Lecture Notes on Temporal Logic

Matt Fredrikson

Carnegie Mellon University

Lecture 22
Thursday, April 13

1 Introduction

In the previous lecture, we observed a potential difference between whether a data structure invariant holds literally always at all times during all runs of all its operations (which is essentially a prerequisite for uncorrupted concurrent usages) compared to whether the data structure invariant merely holds at the end of each of the operations if it was true before. On second thought, what we have seen so far was, from the very semantics, meant as the dynamic logic for proving properties of (all or some) final states of the program under complete ignorance of what happens in between. There are ways of augmenting dynamic logics to temporal dynamic logics [BS01, **?**, **?**] that provide explicit ways of proving formulas that are true, e.g., always throughout an execution.

While this works well and continues the deductive verification principles we saw so far, we will, instead, leverage the motivation of a temporal understanding of programs as a segway into studying temporal logics [Pri57, Pnu77, Eme90] and their use in model checking [CES83, QS82, CGP99, BKL08, CES09].

Learning Goals

After this lecture, you will:

- Learn about Kripke structures, a widely-used finite-state structure for modeling computation in model checking.
- Learn Computation Tree Logic (CTL), a temporal logic that characterizes computations' behavior over time in terms of their paths through a Kripke structure.

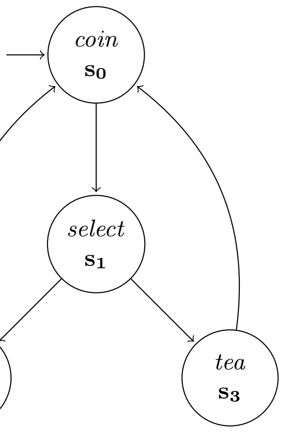


Figure 1: Computation structure describing the operation of a vending machine.

- Understand the semantics of CTL formulas in terms of computation trees that describe the branches within a Kripke structure that a computation might take.
- See how to model temporal properties using CTL.

2 Modeling Computation Over Time

So far we have always modeled computation with programs, written in a language with semantics that allow us to reason about their behavior mathematically. In our study of model checking, we will model computation primarily in terms of transition systems called *Kripke structures*, given in Definition 1.

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 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 3.

The set of states W represented in Figure 3 are $W = \{s_0, s_1, s_2, s_3\}$. The propositional atoms V that appear in those states are $V = \{$coin,select,coffee,tea$\}$. The initial state

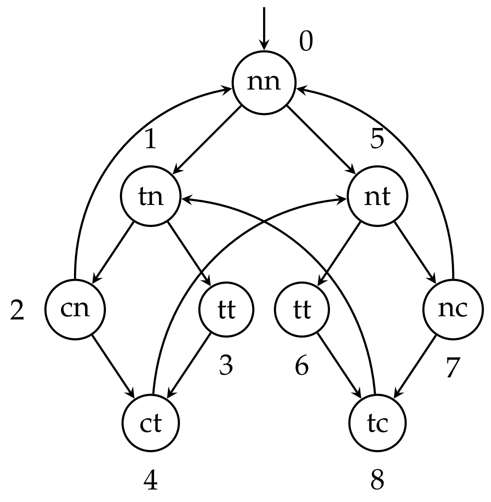


Figure 2: Kripke structure for a mutual exclusion protocol.

$I = \{s_0\}$. The mapping v is represented as follows:

$$\begin{aligned} s_0 &\to \{\text{coin}\} \\ s_1 &\to \{\text{select}\} \\ s_2 &\to \{\text{coffee}\} \\ s_3 &\to \{\text{tea}\} \end{aligned}$$

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 3. In general, this will not be the case, as the number of states and transitions comprising a system are too large to simply inspect. $\Box$

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 setion, 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 all unreachable states, including $c_1 \wedge c_2$, are omitted. A more interesting property has to do with *liveness*: once a process tries to enter the critical section, does it eventually succeed? This is still relatively easy to work out by manually tracing paths through the states, but already in this small example it becomes apparent that checking temporal properties may not always be so easy. □

2.1 Paths and Traces

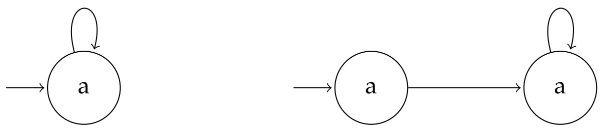
When it comes to the computation paths modeled by Kripke structures, there is a key distinction to keep in mind. We defined a path as a sequence of states under the transition relation. We can also refer to the *traces* of a Kripke structure, by evaluating each of its paths under the labeling function. So for the structure in Figure 3, the paths would be:

$$\begin{array}{l} s_0, s_1, s_2, s_0, s_1, s_2, \ldots \\ s_0, s_1, s_3, s_0, s_1, s_3, \ldots \\ s_0, s_1, s_2, s_0, s_1, s_3, \ldots \\ s_0, s_1, s_3, s_0, s_1, s_2, \ldots \\ \vdots \end{array}$$

And the corresponding traces would be:

$$\begin{array}{l} coin, select, coffee, coin, select, coffee, \ldots \\ coin, select, tea, coin, select, tea, \ldots \\ coin, select, coffee, coin, select, tea, \ldots \\ coin, select, tea, coin, select, coffee, \ldots \\ \vdots \end{array}$$

Given a Kripke structure K, we will write $\mathsf{Pa}(K)$ to denote its paths, and $\mathsf{Tr}(K)$ to denote its traces. Note that paths and traces may not be in one-to-one correspondence, as is demonstrated by the following two structures.



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

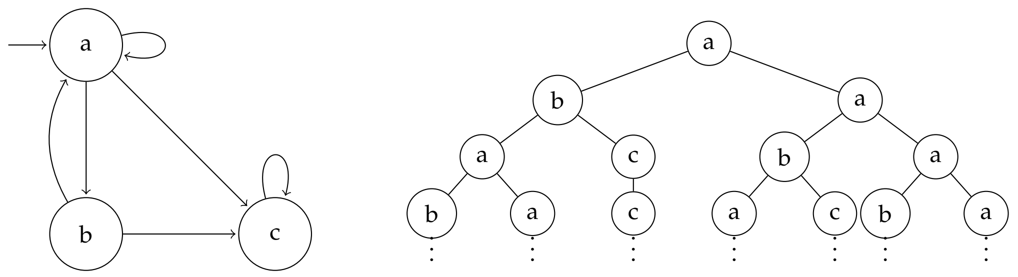


Figure 3: Kripke structure and its computation tree.

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*. By convention, when comparing a CTL formula to a Kripke structure, we always consider the paths reachable from the initial state. CTL formulas use the $\mathbf{E}$ (existential) and $\mathbf{A}$ (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 is a state formula, then the following are all path formulas.

- $\mathbf{X}\,P$: 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.

Putting all of this together, the semantics of the logic is shown in Definition 4.

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

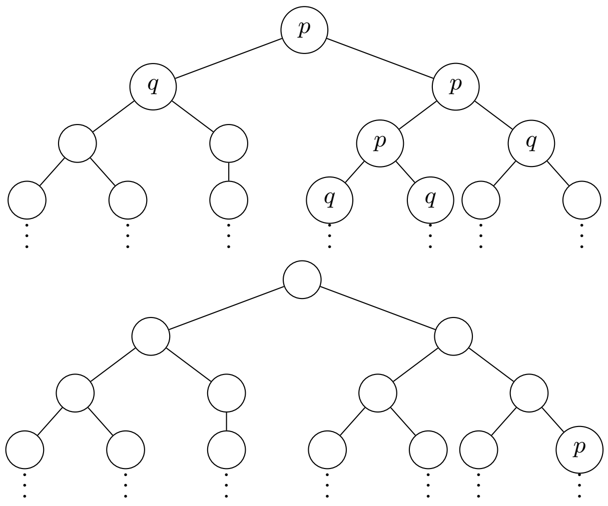


Figure 4: Visualization CTL formuls: $\mathbf{A}[P\,\mathbf{U}\,Q]$ (top) and $\mathbf{EF}\,P$ (bottom).

1. $s \models p$ iff $v(s)(p) = \mathit{true}$ 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. $s \models P \land Q$ 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. $\mathbf{EF}\,P \leftrightarrow \mathbf{E}[\mathit{true}\,\mathbf{U}\,P]$
2. $\mathbf{AF}\,P \leftrightarrow \mathbf{A}[\mathit{true}\,\mathbf{U}\,P]$
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. $\mathbf{A}[P\,\mathbf{U}\,Q] \leftrightarrow \neg\mathbf{E}[\neg Q\,\mathbf{U}\,(\neg P \wedge \neg Q)] \;\wedge\; \neg\mathbf{EG}\,\neg Q$

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. But thanks to these equivalences, when developing reasoning techniques we can simply pretend next and until would be the only temporal operators to worry about. In fact, we can even pretend only the existential path quantifier $\mathbf{E}$ is used, never the universal path quantifier $\mathbf{A}$, but this reduction in the number of different operators comes at quite some expense in the size and complexity in the resulting formulas.

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 always happens". We will how we can 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 deadlock-freedom property: the "good thing" that we want to happen (infinitely often) 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 ccx.
2. Liveness: $\mathbf{AG}\,(t_1 \rightarrow \mathbf{AF}\,c_1) \wedge \mathbf{AG}\,(t_2 \rightarrow \mathbf{AF}\,c_2)$

□

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.
   - $\mathbf{AX}\, \mathbf{AF}\, P \leftrightarrow \mathbf{AF}\, \mathbf{AX}\, P$
   - $\mathbf{EX}\, \mathbf{EF}\, P \leftrightarrow \mathbf{EF}\, \mathbf{EX}\, P$
   - $\mathbf{AG}\, P \leftrightarrow \mathbf{AX}\, \mathbf{AG}\, P$
   - $\neg \mathbf{A}[P\, \mathbf{U}\, P] \leftrightarrow \mathbf{E}[P\, \mathbf{U}\, \neg Q]$
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.

[BS01] Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In *IJCAR*, pages 626–641, 2001.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, 1999.

[Eme90] Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. MIT Press, 1990.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[Pri57] Arthur Prior. *Time and Modality*. Clarendon Press, 1957.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.