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:

$$\begin{array}{lcl} P & ::= & A \mid \mathbf{true} \mid \mathbf{false} \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \mid \mathbf{AX}(P) \mid \mathbf{EX}(P) \mid \mathbf{AG}(P) \mid \mathbf{EG}(P) \\ & & \mathbf{AF}(P) \mid \mathbf{EF}(P) \mid \mathbf{AU}(P_1, P_2) \mid \mathbf{EU}(P_1, P_2) \end{array}$$

The letter **A** stands for "**A**ll", **E** for "**E**xists", **X** for "Ne**X**t", **F** for "**F**uture", **G** for "**G**lobal", and **U** for "**U**ntil".

CTL semantics

A *Kripke structure* (or *model*) is a triple $\mathcal{M} = \langle S, \rightarrow, L \rangle$ 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 fuction 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 $\mathcal{M}$. By a *path starting at a state* s we will mean a non-empty sequence of states $s_1, s_2, \ldots$ such that $s_1 = s$ and $s_i \rightarrow s_{i+1}$ for all i. We will write π_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 $\mathcal{M} = \langle S, \rightarrow, L \rangle$, a state $s \in S$, and a CTL proposition P, we write $\mathcal{M} \models_s P$ to mean that $\mathcal{M}$ *satisfies* P *in state* s, or simply that P *holds in state* s when $\mathcal{M}$ is understood. This relation is defined by induction on the structure of P as follows:

- $\mathcal{M} \models_s \mathbf{true}$; $\mathcal{M} \not\models_s \mathbf{false}$; while $\mathcal{M} \models_s A$ iff $A \in L(s)$.
- $\mathcal{M} \models_s \neg P$ iff $\mathcal{M} \not\models_s P$.
- $\mathcal{M} \models_s P_1 \wedge P_2$ iff $\mathcal{M} \models_s P_1$ and $\mathcal{M} \models_s P_2$.
- $\mathcal{M} \models_s P_1 \vee P_2$ iff $\mathcal{M} \models_s P_1$ or $\mathcal{M} \models_s P_2$.
- $\mathcal{M} \models_s P_1 \Rightarrow P_2$ iff $\mathcal{M} \models_s P_2$ whenever $\mathcal{M} \models_s P_1$.
- $\mathcal{M} \models_s \mathbf{AX}(P)$ iff $\mathcal{M} \models_{s'} P$ for all s' such that $s \rightarrow s'$.
- $\mathcal{M} \models_s \mathbf{EX}(P)$ iff $\mathcal{M} \models_{s'} P$ for some s' such that $s \rightarrow s'$.
- $\mathcal{M} \models_s \mathbf{AG}(P)$ iff for every path π_s we have $\mathcal{M} \models_{s'} P$ for all $s' \in \pi_s$.
- $\mathcal{M} \models_s \mathbf{EG}(P)$ iff there is a path π_s such that $\mathcal{M} \models_{s'} P$ for all $s' \in \pi_s$.
- $\mathcal{M} \models_s \mathbf{AF}(P)$ iff for every path π_s we have $\mathcal{M} \models_{s'} P$ for some $s' \in \pi_s$.
- $\mathcal{M} \models_s \mathbf{EF}(P)$ iff there is a path π_s such that $\mathcal{M} \models_{s'} P$ for some $s' \in \pi_s$.
- $\mathcal{M} \models_s \mathbf{AU}(P_1, P_2)$ iff every path π_s *satisfies* P_1 *until* P_2, which is to say, there is some $s' \in \pi_s$ such that $\mathcal{M} \models_{s'} P_2$ and $\mathcal{M} \models_{s''} P_1$ for every predecessor s'' of s in π_s.
- $\mathcal{M} \models_s \mathbf{EU}(P_1, P_2)$ iff there is a path π_s that satisfies P_1 until P_2.

For a set of propositions Φ, we say that Φ *entails* Q, written $\Phi \models Q$, to mean that for every model $\mathcal{M}$ and every state s of $\mathcal{M}$, we have $\mathcal{M} \models_s Q$ whenever $\mathcal{M} \models_s P$ for all $P \in \Phi$. When Φ 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 **AF**, **EU**, *and* **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* **AX**, **AU**, **EF**, **AG**, *or* **EG**. *Moreover, Q can be obtained from P mechanically.*

Proof: By repeated application of the following equivalences:

1. $\mathbf{AX}(P) \equiv \neg\mathbf{EX}(\neg P)$
2. $\mathbf{AG}(P) \equiv \neg\mathbf{EF}(\neg P)$ (use 4 to eliminate **EF**)
3. $\mathbf{EG}(P) \equiv \neg\mathbf{AF}(\neg P)$
4. $\mathbf{EF}(P) \equiv \mathbf{EU}(\mathbf{true}, P)$
5. $\mathbf{AU}(P_1, P_2) \equiv \neg[\mathbf{EU}(\neg P_2, \neg P_1 \wedge \neg P_2) \vee \mathbf{EG}(\neg P_2)]$ (use 3 to eliminate **EG**)

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

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

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

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 fuctions 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))
  (conj (CTL-Prop T) (CTL-Prop T))
  (disj (CTL-Prop T) (CTL-Prop T))
  (imp (CTL-Prop T) (CTL-Prop T))
  (AX (CTL-Prop T))
  (EX (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 `true`. The following fuctions returns 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 fuction 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] _)) (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)))
```

```
                                (else (loop more results)))))))
    (loop L []))

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

(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) (&& (member? s P1-states)
                                                           (member? s P2-states))))))
      ((disj P1 P2) (join (sat P1 model) (sat P2 model)))
      ((imp P1 P2) (sat (disj (neg P1) P2) model))
      ((EX Q) (let ((Q-states (sat Q model)))
                (select all-states
                  (function (s)
                    (for-some (succ s) (function (s') (member? s' Q-states)))))))
      ((AX Q) (sat (neg (EX (neg Q))) model))
      ((AF Q) (let ((init-states (sat Q model))
                    (step (function (states)
                            (join states
                                  (select all-states
                                    (function (s)
                                      (for-every (succ s)
                                        (function (s') (member? s' states)))))))))
                (fix init-states step)))
      ((EF Q) (sat (EU TRUE Q) model))
      ((AG Q) (sat (neg (EF (neg Q))) model))
      ((EG Q) (sat (neg (AF (neg Q))) model))
      ((EU P1 P2) (let ((P1-states (sat P1 model))
                        (P2-states (sat P2 model))
                        (step (function (states)
                                (join states
                                  (select all-states
                                    (function (s)
                                      (&& (member? s P1-states)
                                          (for-some (succ s)
                                            (function (s') (member? s' states)))))))))
                      (fix P2-states step)))
      ((AU P1 P2) (sat (neg (disj (EU (neg P2) (conj (neg P1) (neg P2)))
                                  (EG (neg P2)))) model)))))
```

Figure 1.1: A model checker for CTL.