Labb 2

10 december 2020

Philip Salqvist

phisal@kth.se
CINTE
1992-04-17

Innehåll

1 Algorithm

The model proving algorithm begins by reading from the input file, and assigns transitions, labeling, current state, currently recorded states, and CTL formula to T, L, S, U and F. Thereafter it runs the predicate check, validating the CTL formula by recursively checking it's content. All of the levels inside the formula are checked by being matched to it's corresponding rule. All levels thereby need to be correct for the formula to be correct in it's entirety.

If the formula is simply a literal, the check predicate uses member to find which variables holds in the current state. It then moves on to control whether our given literal F is a member of the found list of variables.

```
check(T, L, S, [], F) :-
        member([S, Holds], L),
        member(F, Holds).
```

Regarding the case of negation, the predicate wants to see if the literal F doesn't hold in the current state. We do this by using the \+ operation on the previous check predicate that controls if F holds.

```
check(T, L, S, [], neg(F)) :-
        \+ check(T, L, S, [], F).
```

Regarding $and(F, G)$ we want to know whether both F and G holds. Thus, the predicate first runs check recursively using F as formula to check and thereafter does the same thing but with G as formula. For the $or(F, G)$, the predicate should be true if either F , G or both holds. This is done by separating the predicates, first checking if G holds recursively and then in a separate predicate controlling if G holds recursively.

```
check(T, L, S, [], and(F,G)) :-
        check(T, L, S, [], F),
        check(T, L, S, [], G).


check(T, L, S, [], or(F,_)) :-
        check(T, L, S, [], F).
check(T, L, S, [], or(_,G)) :-
        check(T, L, S, [], G).
```

In the EX implementation we have to find at least one adjacent state to the current state S where F holds. Initially we have to find the list containing all adjacent states. This is done using the member predicate. Thereafter, we want to find a member of the list containing adjacent states. Since we are using member to do this, if the following rules stated bellow will fail, Prolog will simply backtrack to find another member that the following rules apply to. Finally if we find a state where F holds, the predicate is valid.

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

There are two rules regarding the EG implementation. Either S is a member of U, containing currently recorded states. If this is true the predicate terminates drawing the conclusion that the formula holds. If not, we begin by checking that current state is not a member of U. Continuing, the predicate finds adjacent states to the current states and again takes advantage of the backtracking functionality by using the member predicate to find a member of the adjacent states that the following rule holds for. Finally, we recursively call the predicate check, using the adjacent state as current state and F as formula, while adding the original current state to U.

```
check(_, _, S, U, eg(_)) :-
  member(S, U).
check(T, L, S, U, eg(F)) :-
  \+ member(S, U),
  check(T, L, S, [], F),
  member([S, Adj], T),
  member(MemberOfAdj, Adj),
  check(T, L, MemberOfAdj, [S|U], eg(F)).
```

EF is implemented in a similar way as EG, with a slight difference redarding the first rule. In this case, we want to control that S is not a member of U, and regursively check that F holds in the current state. The second rule is identical to EG.

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

In order to implement AG, AF and AX, a predicate *check_all* is introduced. The predicate first uses member to search through T and find the list containing adjacent states to current state S. After this, *check_all* splits this list into $Head$ and $Tail$. Thereafter, we control that F holds in $Head$ using *check*. Finally, *check_all* is called recursively on the $Tail$ of the list. The predicate stops when the list containing adjacent states is empty. We thereby check all branches adjacent to S.

```
check_all(_, _, _, _, []).

check_all(T, L, S, U, F) :-
  member([S, Next_branch], T),
  check_all(T, L, U, F, Next_branch).

check_all(T, L, U, F, [Head|Tail]) :-
  check(T, L, Head, U, F),
  check_all(T, L, U, F, Tail).
```

AG is implemented using two rules. First we check if S is a member of U. In the second rule it is verified that S is not a member of U. Moreover, we verify that F holds in the current state. Finally, *check_all* is called while inserting current state into the list U.

```
check(T, L, S, U, ag(F)) :-
  member(S, U).
check(T, L, S, U, ag(F)) :-
  \+ member(S, U),
```

```
  check(T, L, S, [], F),
  check_all(T, L, S, [S|U], ag(F)).
```

AF is implemented using two rules as well. In both rules we check that S is not a member of U. The first rule also verifies that F holds in the current state. To conclude the second rule, *check_all* is called to check that AF holds in all branches adjacent to S while adding S to the list U.

```
check(T, L, S, U, af(F)) :-
  \+ member(S, U),
  check(T, L, S, [], F).
check(T, L, S, U, af(F)) :-
  \+ member(S, U),
  check_all(T, L, S, [S|U], af(F)).
```

AX simply calls *check_all* to verify that F holds in all branches adjacent to the current state.

```
check(T, L, S, [], ax(F)) :-
  check_all(T, L, S, [], F).
```

2 Result

All of the valid and invalid tests are passed by the algorithm. The valid and invalid modelling of an espresso machine are both passed by the algorithm. Details can be found in Appendix B.

3 Table of predicates

Predicate	True	False
check/5	When all of the rules corresponding to the content of the CTL formula has been proven recursively.	If not, false.
check_all/5	When it has been proven that F holds in all branches adjacent to the state S.	If not, false.

4 Appendix A

```
verify(Input) :-
see(Input), read(T), read(L), read(S), read(F), seen,
check(T, L, S, [], F).

check(T, L, S, [], F) :-
        member([S, Holds], L),
        member(F, Holds).

check(T, L, S, [], neg(F)) :-
        \+ check(T, L, S, [], F).

% And
check(T, L, S, [], and(F,G)) :-
        check(T, L, S, [], F),
        check(T, L, S, [], G).

% Or
check(T, L, S, [], or(F,_)) :-
        check(T, L, S, [], F).
check(T, L, S, [], or(_,G)) :-
        check(T, L, S, [], G).

% AX
check(T, L, S, [], ax(F)) :-
         check_all(T, L, S, [], F).

/*Will check all members of adj, due to backtracking. We just have to find one
adjecent state where F holds.*/

%EX
check(T, L, S, [], ex(F)) :-
        member([S, Adj], T),
        member(MemberOfAdj, Adj),
        check(T, L, MemberOfAdj, [], F).

% AG
check(T, L, S, U, ag(F)) :-
  member(S, U).
check(T, L, S, U, ag(F)) :-
  \+ member(S, U),
  check(T, L, S, [], F),
  check_all(T, L, S, [S|U], ag(F)).



/* Does there exist a path where F globally holds?
1. If S is a member of u
2. If S is not a member osv */

% EG
check(_, _, S, U, eg(_)) :-
  member(S, U).
check(T, L, S, U, eg(F)) :-
  \+ member(S, U),
  check(T, L, S, [], F),
  member([S, Adj], T),
  member(MemberOfAdj, Adj),
  check(T, L, MemberOfAdj, [S|U], eg(F)).


% EF
check(T, L, S, U, ef(F)) :-
  \+ member(S, U),
  check(T, L, S, [], F).
check(T, L, S, U, ef(F)) :-
  \+ member(S, U),
  member([S, Adj], T),
  member(MemberOfAdj, Adj),
  check(T, L, MemberOfAdj, [S|U], ef(F)).
```

```
% AF
check(T, L, S, U, af(F)) :-
        \+ member(S, U),
        check(T, L, S, [], F).
check(T, L, S, U, af(F)) :-
        \+ member(S, U),
  check_all(T, L, S, [S|U], af(F)).


% check all
check_all(_, _, _, _, []).

check_all(T, L, S, U, F) :-
  member([S, Next_branch], T),
  check_all(T, L, U, F, Next_branch).

check_all(T, L, U, F, [Head|Tail]) :-
  check(T, L, Head, U, F),
  check_all(T, L, U, F, Tail).
```

5 Appendix B

```
%valid

[[menu, [fill_beans, fill_water, make_espresso]],
 [fill_beans, [menu]],
 [fill_water, [menu]],
 [make_espresso, [menu]]].

[[menu,[]],
 [fill_beans, []],
 [fill_water, []],
 [make_espresso, [beans, water]]].

menu.

ef(and(beans, water)).

%invalid

[[menu, [fill_beans, fill_water, make_espresso]],
 [fill_beans, [menu]],
 [fill_water, [menu]],
 [make_espresso, [menu]]].

[[menu,[]],
 [fill_beans, []],
 [fill_water, []],
 [make_espresso, [beans, water]]].

menu.

eg(and(beans, water)).
```