Lecture 16: Computation Tree Logic (CTL)
Programme for the upcoming lectures

Introducing CTL

Basic Algorithms for CTL

CTL and Fairness; computing strongly connected components

Basic Decision Diagrams

Tool demonstration: SMV
LTL (linear-time logic)

- Describes properties of individual executions.
- Semantics defined as a set of executions.

CTL (computation tree logic)

- Describes properties of a computation tree: formulas can reason about many executions at once. (CTL belongs to the family of branching-time logics.)
- Semantics defined in terms of states.
Computation tree

Let $\mathcal{T} = \langle S, \rightarrow, s^0 \rangle$ be a transition system.
Intuitively, the computation tree of $\mathcal{T}$ is the acyclic unfolding of $\mathcal{T}$.

Formally, we can define the unfolding as the least (possibly infinite) transition
system $\langle U, \rightarrow', u^0 \rangle$ with a labelling $l: U \rightarrow S$ such that

\[
\begin{align*}
u^0 &\in U \text{ and } l(u^0) = s^0; \\
\text{if } u \in U, \ l(u) = s, \text{ and } s \rightarrow s' \text{ for some } u, s, s', &\text{ then there is } u' \in U \text{ with } u \rightarrow' u' \text{ and } l(u') = s'; \\
\end{align*}
\]

$u^0$ does not have a direct predecessor, and all other states in $U$ have exactly
one direct predecessor.

Note: For model checking CTL, the construction of the computation tree will not
be necessary. However, this definition serves to clarify the concepts behind CTL.
Computation tree: Example

A transition system and its computation tree (labelling / given in blue):
CTL = Computation-Tree Logic

Combines temporal operators with quantification over runs

Operators have the following form:

- There exists an execution for all executions
- For all executions
- Next
- Finally
- Globally
- Until
- (and possibly others)
We define a minimal syntax first. Later we define additional operators with the help of the minimal syntax.

Let $AP$ be a set of atomic propositions: The set of CTL formulas over $AP$ is as follows:

- if $a \in AP$, then $a$ is a CTL formula;
- if $\phi_1, \phi_2$ are CTL formulas, then so are
  
  $\neg\phi_1$, $\phi_1 \lor \phi_2$, $EX \phi_1$, $EG \phi_1$, $\phi_1 EU \phi_2$
CTL: Semantics

Let $\mathcal{K} = (S, \rightarrow, s^0, AP, \nu)$ be a Kripke structure.

We define the semantic of every CTL formula $\phi$ over $AP$ w.r.t. $\mathcal{K}$ as a set of states $[\lbrack \phi \rbrack]_\mathcal{K}$, as follows:

\[
\begin{align*}
[\lbrack a \rbrack]_\mathcal{K} &= \nu(a) \quad \text{for } a \in AP \\
[\lbrack \neg \phi_1 \rbrack]_\mathcal{K} &= S \setminus [\lbrack \phi_1 \rbrack]_\mathcal{K} \\
[\lbrack \phi_1 \lor \phi_2 \rbrack]_\mathcal{K} &= [\lbrack \phi_1 \rbrack]_\mathcal{K} \cup [\lbrack \phi_2 \rbrack]_\mathcal{K} \\
[\lbrack \text{EX} \phi_1 \rbrack]_\mathcal{K} &= \{ s \mid \text{there is a } t \text{ s.t. } s \rightarrow t \text{ and } t \in [\lbrack \phi_1 \rbrack]_\mathcal{K} \} \\
[\lbrack \text{EG} \phi_1 \rbrack]_\mathcal{K} &= \{ s \mid \text{there is a run } \rho \text{ with } \rho(0) = s \text{ and } \rho(i) \in [\lbrack \phi_1 \rbrack]_\mathcal{K} \text{ for all } i \geq 0 \} \\
[\lbrack \phi_1 \text{ EU } \phi_2 \rbrack]_\mathcal{K} &= \{ s \mid \text{there is a run } \rho \text{ with } \rho(0) = s \text{ and } k \geq 0 \text{ s.t. } \rho(i) \in [\lbrack \phi_1 \rbrack]_\mathcal{K} \text{ for all } i < k \text{ and } \rho(k) \in [\lbrack \phi_2 \rbrack]_\mathcal{K} \}
\end{align*}
\]
We say that $\mathcal{K}$ satisfies $\phi$ (denoted $\mathcal{K} \models \phi$) iff $s^0 \in \llbracket \phi \rrbracket_{\mathcal{K}}$.

We declare two formulas equivalent (written $\phi_1 \equiv \phi_2$) iff for every Kripke structure $\mathcal{K}$ we have $\llbracket \phi_1 \rrbracket_{\mathcal{K}} = \llbracket \phi_2 \rrbracket_{\mathcal{K}}$.

In the following, we omit the index $\mathcal{K}$ from $\llbracket \cdot \rrbracket_{\mathcal{K}}$ if $\mathcal{K}$ is understood.
CTL: Extended syntax

\[ \phi_1 \land \phi_2 \equiv \neg(\neg\phi_1 \lor \neg\phi_2) \]
\[ \text{true} \equiv a \lor \neg a \]
\[ \text{false} \equiv \neg\text{true} \]
\[ \phi_1 \text{ EW } \phi_2 \equiv \text{ EG } \phi_1 \lor (\phi_1 \text{ EU } \phi_2) \]
\[ \text{EF } \phi \equiv \text{ true EU } \phi \]
\[ \phi_1 \text{ AW } \phi_2 \equiv \neg(\neg\phi_2 \text{ EU } \neg(\phi_1 \lor \phi_2)) \]
\[ \phi_1 \text{ AU } \phi_2 \equiv \neg\text{ EG } \neg\phi \]
\[ \neg\text{ EX } \neg\phi \]
\[ \neg\text{ EF } \neg\phi \]
\[ \neg\text{ EG } \neg\phi \]

Other logical and temporal operators (e.g. \( \to \)), ER, AR, \ldots may also be defined.
We use the following computation tree as a running example (with varying distributions of red and black states):

In the following slides, the topmost state satisfies the given formula if the black states satisfy $p$ and the red states satisfy $q$. 
AX $p$
p AU q
EX p
p EU q
Solving nested formulas: Is $s_0 \in \llbracket \text{AF AG } x \rrbracket$?

To compute the semantics of formulas with nested operators, we first compute the states satisfying the innermost formulas; then we use those results to solve progressively more complex formulas.

In this example, we compute $\llbracket x \rrbracket$, $\llbracket \text{AG } x \rrbracket$, and $\llbracket \text{AF AG } x \rrbracket$, in that order.
Bottom-up method (1): Compute $\llbracket x \rrbracket$
Bottom-up method (2): Compute $[[\text{AG } x]]$
Bottom-up method (3): Compute $\llbracket AF AG \ x \rrbracket$
Five philosophers are sitting around a table, taking turns at thinking and eating.

We shall express a couple of properties in CTL. Let us assume the following atomic propositions:

\[ e_i \equiv \text{philosopher } i \text{ is currently eating} \]

\[ f_i \equiv \text{philosopher } i \text{ has just finished eating} \]
“Philosophers 1 and 4 will never eat at the same time.”
“Philosophers 1 and 4 will never eat at the same time.”

$$AG \neg (e_1 \land e_4)$$

“Whenever philosopher 4 has finished eating, he cannot eat again until philosopher 3 has eaten.”
“Philosophers 1 and 4 will never eat at the same time.”

$$AG \neg (e_1 \wedge e_4)$$

“Whenever philosopher 4 has finished eating, he cannot eat again until philosopher 3 has eaten.”

$$AG (f_4 \rightarrow (\neg e_4 \text{ AW } e_3))$$

“Philosopher 2 will be the first to eat.”
“Philosophers 1 and 4 will never eat at the same time.”

\[ \text{AG} \neg (e_1 \land e_4) \]

“Whenever philosopher 4 has finished eating, he cannot eat again until philosopher 3 has eaten.”

\[ \text{AG}(f_4 \rightarrow (\neg e_4 \text{ AW } e_3)) \]

“Philosopher 2 will be the first to eat.”

\[ \neg (e_1 \lor e_3 \lor e_4 \lor e_5) \text{ AU } e_2 \]
CTL and LTL have a large overlap, i.e. properties expressible in both logics. Examples:

**Invariants** (e.g., “\(p\) never holds.”)

\(\text{AG} \neg p\) or \(G\neg p\)

**Reactivity** ("Whenever \(p\) happens, eventually \(q\) will happen.")

\(\text{AG}(p \rightarrow AF q)\) or \(G(p \rightarrow F q)\)
CTL considers the whole computation tree whereas LTL only considers individual runs. Thus CTL allows to reason about the *branching behaviour*, considering multiple possible runs at once. Examples:

The CTL property $\text{AG EF } p$ ("reset property") is not expressible in LTL.

The CTL property $\text{AF AX } p$ distinguishes the following two systems, but the LTL property $\text{F X } p$ does not:
Expressiveness of CTL and LTL (3/4)

Even though CTL considers the whole computation tree, its state-based semantics is subtly different from LTL. Thus, there are also properties expressible in LTL but not in CTL. Examples:

The LTL property $\mathbf{F} \mathbf{G} \rho$ is not expressible in CTL:

$$\mathcal{K} \models \mathbf{F} \mathbf{G} \rho \quad \text{but} \quad \mathcal{K} \not\models \mathbf{A} \mathbf{F} \mathbf{A} \mathbf{G} \rho$$
Also, **fairness conditions** are not directly expressible in CTL:

\[(G F p_1 \land G F p_2) \rightarrow \phi\]

However, as we shall see later, there is another way to extend CTL with fairness conditions.

**Conclusion:** The expressiveness of CTL and LTL is incomparable; there is an overlap, and each logic can express properties that the other cannot.

**Remark:** There is a logic called **CTL*** that combines the expressiveness of CTL and LTL. However, we will not deal with it in this course.