DD1351 - Model checking with CTL using reduced Temporal Logic

Alexander Söderhäll Manuel Rydholm

December 2019

Lab requirements

The program as required by Lab 3 needs to be a sound algorithm that proves rules of temporal logical statements with CTL in Prolog. Input to the algorithm is given by a .txt file where data is given to us in lists and atoms. As such, the algorithm produced needs to be able to defer from temporal logic how to verify states through $M, s \models \phi$.

Constructing model M: A Ticket Machine

The model M chosen to construct is a ticket machine. The ticket machine has the states:

- start - in this state M is waiting to start the screen.
- choose - in this state M needs that a button is pressed.
- ticket - in this state M needs that you choose a ticket.
- fare - in this state M needs that you choose a fare and have the SL card on the pad.
- pay - in this state M it needs that you pay it in cash.
- print - in this state M it will print the ticket or fare.

Then we have our atoms that influence the states:

- buy - starting the screen.
- button1 - selecting a ticket.
- button2 - selecting a fare.
- cash - giving money to the machine.
- card - the SL card that you might have.

Our model

Our model M with transitions T looks like:

```
[[start, [choose]],
[choose, [ticket, fare]],
[ticket, [pay]],
[fare, [pay]],
[pay, [print]],
[print, [start]]].
```

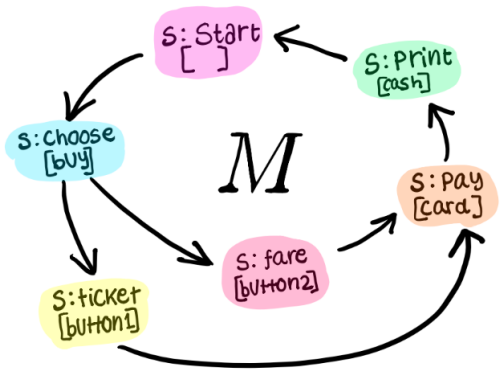

Figure 1: Model M

And our listings L looks like:

```
[[start, []],
[choose, [buy]],
[ticket, [button1]],
[fare, [button2]],
[pay, [card]],
[print, [cash]]]..
```

Valid

If you have chosen a fare and you have the card on the pad you should be able to buy.

```
[[start, [choose]],
[choose, [ticket, fare]],
[ticket, [pay]],
[fare, [pay]],
[pay, [print]],
[print, [start]]].

[[start, []],
[choose, [buy]],
[ticket, [button1]],
[fare, [button2]],
[pay, [card]],
[print, [cash]]].

fare.

ax(card).
```

Invalid

If you haven't pressed choose on start you should not be able to choose a ticket or fare.

```
[[start, [choose]],
[choose, [ticket, fare]],
[ticket, [pay]],
[fare, [pay]],
[pay, [print]],
[print, [start]]].

[[start, []],
[choose, [buy]],
[ticket, [button1]],
[fare, [button2]],
[pay, [card]],
[print, [cash]]].

start.

ax(neg(buy)).
```

How the algorithm works

The algorithm implements the rules using Prologs recursion and back-tracking. Lets look at some examples, starting with the rule EX:

```
check(T, L, S, [], ex(F)) :-
    member([S, List], T),
    member(S2, List),
    check(T, L, S2, [], F).
```

The state needs to be found in our list of transitions. If it does it will bind the transitions to a variable called List. Then it will pick a member of List using the binding in Prolog, member S2 is then to the Check at the bottom. This holds true only if formula F is true for some next state. The exhaustive back-tracking of Prolog makes us able to recursively iterate over this function to look for all possible state S2 from S.

Another example with EF_2.

```
check(T, L, S, U, ef(F)) :-
   \+ (member(S, U)),
   member([S, List], T),
   checkNodes(T, L, List, [S|U], ef(F)).

checkNodes(T, L, [S|_], U, F) :-
   check(T, L, S, U, F).
checkNodes(T, L, [_|S2], U, F) :-
   checkNodes(T, L, S2, U, F).
```

For EF_2 the current state cannot be part of previously recorded states U. We also take the transitions from S and put current state in U. Then checkNodes is called which looks if some state from S holds with formula F. Using checkNodes we are able to iterate over all states from the current state.

Predicates

- check(_, L, S, [], F) - true when formula F holds in the current state S, false otherwise.
- check(T, L, S, [], neg(F)) - true when formula F does not hold in the current state S, false otherwise.
- check(T, L, S, [], and(F,G)) - true when formulas F and G holds in the current state S, false otherwise.
- check(T, L, S, [], or(F, _)) - true when the first formula F in or holds in the current state S, false otherwise.
- check(T, L, S, [], or(_, F)) - true when the second formula F in or holds in the current state S, false otherwise.
- check(T, L, S, [], ax(F)) - true when formula F is true in all next states from current state S, false otherwise.
- check(T, L, S, [], ex(F)) - true when formula F is true in some next state from current state S, false otherwise.
- check(T, L, S, U, ag(F)) - true when formula F is true in all paths from current state S, false otherwise.
- check(T, L, S, U, af(F)) - true when formula F is true for some next statement along any path from current state S, false otherwise.
- check(_, _, S, U, eg(_)) - true when current state is a member of recorded states U, false otherwise.
- check(T, L, S, U, eg(F)) - true when formula F is true for all states in some path from current state S, flase otherwise.
- checkNodes(T, L, [S—_], U, F) - true when first state S holds with formula F, false otherwise.
- checkNodes(T, L, [_—S2], U, F) - true when some next state holds in formula F, flase otherwise.

Results

When we run *run_all_tests('ctl.pl').* in Prolog we get all valid and invalid tests verified, as such the algorithm produced can be viewed as full and sound for the specifications of the lab. It is also nice that we could use the equivalences to be able to manage A easier.

Appendix

Program Code

```
/* 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).

/* check(T, L, S, U, F) */

/*
T - The transitions in form of adjacency lists
L - The labeling
S - Current state
U - Currently recorded states
F - CTL Formula to check.
*/

/* Should evaluate to true iff the sequent below is valid.
(T,L), S |- U F */

/* To execute: consult(your_file. pl ). verify( input . txt ). */

/*Literals
Checks to see if current state S is a member of the labels,
binds L to List and checks that F is a member of List. */
check(_, L, S, [], F) :-
    member([S, List], L),
    member(F, List).

/*
Checks that F is not provable, if so go ahead. */
check(T, L, S, [], neg(F)) :-
    \+ check(T, L, S, [], F).

/* And
Checks that F and G separately are true. */
check(T, L, S, [], and(F,G)) :-
    check(T, L, S, [], F),
    check(T, L, S, [], G).

/* Or1
Checks that F holds. */
check(T, L, S, [], or(F, _)) :-
```

```
    check(T, L, S, [], F).

/* Or2
Checks that F holds. */
check(T, L, S, [], or(_, F)) :-
    check(T, L, S, [], F).

/* AX *
Uses the equivalence between AX and EX */
check(T, L, S, [], ax(F)) :-
    check(T, L, S, [], neg(ex(neg(F)))).

/* EX
Checks that current state is a member of T, bind List to T
and uses List to check that some new state S2 from S can be found
in list aswell with CTL formula F */
check(T, L, S, [], ex(F)) :-
    member([S, List], T),
    member(S2, List),
    check(T, L, S2, [], F).

/* AG
Uses the equivalence between AG and EF */
check(T, L, S, U, ag(F)) :-
   check(T, L, S, U, neg(ef(neg(F)))).

/* AF
Uses the equivalence between AF and EG */
check(T, L, S, U, af(F)) :-
    check(T, L, S, U, neg(eg(neg(F)))).

/* EG 1
Only checks that current state is a member of recorded states */
check(_, _, S, U, eg(_)) :-
   member(S, U).

/* EG 2
First make sure that current state can not be found in recorded states.
Then check that S is a member of T, bind T to List and check current
    state holds for
formula F and check that some next state in some path from S holds. */
check(T, L, S, U, eg(F)) :-
   \+ (member(S, U)),
   member([S, List], T),
   check(T, L, S, [], F),
   checkNodes(T, L, List, [S|U], eg(F)).

/* EF 1
Checks that S is not a member of recorded states, also checks current
    state with
```

```
an empty list */
check(T, L, S, U, ef(F)) :-
   \+ (member(S, U)),
   check(T, L, S, [], F).

/* EF 2
checks that current state is not part of recorded states,
binds List to T that contains transitions from S, sends
these transitions as a list of states to recursively
iterate over in checkNodes */
check(T, L, S, U, ef(F)) :-
   \+ (member(S, U)),
   member([S, List], T),
   checkNodes(T, L, List, [S|U], ef(F)).

/* Helper function for EG 2 and EF 2
Checks if next states of S are true */
checkNodes(T, L, [S|_], U, F) :-
   check(T, L, S, U, F).
checkNodes(T, L, [_|S2], U, F) :-
   checkNodes(T, L, S2, U, F).
```