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
- Problems are solved 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 solve complex problems using techniques like recursion

```
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, so all states must lead to some other state
  - By definition, all paths are infinite

Implementation

- We implement a subset of CTL evaluating modified Kripke structures as models
  - We consider only finite, non-looping execution trees (DAGs)
  - A finite branching set of possible execution paths may be transformed into a DAG for this purpose
- We verify a model through recursive semantic entailment[2]
  - Models are correct for a formula if formula's (φ) entailment reduces to all valid propositions (Vakili, A., & Day, N. A.)

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

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

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.

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

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:

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

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.
- Again, transition graphs MUST be acyclic for this structure to be evaluated.

Semantic Entailment

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

```prolog
%% Semantic entailment is defined recursively on \phi
% entails(State, Formula)

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

% 2 propositions
entails(State, proposition(P)) :-
        label(State, P).
entails(State, p(P)) :- entails(State, proposition(P)).

% 3 not
entails(State, ctl_not(F)) :-
        not(entails(State, F)).

% 4 and
entails(State, ctl_and(F1, F2)) :-
        entails(State, F1), entails(State, F2).

% 5 or
entails(State, ctl_or(F1, F2)) :-
        entails(State, F1); entails(State, F2).

% 6 implies
entails(State, ctl_implies(F1, F2)) :-
        entails(State, ctl_not(F1));
        entails(State, F2).

% 7 iff
entails(State, ctl_iff(F1, F2)) :-
        entails(State, ctl_and(F1, F2));
        entails(State, ctl_and(ctl_not(F1), ctl_not(F2))).
```

```prolog
% 8 AX
entails(State, ctl_AX(F)) :-
        not(end_state(State)), not(entails(State, ctl_EX(ctl_not(F)))).

% 9 EX
entails(State, ctl_EX(F)) :-
        transition(State, S1), entails(S1, F).

% 10 AG
entails(State, ctl_AG(F)) :-
        entails(State, F), end_state(State);
        entails(State, ctl_and(F, ctl_AX(ctl_AG(F)))).

% 11 EG
entails(State, ctl_EG(F)) :-
        entails(State, F), end_state(State);
        entails(State, ctl_and(F, ctl_EX(ctl_EG(F)))).

% 12 AF
entails(State, ctl_AF(F)) :-
        entails(State, ctl_or(F, ctl_AX(ctl_AF(F)))).

% 13 EF
entails(State, ctl_EF(F)) :-
        entails(State, F);
        follows(State, S1), entails(S1, F).

% 14 A[ U ]
entails(State, ctl_AU(F1, F2)) :-
        entails(State, ctl_or(F2, ctl_and(F1, ctl_AX(ctl_AU(F1, F2))))).

% 15 E[ U ]
entails(State, ctl_EU(F1, F2)) :-
        entails(State, ctl_or(F2, ctl_and(F1, ctl_EX(ctl_EU(F1, F2))))).
```

Verification

- We can examine system correctness using an (inexhaustive) suite of unit tests
- Various trees are created and tested against formulae to confirm expected behavior.

```
%% all-empty tree
v_tree(empty, [0, 0, 0, 0, 0, 0, 0], [se1, se2, se3, se4, se5, se6, se7]).

%% all-full tree
v_tree(full, [1, 1, 1, 1, 1, 1, 1], [sf1, sf2, sf3, sf4, sf5, sf6, sf7]).

%% tree with single full state (leaf node 5)
v_tree(one_full, [0, 0, 0, 0, 1, 0, 0], [so1, so2, so3, so4, so5, so6, so7]).

%% tree with one branch full
v_tree(branch_full, [1, 0, 1, 0, 0, 1, 0], [sb1, sb2, sb3, sb4, sb5, sb6, sb7]).

%% tree with single empty state (leaf node 5)
v_tree(one_empty, [1, 1, 1, 1, 0, 1, 1], [soe1, soe2, soe3, soe4, soe5, soe6, soe7]).

%% tree with one branch empty
v_tree(branch_empty, [0, 1, 0, 1, 1, 0, 1], [sbe1, sbe2, sbe3, sbe4, sbe5, sbe6, sbe7]).


%%% ---------------- Verification Model Testing ----------------
test_formulae([
        ctl_AX(proposition(full)),
        ctl_AX(proposition(empty)),
        ctl_EX(proposition(full)),
        ctl_EX(proposition(empty)),
        ctl_AG(proposition(full)),
        ctl_AG(proposition(empty)),
        ctl_EG(proposition(full)),
        ctl_EG(proposition(empty)),
        ctl_AF(proposition(full)),
        ctl_AF(proposition(empty)),
        ctl_EF(proposition(full)),
        ctl_EF(proposition(empty))
]).
```

```
% main entry point for testing a given tree.
% ensure that test_vals are set, and provide the desired tree id to test.
% runs all test propositions centered at the root.
test(Id, R) :- test_vals(Id, Vs), test_tree(Id, Vs, R).
test(Id) :- test_vals(Id, Vs), test_tree(Id, Vs).

test_vals(empty, [false, true, false, true, false, true, false, true, false, true, false, true]).
test_vals(full, [true, false, true, false, true, false, true, false, true, false, true, false]).
test_vals(one_empty, [true, false, true, false, false, false, true, false, true, false, true, true]).
test_vals(one_full, [false, true, false, true, false, false, false, true, false, true, true, true]).
test_vals(branch_empty, [false, false, true, true, false, false, false, true, false, true, true, true]).
test_vals(branch_full, [false, false, true, true, false, false, true, false, true, false, true, true]).

%% basic operator testing

test_tautology :-
        entails(full, 1, ctl_true),
        not(entails(full, 1, ctl_false)).

test_proposition :-
        entails(full, 1, proposition(full)).

test_not :-
        entails(full, 1, ctl_not(ctl_false)),
        not(entails(full, 1, ctl_not(ctl_true))).

test_and :-
        entails(full, 1, ctl_and(ctl_true, ctl_true)),
        not(entails(full, 1, ctl_and(ctl_true, ctl_false))),
        not(entails(full, 1, ctl_and(ctl_false, ctl_true))),
        not(entails(full, 1, ctl_and(ctl_false, ctl_false))).

test_or :-
        entails(full, 1, ctl_or(ctl_true, ctl_true)),
        entails(full, 1, ctl_or(ctl_true, ctl_false)),
        entails(full, 1, ctl_or(ctl_false, ctl_true)),
        not(entails(full, 1, ctl_or(ctl_false, ctl_false))).

test_implies :-
        entails(full, 1, ctl_implies(ctl_true, ctl_true)),
        not(entails(full, 1, ctl_implies(ctl_true, ctl_false))),
        entails(full, 1, ctl_implies(ctl_false, ctl_true)),
        entails(full, 1, ctl_implies(ctl_false, ctl_false)).

test_iff :-
        entails(full, 1, ctl_iff(ctl_true, ctl_true)),
        not(entails(full, 1, ctl_iff(ctl_true, ctl_false))),
        not(entails(full, 1, ctl_iff(ctl_false, ctl_true))),
        entails(full, 1, ctl_iff(ctl_false, ctl_false)).
```

Example: Complex Queries

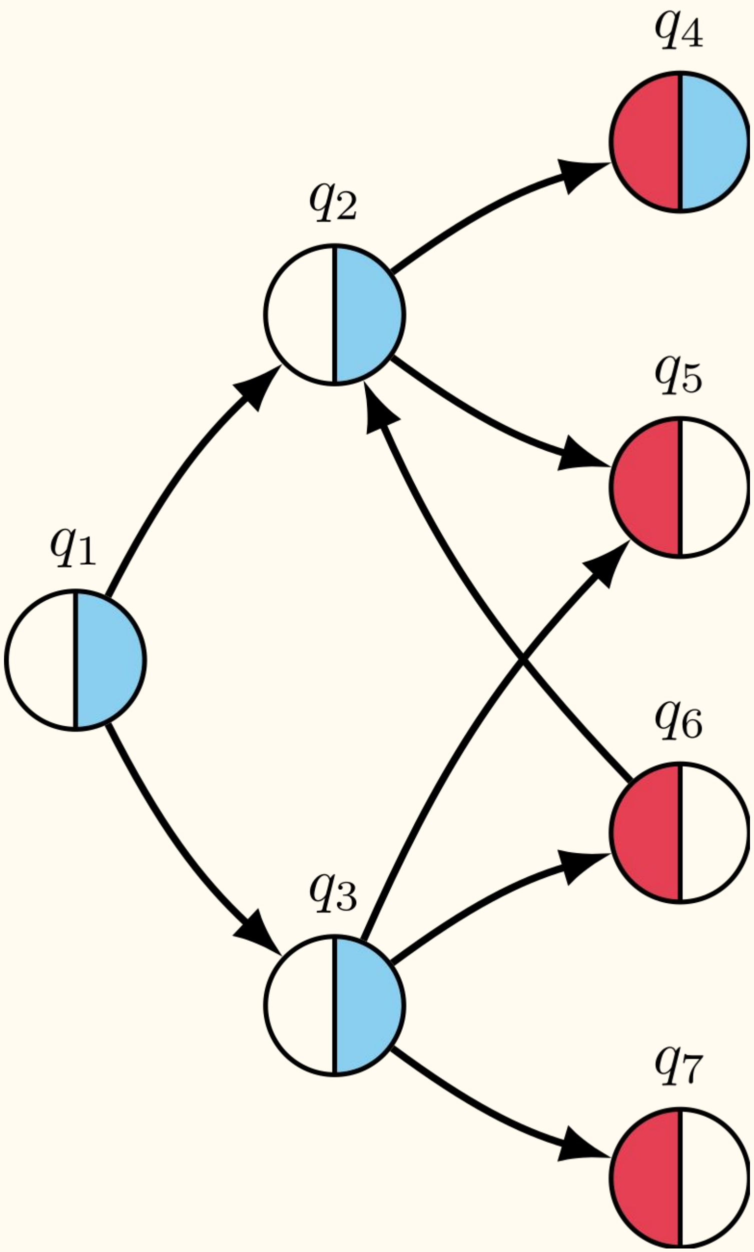
```
:- use_module(library(lists)).
:- ensure_loaded(ctl).

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

transition(S1, S2) :-
        ls_transitions(S1, L),
        member(S2, L).

ls_transitions(q1, [q2, q3]).
ls_transitions(q2, [q4, q5]).
ls_transitions(q3, [q5, q6, q7]).
transition(q6, q2).

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

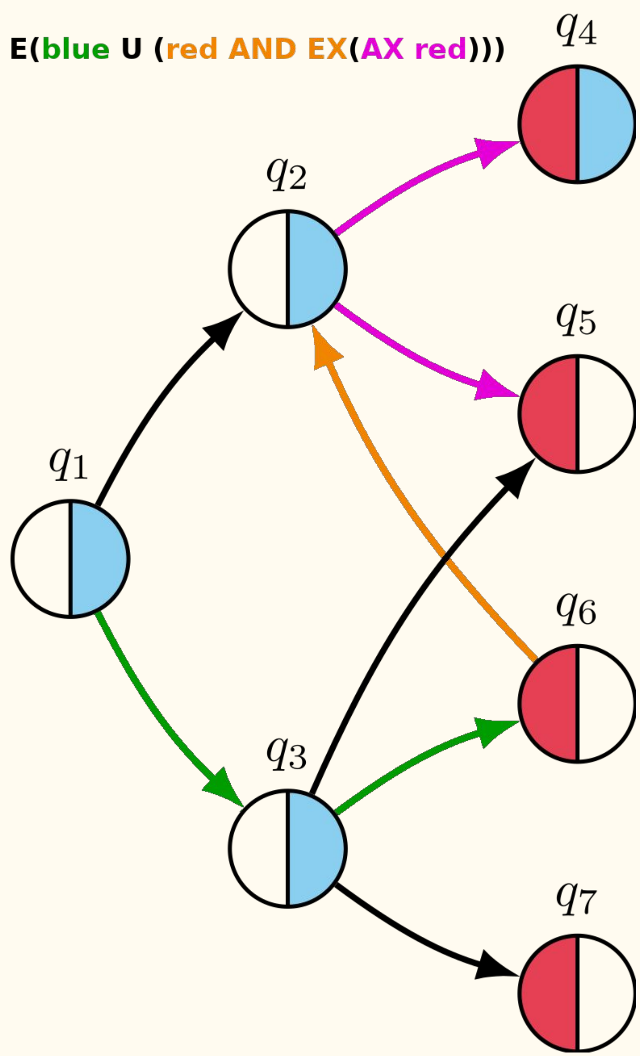


Example: Complex Queries

```
1 ?- entails(q1, ctl_AF(p(red))).
true.

2 ?- entails(q1, ctl_AG(p(red))).
false.

3 ?- entails(q1, ctl_EU(p(blue), ctl_and(p(red), ctl_EX(ctl_AX(p(red)))))).
true .
```



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.