Notes on CTL

DPL Seminar, Spring 2002
Handout 12

Konstantine Arkoudas

November 27, 2003
CTL syntax

We assume that we are given a set of atomic propositions (or simply “atoms”). We will use the letter
A to denote a typical atom. The propositions of CTL have the following abstract syntax:

\[ P ::= A | \text{true} | \text{false} | \neg P | P_1 \land P_2 | P_1 \lor P_2 | P_1 \rightarrow P_2 | \mathbf{AX}(P) | \mathbf{EX}(P) | \mathbf{AG}(P) | \mathbf{EG}(P) \]

\[ \mathbf{AF}(P) | \mathbf{EF}(P) | \mathbf{AU}(P_1; P_2) | \mathbf{EU}(P_1; P_2) \]

The letter \( \mathbf{A} \) stands for “All”, \( \mathbf{E} \) for “Exists”, \( \mathbf{X} \) for “Next”, \( \mathbf{F} \) for “Future”, \( \mathbf{G} \) for “Global”, and \( \mathbf{U} \) for “Until”.

CTL semantics

A Kripke structure (or model) is a triple \( M = (S, \rightarrow, L) \) consisting of a set of states \( S \), a total binary
relation \( \rightarrow \) on \( S \), which we will call the successor relation, and a labelling \( L \), which is a total function
on \( S \) that maps every state \( s \in S \) to a set of atomic propositions. Intuitively, \( A \in L(s) \) iff \( A \) holds in
state \( s \). In addition, some states in \( S \) may be designated as initial states, though this is not reflected
in the formal definition of \( M \). By a path starting at a state \( s \) we will mean a non-empty sequence
of states \( s_1, s_2, \ldots \) such that \( s_i = s \) and \( s_i \rightarrow s_{i+1} \) for all \( i \). We will write \( \pi_s \) for an arbitrary path
starting at \( s \). Occasionally we will treat such a path as a set, writing \( s' \in \pi_s \) to mean that \( s' = s_i \) for
some \( i \).

Given a model \( M = (S, \rightarrow, L) \), a state \( s \in S \), and a CTL proposition \( P \), we write \( M \models_s P \) to
mean that \( M \) satisfies \( P \) in state \( s \), or simply that \( P \) holds in state \( s \) when \( M \) is understood. This
relation is defined by induction on the structure of \( P \) as follows:

\begin{itemize}
  \item \( M \models_s \text{true} \); \( M \not\models_s \text{false} \); while \( M \models_s A \) iff \( A \in L(s) \).
  \item \( M \models_s \neg P \) iff \( M \not\models_s P \).
  \item \( M \models_s P_1 \land P_2 \) iff \( M \models_s P_1 \) and \( M \models_s P_2 \).
  \item \( M \models_s P_1 \lor P_2 \) iff \( M \models_s P_1 \) or \( M \models_s P_2 \).
  \item \( M \models_s P_1 \rightarrow P_2 \) iff \( M \models_s P_2 \) whenever \( M \models_s P_1 \).
  \item \( M \models_s \mathbf{AX}(P) \) iff \( M \models_{s'} P \) for all \( s' \) such that \( s \rightarrow s' \).
  \item \( M \models_s \mathbf{EX}(P) \) iff \( M \models_{s'} P \) for some \( s' \) such that \( s \rightarrow s' \).
  \item \( M \models_s \mathbf{AG}(P) \) iff for every path \( \pi_s \) we have \( M \models_{s'} P \) for all \( s' \in \pi_s \).
  \item \( M \models_s \mathbf{EG}(P) \) iff there is a path \( \pi_s \) such that \( M \models_{s'} P \) for all \( s' \in \pi_s \).
  \item \( M \models_s \mathbf{AF}(P) \) iff for every path \( \pi_s \) we have \( M \models_{s'} P \) for some \( s' \in \pi_s \).
  \item \( M \models_s \mathbf{EF}(P) \) iff there is a path \( \pi_s \) such that \( M \models_{s'} P \) for some \( s' \in \pi_s \).
  \item \( M \models_s \mathbf{AU}(P_1; P_2) \) iff every path \( \pi_s \) satisfies \( P_1 \) until \( P_2 \), which is to say, there is some \( s' \in \pi_s \) such that \( M \models_{s'} P_2 \) and \( M \models_{s''} P_1 \) for every predecessor \( s'' \) of \( s \) in \( \pi_s \).
  \item \( M \models_s \mathbf{EU}(P_1; P_2) \) iff there is a path \( \pi_s \) that satisfies \( P_1 \) until \( P_2 \).
\end{itemize}
For a set of propositions $\Phi$, we say that $\Phi$ entails $Q$, written $\Phi \models Q$, to mean that for every model $\mathcal{M}$ and every state $s$ of $\mathcal{M}$, we have $\mathcal{M} \models s P$ whenever $\mathcal{M} \models s P$ for all $P \in \Phi$. When $\Phi$ is a singleton $\{P\}$ we simply write $P \models Q$ rather than $\{P\} \models Q$. We say that $P$ and $Q$ are logically equivalent (or just “equivalent”), written $P \equiv Q$, iff each entails the other, i.e., iff $P \models Q$ and $Q \models P$.

**Theorem 1.1** $\mathbf{AF}$, $\mathbf{EU}$, and $\mathbf{EX}$ form an adequate set of temporal connectives for CTL. More precisely, every CTL proposition $P$ is equivalent to a proposition $Q$ that has no occurrences of $\mathbf{AX}$, $\mathbf{AU}$, $\mathbf{EF}$, $\mathbf{AG}$, or $\mathbf{EG}$. Moreover, $Q$ can be obtained from $P$ mechanically.

**Proof:** By repeated application of the following equivalences:

1. $\mathbf{AX}(P) \equiv \neg \mathbf{EX}(-P)$
2. $\mathbf{AG}(P) \equiv \neg \mathbf{EF}(-P)$ \hspace{1em} (use 4 to eliminate $\mathbf{EF}$)
3. $\mathbf{EG}(P) \equiv \neg \mathbf{AF}(-P)$
4. $\mathbf{EF}(P) \equiv \mathbf{EU}(\text{true}, P)$
5. $\mathbf{AU}(P_1, P_2) \equiv \neg [\mathbf{EU}(-P_2, -P_1 \land -P_2) \lor \mathbf{EG}(-P_2)]$ \hspace{1em} (use 3 to eliminate $\mathbf{EG}$)

The first four of the above are readily derivable from the definition of satisfaction. We will demonstrate the fifth in the sequel. \hfill \blacksquare

**Theorem 1.2** (Fixed-point characterization of CTL) We have:

- $\mathbf{AF}(P) \equiv P \lor \mathbf{AX}(\mathbf{AF}(P))$
- $\mathbf{EU}(P_1, P_2) \equiv P_2 \lor [P_1 \land \mathbf{EX}(\mathbf{EU}(P_1, P_2))]$

Both of the above can be proved directly from the given semantics of CTL. Note that these equivalences can be viewed as recursive equations. Specifically, if we write $[P]$ for the set of all states that satisfy a proposition $P$ (for a fixed model $\mathcal{M}$), then the above equivalences entail

$$[\mathbf{AF}(P)] = [P] \cup \{s \mid s' \in [\mathbf{AF}(P)] \text{ for all } s' \text{ such that } s \rightarrow s' \}$$

and

$$[\mathbf{EU}(P_1, P_2)] = [P_2] \cup \{[P_1] \cap \{s \mid s' \in [\mathbf{EU}(P_1, P_2)] \text{ for some } s' \text{ such that } s \rightarrow s' \} \}$$

respectively, which means that $[\mathbf{AF}(P)]$ and $[\mathbf{EU}(P_1, P_2)]$ must be fixed points of the recursive equations

$$S = [P] \cup \{s \mid s' \in S \text{ for all } s' \text{ such that } s \rightarrow s' \}$$

and

$$S = [P_2] \cup \{[P_1] \cap \{s \mid s' \in S \text{ for some } s' \text{ such that } s \rightarrow s' \} \}$$

respectively. Indeed, the expressions on the right-hand sides are monotonic functions on bounded cpos, and hence the existence of fixed-point solutions is guaranteed by Tarski’s fixed-point theorem. The Athena implementation of the model checker will use the customary bottom-up iteration algorithm to compute $[\mathbf{AF}(P)]$ and $[\mathbf{EU}(P_1, P_2)]$ as the least fixed points of the above equations.
Implementation

The following Athena structure models the abstract syntax of CTL:

(structure (CTL-Prop T)
  TRUE
  FALSE
  (atom T)
  (neg (CTL-Prop T))
  (con (CTL-Prop T) (CTL-Prop T))
  (dis (CTL-Prop T) (CTL-Prop T))
  (imp (CTL-Prop T) (CTL-Prop T))
  (AX (CTL-Prop T))
  (AX (CTL-Prop T))
  (AF (CTL-Prop T))
  (EF (CTL-Prop T))
  (AG (CTL-Prop T))
  (EG (CTL-Prop T))
  (AU (CTL-Prop T) (CTL-Prop T))
  (EU (CTL-Prop T) (CTL-Prop T)))

A finite Kripke structure will be represented as a list of triples of the form

\[ [s, [s_1, \ldots, s_k], f] \]

where \( s \) is a state, \( s_1, \ldots, s_k \) are its successors, and \( f \) is a total predicate on atoms. The atoms that hold in \( s \) will be all and only those atoms \( A \) for which \( f(A) \) returns \texttt{true}. The following functions return a list of all the states of a given model, and a list of all the successors of a given state in a given model, respectively:

(define (states model)
  (letrec ((loop (function (model results))
             (match model
                   ([] results)
                   ((list-of [s _] rest) (loop rest (add s results)))))
    (loop model [])))

(define (succ s model)
  (match model
            (([split _ (list-of [(val-of s) succ-list _] _)] succ-list))))

The following function determines whether an atom holds in a given state of a given model:

(define (sat-atom? A s model)
  (match model
           (([split _ (list-of [(val-of s) _] _)] ) (f A)))))

The following are classic list functions that we will need:

(define (member? x L)
  (match L
            (([split _ (list-of (val-of x _) ) ] true)
             (_ false))))

(define (select L f)
  (letrec ((loop (function (L results)
                      (match L
                               ([] results)
                               ((list-of x more)
                                (check ((f x) (loop more (add x results)))))))))

3
(define (negate x)
  (match x
    (true true) (false false)))

(define (filter-out L f)
  (select L (function (x) (negate (f x))))))

(define (for-every L f)
  (match L
    ([] true)
    ((list-of x rest) (&& (f x) (for-every rest f))))))

(define (for-some L f)
  (negate (for-every L (function (x) (negate (f x)))))))

(define (subset? L1 L2)
  (for-every L1 (function (x) (member? x L2))))

(define (equal-state-sets L1 L2)
  (&& (subset? L1 L2)
       (subset? L2 L1)))

The last auxiliary function we will need is a least-fixed-point finder:

(define (fix init-states step)
  (letrec ((loop (function (states)
      (let ((new-states (step states)))
        (check ((equal-state-sets? new-states states) states)
          (else (loop new-states)))))
      (loop init-states)))

Finally, the code for the model checker is given in Figure ?? on the next page.
(define (sat P model)
  (let ((all-states (states model)))
    (match P
      [(TRUE all-states)
        (FALSE [])
        [(atom A) (select all-states (function (s) (sat-atom? A s model)))]
        [(neg Q) (let ((Q-states (sat Q model)))
                   (filter-out all-states (function (s) (member? s Q-states))))]
        [(conj P1 P2) (let ((P1-states (sat P1 model))
                           (P2-states (sat P2 model)))
                       (select all-states (function (s) (and (member? s P1-states)
                                                      (member? s P2-states)))))
        [(disj P1 P2) (let ((P1-states (sat P1 model))
                           (P2-states (sat P2 model)))
                       (select all-states (function (s) (or (member? s P1-states)
                                                      (member? s P2-states)))))
        [(imp P1 P2) (let ((P1-states (sat (neg P1) model)))
                       (select all-states (function (s) (for-some (s' (member? s' states))))))]
        [(EX Q) (let ((Q-states (sat Q model)))
                  (select all-states (function (s) (for-every (s' (member? s' states))))))]
        [(AF Q) (let ((init-states (sat Q model)))
                   (step (function (states)
                           (join states
                                (select all-states (function (s) (for-some (succ s) (member? s' states))))))]
                   (fix (fix init-states step))))]
        [(EF Q) (let ((init-states (sat Q model)))
                   (step (function (states)
                           (join states
                                (select all-states (function (s) (for-some (succ s) (member? s' states))))))]
                   (fix (fix init-states step))))]
        [(AG Q) (let ((init-states (sat (neg Q) model)))
                   (step (function (states)
                           (join states
                                (select all-states (function (s) (for-some (succ s) (member? s' states))))))]
                   (fix (fix init-states step))))]
        [(EU P1 P2) (let ((P1-states (sat P1 model)))
                       (step (function (states)
                               (join states
                                    (select all-states (function (s) (for-some (succ s) (member? s' states))))))]
                               (fix (fix init-states step))))]
        [(AU P1 P2) (let ((P1-states (sat (neg (disj (EU (neg P1) (conj (neg P1) (neg P2))) (EG (neg P2))) model)))
                           (step (function (states)
                                   (join states
                                        (select all-states (function (s) (for-some (succ s) (member? s' states))))))]
                                        (fix (fix init-states step))))])])])])]))])])])]

Figure 1.1: A model checker for CTL.