2^V$, where 2^V

is the powerset of V assigning truth-values to all the propositional atoms in all states. Moreover, a Kripke structure has a set of initial states $I \subseteq W$.

A (computation) *path* is an infinite sequence $s_0, s_1, s_2, s_3, \ldots$ of states $s_i \in W$ such that $s_i \curvearrowright s_{i+1}$ for all i. We will always assume that the structures used in model checking are computation structures, unless otherwise noted. We can also refer to the *traces* of a Kripke structure, by evaluating each of its paths under the labeling function.

Like CTL, the temporal modalities of LTL allow us to formalize properties that involve time and sequencing. While the semantics of CTL formulas are defined over the states of a transition structure, the truth value of LTL formulas is defined over traces. Definition 2 gives the meaning of an LTL formula over a trace. Definition 3 extends the semantics to transition systems, where we require that for all traces σ obtained by running a computation structure K, $\sigma \models P$.

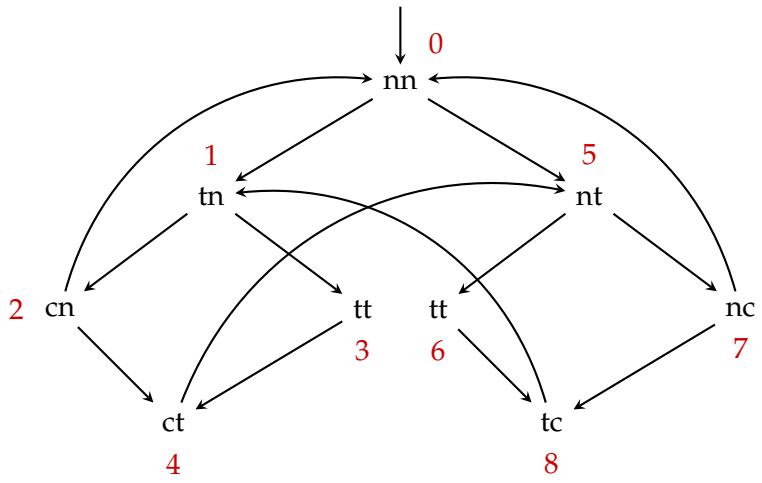
Definition 2 (LTL semantics (traces))**.** The truth of LTL formulas in a trace σ is defined inductively as follows:

1. $\sigma \models p$ iff $\sigma_0 \models p$ for atomic propositions p provided that $\sigma_0 \neq \Lambda$
2. $\sigma \models \neg P$ iff $\sigma \not\models P$, i.e. it is not the case that $\sigma \models P$
3. $\sigma \models P \wedge Q$ iff $\sigma \models P$ and $\sigma \models Q$
4. $\sigma \models \mathbf{X}P$ iff $\sigma^1 \models P$
5. $\sigma \models \Box P$ iff $\sigma^i \models P$ for all $i \geq 0$
6. $\sigma \models \Diamond P$ iff $\sigma^i \models P$ for some $i \geq 0$
7. $\sigma \models \mathbf{U}PQ$ iff there is an $i \geq 0$ such that $\sigma^i \models Q$ and $\sigma^j \models P$ for all $0 \leq j < i$

In all cases, the truth-value of a formula is, of course, determined by the state at which σ is at any given time.

Definition 3 (LTL semantics (transition systems))**.** Let $K = (W, \curvearrowright, v)$ be a transition system and P an LTL formula over the set of atomic propositions Σ. Then $K \models P$ iff for all traces σ of K, $\sigma \models P$.

To make this more concrete, consider the two-process mutual exclusion computation structure shown below. The atomic propositions c_i, t_i, n_i correspond to process i being in its critical section, trying to enter, and being in the noncritical section, respectively. The initial state is the root of the tree, and transitions represent steps taken by exactly one process at a time.



We can express some useful properties about the potential behavior of this computation using LTL formulas.

- The mutual exclusion safety property $\Box(\neg c_1 \vee \neg c_2)$ characterizes traces where it is never the case that both processes are in the critical section at the same time. Equivalently, traces where at all times it is true that either $\neg c_1$ or $\neg c_2$.
- The liveness property $\Box(t_1 \rightarrow \Diamond c_1) \wedge \Box(t_2 \rightarrow \Diamond c_2)$ characterizes traces that satisfy the requirement that whenever a process tries to enter its critical section (t_i is true), it eventually succeeds (c_i becomes true).

Let's first consider the mutual exclusion safety property. In order to check that the transition structure satisfies it, we need to verify that all traces in the structure satisfy $\neg c_1 \vee \neg c_2$. As the set of traces in this structure is infinite, approaching this directly by exhaustive enumeration will not be productive. Indeed, we could proceed inductively as we have for other unbounded computations in this course.

But our experience with induction has always relied heavily on providing an invariant from which we can build a sufficiently strong inductive hypothesis. We want to develop a completely automatic technique for verifying LTL formulas, so thinking about this problem differently might yield something more suitable.

A formal language perspective Recalling that the semantics of LTL formulas are defined over traces, we can define the language $\mathcal{L}(P)$ of an LTL formula P as the set of traces that satisfy P.

Definition 4 (LTL Semantics (language over traces))**.** Let P be an LTL formula and Σ a set of atomic propositions. Then the language of P is defined as:

$$\mathcal{L}(P) = \{\sigma \in \Sigma^\omega \ : \ \sigma \models P\}$$

where Σ^ω is the set of infinite strings over Σ, and the truth relation $\models$ is defined inductively in Definition 2.

Definition 4 equates the meaning of an LTL formula with a language that describes every behavior that is allowed by the property. Viewing this set as a language, each word in the language is an infinite-length string with characters that correspond to atomic propositions. For example, the mutual exclusion property from earlier has the following word in its language:

$$\sigma = (\{\}, \{c_2\}, \{c_1\}, \{\}, \dots \text{ (repeated infinitely)})$$

In the above, we use the convention that any atomic proposition not appearing in a state is assumed to be false; so the appearance of $\{\}$ means that no atomic proposition is true, whereas $\{c_1\}$ means that c_1 is true but c_2 is false.

The following word is not in its language, because c_1 and c_2 are simultaneously true in the fourth state:

$$\sigma = (\{\}, \{c_2\}, \{c_1\}, \{c_1, c_2\}, \dots \text{ (repeated infinitely)})$$

We can also define the set of traces $\mathcal{L}(K)$ of a computation structure K, as the set of all infinite-length words over atomic propositions obtained by following transitions in K from an initial state. $\mathcal{L}(K)$ corresponds to all of the possible behaviors that K might exhibit in its execution.

Definition 5 (Language of a computation structure)**.** Let $K = (W, \curvearrowright, v)$ be a computation structure defined over a set of atomic propositions Σ. Then the language of K, denoted $\mathcal{L}(K)$, is: $\mathcal{L}(K) = \{\sigma \in \Sigma^\omega \ : \ s_0, s_1, \dots \text{ a path in } K \text{ and } \sigma_i = v(s_i)\}$.

In the computation structure given above, one such behavior (i.e. word in the language) would be:

$$\sigma = (\{n_1, n_2\}, \{n_1, t_2\}, \{n_1, c_2\}, \dots \text{ (repeated infinitely)})$$

Interpreting the LTL formula and computation structure as languages gives us a new way to think about the model checking problem. Namely, we can reason that in order for a transition structure K to satisfy formula P, it must be that every trace of K satisfies P. The languages $\mathcal{L}(P)$ gives us exactly the set of traces that satisfy P, so we have only to check that the language $\mathcal{L}(K)$ is contained in $\mathcal{L}(P)$:

$$\mathcal{L}(K) \subseteq \mathcal{L}(P) \tag{1}$$

Equation 1 equivalent to saying that all of the behaviors of K are among the set of behaviors that are allowed by P.

Checking by complement How can we check whether Equation 1 holds for a given K and P? Suppose for the moment that $\mathcal{L}(K)$ and $\mathcal{L}(P)$ were regular languages containing only finite words. Then we could exploit the fact that regular languages are closed under intersection and complementation, in addition to the following fact:

$$\mathcal{L}(K) \subseteq \mathcal{L}(P) \text{ if and only if } \mathcal{L}(K) \cap \overline{\mathcal{L}(P)} = \emptyset \tag{2}$$

$\overline{\mathcal{L}(P)}$ is the complement of $\mathcal{L}(P)$, i.e., the set of all behaviors that are not allowed by P. We can check that Equation 2 matches the intuition developed so far: if $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)}$ is empty, then there are no behaviors of K that are *not* allowed by P. Removing the double negative, *all* behaviors of K are allowed by P.

Assuming we have the finite-state machine corresponding to a regular language, checking whether that language is empty is a reachability problem: we simply look for a path through the automaton from an initial state to an accepting state. This suggests the following algorithm for checking property P against transition structure K (assuming both are equivalent to regular languages):

1. Construct finite-state machines A_K and $A_{\overline{P}}$ corresponding to $\mathcal{L}(A)$ and $\overline{\mathcal{L}(P)}$, respectively. We know that $A_{\overline{P}}$ exists because regular languages are closed under complementation.

2. Use the fact that regular languages are closed under intersection to compute $A_{K \cap \overline{P}}$ from A_K and $A_{\overline{P}}$.

3. Check whether $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)}$ is empty by looking for a path in $A_{K \cap \overline{P}}$ from an initial state to an accepting state.
   a) If $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)} = \emptyset$, then conclude that $\mathcal{L}(K) \subseteq \mathcal{L}(P)$ so K satisfies P ($K \models P$).
   b) If $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)} \neq \emptyset$, then conclude that $K \not\models P$. Any word in $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)}$ corresponds to a counterexample of P, i.e., a trace exhibiting a behavior in K that is not allowed by P.

This procedure is appealing for several reasons. It is completely automatic, and reduces model checking to a reachability problem over the graph of an automata. In cases where the transition structure does not satisfy the property in question, there is a simple procedure for extracting counterexamples that witness this fact; such counterexamples can be useful in practice for diagnostic reasons by highlighting behaviors that violate the property.

Of course, we can't actually use this procedure to check LTL formulas against computation structures because we know that $\mathcal{L}(P)$ and $\mathcal{L}(K)$ are not regular languages—their words are infinite, and can't be recognized by finite state machines.