15-414: Bug Catching: Automated Program Verification

Lecture Notes on Computations & Computation Tree Logic

Matt Fredrikson Ruben Martins

Carnegie Mellon University
Lecture 17

1 Introduction

Linear temporal logic is a very important logic for model checking [Eme90, CGP99, BKL08] but has the downside that its verification algorithms are rather complex. To get a good sense of how model checking works, we, thus, consider the closely related but different(!) Computation Tree Logic (CTL) instead. Both LTL and CTL are common in model checking even if they have different advantages and downsides.

The main point about LTL is that its semantics fixes a trace and then talks about temporal properties along that particular trace. CTL instead switches to a new trace every time a temporal operator is used. CTL has the advantage of having a pretty simple model checking algorithm.

2 Kripke Structures

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* $K = (W, \curvearrowright, v)$ is a Kripke frame $(W, \curvearrowright)$ with a mapping $v : W \to \Sigma \to \{true, false\}$ assigning truth-values to all the propositional atoms in all states. Note that the states W are composed by a set of propositional atoms Σ. Moreover, a Kripke structure has a set of initial states $I \subseteq W$.

Definition 2 (Computation structure)**.** A Kripke structure $K = (W, \curvearrowright, v)$ 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* in a computation structure 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.

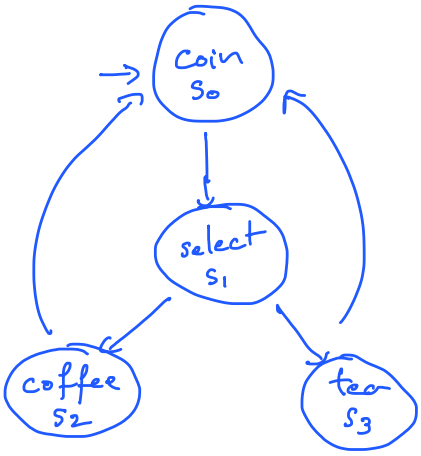


Figure 1: An example of a Kripke structure

The program semantics $[\![\alpha]\!]$ which was defined as a relation of initial and final states in Lecture 3 is an example of a Kripke structure. Another example of a Kripke structure that represents a coffee and tea machine is shown in Figure 1.

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

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

Note that we only shown 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 follows.

$$s_0 \curvearrowright s_1$$
$$s_1 \curvearrowright s_2$$
$$s_1 \curvearrowright s_3$$
$$s_2 \curvearrowright s_0$$
$$s_3 \curvearrowright s_0$$

3 Computation Tree Logic

Definition 3. In a fixed computation structure $K = (W, \curvearrowright, v)$, the truth of CTL formulas in state s is defined inductively as follows:

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. $s \models P \wedge Q$ 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. $s \models \mathbf{AU}PQ$ 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. $s \models \mathbf{EU}PQ$ 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 $0 \leq j < i$

4 Definables

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 4. *The following are valid CTL equivalences:*

1. $\mathbf{EF}P \leftrightarrow \mathbf{EU}\,\mathit{true}P$
2. $\mathbf{AF}P \leftrightarrow \mathbf{AU}\,\mathit{true}P$
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. $\mathbf{AU}PQ \leftrightarrow \neg\mathbf{EU}\neg Q(\neg P \wedge \neg Q) \wedge \neg\mathbf{EG}\neg Q$

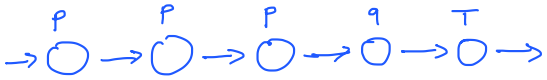


Figure 2: Visualization of a LTL formula: $P\mathbf{U}Q$

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.

5 Comparison between LTL and CTL

We mentioned before that LTL universally quantifies over paths. However, it is also possible to have existential path quantification. As we have seen in the previous sections, in CTL we can have either existential or universal quantification:

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

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

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

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

While LTL formulas describe linear-time properties (single paths), CTL formulas describe branching-time properties and can describe multiple possible futures. We can visualize LTL formulas as a sequence of states in a single line where CTL corresponds to a transition of states in a tree.

Figure 2 shows the visualization of the LTL formula $P\mathbf{U}Q$, whereas Figure 3 shows the visualization of the CTL formula $\mathbf{AU}PQ$ and Figure 4 the visualization of $\mathbf{AE}PQ$.

LTL and CTL are incomparable in terms of expressiveness. There are formulas in both logics that cannot be expressed in the other. Consider the CTL formula $\mathbf{AFAG}P$. One may think that $\mathbf{FG}P$ would be the equivalent LTL formula. Consider the language of the automaton presented in Figure 5. The language of this automaton satisfies $\mathbf{FG}P$ but does not satisfy $\mathbf{AF}(\mathbf{AG}P)$. Note that we can visualize the CTL formula in Figure 6 where we can see that there is a run in which the system will always be in the state from which a run finally goes in a non P state.

An example of a CTL formula that cannot be expressed in LTL is $\mathbf{AG}(\mathbf{EF}P)$. This formula states that there is always the possibility that a state can be reached during a

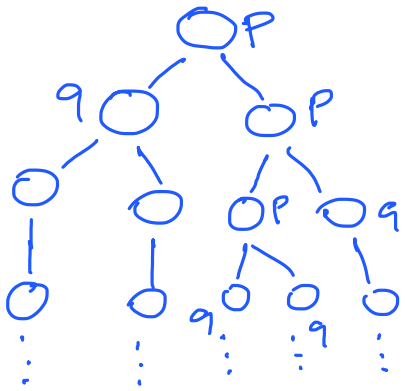


Figure 3: Visualization of a CTL formula: $\mathbf{AU}PQ$

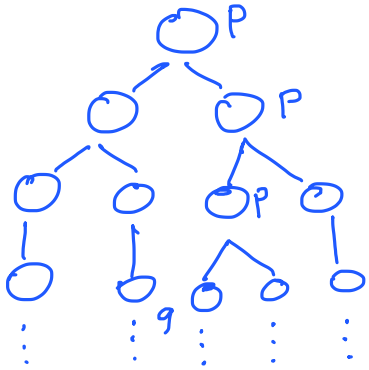


Figure 4: Visualization of a CTL formula: $\mathbf{AE}PQ$

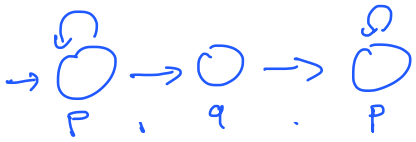


Figure 5: An example of an automaton that satisfies $\mathbf{FG}P$

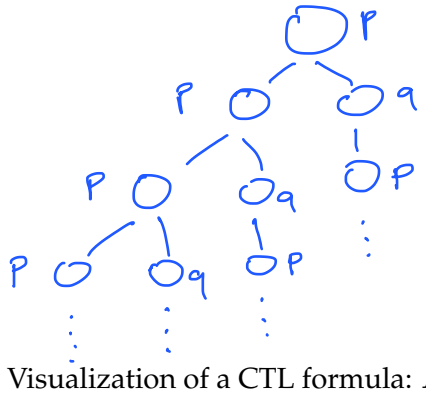


Figure 6: Visualization of a CTL formula: $\mathbf{AF}(\mathbf{AG}P)$

run, even if it is never actually reached. However, the natural corresponding LTL formula $\mathbf{GF}P$ states that at all times, P will eventually be reached. Note that this formula is stronger than the previous one since we just need the possibility of returning to P.

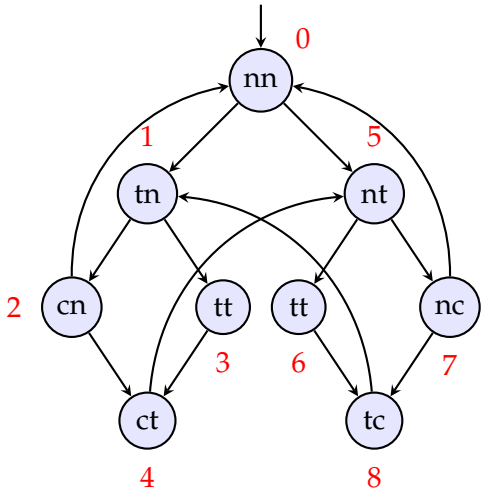
6 Example: Mutual Exclusion

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.

The notation in the following transition diagram is nt for: the first process is in the noncritical section while the second process is trying to get into its critical section.

- n noncritical section of an abstract process
- t trying to enter critical section of an abstract process
- c critical section of an abstract process

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). Consider Kripke structure



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

References

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

[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.