Lab 3

Alexander Wigren & Pontus Hector

Fall 2022

Introduction

This report covers the third lab assignment in the course DD1351, given during the fall semester of 2022. The assignment entailed building a proof-checking program for Computation Tree Logic (CTL) in Prolog. In the following sections we present how the algorithm works and a list of each predicate and its success criteria. We also present a CTL model and two proofs; one true and one false, to go along with it. The complete program code and example proofs may be found in the appendices.

The proof checking algorithm

In this section we present the general workings of the Prolog program. CTL proofs which may be checked by the program are structured as a collection of lists: One list consisting of states and their neighboring states and one other list which specifies which properties hold for each state. (The appendices may be viewed to see the exact structure). The proof checking we have implemented in the program may employ many different predicates while attempting to solve queries, but the general procedure is recursive in its nature. The program gradually works its way through the proof, stepping through the model while searching for the required information specified by each formula.

Bootstrapping

The program starts with a simple bootstrapping predicate `verify` which simply reads the input file and separates the content into the useful terms `T` (State adjacency list), `L` (Labeling, state properties), `S` (Initial current state) and `F` (Formula to be checked). It then calls the predicate `check` which starts the proof checking.

Checking the proof

The predicate `check` contains clauses for all the rules specified in the lab instruction: AX, EG, negation and so on. The rule-specific clauses may have some unique sub-predicates that they may call, or they may just recursively call other `check` clauses before eventually stopping at some base clause. The algorithm is in fact quite simple and we can describe an example run of it as follows:

Assume we should check whether or not the formula ag(and(x,y)) holds: The ag rule has two clauses, the first of which investigates whether or not the current state has previously been checked and if the program thus has encountered a loop. This is the base clause and results in the termination of recursion. The second clause is used if the current state has not been previously investigated, and will then call recursively call `check` with the formula and(x,y). This results in the clause for "and" being called, which in turn calls the `check` clause for literals twice; once for x and once for y. If x and y holds, we then move back up to the ag clause again and collect the neighboring states of the current state with a call to `member([S, Transitions], T)`. Afterwards we call the predicate `checkAGtransitions` which contains two clauses: A base clause if only one neighbor remains and another which is used if more than one neighbor remains. The checks are then repeated for each neighbor until all of them have been checked. If and(x,y) holds for every state until loops are detected and we exhaust every possible road, then ag(and(x,y)) is successful. If and(x,y) does not hold for any state encountered, ag(and(x,y)) will fail.

The model

The model was created around a scaled-off version of a modern smartphone, in particular around the fact that certain applications can be reached from a locked state. The model itself consists of six different states. The first state, `s0`, represents the smartphone being turned off. The second state, `s1`, represents the smartphone being turned on and locked. The third state, `s2`, represents the smartphone being turned on and unlocked. The fourth state, `s3`, represents the smartphone being turned on, locked and running the camera application. The fifth state, `s4`, represents the smartphone being turned on and running the camera application. The last state, `s5`, represents the smartphone being turned on and running the gallery application.

The only state that can be reached from the turned-off state (s0) is the turned on and locked state (s1). From the locked state one may access the unlocked state (s2), which represents the "home screen", but also the camera application directly (s3). When running the camera application from a locked state one may **not** access the gallery application (s5) directly from that state, which is possible when running the camera application from an unlocked state. Hence one can only access the locked state (s1) from that state. From the unlocked state (s2) one may access the locked state (s1), the camera application (s4), and the gallery application (s5). From the camera application (s4), where the phone is unlocked, one may access the locked state (s1), the unlocked state (s2) and the gallery application (s5). From the gallery application (s5) one may access the locked state (s1) and the unlocked state (s2). Similarly to most modern smartphones, the turned-off state can be reached from any other state.

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

[[s0, []],
 [s1, [p, q]],                       // p = ON
 [s2, [p]],                         // q = LOCKED
 [s3, [p, q, r]],                  // r = CAMERA APPLICATION RUNNING
 [s4, [p, r]],                    // s = GALLERY APPLICATION RUNNING
 [s5, [p, s]]].
```

The model was then put to a test through the model checker in which we determine if the gallery application can be accessed at all (ef(s)) from the locked state and whether one can run the gallery application while the phone is locked (ef(and(q, s))).

List of predicates and their success criteria

`Predicate` — *Success criteria*

Literals
`check` — The literal X is found in the state

Neg
`check` — The recursive call to `check` fails

And
`check` — The contents of both F and G are determined to be valid

Or
`check` — The contents of either F or G are determined to be valid

AX

`check` — The contents of X are determined to be valid throughout all transients to the current state

EX

`check` — The contents of X are determined to be valid throughout at least one other transient to the current state

AG

`check` — Base case - Loop is found

`check` — The contents of G are determined to be valid for the current state and all transient states recursively.

EG

`check` — Base case - Loop is found

`check` — The contents of G are determined to be valid for the current state or any of the transient states recursively.

AF

`check` — Base case - Loop is not found and the contents are determined to be valid.

`check` — The contents of F are determined to be valid at some transient state, for all initially following transients, and their following transients.

EF

`check` — Base case - Loop is not found and the contents are determined to be valid.

`check` — The contents of F are determined to be valid at some transient state, for at least one of the initially following transients, and their following transients.

Appendix A - Prolog Code

```
% For SICStus, uncomment line below: (needed for member/2)
%:- use_module(library(lists)).
:- discontiguous check/5.

% 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 |- F
% U
% To execute: consult('your_file.pl'). verify('input.txt').

% XXXXX Literals XXXXX
check(_, L, S, [], X) :-
    member([S, Labels], L),
    member(X, Labels).

% XXXXX Neg XXXXX
check(T, L, S, [], neg(X)) :-
     \+check(T, L, S, [], X).

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

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

% XXXXX AX XXXXX
check(T, L, S, [], ax(X)) :-
     member([S, Transitions], T),
     checkAXtransitions(T, L, Transitions, [], X).

checkAXtransitions(T, L, [LastState|[]], [], X) :-
    check(T, L, LastState, [], X).

checkAXtransitions(T, L, [State|MoreStates], [], X) :-
     check(T, L, State, [], X),
     checkAXtransitions(T, L, MoreStates, [], X).

% XXXXX EX XXXXX
check(T, L, S, [], ex(X)) :-
     member([S, Transitions], T),
     checkEXtransitions(T, L, Transitions, [], X).

checkEXtransitions(T, L, [LastState|[]], [], X) :-
    check(T, L, LastState, [], X).

checkEXtransitions(T, L, [State|MoreStates], [], X) :-
    check(T, L, State, [], X) ;
    checkEXtransitions(T, L, MoreStates, [], X).

% XXXXX AG XXXXX

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

% AG2
check(T, L, S, U, ag(G)) :-
    \+member(S, U),
    check(T, L, S, [], G),
    member([S, Transitions], T),
    checkAGtransitions(T, L, Transitions, [S|U], G).

checkAGtransitions(T, L, [LastState|[]], U, G) :-
    check(T, L, LastState, U,  ag(G)).
```

```
checkAGtransitions(T, L, [State|MoreStates], U, G) :-
    check(T, L, State, U, ag(G)),
    checkAGtransitions(T, L, MoreStates, U,  G).

% XXXXX EG XXXXX

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

% EG2
check(T, L, S, U, eg(G)) :-
    \+member(S, U),
    check(T, L, S, [], G),
    member([S, Transitions], T),
    checkEGtransitions(T, L, Transitions, [S|U], G).

checkEGtransitions(T, L, [LastState|[]], U, G) :-
    check(T, L, LastState, U, eg(G)).

checkEGtransitions(T, L, [State|MoreStates], U, G) :-
    check(T, L, State, U, eg(G)) ;
    checkEGtransitions(T, L, MoreStates, U, G).

% XXXXX AF XXXXX

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

% AF2
check(T, L, S, U, af(F)) :-
    \+member(S, U),
    member([S, Transitions], T),
    checkAFtransitions(T, L, Transitions, [S|U], F).

checkAFtransitions(T, L, [LastState|[]], U, F) :-
    check(T, L, LastState, U,  af(F)).

checkAFtransitions(T, L, [State|MoreStates], U, F) :-
    check(T, L, State, U, af(F)),
    checkAFtransitions(T, L, MoreStates, U,  F).
```

```
% XXXXX EF XXXXX

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

% EF2
check(T, L, S, U, ef(F)) :-
    \+member(S, U),
    member([S, Transitions], T),
    checkEFtransitions(T, L, Transitions, [S|U], F).

checkEFtransitions(T, L, [LastState|[]], U, F) :-
    check(T, L, LastState, U, ef(F)).

checkEFtransitions(T, L, [State|MoreStates], U, F) :-
    check(T, L, State, U, ef(F)) ;
    checkEFtransitions(T, L, MoreStates, U, F).
```

Appendix B - Example proofs

Can a state in which the gallery application is running be accessed from the locked state?

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

[[s0, []],
 [s1, [p, q]],
 [s2, [p]],
 [s3, [p, q, r]],
 [s4, [p, r]],
 [s5, [p, s]]].

s1.

ef(s).
```

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- cd('C:/Users/tjeck/OneDrive/Dokument/Prolog/lab3-tests/tests').
true.

?- ['modellprovare.pl'].
true.

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

Can one reach a state in which the gallery application is running and the phone is locked?

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

[[s0, []],
 [s1, [p, q]],
 [s2, [p]],
 [s3, [p, q, r]],
 [s4, [p, r]],
 [s5, [p, s]]].
```

```
s1.

ef(and(q, s)).
```

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- cd('C:/Users/tjeck/OneDrive/Dokument/Prolog/lab3-tests/tests').
true.

?- ['modellprovare.pl'].
true.

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