Laborationsrapport: Lab 3 Logik för Dataloger (DD1351)

1 Modell provning för CTL

Algoritmen för att bevisa modeller består fyra delar delar verify, check, och check_a/e. Dessa läser in modellen, provar tillstånds kvantifierare och provar väg kvantifierare.

2 Algoritmen

Verify är den delen av koden som läser in ett givet dokument och ger värden till T (övergångar), L (märkning), S (nuvarande tillstånd) och F (CTL formeln) därefter kallas även predikatet check med alla dessa element med en tom lista som senare blir till U (nuvarande sparade tillstånd). Check kollar rekursivt en delformel och dess subrutiner för att se att de stämmer. De olika versionerna av check matchas till varje ny kallelse och vid olika situationer kallas de andra delarna av programmet och/eller så jämförs innehållet mellan de olika medskickade argumenten och SAVED. SAVED innehåller grannarna till S. Om formeln är en A- eller E-formel så kommer check_a respektive check_e kallas i slutet av check. Check_a går rekursivt genom SAVED och kollar med hjälp av check om varje väg leder till B. Check_e å andra sidan försöker endast kolla om någon väg i SAVED leder till B. B är den formel som evalueras vid detta tillfälle.

3 Modellen

Vi har skapat en modell som beskriver olika tillstånd vid inloggning till en e-mailadress. Modellen har de fem följande tillstånden: startsida, inmatning, verifiera, inloggad och försök igen. Modellen har de fyra följande atomerna: användarnamn, nekas, verifieras och lösenord. Modellen är illustrerad i grafen i appendix och dess prolog kod följer nedan. För att logga in till sin e-mailadress enligt modellen presenteras man först med en startsida som sedan låter en skriva in sitt användarnamn om användarnamnet nekas skickas man tillbaka till startsidan. Däremot om användarnamnet verifieras får man skriva in sitt lösenord och om det inte är rätt lösenord förs man tillbaka till startsidan igen. Å andra sidan, om lösenordet verifieras så loggas man in.

3.1 Prolog Koden Till Modellen

%Tillstånden kallas för: Startsida -> s, Inmatning -> i, Verifiera -> v, Inloggad -> in, Försök igen -> f
%Atomerna kallas för: Användarnamn -> a, Nekas -> n, Verifieras -> ve, Rätt lösenord -> rl

```
[
[s, [i]],
[i, [s, f, v, i]],
[f, [s]],
[v, [in, f]],
[in, []]
].
```

```
[
[s, []],
[i, [a]],
[f, [a, n]],
[v, [a, ve]],
[in, [a, ve, rl]]
].
```

s.

ef(and(and(a,ve),rl)).

Hållbar systemegenskap:

EF(and(and(a,ve),rl)).

Ohållbar systemegenskap:

EF(and(and(a,ve),not(rl))

Tabell

Predikat	**När det är sant**	**När det är falskt**	**Användning**
verify/1	Då filen har rätt filformat.	Då filen har inkorrekt format.	För att läsa indatan som sedan ska användas för att kontrollera modellen.
check/5	När den formel som utvärderas och alla följande subrutiner också utvärderas till sant.	När den formel som utvärderas eller någon av följande subrutiner utvärderas till falskt.	För att kontrollera tillståndet som vi är i just nu.
check_a/6	Då en given tillståndskvantifierare gäller över alla vägar eller om den listan som fylls med dess grannar är tom.	Då en given tillståndskvantifierare inte gäller över alla vägar.	För att kolla om det tillståndet som utvärderas gäller hela listan.
check_e/6	Då en given tillståndskvantifierare gäller över någon väg och listan som fylls med dess grannar inte är tom.	Då en given tillståndskvantifierare inte gäller över någon väg.	För att kolla om något i listan gäller för det nuvarande tillståndet.

Appendix

Programkoden

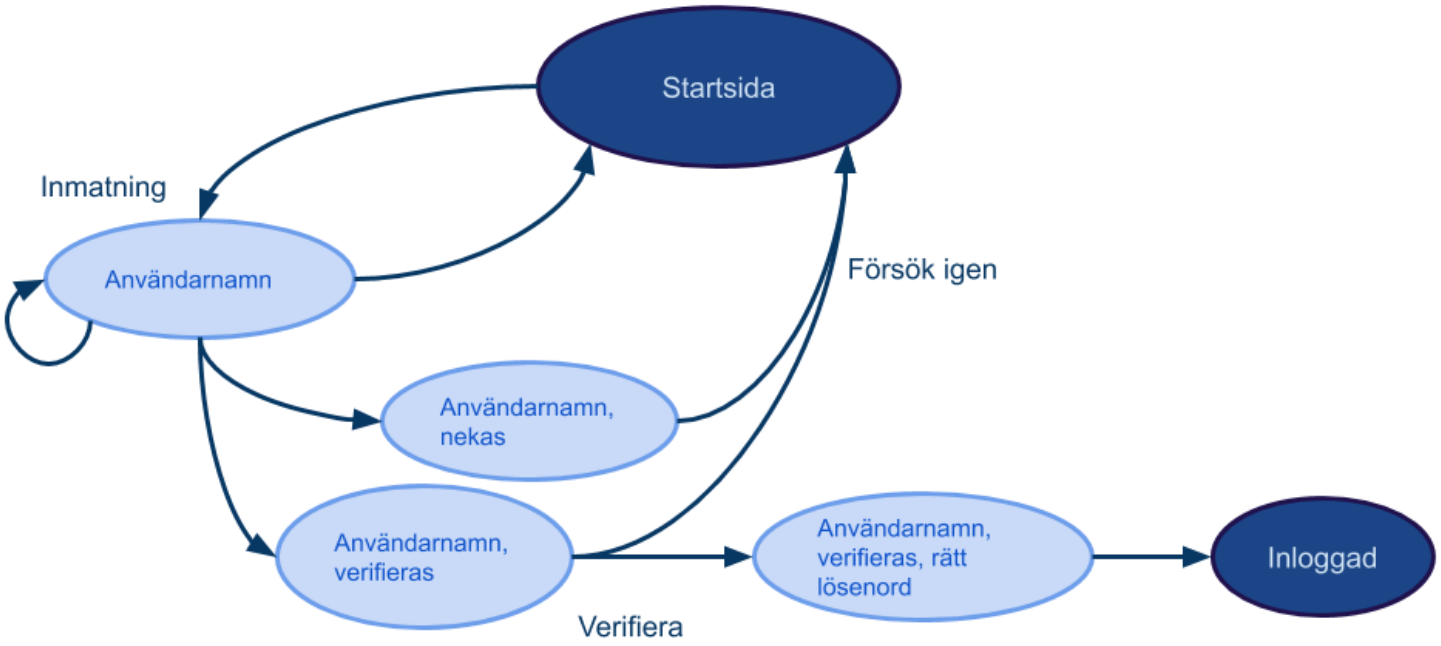
```
% For SICStus, uncomment line below: (needed for member/2)
:- use_module(library(lists)).
% Load model, initial state and formula from file.

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 if the sequent below is valid.
%
% (T,L), S  |-    F
%             U
% To execute: consult('your_file.pl'). verify('input.txt').

% Literals                  Checking S and it's list of axioms that X is present.

check(_, L, S, [], F):-
          %writeln('literals'),
          member([S, SAVED], L),
          member(F, SAVED).

%neg(X)                    Checking S and it's list of axioms that X is NOT present.

check(_, L, S, [], neg(F)) :-
                    %writeln('neg'),
                    member([S, SAVED], L),
                    \+ member(F, SAVED).

% And  Check S and it's list of axioms so that Both F & G are valid.

check(T, L, S, [], and(F,G)) :-
                    %writeln('and'),
                    check(T, L, S, [], F),
                    check(T, L, S, [], G).

% Or     Check S and it's list of axioms so thar either F or G  or both are valid.

check(T, L, S, [], or(F,G)) :-
                    %writeln('or'),
                    check(T, L, S, [], F);
                    check(T, L, S, [], G).

% AX

check(T, L, S, [], ax(F)) :-
                    %writeln('AX'),
                    member([S, SAVED], T),                                  %a list, SAVED, with all of
S neighbours from T is fetched.
                    check_a(T, L, SAVED, [], F, F).             %all neighbors in SAVED are checked.


% EX                checks whether or not the next state will satisfy F

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

```
                %writeln('EX'),
                member([S, SAVED], T),
                check_e(T, L, SAVED, [], F, F).


% AG            Checks is F is satisfied in all possible future states. Success if a loop is found.

check(T, L, S, U, ag(F)) :-
                %writeln('AG'),
                member(S, U).

check(T, L, S, U, ag(F)) :-
                %writeln('AG'),
                \+ member(S, U),
                check(T, L, S, [], F),                  %Checks whether or not F is true in the
current state.
                member([S, SAVED], T),                  %a list, SAVED, with all the
neighbours to S from T is fetched.
                check_a(T, L, SAVED, [S|U], F, ag(F)).  %all neighbours SAVED are
checked and S is added to the states that have been recorded as U.


% EG            checks if there is state left that has not been U that will satisfy F.

check(T, L, S, U, eg(F)) :-
                %writeln('EG'),
                member(S, U).

check(T, L, S, U, eg(F)) :-
                %writeln('EG'),
                \+ member(S, U),
                check(T, L, S, [], F),      %Checks if F is true in the current state which is S.
                member([S, SAVED], T),      %a list, SAVED, with all the neighbours to S from T
is fetched.
                check_e(T, L, SAVED, [S|U], F, eg(F)).

% EF            checks if there is any path that would eventually satisfy F.

check(T, L, S, U, ef(F)) :-
                %writeln('EF1'),
                \+ member(S, U),
                check(T, L, S, [], F).      %check if F is satisfied in the current state

check(T, L, S, U, ef(F)) :-
                %writeln('EF2'),
                %write(' S: '), write(S), write(' U: '), writeln(U),
                \+ member(S,U),
                %writeln('member1'),
```

```
		member([S, SAVED], T),			%a list, SAVED, with all the neighbours to S from T is fetched.
		%writeln('member2'), write('T: '), write(T), write(' L: '), write(L), write(' S: '), write(S), write(' U: '), write(U), write(' F: '), writeln(F),
		check_e(T, L, SAVED, [S|U], F, ef(F)).	%all neighbours SAVED are checked and S is added to the states that have been recorded as U.

% AF		check all possible paths and if they will eventually satisfy F, it fails if a loop is ever found.

check(T, L, S, U, af(F)) :-
		%writeln('AF'),
		\+ member(S, U),
		check(T, L, S, [], F).

check(T, L, S, U, af(F)) :-		%Checks whether or not it's true in the current state.
		%writeln('AF'),
		\+ member(S, U),
		member([S, SAVED], T),			%a list, SAVED, with all the neighbours to S from T is fetched.
		check_a(T, L, SAVED, [S|U], F, af(F)).	%all neighbours SAVED are checked and S is added to the states that have been recorded as U.

%A predicate which helps the A-formulas.
check_a(_, _, [], _, _, _).			%Gone through all the neighbours

check_a(T, L, [HUVUD|SVANS], U, A, B) :-
		%writeln('check_a'),
		%write('U: '), writeln(U),
		check(T, L, HUVUD, U, B),			%checks if it is true in HUVUD of the list of neighbours.
		check_a(T, L, SVANS, U, A, B).%Checks if it is true in the rest of the list of neighbours.

%A predicate which helps the E-formulas
check_e(_, _, [], _, _, _) :- fail.	%If the list is empty it fails

check_e(T, L, [HUVUD|SVANS], U, A, B) :-
		%writeln('check_e'),
		%write('U: '), writeln(U),
		 check(T, L, HUVUD, U, B); % skickar vidare för att verifiera current state H. försöker matcha något nästkommande states
		 check_e(T, L, SVANS, U, A, B),!.
```

Modellgrafen



Atomer :

- Användarnamn
- Nekas
- Verifieras
- Rätt lösenord

Tillstånd:

- Startsida
- Inmatning
- Verifiera
- Inloggad
- Försök igen

Intuitiv Semantik

AX Φ	-	I nästa tillstånd Φ
AG Φ	-	Alltid Φ
AF Φ	-	Så småningom Φ
EX Φ	-	I något nästa tillstånd Φ
EG Φ	-	Det finns en väg där alltid Φ
EF Φ	-	Det finns en väg där så småningom Φ