DD1351 Logic for Computer Scientists
Lab 3: Model Checking For CTL

Ayda Ghalkhanbaz
aydag@kth.se

Fatima Mohammad Ali
fatimama@kth.se

December 6, 2023

Introduction

The third lab of the course DD1351 covers the implementation details of an algorithm that proves the validity of a model in Computation Tree Logic (CTL) in Prolog. In the upcoming sections, the function of the algorithm is explained in detail. Additionally, a list of employed predicates, the provided code in Prolog and some run tests are presented for a comprehensive demonstration of the algorithm's validity.

Model Checking Algorithm

The validity of a CTL-model is checked by first reading a given file and saving the model, initial state, and formula as `T` (transitions in form of adjacency lists), `L` (the labeling), `S` (current state), and `F` (formula to check). What is required next is to evaluate the model by using a `check(T, L, S, U, F)` predicate where `U` is a list containing the already visited states.

The aim is to check whether the given formula is valid in the current state. This is done by handling the rules in CTL with the help of `check`\5 predicate.

The first rule is to check the literals, it is sufficient to check whether the current state `S` is present in the labels list. This is done by checking if `S`, with its next states (iterative variable '`Labels`'), is a member of `L` (the labeling lists). Followingly, it is checked whether the given state `X` is a member of the '`Labels`' variable. If the two member checks pass, then the rule holds for the given state.

Same procedure was followed when checking the negation rule. Instead of receiving state `X` as an argument, the check predicate receives '`neg(X)`' to pattern-match with the given state. The negation rule holds if `X` is not a member of '`Labels`'.

Checking the `AND` and `OR` rules is as straightforward as the atomic rules. Besides pattern matching '`and(X,Y)`' in `check(T, L, S, [], and(X,Y))`, the `AND` rule will evaluate each state `X` and `Y` by sending those as arguments to the check predicate. If both states pass their rule check, the `AND` rule holds for the given formula. The `OR` rule was implemented in a similar manner, with the difference being that only one state had to pass its check for the rule to hold (either the check for `X` or `Y` or both could be true).

The atomic and operator rules do not require considering the passed states. Thus, an empty list was sent as an argument instead of `U`.

The A-rules require checking all states from a given state, therefore the `check_all`\5 predicate doesn't take a single state, but a list of states to check. Everytime the predicate is called, it extracts the head state, if the formula is true in that state then it calls itself recursively with the rest of the state lists until it reaches the base case.

The E-rules work in a similar way when it comes to handling the states. The only difference is that E-rules can be valid/true if the formula is valid for at least one state and not all of them. Thus, the `check_exist`\5 predicate returns true if the `check`\5 for the head state holds `OR` the recursive call of the `check_exist`\5 predicate for the rest of heads holds for at least one state.

All other rules utilize `check_all`\5 or `check_exist`\5 to validate the formula. A list of all employed predicates is demonstrated in table 1 and 2. Additionally the entire program can be found in Appendix A: Prolog Code.

Design of a Model

The procedure of shopping on an online website was modelled in this task. The model and two implementations of CTL-formulas can be seen in Appendix B: Model. The model shows a website with the starting state at s0, the 'logged out' atom. Only if the user is logged in (s1), it is possible to add items to a cart (s2). The user can add to the cart forever or go to the payment site (s3). The payment at the payment site can either be accepted (s5) or declined (s4). If the payment declines, the user is redirected to the payment site (back to s3 from s4). If the payment is accepted, the user is directed to the logged in state and they can start adding to the cart again.

For simplicity, we have decided that the user can directly reach the log out state only if they are in state s1 (the logged in site). If the user is adding to cart, it is possible to log out by passing by state s1 (to reach s0). If the payment site is reached, it is possible to log out only through passing s2 and s1 to reach s0. Both s2 and s3 could in theory have a direct connection with s0, but for this specific website the current logout procedure is preferred.

Two CTL-formulas were written for the model. Example 1 in Appendix B: Model shows a valid formula on the current state `s2`, being '`ag(and(ef(i),ef(ps)))`'. This line indicates that for all global states from s2, there exist a future where both 'i' and 'ps' are valid. In other words, the formula indicates that starting from state s2 and subsequently all other states, the user can eventually reach both the log in page and payment site via at least one path, which is '`true`' for the model.

The second example indicates an invalid formula for the model; that is '`af(and(ps,pa))`' from state `s2`. This formula says that starting from state `s2`, the user can reach both '`payment site`' and '`payment accepted`' states in all future states along the paths; in other words, the user always reaches the '`payment site`' and '`payment accepted`' states eventually along all paths, which is '`false`' for this model since it is not guaranteed that '`payment accepted`' state can be reached from state `s2`.

List of Predicates

The main predicates are presented in table 1 showing in which scenarios they are true or false. Additionally, all helper predicates for checking the validation of the formula are presented in table 2.

Table 1: A list of the main predicates and their success and failure criteria

Main Predicates		
Predicate	**Success Criteria**	**Failure Criteria**
`appendEl`\3	an element has been added to a list	-
`check_all`\5	the formula is valid in all states	the formula is invalid in at least one state
`check_exist`\5	the formula is valid in at least one state	the formula is not valid in any state
`check`\5	the formula is valid given a state	the formula is not valid in the given state

Table 2: A list of helper predicates for checking the validation of the arbitrary formulas X and Y in given states

Helper Predicates	
Predicate	**Success Criteria**
`check(_,L,S,[],X)`	true when the formula holds for X
`check(_,L,S,[],neg(X))`	true when the formula doesn't hold for X
`check(T,L,S,[],and(X,Y))`	true when the formula holds for X and Y
`check(T,L,S,[],or(X,Y))`	true when the formula holds for X or Y
`check(T,L,S,[],ax(X))`	true when the formula holds for all the next states
`check(T,L,S,[],ex(X)`	true when the formula holds for some of the next states
`check(_,_,S,U,ag(_)`	true if the current state already exists in U
`check(T,L,S,U,ag(X)`	true when the formula holds for the current state and all global states
`check(_,_,S,U,eg(_)`	true if the current state already exists in U
`check(T,L,S,U,eg(X)`	true when the formula holds for some of the global states
`check(T,L,S,U,ef(X)`	true when the formula holds for the curent state
`check(T,L,S,U,ef(X)`	true when the formula holds for some of the states in future
`check(T,L,S,U,af(X)`	true when the formula holds for the current state
`check(T,L,S,U,af(X)`	true when the formula holds for all the states in future

Appendix A: Prolog Code

```prolog
% helper predicate; appending an element to a list
appendEl(X, [], [X]).
appendEl(X, [H | T], [H | Y]) :-
appendEl(X, T, Y).

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

/***************** check all states *********************/
% base case
check_all(_,_,[],_,_).
% check all states. the formula has to be true for all of them
check_all(T, L, [S|Rest],U, X) :- check(T,L, S, U,X),
                                  check_all(T, L, Rest, U, X).

/******************* check existence ******************/
% check if such a state exists where the formula is true
check_exist(T, L, [S|Rest], U, X) :- check(T, L, S, U, X);
                                     check_exist(T, L, Rest, U, X).

/************* check atomic propositions ***************/

check(_, L, S, [], X) :- member([S, Lables], L),
                        member(X, Lables).
% negation
check(_, L, S, [], neg(X)) :- member([S, Lables], L),
                             \+(member(X, Lables)).

/***************** check CTL operators *****************/
% And
check(T, L, S, [], and(X,Y)) :- check(T, L, S, [], X),
                               check(T, L, S, [], Y).

% Or
check(T, L, S, [], or(X,Y)) :- check(T, L, S, [], X);
                              check(T, L, S, [], Y).

% AX
check(T, L, S, [], ax(X)) :- member([S, NextStates], T),
                            check_all(T, L, NextStates, [], X).
```

```
% EX
check(T, L, S, [], ex(X)) :- member([S, NextStates], T),
                             check_exist(T, L, NextStates, [], X).

% AG1
check(_, _, S, U, ag(_)) :- member(S, U).

% AG2
check(T, L, S, U, ag(X)) :- \+member(S, U),
                            check(T, L, S, [], X),
                            member([S, States], T),
                            appendEl(S, U, U1),
                            check_all(T, L, States, U1, ag(X)).

% EG1
check(_, _, S, U, eg(_)) :- member(S, U).

% EG2
check(T, L, S, U, eg(X)) :- \+member(S, U),
                            check(T, L, S, [], X),
                            member([S, States], T),
                            appendEl(S, U, U1),
                            check_exist(T, L, States, U1, eg(X)).

% EF1
check(T, L, S, U, ef(X)) :- \+member(S, U),
                            check(T, L, S, [], X).

% EF2
check(T, L, S, U, ef(X)) :- \+member(S, U),
                            member([S, States], T),
                            appendEl(S, U, U1),
                            check_exist(T, L, States, U1, ef(X)).

% AF1
check(T, L, S, U, af(X)) :- \+member(S, U),
                            check(T, L, S, [], X).

% AF2
check(T, L, S, U, af(X)) :- \+member(S, U),
                            member([S, States], T),
                            appendEl(S, U, U1),
                            check_all(T, L, States, U1, af(X)).
```

Appendix B: Model

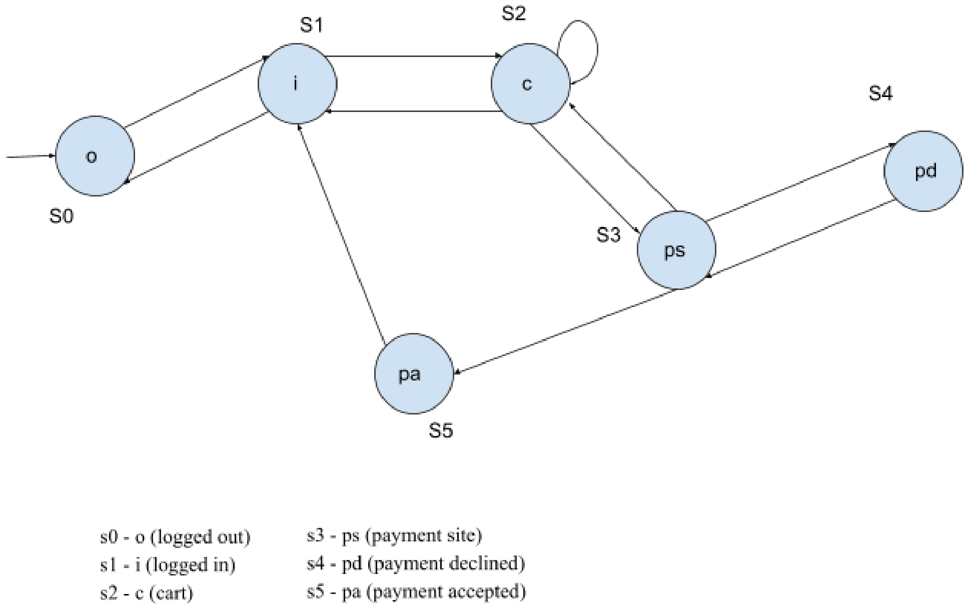

Figure 1: Model of shopping procedure on an online website.

Example of a valid proof

```
[[s0, [s1]],
 [s1, [s0, s2]],
 [s2, [s2, s1, s3]],
 [s3, [s4, s5]],
 [s4, [s3]],
 [s5, [s1]]].

[[s0, [o]],
 [s1, [i]],
 [s2, [c]],
 [s3, [ps]],
 [s4, [pd]],
 [s5, [pa]]].

s2.
```

```
ag(and(ef(i),ef(ps))).
```

Run of the valid proof

```
?- [lab3].
true.

?- verify('valid.txt').
true.
```

Example of an invalid proof

```
[[s0, [s1]],
 [s1, [s0, s2]],
 [s2, [s2, s1, s3]],
 [s3, [s4, s5]],
 [s4, [s3]],
 [s5, [s1]]].

[[s0, [o]],
 [s1, [i]],
 [s2, [c]],
 [s3, [ps]],
 [s4, [pd]],
 [s5, [pa]]].

s2.

af(and(ps, pa)).
```

Run of the invalid proof

```
?- [lab3].
true.

?- verify('invalid.txt').
false.
```