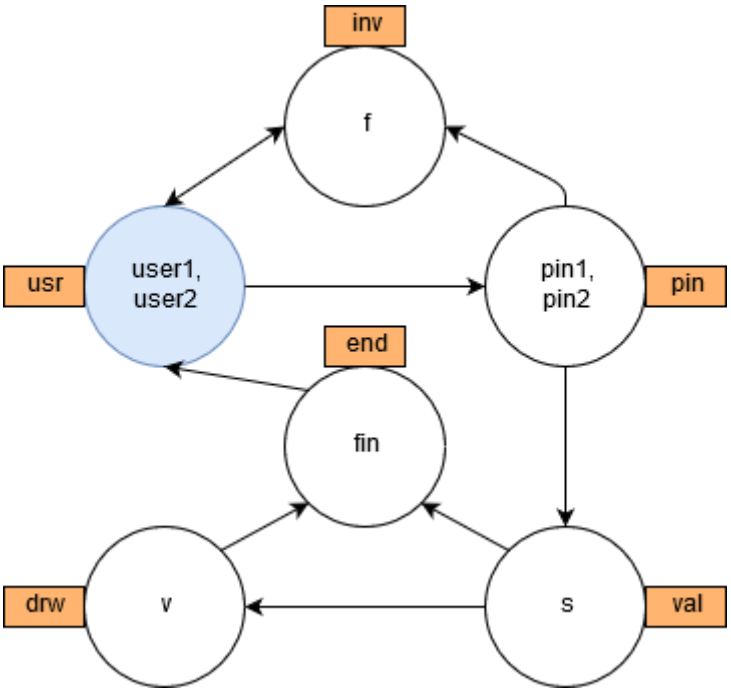
Diar Sabri, 19940227-4196
DD1351 Lab 3

Inledning

Syftet med laborationen var att skapa ett Prolog-program för modellprovning. Programmet bekräftar vare sig en logisk formel gäller för ett visst tillstånd i den givna modellen eller ej. Metoden för att skapa denna modellprovare utnyttjar främst pattern matching samt rekursion.

Modell

Modellen nedan är inspirerad av laborationen från programmeringsparadigm-kursens kallad INET. Det är en bankautomat för uttag samt saldokontroll. Användaren börjar i tillståndet *usr*. Slår användaren in korrekt användar-id går denne vidare till *pin*. Slår användaren in fel pin eller fel usr, går denne till *inv* vilket endast har en väg tillbaks, via *usr*. Är båda korrekta är nästa tillstånd *val*, och användaren kan nu se sitt saldo. Antingen slår användaren in giltigt uttagsbelopp, och går via *bal* till *end*, eller så går användaren direkt till *end*. Därefter repeteras processen.



Tillstånd

Användare:	*usr*
Pinkod:	*pin*
Ogiltigt inlogg:	*inv*
Giltigt inlogg:	*val*
Saldo > uttagsbelopp:	*drw*
Avskedsmeddelande:	*end*

Atomer

Användare:	*user1, user2*
Pinkod:	*pin1, pin2*
Ogiltigt inlogg:	*f*
Giltigt inlogg:	*s*
Saldo > uttagsbelopp:	*v*
Avskedsmeddelande:	*fin*

Prolog

```
% states
[
        [usr, [pin,inv]],
        [pin, [inv,val]],
        [inv, [usr]],
```

```
        [val, [drw,end]],
        [drw, [end]],
        [end, [urs]]
].

% atoms
[
        [usr, [user1,user2]],
        [pin, [pin1,pin2]],
        [inv, [f]],
        [val, [s]],
        [drw, [v]],
        [end, [fin]]
].

% starting state
usr.
```

Modellformler

Nedan följer två egenskaper uttryckta som CTL-formler relaterade till vår modell. Den första är en giltig egenskap, och den andra en ogiltig.

EF(EX(v)) - Det existerar en väg sådan att vi kommer till ett tillstånd där det existerar en väg där nästa tillstånd är v. I klarspråk betyder det att ifall vi lyckas logga in finns det en möjlighet för oss att i nästa steg göra ett uttag, förutsatt att det finns tillräckligt med pengar på kontot.

AF(fin) - Alla vägar vi tar kan leda till tillståndet fin. I princip innebär det att vi alltid kan logga in och antingen kolla saldot eller göra ett uttag, detta stämmer givetvis inte ifall vi har glömt koden eller glömt användar-id:t.

Predikattabell

Predikat	Sant
verify	Formeln gäller i modellen
checkAll	Kontrollerar att F håller i kommande tillstånd från S
list	Returnerar en lista
check	Om F är en atom i S

check(neg)	Om F är falskt i S
check(and)	Sant då F1 & F2 är sanna i S
check(or 1/2)	Sann då F är sann i S,
check(ax)	Sann om F är sann i alla tillstånd precis efter S
check(ag)	Sann om F är sann i alla tillstånd från S
check(af)	Sant om det från alla vägar från S existerar ett tillstånd där F är sann
check(ex)	Sann om F är sann i något av tillstånden precis efter S
check(eg)	Sann om det finns en väg där F alltid är sann
check(ef)	Sann om det existerar ett tillstånd där F är sann och kan nås från S

Alla predikat ovan är falska annars.

Kod

```
% Diar Sabri
% Lab3 DD1351

% Load model, initial state and formula from file.
verify(Input) :-
 see(Input), read(T), read(L), read(S), read(F), seen,
 check(T,       % T - The transitions in form of adjacency lists
       L,       % L - The labeling
       S,       % S - Current state
       [],      % U - Currently recorded states (empty at first)
       F).      % F - CTL Formula to check.
% Should evaluate to true iff the sequent below is valid.
% (T,L), S |- F
%           U

% Returns the list for given state
list([[S, P]|_], S, P) :- !.          % Base
list([_|T], S, P) :- list(T, S, P).   % Recursive

% Basic
check(_, L, S, [], F) :-
 list(L, S, Fres),              % Calls function above to retreive the
list
```

```
 member(F, Fres).                % Control for membership

% Iteration over all states, recursive
checkAll(_, _, [], _, _).                  % Base case
checkAll(T, L, [S|States], U, F) :-
 check(T, L, S, U, F), !,                     % Controls that the
formula holds in current state.
 checkAll(T, L, States, U, F).             % Recursive

% Negation
check(_, L, S, [], neg(F)) :-
 list(L, S, Fres),              % Returns a list with formulas to the
current state
 \+ member(F, Fres).            % Controls that the contents of F are
not a part of the list from above

% And
check(T, L, S, [], and(F1, F2)) :-
 check(T, L, S, [], F1),
 check(T, L, S, [], F2).

% Or 1
% Controls whether F is among the states
check(T, L, S, [], or(F, _)) :-
 check(T, L, S, [], F), !.

% Or 2
% Same
check(T, L, S, [], or(_, F)) :-
 check(T, L, S, [], F), !.


% AX
check(T, L, S, U, ax(F)) :-
 list(T, S, P),              % Gets list with paths from current to all
other states
 checkAll(T, L, P, U, F).    % Controls that F is valid in all these
states

% AG (I)
```

```
check(_, _, S, U, ag(_)) :-
 member(S, U).                    % Quit if current already visited
% AG (II)
check(T, L, S, U, ag(F)) :-
 check(T, L, S, [], F),           % Check F holds in current state
                                  % Add current state as visited,
check rest of states
 check(T, L, S, [S|U], ax(ag(F))).  % from current. AG should hold
there as well

% AF (I)
check(T, L, S, U, af(F)) :-
 \+ member(S, U),                 % Check current state not visited
 check(T, L, S, [], F).           % Check F holds in current state
% AF (II)
check(T, L, S, U, af(F)) :-
 \+ member(S, U),                 % Check current state not visited
 check(T, L, S, [S|U], ax(af(F))).


% EX
check(T, L, S, U, ex(F)) :-
 list(T, S, States),          % Gets list with paths from current to
all other states
 member(S1, States),          % Finds a state where F holds
 check(T, L, S1, U, F).       % Loops through all states until
satisfactory (F holds)

% EG (I)
check(_, _, S, U, eg(_)) :-
 member(S, U), !.               % Quit if current already visited
% EG (II)
check(T, L, S, U, eg(F)) :-
 check(T, L, S, [], F),         % Check F holds in current state
 list(T, S, P),                 % Gets list with paths from current to
all other states
 member(S1, P),                 % Finds a state where F holds
                                % Loops through all states until
satisfactory (F holds)
 check(T, L, S1, [S|U], eg(F)). % Mark current state as visited
```

```
% EF (I)
check(T, L, S, U, ef(F)) :-
 \+ member(S, U),                    % Check current state not visited
 check(T, L, S, [], F).              % Check F holds in current state
% EF (II)
check(T, L, S, U, ef(F)) :-
 \+ member(S, U),                    % Check current state not visited
 list(T, S, P),                      % Gets list with paths from current
to all other states
 member(S1, P),                      % Finds a state where F holds
                                     % Loops through all states until
satisfactory (F holds)
 check(T, L, S1, [S|U], ef(F)).      % Mark current state as visited
```