Lab 3 Report

Logic for Computer Scientists, DD1351

Group Allan & Mateusz

2024-12-10

Project members:
Mateusz Kaszyk
Allan Al Saleh

Introduction

The main purpose of this assignment is exploring (model checking) techniques for computational tree logic (CTL), and implementing a proof system tailored for CTL formulas. Model checking is used for verification of CTL formulas and is here implemented recursively while addressing the challenges posed by loops within the model, which is critical for ensuring termination.

Approach

The approach to this assignment has three steps: understanding the system, implementing the system, and validating its functionality through different scenarios.

We started out by making a skeleton version of the program that could handle the easiest possible case by implementing the literals and checking whether a given start node had a specific property or not.

Step two was implementing every subsequent rule and testing them separately in different scenarios by writing simple models and statements.

The final step of the model checker development was testing with more complicated statements nesting multiple rules and running through the provided test suite. We made a copy of our model checker that would write out every rule the model judged as being true while testing a statement and used it if the program gave a incorrect answer or froze on a test file to isolate the problem. Then we ran the updated solution through various similar scenarios.

Results

Overview

The proof system uses recursive evaluation of CTL formulas.
The ”check(T, L, S, U, F)” predicate has variables that represent:
T - The transitions in form of adjacency lists
L - The labeling
S - Current state
U - Currently controlled states
F - CTL formula to check

The CTL proof system evaluates formulas recursively while managing loops to ensure proper termination. The system uses G(globally), F(future) and X(next) for A(always) rules to ensure validity for all successors of a state and for E(exists) rules to handle specific successors. The program applies proof rules iteratively, tracking visited states with a list.

The model checker starts of with:

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

The verify_1 predicate is used when running the program to specify a .txt file with a model, staring node and statement to be tested. The second row is responsible for extracting the lists and statements necessary for the program. Finally the check_1 predicate is responsible for actually checking whether the given statement is correct.

All of the CTL rules are at least partially defined by the check_1 predicate with all the rules containing A, E, X, F and G having their own predicates responsible for handling the recursive parts of the rule implementations as well.

For the definition of the rules we used the following table provided in the assignment:

$$p\ \frac{-}{\mathcal{M}, s \vdash_{[]} p}\ p \in L(s) \qquad \neg p\ \frac{-}{\mathcal{M}, s \vdash_{[]} \neg p}\ p \notin L(s)$$

$$\wedge\ \frac{\mathcal{M}, s \vdash_{[]} \phi \qquad \mathcal{M}, s \vdash_{[]} \psi}{\mathcal{M}, s \vdash_{[]} \phi \wedge \psi}$$

$$\vee_1\ \frac{\mathcal{M}, s \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{[]} \phi \vee \psi} \qquad \vee_2\ \frac{\mathcal{M}, s \vdash_{[]} \psi}{\mathcal{M}, s \vdash_{[]} \phi \vee \psi}$$

$$\mathsf{AX}\ \frac{\mathcal{M}, s_1 \vdash_{[]} \phi \quad \cdots \quad \mathcal{M}, s_n \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{[]} \mathsf{AX}\ \phi}$$

$$\mathsf{AG}_1\ \frac{-}{\mathcal{M}, s \vdash_{U} \mathsf{AG}\ \phi}\ s \in U \qquad \mathsf{AF}_1\ \frac{\mathcal{M}, s \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{U} \mathsf{AF}\ \phi}\ s \notin U$$

$$\mathsf{AG}_2\ \frac{\mathcal{M}, s \vdash_{[]} \phi \qquad \mathcal{M}, s_1 \vdash_{U,s} \mathsf{AG}\ \phi\ \cdots\ \mathcal{M}, s_n \vdash_{U,s} \mathsf{AG}\ \phi}{\mathcal{M}, s \vdash_{U} \mathsf{AG}\ \phi}\ s \notin U$$

$$\mathsf{AF}_2\ \frac{\mathcal{M}, s_1 \vdash_{U,s} \mathsf{AF}\ \phi \quad \cdots \quad \mathcal{M}, s_n \vdash_{U,s} \mathsf{AF}\ \phi}{\mathcal{M}, s \vdash_{U} \mathsf{AF}\ \phi}\ s \notin U$$

$$\mathsf{EX}\ \frac{\mathcal{M}, s' \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{[]} \mathsf{EX}\ \phi} \qquad \mathsf{EG}_1\ \frac{-}{\mathcal{M}, s \vdash_{U} \mathsf{EG}\ \phi}\ s \in U$$

$$\mathsf{EG}_2\ \frac{\mathcal{M}, s \vdash_{[]} \phi \qquad \mathcal{M}, s' \vdash_{U,s} \mathsf{EG}\ \phi}{\mathcal{M}, s \vdash_{U} \mathsf{EG}\ \phi}\ s \notin U$$

$$\mathsf{EF}_1\ \frac{\mathcal{M}, s \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{U} \mathsf{EF}\ \phi}\ s \notin U \qquad \mathsf{EF}_2\ \frac{\mathcal{M}, s' \vdash_{U,s} \mathsf{EF}\ \phi}{\mathcal{M}, s \vdash_{U} \mathsf{EF}\ \phi}\ s \notin U$$

Figure 1: CTL rules given in the assignment

When implementing the rules in our code we referred back to the table "translating" the rules into code.

AG

The code below has been reorganized to have everything at the same place but as a example of one of the more complicated rules we can look more closely on the implementation of the "AG" rule(s):

```
check(_, _, S, U, ag(_)) :-
    member(S, U), !.
check(T, L, S, U, ag(X)) :-
    \+member(S,U),
    check(T, L, S, [], X),
    member([S, Transitions], T),
    ag(Transitions, T, L, [S|U], X), !.

ag([TransitionsHead|[]], T, L, U, X) :-
    check(T, L, TransitionsHead, U, ag(X)), !.
ag([TransitionsHead|TransitionsTail], T, L, U, X) :-
    check(T, L, TransitionsHead, U, ag(X)),
    ag(TransitionsTail, T, L, U, X), !.
```

The topmost definition of the "check" predicate checks wether a given node already has been accepted before.

The second definition first makes sure that the node hasn't already been accepted, sees if the condition is true for the start node and if yes prepares a list of all directly reachable nodes and sends it of to the "ag" predicate adding the controlled node to the originally empty "U" list.

The first definition of the "ag" predicate handles the base case of only having one reachable node either total or left to control. It sends it back to the "check" part of the rule implementation to make sure that AG for a given statement holds there as well.

The second definition takes the first reachable node in the list and starts of by doing the same thing as the first definition. Crucially it then follows it up by sending the rest of the list of directly reachable nodes back to itself to make sure that AG for a given statement holds for all nodes that can be reached as well.

Table with predicates

Below is a table with all predicates implemented by us along with their validity criteria:

Predicate	**Validity criteria**
verify \1	True if the given document is found, the data from it is divided into 2 lists and 2 statements and if the "check \5" predicate is true given those inputs.
check \5	True if a given given formula is true for a given model.
ax \4	True if all nodes directly reachable from the starting one satisfy a given condition.
ex \4	True if at least 1 node directly reachable from the starting one satisfies a given condition.
ag \5	True if all nodes reachable from the starting one satisfy a given condition.
eg \5	True if a path from the starting node exists along with all the nodes satisfy a given condition.
ef \5	True if a path from the starting node exist along with some node satisfies a given condition.
af \5	True if all paths from the starting node eventually have some node that satisfies a given condition.

Model

Our model (shown in appendix B) consists of 12 states and represents a simplified version of a ATM-machine. The states and properties are named in a way that makes them pretty self-explanatory. In short the model shows the different states in accessing a account and either displaying the account balance or withdrawing money. It does also account for the possibility of the withdrawal being denied as well as having a branch for showing the different states and possible transitions in the eventuality of a incorrect pin being entered.

Appendix A - Source code

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

% Literals
check(_, L, S, [], X) :-
    member([S, Labeling], L),
    member(X, Labeling), !.
check(_, L, S, [], neg(X)) :-
    member([S, Labeling], L),
    \+member(X, Labeling), !.

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

% Or1
check(T, L, S, [], or(X,_)) :-
    check(T, L, S, [], X), !.

% Or2
check(T, L, S, [], or(_,Y)) :-
    check(T, L, S, [], Y), !.

% AX
```

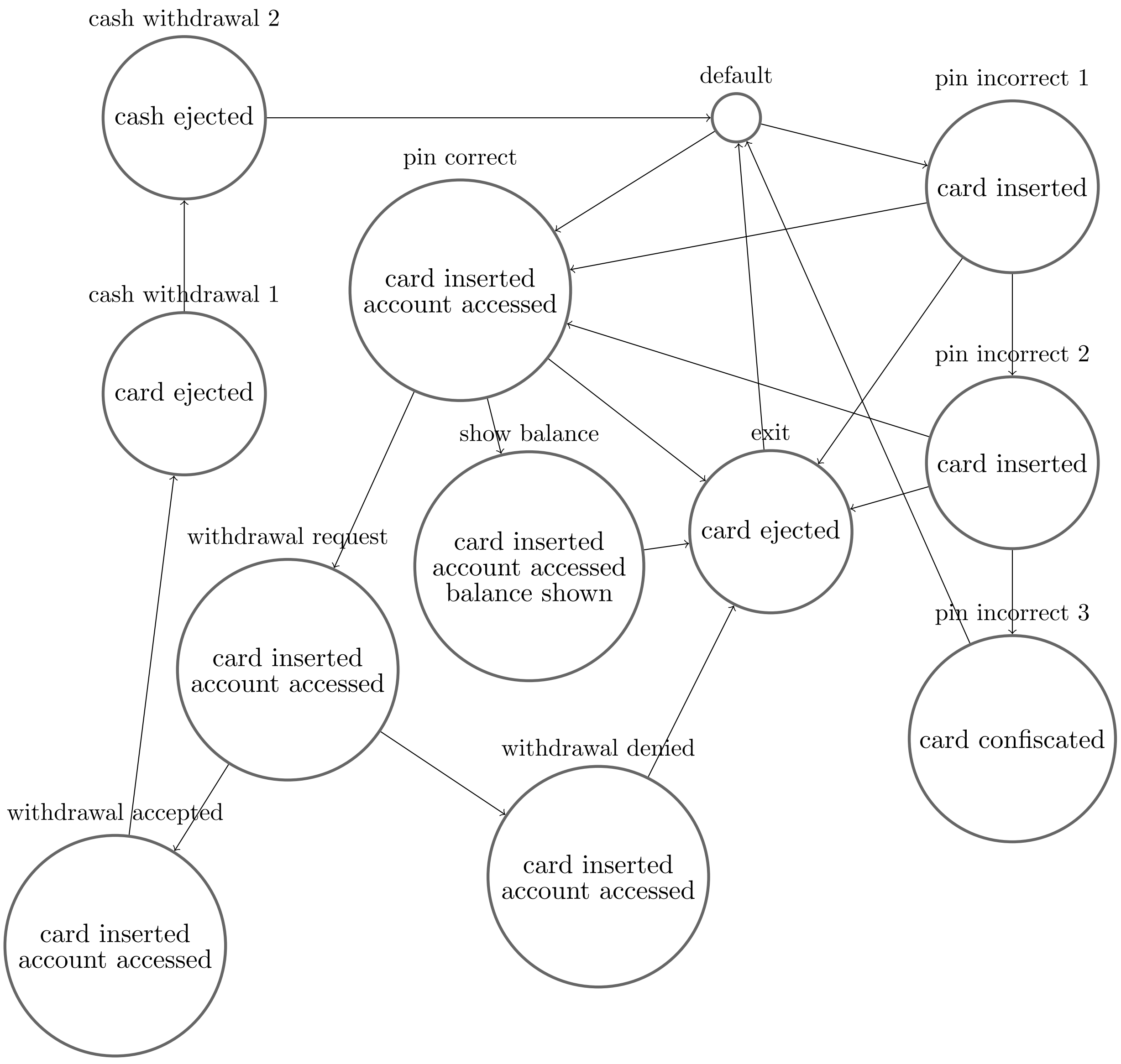
```
check(T, L, S, [], ax(X)) :-
    member([S, Transitions], T),
    ax(Transitions, T, L, X), !.

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

% AG1
check(_, _, S, U, ag(_)) :-
    member(S, U), !.
    %Om noden redan är med i listan U (om den har kontrollerats innan)

% AG2
check(T, L, S, U, ag(X)) :-
    \+member(S,U),
    check(T, L, S, [], X),
    %Om X uppfylls i startnoden
    member([S, Transitions], T),
    %Få fram lista över tillgängliga noder
    ag(Transitions, T, L, [S|U], X), !.
    %Lägg till att S har kollats o skicka vidare

% EG1
check(_, _, S, U, eg(_)) :-
    member(S, U), !.
    %Om noden redan är med i listan U (om den har kontrollerats innan)

% EG2
check(T, L, S, U, eg(X)) :-
    check(T, L, S, [], X),
    %Om X uppfylls i startnoden
    member([S, Transitions], T),
    %Få fram lista över tillgängliga noder
    eg(Transitions, T, L, [S|U], X), !.
    %Lägg till att S har kollats o skicka vidare

% EF1
check(T, L, S, U, ef(X)) :-
%Om noden inte redan är med i listan U och uppfyller X.
    \+member(S, U),
    check(T, L, S, [], X), !.
```

```
% EF2
check(T, L, S, U, ef(X)) :-
    \+member(S, U),
    member([S, Transitions], T),
    %Få fram lista över tillgängliga noder
    ef(Transitions, T, L, [S|U], X), !.
    %Lägg till att S har kollats o skicka vidare

% AF1
check(T, L, S, U, af(X)) :-
%Om noden inte redan är med i listan U och uppfyller X.
    \+member(S, U),
    check(T, L, S, [], X), !.

% AF2
check(T, L, S, U, af(X)) :-
    \+member(S, U),
    member([S, Transitions], T),
    %Få fram lista över tillgängliga noder
    af(Transitions, T, L, [S|U], X), !.
    %Lägg till att S har kollats o skicka vidare




%AX
ax([TransitionsHead|[]], T, L, X) :-
    check(T, L, TransitionsHead, [], X), !.
    %Om sista/enda noden uppfyller X enligt Literals
ax([TransitionsHead|TransitionsTail], T, L, X) :-
    check(T, L, TransitionsHead, [], X),
    %Om första noden uppfyller X enligt Literals
    ax(TransitionsTail, T, L, X), !.
    % Om de andra nåbara noderna uppfyller det

%EX
ex([], _, _, _) :- fail.
%Om vi bara har en tom lista eller om vi har stegat genom alla nåbara noder
%upfylls inte ex(X)
ex([TransitionsHead|TransitionsTail], T, L, X) :-
    check(T, L, TransitionsHead, [], X);
    %Om första noden uppfyller X enligt Literals   ELLER
    ex(TransitionsTail, T, L, X), !.
    % Om de andra nåbara noderna uppfyller det
```

```
%AG2
ag([TransitionsHead|[]], T, L, U, X) :-
    check(T, L, TransitionsHead, U, ag(X)), !.
    %Om sista/enda noden uppfyller ag(X) enligt AG1/AG2
ag([TransitionsHead|TransitionsTail], T, L, U, X) :-
    check(T, L, TransitionsHead, U, ag(X)),
    %Om första noden uppfyller ag(X) enligt AG1/AG2
    ag(TransitionsTail, T, L, U, X), !.
    % Om de andra nåbara noderna uppfyller det

%EG2
eg([TransitionsHead|[]], T, L, U, X) :-
    check(T, L, TransitionsHead, U, eg(X)), !.
    %Om sista/enda noden uppfyller tillståndet eg(X) enligt EG1/EG2
eg([TransitionsHead|TransitionsTail], T, L, U, X) :-
    check(T, L, TransitionsHead, U, eg(X));
    %Om första noden uppfyller eg(X) enligt EG1/EG2   ELLER
    eg(TransitionsTail, T, L, U, X), !.
    % Om nån av de andra nåbara noderna uppfyller det

%EF2
ef([TransitionsHead|[]], T, L, U, X) :-
    check(T, L, TransitionsHead, U, ef(X)), !.
    %Om sista/enda noden uppfyller tillståndet ef(X) enligt EF1/EF2
ef([TransitionsHead|TransitionsTail], T, L, U, X) :-
    check(T, L, TransitionsHead, U, ef(X));
    %Om första noden uppfyller ef(X) enligt EF1/EF2   ELLER
    ef(TransitionsTail, T, L, U, X), !.
    % Om nån av de andra nåbara noderna uppfyller det

%AF2
af([TransitionsHead|[]], T, L, U, X) :-
    check(T, L, TransitionsHead, U, af(X)), !.
    %Om sista/enda noden uppfyller tillståndet af(X) enligt AF1/AF2
af([TransitionsHead|TransitionsTail], T, L, U, X) :-
    check(T, L, TransitionsHead, U, af(X)),
    %Om första noden uppfyller af(X) enligt AF1/AF2
    af(TransitionsTail, T, L, U, X), !.
    % Om de andra nåbara noderna uppfyller det
```

Appendix B ATM Model

We selected to model a ATM-machine, below is a drawn model as well as the adjacency and labeling lists for it.



```
[[default, [pin_incorrect_1, pin_correct]],
 [pin_incorrect_1, [exit, pin_correct, pin_incorrect_2]],
 [pin_incorrect_2, [exit, pin_correct, pin_incorrect_3]],
 [pin_incorrect_3, [default]],
 [pin_correct, [exit, show_balance, withdrawal_request]],
 [show_balance, [exit]],
 [withdrawal_request, [withdrawal_accepted, withdrawal_denied]],
 [withdrawal_accepted, [cash_withdrawal_1]],
 [cash_withdrawal_1, [cash_withdrawal_2]],
 [cash_withdrawal_2, [default]],
 [withdrawal_denied, [exit]],
 [exit, [default]]].

[[default, []],
 [pin_incorrect_1, [card_inserted]],
 [pin_incorrect_2, [card_inserted]],
 [pin_incorrect_3, [card_confiscated]],
 [pin_correct, [card_inserted, account_accessed]],
 [show_balance, [card_inserted, account_accessed, balance_shown]],
 [withdrawal_request, [card_inserted, account_accessed]],
 [withdrawal_accepted, [card_inserted, account_accessed]],
 [cash_withdrawal_1, [card_ejected]],
 [cash_withdrawal_2, [cash_ejected]],
 [withdrawal_denied, [card_inserted, account_accessed]],
 [exit, [card_ejected]]].
```

Our model checker accepted the following starting node along with a correct statement:

```
default.

and(af(or(card_ejected, card_confiscated)), eg(neg(account_accessed))).
```

The statement says that starting at "default" we will always reach a state where "card_ejected" or "card_confiscated" is true and that there is a path where "account_accessed" is always false.

Our model checker rejected the following starting node along with a false statement:

```
default.

af(card_ejected).
```

This statement simply says that starting at "default" we will always reach a state where "card_ejected" is true.