15-414: Bug Catching: Automated Program Verification

Lecture Notes on Branching-Time Properties

Matt Fredrikson

Carnegie Mellon University
Lecture 20
Thursday, November 12, 2026

1 Introduction

In the last two lectures, we used Linear Temporal Logic (LTL) to describe properties of individual computations and developed a model checking algorithm using Büchi automata. When we check an LTL formula against a computation structure, we ask whether it holds on every trace. But a trace records only the choices that were made during one execution. It does not record the choices that were available along the way.

Today we will study Computation Tree Logic (CTL), which can express properties of those choices. Its formulas are evaluated at states, where they can quantify over all possible future paths or ask whether some future path exists. We will use the same Kripke structures as before, but view their behavior as a branching computation tree. Throughout these notes, we write $\Box$ and $\Diamond$ for LTL's always and eventually operators, and $\mathbf{G}$ and $\mathbf{F}$ for the corresponding CTL temporal operators.

Learning Goals

After this lecture, you will:

- Model computations with Kripke structures and distinguish paths from traces.
- Understand CTL formulas and their semantics on computation trees.
- Express temporal properties using CTL.
- Compare properties of traces with properties of branching choices.

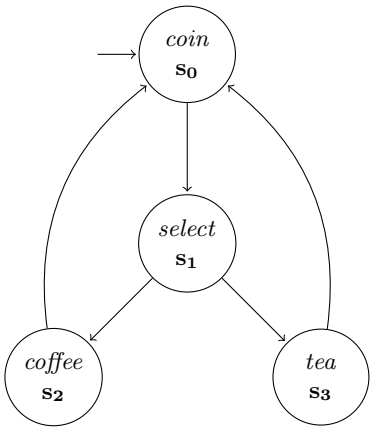


Figure 1: Computation structure describing the operation of a vending machine.

2 Modeling Computation Over Time

We began by modeling computation with programs, written in a language whose semantics allow us to reason about their behavior mathematically. For model checking, we use transition systems called *Kripke structures*. Definition 1 recalls this model.

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 finite set of atomic propositions V and a labeling function $v : W \to 2^V$. The set $v(s)$ contains exactly the propositions true at state s. Moreover, a Kripke structure has a set of initial states $I \subseteq W$.

A Kripke structure $K = (W, \curvearrowright, v, I)$ is called a *computation structure* if W is a finite set of states and every element $s \in W$ has at least one direct successor $t \in W$ with $s \curvearrowright t$. 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.

Example 2. An example of a Kripke structure that represents a vending machine is shown in Figure 1.

The states represented in Figure 1 are $W = \{s_0, s_1, s_2, s_3\}$. The propositional atoms V that appear in those states are $V = \{\text{coin,select,coffee,tea}\}$. The initial-state set is $I = \{s_0\}$. The mapping v is represented as follows:

$$s_0 \to \{\text{coin}\}$$
$$s_1 \to \{\text{select}\}$$
$$s_2 \to \{\text{coffee}\}$$
$$s_3 \to \{\text{tea}\}$$

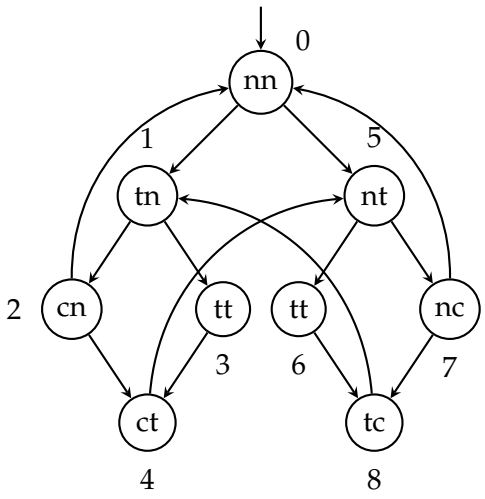


Figure 2: Kripke structure for a mutual exclusion protocol.

Note that we only show the propositional atoms that are assigned the truth value $true$ but the remaining atoms would be assigned truth value $false$. Finally, the transition relation $\curvearrowright$ is defined as: $\{s_0 \curvearrowright s_1, s_1 \curvearrowright s_2, s_1 \curvearrowright s_3, s_2 \curvearrowright s_0, s_3 \curvearrowright s_0\}$.

What are examples of properties that we might want to check against such a computation? For one, being able to conclude that the vending machine does not dispense coffee or tea until after a user produces a coin would be helpful. Likewise, we may want to check that once a coin has been presented, the machine will eventually dispense *some* beverage. These are easy to check by visual inspection of Figure 1. In general, this will not be the case, as the number of states and transitions comprising a system are too large to simply inspect. □

Example 3. Figure 2 shows another example of a Kripke structure, which models a mutual exclusion protocol.

This models two abstract processes vying for some resource denoted by a *critical section*. The aim of the protocol is to ensure that both processes cannot access the critical section at the same time. The states denote the current status of each process, which are either currently in the **n**oncritical section, **t**rying to enter the critical section, or are in the **c**ritical section. Those atomic propositional letters are used with suffix 1 to indicate that they apply to process 1 and with suffix 2 to indicate process 2. For example the notation nt indicates a state in which $n_1 \wedge t_2$ is true (and no other propositional letters).

One of the properties of interest is obviously the mutual exclusion property itself: that both processes don't enter the critical section simultaneously. This is apparent from Figure 2, where none of the reachable states has both c_1 and c_2 in its label. A more interesting property has to do with *liveness*: once a process tries to enter the critical section, does it eventually succeed? Even this small example calls for a systematic check. □

2.1 Paths and Traces

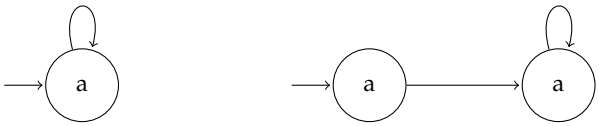
When it comes to the computation paths modeled by Kripke structures, there is a key distinction to keep in mind. We defined a path as a sequence of states under the transition relation. We can also refer to the *traces* of a Kripke structure, by evaluating each of its paths under the labeling function. So for the structure in Figure 1, the paths would be:

$$
\begin{aligned}
&s_0, s_1, s_2, s_0, s_1, s_2, \ldots \\
&s_0, s_1, s_3, s_0, s_1, s_3, \ldots \\
&s_0, s_1, s_2, s_0, s_1, s_3, \ldots \\
&s_0, s_1, s_3, s_0, s_1, s_2, \ldots \\
&\vdots
\end{aligned}
$$

And the corresponding traces would be:

$$
\begin{aligned}
&\mathit{coin}, \mathit{select}, \mathit{coffee}, \mathit{coin}, \mathit{select}, \mathit{coffee}, \ldots \\
&\mathit{coin}, \mathit{select}, \mathit{tea}, \mathit{coin}, \mathit{select}, \mathit{tea}, \ldots \\
&\mathit{coin}, \mathit{select}, \mathit{coffee}, \mathit{coin}, \mathit{select}, \mathit{tea}, \ldots \\
&\mathit{coin}, \mathit{select}, \mathit{tea}, \mathit{coin}, \mathit{select}, \mathit{coffee}, \ldots \\
&\vdots
\end{aligned}
$$

Here each displayed proposition abbreviates a singleton label, such as $\{\mathit{coin}\}$. Given a Kripke structure K, we write $\mathsf{Pa}(K)$ for the paths starting in its initial states and $\mathsf{Tr}(K)$ for their traces. Different paths can have the same trace when the labels hide distinctions between states. Even distinct structures can have the same trace set: each of the following structures generates only the trace $\{a\}, \{a\}, \ldots$.



3 Computation Tree Logic

The types of properties discussed in the previous examples all aim to characterize what takes place in a computation over time. To formally model such properties, we will rely on temporal logic. Computation Tree Logic (CTL) is a widely-used temporal logic, which views the computations embodied in Kripke structures in terms of the branching possibilities at each state. Consider the Kripke structure shown on the left in Figure 3. The tree on the right of the figure represents each of the paths that the computation could take; for example it always begins in a, and one possible path repeatedly enters b, then $a, b, a, \ldots$ infinitely often. Another path goes directly from a to c, and stays in c forever.

Formulas in CTL represent properties of paths that are reachable from a given state, and are thus called *state formulas*. We write $K \models P$ when $K, s \models P$ for every initial state

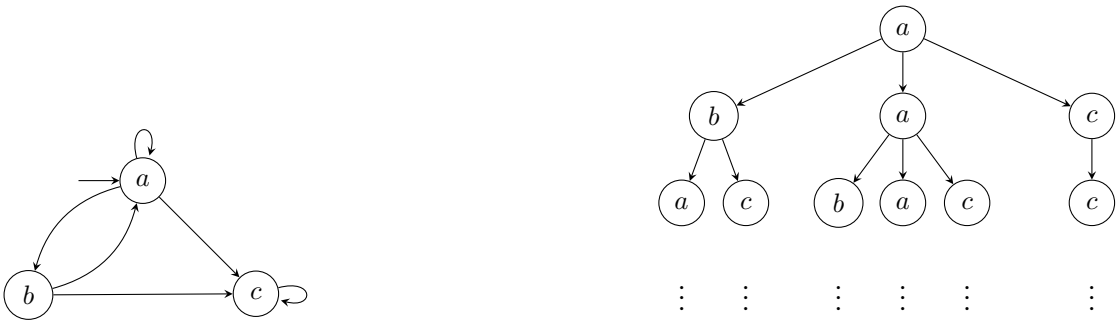


Figure 3: A Kripke structure and the first two levels of its computation tree. State labels are shown inside the nodes. Every occurrence of a state has all of that state's successors.

$s \in I$. CTL formulas use the **E** (existential) and **A** (universal) *path quantifiers*, which ask whether there exists a path with a given property, or whether all paths exhibit a given property.

- $\mathbf{E}P$ is a state formula where for a given Kripke structure K we have the following:

$$K, s \models \mathbf{E}P \leftrightarrow \text{ there } \textbf{exists} \text{ a path } \pi \text{ starting at } s \text{ where } \pi \models P$$

- $\mathbf{A}P$ is a state formula where for a given Kripke structure K we have the following:

$$K, s \models \mathbf{A}P \leftrightarrow \text{ for } \textbf{all} \text{ paths } \pi \text{ starting at } s, \pi \models P$$

Path quantifiers are always paired with a *path formula*, which specifies a property over a given single path. If P and Q are state formulas, then the following are the path formulas allowed directly after a CTL path quantifier.

- $\mathbf{X}P$: The next state in the path satisfies P.
- $\mathbf{G}P$: All states in the path satisfy P.
- $\mathbf{F}P$: There exists some state on the path that satisfies P.
- $P\mathbf{U}Q$: There exists some state on the path that satisfies Q. Until then, all states satisfy P.

Each temporal operator in CTL must be immediately preceded by a path quantifier. For example, $\mathbf{AFAG}P$ is a CTL formula, whereas $\mathbf{AFG}P$ is not: the inner $\mathbf{G}$ has no path quantifier. Putting all of this together, the semantics of the logic is shown in Definition 4.

Definition 4. In a fixed computation structure $K = (W, \curvearrowright, v, I)$, the truth of CTL formulas in state s is defined inductively as follows:

1. $s \models p$ iff $p \in v(s)$ for atomic propositions p
2. $s \models \neg P$ iff $s \not\models P$, i.e. it is not the case that $s \models P$
3. $s \models P \wedge Q$ iff $s \models P$ and $s \models Q$
4. $s \models \mathbf{AX}P$ iff all successors t with $s \curvearrowright t$ satisfy $t \models P$
5. $s \models \mathbf{EX}P$ iff at least one successor t with $s \curvearrowright t$ satisfies $t \models P$
6. $s \models \mathbf{AG}P$ iff all paths $s_0, s_1, s_2, \ldots$ starting in $s_0 = s$ satisfy $s_i \models P$ for all $i \geq 0$
7. $s \models \mathbf{AF}P$ iff all paths $s_0, s_1, s_2, \ldots$ starting in $s_0 = s$ satisfy $s_i \models P$ for some $i \geq 0$
8. $s \models \mathbf{EG}P$ iff some path $s_0, s_1, s_2, \ldots$ starting in $s_0 = s$ satisfies $s_i \models P$ for all $i \geq 0$
9. $s \models \mathbf{EF}P$ iff some path $s_0, s_1, s_2, \ldots$ starting in $s_0 = s$ satisfies $s_i \models P$ for some $i \geq 0$
10. $s \models \mathbf{A}[P\mathbf{U}Q]$ iff all paths $s_0, s_1, s_2, \ldots$ starting in $s_0 = s$ have some $i \geq 0$ such that $s_i \models Q$ and $s_j \models P$ for all $0 \leq j < i$
11. $s \models \mathbf{E}[P\mathbf{U}Q]$ iff some path $s_0, s_1, s_2, \ldots$ starting in $s_0 = s$ has some $i \geq 0$ such that $s_i \models Q$ and $s_j \models P$ for all $0 \leq j < i$

Returning to the notion of computation trees from before, Figure 4 visualizes the semantics of two CTL formulas.

3.1 Useful equivalences

Some of the CTL formulas are redundant in the sense that they are definable with other CTL formulas already. But the meaning of the original formulas is usually much easier to understand than the meaning of its equivalent.

Lemma 5. *The following are valid CTL equivalences:*

1. $\mathbf{EF}P \leftrightarrow \mathbf{E}[true\mathbf{U}P]$
2. $\mathbf{AF}P \leftrightarrow \mathbf{A}[true\mathbf{U}P]$
3. $\mathbf{EG}P \leftrightarrow \neg\mathbf{AF}\neg P$
4. $\mathbf{AG}P \leftrightarrow \neg\mathbf{EF}\neg P$
5. $\mathbf{AX}P \leftrightarrow \neg\mathbf{EX}\neg P$
6. $\mathbf{A}[P\mathbf{U}Q] \leftrightarrow \neg\mathbf{E}[\neg Q\mathbf{U}(\neg P \wedge \neg Q)] \wedge \neg\mathbf{EG}\neg Q$

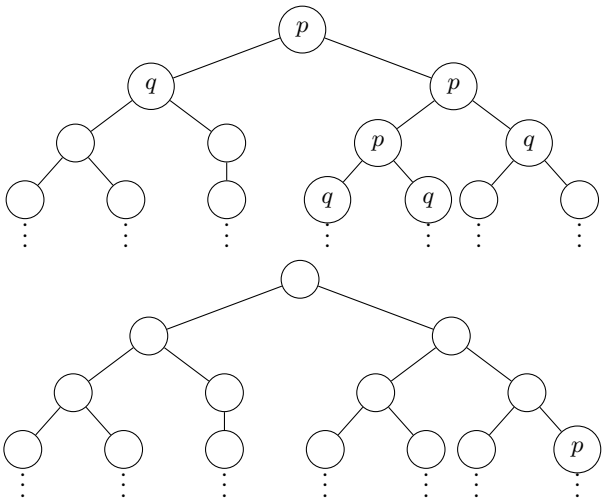


Figure 4: CTL formulas $\mathbf{A}[p\mathbf{U}q]$ (top) and $\mathbf{EF}p$ (bottom). Blank nodes have no required label; the drawings can be extended to infinite trees.

Most of these cases except the last are quite easy to prove. So as not to confuse ourselves, we will definitely make use of the finally and globally operators in applications. The equivalences let us reduce the operators used by a model checking algorithm. With both path quantifiers, next and until suffice. If we also eliminate the universal path quantifier, a convenient basis is **EX**, **EG**, and **EU**, together with the Boolean connectives.

Example 6. Temporal logic is particularly helpful to verify properties of distributed systems. For example, we may want to reason about safety or liveness. Safety properties state that "nothing bad would ever happen", whereas liveness properties state that "something good eventually happens". We will see how to encode safety and liveness using CTL for a mutual exclusion protocol.

Returning to the mutual exclusion example from before, the properties that we proposed exemplify these ideas. Safety corresponds to the mutual exclusion property: a "bad thing" happens if both processes ever enter the critical section at the same time. Liveness corresponds to the starvation-freedom property: the requirement is that once a process tries to enter the critical section, it will eventually succeed.

1. Safety: $\neg\mathbf{EF}(c_1 \wedge c_2)$ is trivially true since there is no state labelled cc.
2. Liveness: $\mathbf{AG}(t_1 \rightarrow \mathbf{AF}c_1) \wedge \mathbf{AG}(t_2 \rightarrow \mathbf{AF}c_2)$

□

4 Comparison between LTL and CTL

LTL describes what happens along a single path. When we say that a computation structure satisfies an LTL formula, we require that formula to hold on every initial trace. CTL makes path quantification part of the formula itself. In particular, a nested CTL formula can ask about the alternatives available at a state reached along a path, even if those alternatives are never taken on that path.

For example, the LTL formula $P \,\mathbf{U}\, Q$ says that Q eventually holds on the path being considered and that P holds at every earlier position. The CTL formula $\mathbf{A}[P\mathbf{U}Q]$ requires this on every path from the current state, while $\mathbf{E}[P\mathbf{U}Q]$ requires it on at least one path. Figure 5 illustrates the distinction. In the universal case, different paths can reach Q at different times.

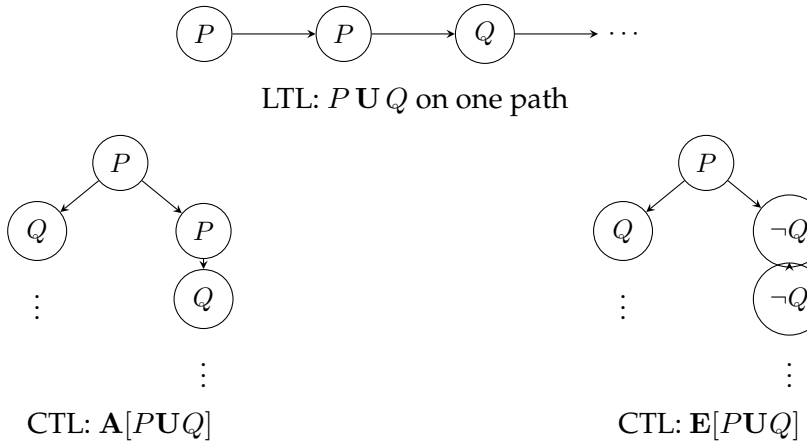


Figure 5: Until on a path and on a computation tree. The right-hand tree has a path that never reaches Q, but its left branch witnesses the existential formula. Nodes below a Q node are unrestricted.

Eventually always. It is tempting to translate an LTL formula to CTL by adding a universal path quantifier before each temporal operator. This works for some formulas, but it does not work in general. Consider

$$\underbrace{\Diamond\Box P}_{\text{LTL}} \qquad \text{and} \qquad \underbrace{\mathbf{AFAG}P}_{\text{CTL}}.$$

For a computation structure to satisfy $\Diamond\Box P$, each of its initial traces must eventually reach a point after which P is always true *on that trace*. The CTL formula asks for more: every initial path must eventually reach a state from which P is true *on every possible continuation*.

Figure 6 separates these formulas. A path can stay at s_0 forever, in which case P is always true. Otherwise it eventually moves through s_1 to s_2, where P remains true forever. Thus every initial trace satisfies $\Diamond\Box P$.

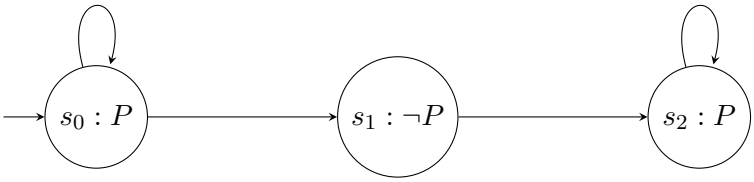


Figure 6: Every initial trace satisfies $\Diamond\Box P$, but the initial state does not satisfy $\mathbf{AFAG}P$.

However, s_0 does not satisfy $\mathbf{AG}P$, since it has a continuation through s_1, where P is false. The path that stays at s_0 forever therefore never reaches a state satisfying $\mathbf{AG}P$. This path refutes $\mathbf{AFAG}P$, even though it never actually takes the transition to s_1.

The possibility of progress. Now consider the CTL formula

$$\mathbf{AGEF}P.$$

It says that from every reachable state there is some path to a state satisfying P. For example, if P means that a server has returned to its ready state, the formula says that recovery remains possible. It does not say that the server actually recovers on every execution. The LTL formula $\Box\Diamond P$ would require P to recur infinitely often on every initial trace.

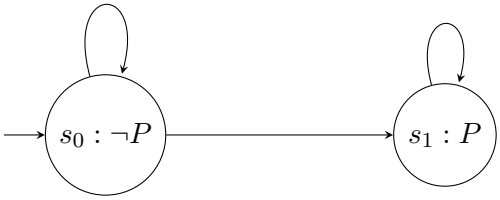


Figure 7: Reaching P is always possible, so $\mathbf{AGEF}P$ holds. The path that stays at s_0 forever shows that the LTL property $\Box\Diamond P$ does not hold.

The distinction matters when choosing a specification. An option to make progress is weaker than a guarantee that progress will occur. In Figure 7, the option exists at every state, but an execution can keep postponing it forever.

LTL and CTL are incomparable in expressive power: each can express properties that the other cannot. In particular, no CTL formula expresses the LTL property $\Diamond\Box P$, and no LTL formula expresses the CTL property $\mathbf{AGEF}P$ over all computation structures [BKL08]. The examples above distinguish particular candidate translations; the full inexpressibility results require stronger arguments. The practical lesson is to distinguish a property of the path taken from a property of the choices available along that path.

Exercises

1. Write a Kripke structure that models a traffic light that is allowed to blink yellow for arbitrarily long periods of time. The atomic propositions are G (green), Y (yellow), R (red), and B (black), where the meaning of "blink" is to alternate between yellow and black.
   - Write a CTL formula that expresses the safety property: the light will never switch from green to red without first becoming yellow.
   - Write a CTL formula that expresses the liveness property: the light will never stay red for indefinitely long.
2. Draw a computation tree that expresses the semantics of the following CTL formulas:
   - $\mathbf{AF}a$
   - $\mathbf{EG}a$
   - $\mathbf{EG}a \rightarrow \mathbf{AF}b$
   - $\mathbf{A}[a\mathbf{U}\mathbf{A}[b\mathbf{U}c]]$
3. Draw Kripke structures that model the four formulas in the previous question.
4. Determine which of the following equivalences are correct.
   - $\mathbf{AXAF}P \leftrightarrow \mathbf{AFAX}P$
   - $\mathbf{EXEF}P \leftrightarrow \mathbf{EFEX}P$
   - $\mathbf{AG}P \leftrightarrow \mathbf{AXAG}P$
   - $\neg\mathbf{A}[P\mathbf{U}P] \leftrightarrow \mathbf{E}[P\mathbf{U}\neg Q]$
5. Provide a Kripke structure which shows that the following CTL formulas are not equivalent: $\mathbf{AF}P \vee Q$ and $\mathbf{AF}P \vee \mathbf{AF}Q$.
6. Provide two Kripke structures K_0 and K_1 where $\mathsf{Tr}(K_0) = \mathsf{Tr}(K_1)$, and a CTL formula P that is true in the initial state of K_0 but not in K_1. Is your example significant in terms of modeling correct system behavior, i.e. is your example merely a theoretical exercise or can you think of a scenario where distinguishing between such formulas might matter?

References

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking*. MIT Press, 2008.