# Laboration 3: CTL-checker using Prolog

Emil Göransson

June 11, 2024

## Introduction

This report will cover one of the many ways you can implement a CTL-model-checker in Prolog. Given transitions in the form of adjacency lists, the labeling, the current state, and the CTL formula to check, the CTL-checker can say if the formula is true or false for the given adjacency and labeling lists.

## Utilizing recursion

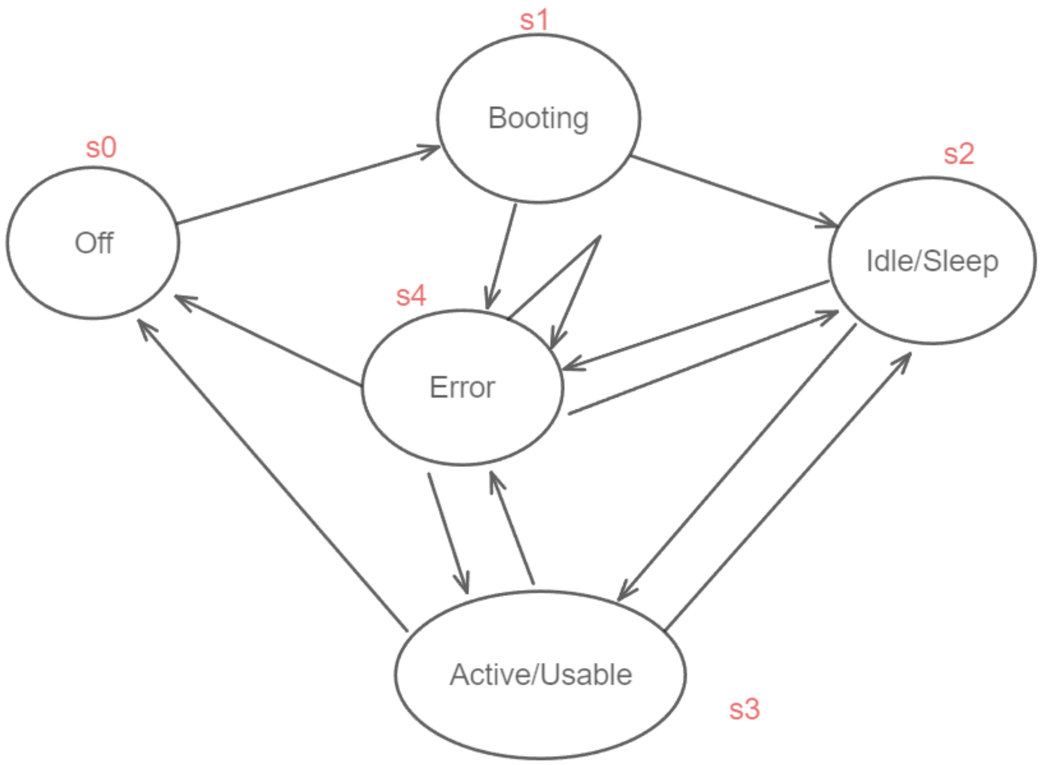
The algorithm uses backtracking and pattern matching to find the correct predicates.

```
check(_, L, S, [], F):-
```

This predicate is the first step of the program, which will fail, given that almost every valid proof contains multiple formulas to check, often chained together. Given the nature of prologs-backtracking, it is checked, however, another alternative is "chosen" since if given a valid proof, that it will find a "better" - predicate.

It breaks down the formulas into smaller bits, for example given the example ef(neg(r). it will first match with the ef-predicate check which will call the neg-predicate check which will either say true or false.

# 1 Computer-model




### Transition graph

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

### Labels

[[s0, []], [s1, [f,u]], [s2, [f,i]], [s3, [f,i,u]], [s4, []].

### Atoms explanation

1. i = connection to internet possible
2. u = takes user input
3. f = force shutdown possible

### Non-trivial CTL

#### True CTL

$$ag(neg(and(f, i))$$

in every state globally, it is not possible to both force shut down the PC and connect to the internet at the same time.

#### False CTL

$$ag(and(f, i))$$

in every state globally, it is possible to force shutdown and connect to the internet at the same time.

## Table of predicates

| **Predicate/#arguments** | **True** | **False** |
|---|---|---|
| verify/1 | When the file is supplied in the correct format | If it can't find the file or it isn't in the correct format |
| check/5 | largest part of program, true if it finds F in state S and it can find a predicate that is evaluated as true "In the end" (after backtracking) / recursion. | if there is no matching predicate that results in true |
| check_all_states/5 | true if check/5 is true for every connected path given via [CurPath—Rest] fulfills F. Uses check/5 | if not every connected state fulfills the criteria and check/5 fails |
| check_all_atleast_one_states/5 | if the "states/path CurPath or something in Rest from [CurPath—Rest] fulfills state F that is checked. | check/5 fails AND there are no more states to check |
| member/2 | If argument_1 exists inside the list argument_2 | Argument_1 does not exist in the list argument_2 |

Table 1: Explanation of Predicates

## Appendix

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

```

```prolog
%base case, checks if F is true in state S, (initial check that will fail unless its a SINGLE variable)
check(_, L, S, [], F):-
    member([S, LabelState], L), % Fetches LabelState (list of lables from states (p, r) ex.)
    member(F, LabelState).

%Not/neg
check(_, L, S, [], neg(F)):-
    member([S, LabelState], L),
    \+member(F, LabelState).
% 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,G)):-
    (
        check(T, L, S, [], F)
    ;   check(T, L, S, [], G)
    ).
% AX - All Next state
check(T, L, S, [], ax(F)):-
    %gets connected paths
    member([S, Paths], T),
    %checks all elements in Paths list
    check_all_states(T, L, Paths, [], F).
% EX
check(T, L, S, [], ex(F)):-
    member([S, Paths], T),
    check_all_atleast_one_states(T, L, Paths, [], F).


% AG1
check(_, _, S, U, ag(_)):-
    member(S, U).
%AG2
check(T, L, S, U, ag(F)):-
    \+member(S,U),
    check(T, L, S, [], F),
    member([S, Paths], T),
    check_all_states(T, L, Paths, [S|U], ag(F)).
% EG1 Basecase
```

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

% EG2, Same as AG2 just at least one state
check(T, L, S, U, eg(F)):-
    \+member(S,U),
    check(T, L, S, [], F),
    member([S, Paths], T),
    check_all_atleast_one_states(T, L, Paths, [S|U], eg(F)).

% EF1, basecase ish
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, Paths], T),
    check_all_atleast_one_states(T, L, Paths, [S|U], ef(F)).

% 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, Paths], T),
    check_all_states(T, L, Paths, [S|U], af(F)).

check_all_states(_, _, [], _, _).
check_all_states(T, L, [CurPath|Rest], U, F):-
    check(T, L, CurPath, U, F),
    check_all_states(T, L, Rest, U, F).

check_all_atleast_one_states(T, L, [CurPath|Rest], U, F):-
    (
        check(T, L, CurPath, U, F)
    ;   check_all_atleast_one_states(T, L, Rest, U, F)
    ).
```