Jacob Heden Malm
KTH CINTE
199804051499
Prolog Labb 3
Modellprovning för CTL

| Predikat | Vad gör det? | När är predikatet sant? |
|---|---|---|
| UpdateList | Accepterar ett element och en lista, och lägger in elementet i början på listan. | När första elementet är en lista och andra elementet är ett element. |
| check | Accepterar en CTL formel, och kollar om den är sann, delvis. | CTL formeln bryts delvis ner och matchas till olika instanser av check predikatet, när en del är korrekt utformad blir det motsvarande predikatet korrekt. |
| getSublist | Accepterar ett tillstånd från modellen, och två listor. Beroende på vilken lista som staten skickas med, grannlistan eller listan som innehåller atomerna som är sanna i de olika tillstånden, returneras den mostvarande dellistan till tillståndet. | |
| iterate | Ett predikat som används för check(_, _, _, _, ax(F)). Itererar genom grannlistan och kollar om F håller för alla grannar till det nuvarande tillståndet. | |

**Hur är programmet strukturerat?**

Programmet bryter rekursivt ner formeln som vi vill kolla i den givna modellen. Det vill säga, om vi får en formel som är strukturerad som af(ex(ag(not(q)))) och vi vill kolla att den gäller i ett tillstånd, skickas först formeln till check_af predikatet, här kollar vi om ex(_) gäller, så detta skickas vidare till ex predikatet, här kollar vi om ag(_) gäller, vilket skickas vidare till ag predikatet, osv. På så sätt bygger alla formler vi får på antingen check(not(F)) eller check(F), där F är en atom. Tillståndet/n vi kollar om F gäller i beror på strukturen av den hela givna CTL formeln.

Hur är de olika check() predikaten strukturerade?

And och Or predikaten bygger direkt på check(F) predikatet. Skillnaden är att det accepterar två CTL formler X och G. Antingen and(F, G) eller or(F, G). Det vi gör är att kollar om både F och G gäller, eller att om endast ett av dem håller.

check(ax) predikatet är på liknande sätt som check(F) predikatet en byggsten för andra check predikat. Detta är på grund av att det tillåter oss att kolla om någon formel håller i alla nästa tillstånd, vilket alla A_ predikat kan dra nytta av. Det vi gör här är accepterar en formel och ett tillstånd, hämtar adjacency list för tillståndet, sedan itererar vi igenom denna listan med iterate predikatet, och för varje nästa tillstånd kollar vi om formeln håller med check(F) predikatet.

check(ef) fungerar på ett liknande sätt. Vi hämtar adjacency listan som det första vi gör för det givna tillståndet, fast istället för att iterera igenom varje nästa tillstånd använder vi det inbyggda member predikatet och prologs backtracking funktion. Vi använder adjacency listan och ett hypotetiskt nästa tillstånd i member. Sedan kollar vi om formeln gäller för detta hypotetiska tillstånd. Om detta inte uppfylls kommer prologs backtracking att matcha det hypotetiska tillståndet med något annat tillstånd i adjacency listan.

check(ag) använder som sagt ax predikatet och listan över tillstånd vi redan varit i. Det vi gör är att vi kollar om formeln gäller i det nuvarande tillståndet, sedan kollar vi om ag(F) gäller i alla tillstånd som det går att nå från det nuvarande genom ax() predikatet. Detta rekursiva strängar för varje nästa tillstånd. En sådan rekursiv sträng terminerar när vi ser att vi redan varit i tillståndet vi granskar. Detta är basfallet. På så sätt är ag() predikatet en prolog variant av en Depth First Search.

eg() predikatet fungerar på samma sätt som ag predikatet, fast istället för att kolla varje nästa tillstånd genom ax() predikatet använder vi samma strategi som i ex predikatet.

Basfallet för af predikatet kollar först om vi inte har varit i det nuvarande tillståndet, sedan kollar det om formeln F gäller i detta tillstånd. För att generera nya rekursiva strängar kollar vi om af() gäller i alla nästa tillstånd genom ax predikatet.

Ef fungerar på samma sätt, fast istället för att använda ax använder vi samma metod som i ex predikatet.

**MIN MODELL**

Jag valde att modellera en dator. Jag har skapat 6 tillstånd, s1-6.

S1: startup av datorn
S2: login skärmen
S3: inloggad och använder datorn på ett normalt sätt
S4: när datorn förbereder sig själv för avstängning
S5: force shutdown, terminerar alla processer
S6: när datorn är avstängd
Atomer: p är datorns anslutning till internet, q är datorns kapabilitet att spela upp musik, z är datoranvändarens rätt att manuellt påbörja nya processer.

Den valida formeln jag testade var s1 EF EG p.
Den invalida formeln jag testade var s5 EX z.

APPENDIX

Sann modell:
[[s1, [s2, s5]],
 [s2, [s5, s3]],
 [s3, [s3, s5, s4]],
 [s4, [s6]], [s5, [s6]], [s6, [s1]]].

[[s1, []],
 [s2, [p, z]],
 [s3, [p, z, q]],
 [s4, [q]], [s5, []], [s6, []]].

s1.

ef(eg(p)).

Falsk modell:
[[s1, [s2, s5]],
 [s2, [s5, s3]],
 [s3, [s3, s5, s4]],

[s4, [s6]], [s5, [s6]], [s6, [s1]]].

[[s1, []],
[s2, [p, z]],
[s3, [p, z, q]],
[s4, [q]], [s5, []], [s6, []]].

s5.

ex(z).

Programkod:

```
% Load model, initial state and formula from file.
verify(Input) :-
see(Input), read(T), read(L), read(S), read(F), seen,
check(T, L, S, [], F).



/**
A predicate which accepts a state and either the list of adjacency lists, and returns the specific adjacency list in the last predicate
or accepts the mappings of values and returns a list of values that hold for the given state.
*/

getSublist(Head, [[Head, Tail]| _], Tail):- !.
getSublist(State, [H|T], List):-
getSublist(State, T, List).


% And
% Check the two formulas independently, both need to hold for the compound formula to be true.
check(Transitions, Values, State, _, and(F, G)) :-
check(Transitions, Values, State, [], F), check(Transitions, Values, State, [], G).

% Or
% Check the two formulas independently, however only one needs to hold for the compound formula to be true.
check(Transitions, Values, State, _, or(F, G)):-
(check(Transitions, Values, State, [], F); check(Transitions, Values, State, [], G)).
```

```
% AX
/*
We get the adjacency list of the current state, then we iterate through this adjacency list and check that X holds for each state
immediately reachably throught the current state. This is done in the iterate predicate.
*/
check(Transitions, Values, State, Visited, ax(X)):-
getSublist(State, Transitions, XStates), iterate(Transitions, Values, XStates, Visited, X).

% EX
/*
Get the adjacency list of the current predicate. We then use the member predicate, a hypothetical state, and the adjacency list
to check if any immediately reachable states from the current hold for X. Prolog will try to match the hypothetical state with each of the
members of the adjacency list until it finds one that holds, relying on backtracking.
*/
check(Transitions, Values, State, Visited, ex(X)):-
getSublist(State, Transitions, Neighbours), member(HypState, Neighbours), check(Transitions, Values, HypState, Visited, X).

% AG
/*
The basecase for ag, if we are checking a state we have already reached, this recursive thread can terminate successfully. Success occurs when
each recursive thread terminates successfully.
*/
check(_, _, State, Visited, ag(_)):- member(State, Visited), !.

/*
The first thing we do is check the current state, then we add it to the list of visited states. We then use the ax predicate to check if ag(x)
holds for every next state.
*/
check(Transitions, Values, State, Visited, ag(X)):-
check(Transitions, Values, State, [], X), check(Transitions, Values, State, [State|Visited], ax(ag(X))).

% EG
% Similar to above, however there is only one recursive thread that needs to be satisfied, because we use ex instead of ax.
check(_, _, State, Visited, eg(_)):- member(State, Visited), !.
```

```
% Similar to ag, however, instead of checking whether eg holds for all of the current states neighbours, we use ex to check if it holds for at least one.
check(Transitions, Values, State, Visited, eg(X)):-
check(Transitions, Values, State, [], X), getSublist(State, Transitions, Neighbours), member(HypState, Neighbours), check(Transitions, Values, HypState, [State|Visited], eg(X)).


% EF
% The basecase is if we find a state where x is satisfied somewhere in the future, i.e. a state we have not yet visited.
check(Transitions, Values, State, Visited, ef(X)):-
not(member(State, Visited)), check(Transitions, Values, State, [], X).

/*
First we make sure that we have not yet visited the current state, then we add it to the states visited, then we check whether or not
any states reachable from the current state have a state where X holds in the future.
*/
check(Transitions, Values, State, Visited, ef(X)):-
not(member(State, Visited)), getSublist(State, Transitions, Neighbours), member(HypState, Neighbours), check(Transitions, Values, HypState, [State|Visited], ef(X)).


% AF
%Again, the basecase is when we find a state where X holds. We check that we have not visited the state so we guarantee that the program terminates.

check(Transitions, Values, State, Visited, af(X)):-
not(member(State, Visited)), check(Transitions, Values, State, [], X).

%Same as above, except we check whether it holds for all of the immediately reachable states from the current state.

check(Transitions, Values, State, Visited, af(X)):-
not(member(State, Visited)), check(Transitions, Values, State, [State|Visited], ax(af(X))).


% The negation of the basecase

check(_, Values, State, _, neg(X)) :-
getSublist(State, Values, Vals), not(member(X, Vals)).

/*
```

```
The basecase, check if the formula holds for this current state.
*/
check(_, Values, State, _, X) :-
getSublist(State, Values, Vals), member(X, Vals).

%A predicate used by ax, we iterate through all adjacent states and check whether x holds for
all of them.
iterate(_, _, [], _, _).
iterate(Transitions, Values, [State|XState], Visited, X):-
check(Transitions, Values, State, Visited, X), !, iterate(Transitions, Values, XState, Visited, X).
```