Azer Hojlas

Lab assignment 3 - Modelling

Purpose:
This report describes a program (written in prolog) that deals with the implementation and testing of a control program in temporal logic. The program input comes in the form of a predetermined file that is formatted in a way that includes a list of truth values, a list of transitions between the states and the desired formula that we wish to prove.

Description of algorithm:
The program starts with the already included predefined code that loads the necessary files into the program. I will assume that the reader already has an understanding of this base code and I shall not elaborate any further on this. My code starts and ends with the predicate “check”. Throughout the program however, the predicate changes in function. Initially check deals with matching the desired formula with it’s appropriate counterparts in the lists. Then comes the rules that handle the listed rules of CTE in the lab requirements. Lastly we have the block of code that handles transitions within the appropriate quantifier rules (“all” and “exists”). When it comes to loops, there exist a predefined variable that stores already verified transitions (resembling the blocked list in CTE).

Predicates:

Predicate	**Purpose**	**True**	**False**
verify	Loads the input into the program and saves them in different variables.	When the file is formatted correctly.	When it isn't.
check	Checks all the rules of the CTE version described in the lab specifications	When the rules are true	When the rules are false.
all_trans_current_ state	Finds the list of transitions for the current state, then forwards the list to the predicate below. Used for rules with quantifier A	When the current state and it's transitions match the ones in the transition list T	When it doesn't match.
all_states_iterator	Validates and iterates through all the transitions for the current state, then the transitions of those states aswell, i.e a loop. Used for rules with quantifier A.	When it has iterated through all the states in that path.	When there is a state that doesn't have a label that matches the sought after formula
exists_trans_current_ state	Finds the list of transitions for the current state, then forward the list to the predicate below. Used for rules with quantifier E.	When the current state and it's transitions match the ones in the transition list T	When it doesn't match.
exists_one_state	Validates and iterates through the states. Simillar to all_states_iterator, but this predicate will cut once one path or state holds true.	When one state or path becomes true with the check predicate.	When no next paths or states hold true with the check predicate.

Model: Soda machine

I have chosen to construct a model that resembles a soda machine. The consumer is greeted by a start screen at the machine. Then they can proceed to choosing a drink, after which they will be asked to pay. If they chose wrong however, they can go back on the terminal and choose again, or proceed with payment. The card will either be declined or accepted, after which the user can choose to try to pay again or return to the start screen. If the card is accepted then the user will receive their drink and the terminal will proceed to the start screen again.

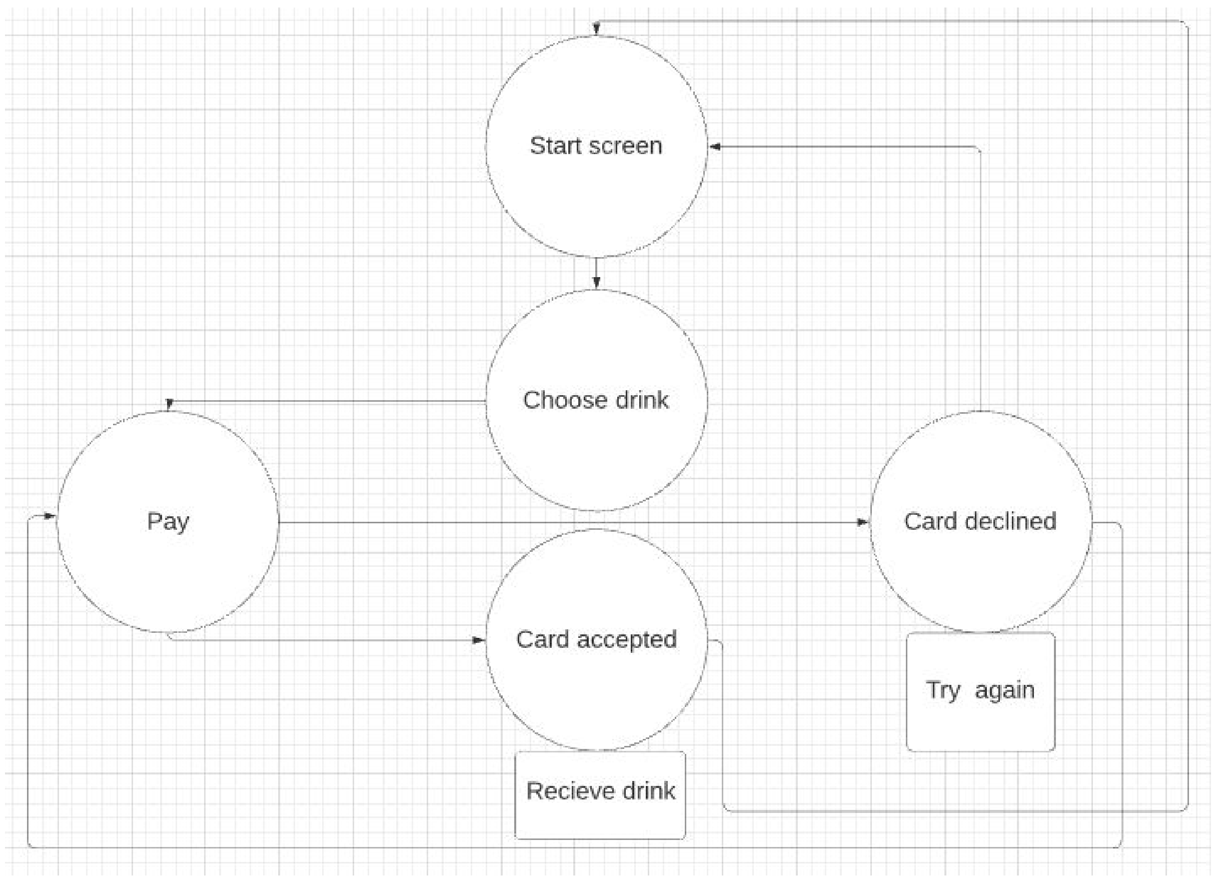
Below is an illustration of the Model:

States:

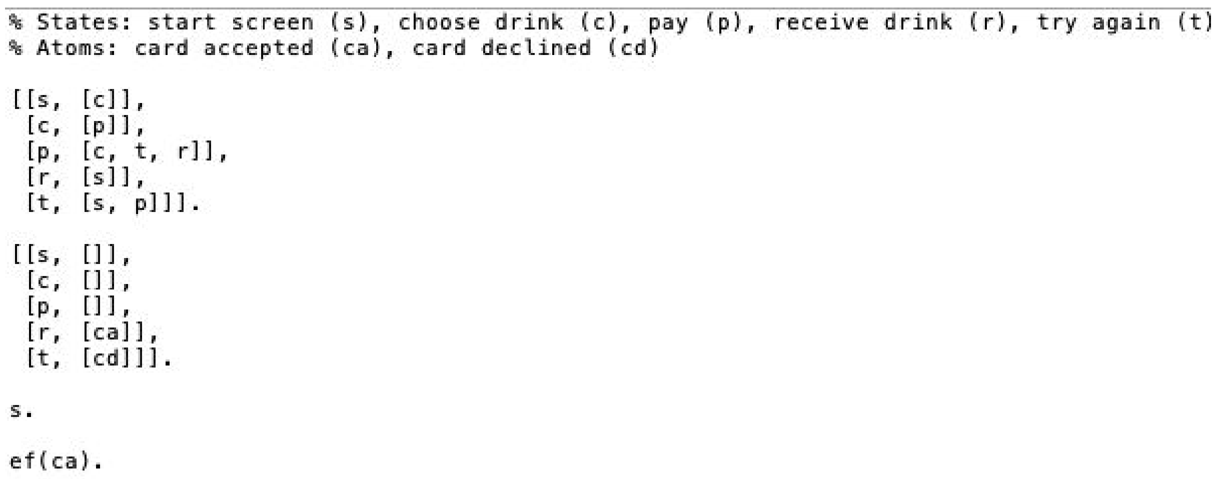
- Start screen
- Choose drink
- Pay
- Receive drink
- Try again

Atoms:

- Card accepted
- Card declined



The model in code:



```
% States: start screen (s), choose drink (c), pay (p), receive drink (r), try again (t)
% Atoms: card accepted (ca), card declined (cd)

[[s, [c]],
 [c, [p]],
 [p, [c, t, r]],
 [r, [s]],
 [t, [s, p]]].

[[s, []],
 [c, []],
 [p, []],
 [r, [ca]],
 [t, [cd]]].

s.

ef(ca).
```

Formulas:

Acceptable formula:
Buying and receiving a drink successfully:
ef(ca).

Unacceptable formula:
Getting a drink while simultaneously having the card declined:
ef(and(ca,cd)).

The code:

```
% Given code

% Loads the necessary inputs from the file
verify(Input) :-
   see(Input),
   read(T),
   read(L),
   read(S),
   read(F),
   seen,
         check(T, L, S, [], F).



% Rules

%___________Check match_______%
check(_,L,S,[],Formula):-
   member([S,LabelNames],L),
   member(Formula,LabelNames).


%_______Check NOT match_______________%
check(_,L,S,[],neg(Formula)):-
   member([S,LabelNames],L),
   \+ member(Formula,LabelNames).



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



%______AND match______%
check(T,L,S,[],and(X,Y)):-
   check(T,L,S,[],X),
   check(T,L,S,[],Y).



%___ AX - In all next states _________%
check(T,L,S,[],ax(Formula)):-
   all_trans_current_state(T,L,S,[],Formula).
```

```
%_________AG - Always________%
check(_,_,S,U,ag(_)):-
   member(S,U).

check(T,L,S,U,ag(Formula)):-
   \+member(S,U),
   check(T,L,S,[],Formula),
   all_trans_current_state(T,L,S,[S|U],ag(Formula)).



%________ AF - Eventually it will happen______%
check(T,L,S,U,af(Formula)):-
   \+member(S,U),
   check(T,L,S,[],Formula).

check(T,L,S,U,af(Formula)):-
   \+member(S,U),
   all_trans_current_state(T,L,S,[S|U],af(Formula)).

%_____EF - there exists some next state_____%

check(T,L,S,[],ex(Formula)):-
   exists_trans_current_state(T,L,S,[],Formula).



%_________EG - A path exists which always holds true___________%
check(_,_,S,U,eg(_)):-
   member(S,U).

check(T,L,S,U,eg(Formula)):-
   \+member(S,U),
   check(T,L,S,[],Formula),
   exists_trans_current_state(T,L,S,[S|U],eg(Formula)).


%____EF -  A path exists which will eventually hold true____%
check(T,L,S,U,ef(Formula)):-
   \+member(S,U),
   check(T,L,S,[],Formula).


check(T,L,S,U,ef(Formula)):-
   \+member(S,U),
    exists_trans_current_state(T,L,S,[S|U],ef(Formula)).

% Transitions

% All quantifier rules

all_trans_current_state(T,L,S,U,F):-
    member([S,Transitions],T),
    all_states_iterator(T,L,U,F,Transitions).


all_states_iterator(_,_,_,_,[]).

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

%Exists quantifier rules

exists_trans_current_state(T,L,S,U,F):-
    member([S,Transitions],T),
    exists_one_state(T,L,U,F,Transitions),!.

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