Reasoning in Time

A Prolog Implementation of Computation Tree Logic

hama

Real Time Systems
COSC 4331

Prolog; A Brief Overview

- Prolog is a logical, imperative programming language
- Ideas are built up from relations
- Relations are either facts or predicates
- Questions are answered by querying possible solutions.

Facts:

```
animal(pig, mammal).
animal(lizard, reptile).
animal(tuna, fish).
```

Predicates:

```
cold_blooded(Name) :-
   animal(Name, reptile);
   animal(Name, fish).
```

Problem Solving

- We can answer complex questions by thinking in terms of relations

```
cold_blooded_animals_only([]).
cold_blooded_animals_only([Animal | Rest]) :-
   cold_blooded(Animal),
   cold_blooded_animals_only(Rest).
```

% Querying the system at runtime:

```
?- cold_blooded_animals_only([pig, lizard]).
   false.
?- cold_blooded_animals_only([tuna, lizard]).
   true.
```

Computation Tree Logic

- Popular in industrial contexts for model checking[1]

- Evaluated over *Kripke Structures*, state machines defined by:
  - A set of possible states
  - A set of initial states
  - A set of transitions relations between all states
  - A labelling function attaching propositions to certain states

- Transition relation is left-total, all states must lead to some other state
  - By definition, all paths are infinite

Implementation

- We implement Kripke structures as models
- We verify a model through recursive semantic entailment[2]
  - Models are correct for a formula φ if its entailment reduces to all valid propositions

The syntax of CTL is defined for a given set of labelling predicates $\mathbb{P}$:

$$
\begin{aligned}
\varphi &::= P \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \text{ where } P \in \mathbb{P} \\
&::= EX\varphi \mid AX\varphi \mid EF\varphi \mid AF\varphi \mid EG\varphi \mid AG\varphi \\
&::= \varphi_1 AU \varphi_2 \mid \varphi_1 EU \varphi_2 \qquad (3)
\end{aligned}
$$

Definition 2: **(Semantics of FOL)** Let $B = \langle F, R\rangle$ be a base for FOL and $\mathcal{I} = \langle \mathcal{D}, .^{\mathcal{I}}\rangle$ an interpretation for B. The satisfiability relation over formulae and interpretations, $\Vdash$, is defined by using structural induction on Φ and t. In the following, $\mathcal{I}^{x:=d}$ is an interpretation over B that is same as $\mathcal{I}$ except that it maps the variable x to d.

$$
\begin{array}{lcl}
\mathcal{I} \Vdash r(t_1,\ldots,t_n) & \Longleftrightarrow & r^{\mathcal{I}}(t_1^{\mathcal{I}},\ldots,t_n^{\mathcal{I}}) \text{ holds,} \\
\mathcal{I} \Vdash t_1 = t_2 & \Longleftrightarrow & t_1^{\mathcal{I}} \text{ is equal to } t_2^{\mathcal{I}}, \\
\mathcal{I} \Vdash \neg\Phi & \Longleftrightarrow & \mathcal{I} \Vdash \Phi \text{ does } \textit{not} \text{ hold,} \\
\mathcal{I} \Vdash \Phi_1 \vee \Phi_2 & \Longleftrightarrow & \mathcal{I} \Vdash \Phi_1 \text{ or } \mathcal{I} \Vdash \Phi_2, \\
\mathcal{I} \Vdash \exists x : \Phi & \Longleftrightarrow & \text{there exists a } d \in \mathcal{D} \\
 & & \text{such that } \mathcal{I}^{x:=d} \Vdash \Phi. \\
(f(t_1,\ldots,t_n))^{\mathcal{I}} & = & f^{\mathcal{I}}(t_1^{\mathcal{I}},\ldots,t_n^{\mathcal{I}})
\end{array}
$$

Definition 5: **(Semantics of CTL)** Let $\mathcal{K} = \langle S_{\mathcal{K}}, I_{\mathcal{K}}, N_{\mathcal{K}}, \mathbb{P}_{\mathcal{K}}\rangle$ be a Kripke structure and φ a CTL formula. The satisfiability relation for CTL, $\models_c$, is defined by structural induction on φ:

$$
\begin{array}{lcl}
\mathcal{K}, s \models_c P & \Longleftrightarrow & P(s) \text{ holds, where } P \in \mathbb{P}_{\mathcal{K}} \\
\mathcal{K}, s \models_c \neg\varphi & \Longleftrightarrow & \mathcal{K}, s \not\models_c \varphi \\
\mathcal{K}, s \models_c \varphi_1 \vee \varphi_2 & \Longleftrightarrow & \mathcal{K}, s \models_c \varphi_1 \;\vee\; \mathcal{K}, s \models_c \varphi_2 \\
\mathcal{K}, s \models_c EX\varphi & \Longleftrightarrow & \exists s' \in S : N_{\mathcal{K}}(s, s') \wedge \mathcal{K}, s' \models_c \varphi \\
\mathcal{K}, s \models_c EG\varphi & \Longleftrightarrow & \text{there exists a path } s_0 \mapsto s_1 \mapsto . \\
 & & \text{such that } s_0 = s \text{ and} \\
 & & \text{for all } i\text{'s } \mathcal{K}, s_i \models_c \varphi. \\
\mathcal{K}, s \models_c \varphi_1 EU \varphi_2 & \Longleftrightarrow & \text{there exists a } j \text{ and a path,} \\
 & & s_0 \mapsto s_1 \mapsto \ldots, \text{ such that} \\
 & & s = s_0,\ \mathcal{K}, s_j \models_c \varphi_2, \text{ and} \\
 & & \text{for all } i < j\ \mathcal{K}, s_i \models_c \varphi_1.
\end{array}
$$

Requirements

- A valid CTL system model in Prolog is a Kripke structure with these user-defined relations:
  - “State”: any distinct value is suitable as a state provided that it is unique to the model.
  - `transition(S1, S2)` : a directed relation S1 -> S2.
  - `label(State, Proposition)` : a relation between a state and a proposition.
    - If the label is a predicate, it may be used to label the state based on its properties.
- All nodes in a Kripke structure must have at least one transition. Terminal nodes are constructed with single transitions back to themselves.

Semantic Entailment

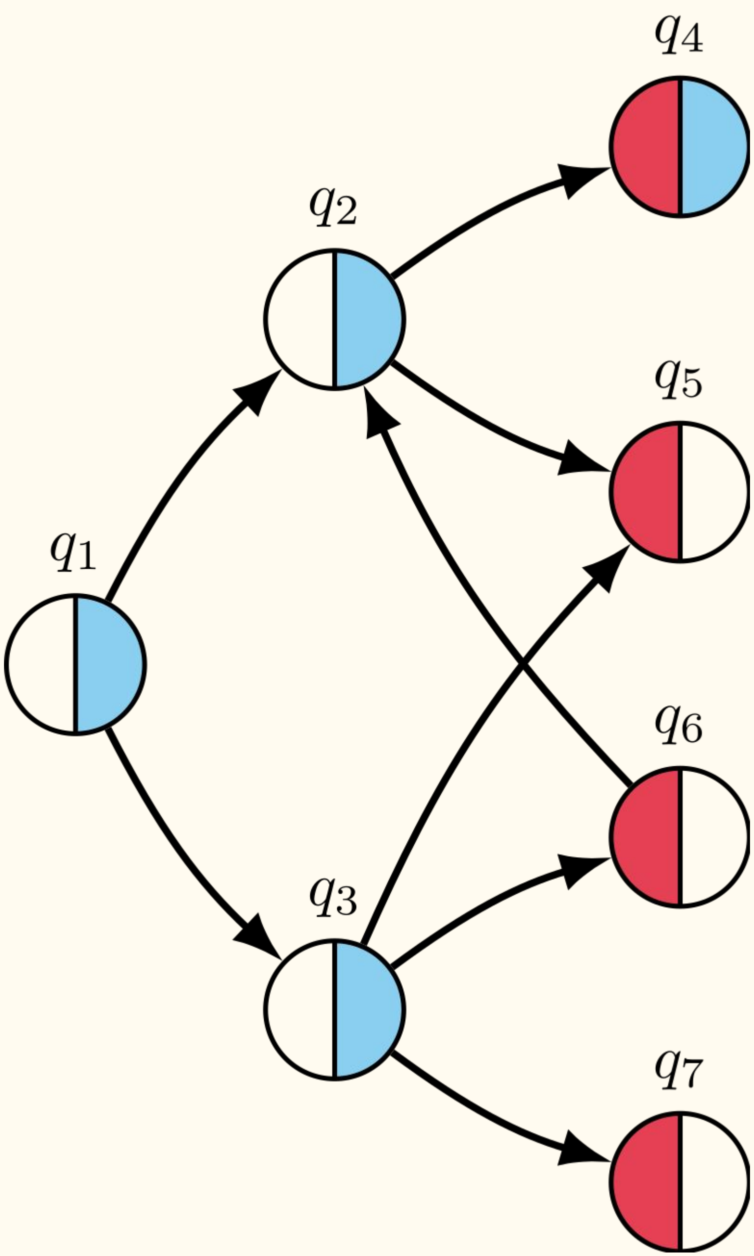
- Rather than evaluating formulas immediately, semantic entailment is defined using predicates accepting a state and a well-formed CTL formula
- Verification proceeds by recursively “unwrapping” formulas

```
%% ctl rules
% proposition
entails(S, P) :- label(S, P).

% tautology
entails(_, ctl_true) :- true.
entails(_, ctl_false) :- false.

% negation
entails(S, ctl_not(P)) :- not(entails(S, P)).

% connectives
entails(S, or(P1, P2)) :- entails(S, P1); entails(S, P2).
entails(S, and(P1, P2)) :- entails(S, P1), entails(S, P2).
entails(S, implies(P1, P2)) :- entails(S, or(P2, ctl_not(P1))).
entails(S, iff(P1, P2)) :- entails(S, or(
    and(P1, P2),
    and(not(P1), not(P2))
)).
```

```
% quantifiers
entails(S, ex(P)) :- transition(S, S2), entails(S2, P).
entails(S, ax(P)) :- not(entails(S, ex(ctl_not(P)))).

% exists
entails(S, ef(P)) :-
    entails(S, P);
    follows(S, S2), entails(S2, P).
entails(S, eg(P)) :- entails(S, eg(P), []).
entails_(S, eg(P), C) :-
    entails(S, P), accumulate(S, S2, C, C2), entails(S2, eg(P), C2).
entails(S, eu(P1, P2)) :-
    entails(S, or(P2, and(P1, ex(eu(P1, P2))))).

% all
entails(S, ag(P)) :- entails(S, ctl_not(ef(ctl_not(P)))).
entails(S, af(P)) :- entails(S, ctl_not(eg(ctl_not(P)))).
entails(S, au(P1, P2)) :- entails(S, or(P2, and(P1, ax(au(P1, P2))))).
```

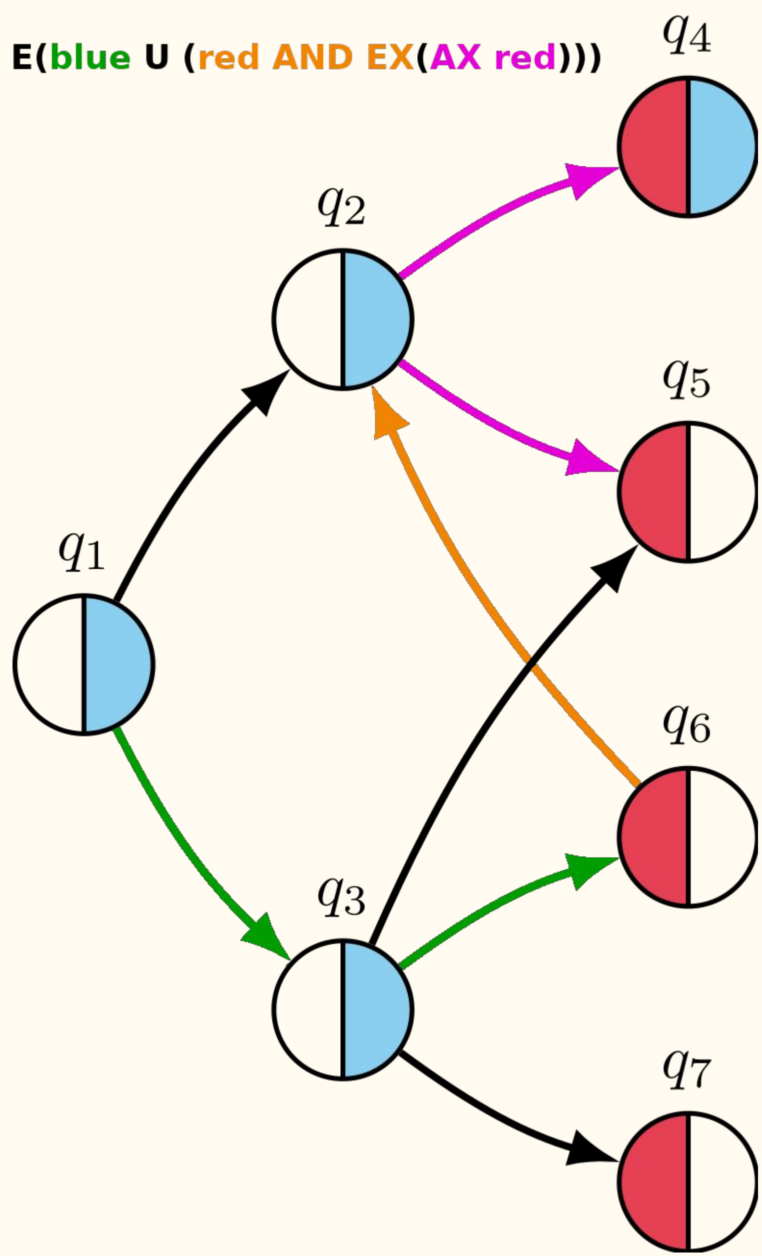
Example: Complex Queries

```
state(complex, q1, 0, 1).
state(complex, q2, 0, 1).
state(complex, q3, 0, 1).
state(complex, q4, 1, 1).
state(complex, q5, 1, 0).
state(complex, q6, 1, 0).
state(complex, q7, 1, 0).

transition(complex, q1, q2).
transition(complex, q1, q3).
transition(complex, q2, q4).
transition(complex, q2, q5).
transition(complex, q3, q5).
transition(complex, q3, q6).
transition(complex, q3, q7).
transition(complex, q4, q4).
transition(complex, q5, q5).
transition(complex, q6, q6).
transition(complex, q7, q7).
transition(complex, q6, s1).

label(State, red)  :- state(complex, State, 1, _).
label(State, blue)  :- state(complex, State, _, 1).
```



Example: Complex Queries

```
?- entails(complex_state(q1), af(p(red))).
true.

?- entails(complex_state(q1), ag(p(red))).
false.

?- entails(complex_state(q1), eu(p(blue), and(p(red), ex(ax(p(red)))))).
true .
```



Benefits of Prolog/Logic Programming CTL

- Simple to evaluate, understand and verify
    - Implementation requires < 50 LOC

- Easy integration with Prolog-based expert systems and solvers

- Embeddable in any C-compatible system without dangers of raw C code

Citations

(1) Vardi, Moshe Y. (2001). "Branching vs. Linear Time: Final Showdown" (PDF). Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science. 2031. Springer, Berlin: 1–22. doi:10.1007/3-540-45319-9_1

(2) Vakili, A., & Day, N. A. (2014). Reducing CTL-live model checking to semantic entailment in first-order logic (version 1). Cheriton School of Comp. Sci., University of Waterloo, Tech. Rep. CS-2014-05.