Modellprovning för CTL

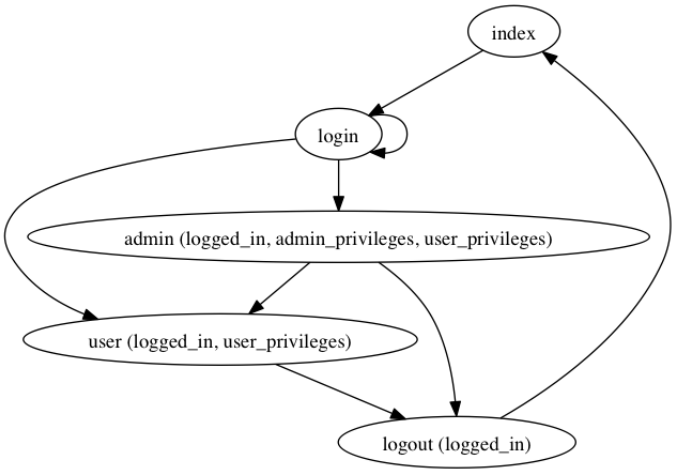
Hampus Fristedt (hamfri@kth.se)
Anton Lindqvist (antoli@kth.se)

1 Inledning

Vi har implementerat ett bevissystem för CTL i Prolog. Vår implementerar stöder det givna, begränsade, bevissystemet och klarar alla testfall. Vår implementation använder sig i hög utsträckning av s.k. *pattern matching* för att identifiera och hantera alla givna regler.

2 Modell

Vi har valt att modellera en webbsida med stöd för användarhantering i den utsträckningen att man kan logga in och ut. Det finns två typer av användare; administrator och en regelrätt användare. Både användarna har befogenhet att logga in och ut. Till skillnad från den regelrätta användaren har administören tillgång till en skyddad administrationssida.



Figur 1: En webbsida med användarhantering.

2.1 Tillstånd

- in: index: Startsidan synlig alla.
- li: login: Sida för att logga in, synlig för alla.
- us: user: Sida för inloggade användare.
- ad: admin: Sida för inloggade administratörer.
- lo: logout: Sida för inloggade att logga ut.

2.2 Atomer

- ap: admin_privileges: Administratörsrättigheter.
- up: user_privileges: Användarrättigheter.
- le: logged_in: Inloggad oavsett rättigheter.

2.3 Systemegenskaper

Vi har valt att konstruera två verklighetsförankrade systemegenskaper för vår modell.

2.3.1 Giltig egenskap

Är du inloggad så har du antingen administratör- eller användarrättigheter. Detta utrycks enklast med en implikation, men vårt bevissystem för CTL saknar stöd för implikationer, därav har vi använt ekvivalensen:

$$P \Rightarrow Q \equiv \neg P \vee Q$$

```
[[in, [li]],
 [li, [li, ad, us]],
 [us, [lo]],
 [ad, [us, lo]],
 [lo, [in]]].

[[in, []],
 [li, []],
 [us, [le, up]],
 [ad, [le, up, ap]],
 [lo, [le]]].

% If you are logged in, you either have user or admin privileges.
% This is equivalent to af(le => or(ap, up)).
in.

af(or(neg(le), or(ap, up))).
```

2.3.2 Ogiltig egenskap

Har du loggat ut kan du inte logga in igen. Detta är givetvis falskt.

```
[[in, [li]],
 [li, [li, ad, us]],
 [us, [lo]],
 [ad, [us, lo]],
 [lo, [in]]].

[[in, []],
 [li, []],
 [us, [le, up]],
 [ad, [le, up, ap]],
 [lo, [le]]].

% You can not login after logging out. Obviously false.
lo.

ag(neg(le)).
```

3 Predikat

- **check(Transitions, Labels, State, [], and(F1, F2))**: Sant då femte parametern är en and-sats och både F1 och F2 är sanna.
- **check(Transitions, Labels, State, [], or(F, _))**: Sant då den första formeln i or-satsen är sann i det givna tillståndet.
- **check(Transitions, Labels, State, [], or(_, F))**: Sant då den andra formeln i or-satsen är sann i det givna tillståndet.
- **check(Transitions, Labels, State, U, ax(F))**: Sant då formeln F är sann i samtliga nästkommande tillstånd från tillståndet State.
- **check(Transitions, Labels, State, U, ex(F))**: Sant då formeln F är sann i något nästkommande tillstånd från tillståndet State.
- **check(Transitions, Labels, State, U, ag(F)**: Sant då formeln F är sann i samtliga tillstånd längs alla vägar tillgängliga från tillståndet State.
- **check(Transitions, Labels, State, U, eg(F))**: Sant då formeln F är sann för samtliga tillstånd längs en väg tillgänglig från tillståndet State.
- **check(Transitions, Labels, State, U, ef(F))**: Sant då formeln F är sann i något tillstånd tillgängligt från tillståndet State.
- **check(Transitions, Labels, State, U, af(F))**: Sant då formeln F är sann i samtliga framtida tillstånd tillgängliga från tillståndet State.
- **check(_, Labels, State, [], neg(F))**: Sant då formeln F inte håller.
- **check(_, Labels, State, [], F)**: Sant då formeln F är en atom i tillståndet State.
- **checkAllNext(T, L, [State|States], U, F)**: Hjälpmetod för att verifiera att F håller i nästkommande tillstånd från tillståndet State. Kan användas rekursivt för att verifiera samtliga states.
- **getList([[State, Paths]|_], State, Paths)**: Hjälpmetod för att hämta en lista ur en lista. Se exempel i programkoden.

4 Programkod

```
% 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).

% and
check(Transitions, Labels, State, [], and(F1, F2)) :-
  check(Transitions, Labels, State, [], F1),
  check(Transitions, Labels, State, [], F2).

% or
check(Transitions, Labels, State, [], or(F, _)) :-
  check(Transitions, Labels, State, [], F), !.
check(Transitions, Labels, State, [], or(_, F)) :-
  check(Transitions, Labels, State, [], F), !.

% ax
```

```
check(Transitions, Labels, State, U, ax(F)) :-
  getList(Transitions, State, Paths),
  checkAllNext(Transitions, Labels, Paths, U, F).

% ex
check(Transitions, Labels, State, U, ex(F)) :-
  % Get all available paths from the given State.
  getList(Transitions, State, States),
  % Find a next state (S1) in the list of states (States) where F holds.
  % Since member S1 is unbound at this point Prolog will iterate over the given
  % list of states (States) until the next predicate is true which then implies
  % that F holds in found next state.
  member(S1, States),
  check(Transitions, Labels, S1, U, F).

% ag
check(_, _, State, U, ag(_)) :-
  % Stop if the current state has been visited before.
  member(State, U).
check(Transitions, Labels, State, U, ag(F)) :-
  % Ensure F hold in the current state.
  check(Transitions, Labels, State, [], F),
  % Use the fact that ax(ag(F)) will traverse all next paths and
  % ensure that ag(F) holds in all those paths.
  check(Transitions, Labels, State, [State|U], ax(ag(F))).

% eg
check(_, _, State, U, eg(_)) :-
  % Stop if the current state has been visited before.
  member(State, U), !.
check(Transitions, Labels, State, U, eg(F)) :-
  % Ensure the formula holds in the current state.
  check(Transitions, Labels, State, [], F),
  % Get all paths from the current state.
  getList(Transitions, State, Paths),
  % See ex/5 for further comments.
  member(S1, Paths),
  check(Transitions, Labels, S1, [State|U], eg(F)).

% ef
check(Transitions, Labels, State, U, ef(F)) :-
  % Ensure the current state has not been visited before.
  \+ member(State, U),
  % Ensure the formula holds in the current state.
  check(Transitions, Labels, State, [], F).
check(Transitions, Labels, State, U, ef(F)) :-
  % Ensure the current state has not been visited before.
  \+ member(State, U),
  % Get all paths from the current state.
  getList(Transitions, State, Paths),
  % See ex/5 for furter comments.
  member(S1, Paths),
  check(Transitions, Labels, S1, [State|U], ef(F)).

% af
check(Transitions, Labels, State, U, af(F)) :-
  % Ensure the current state has not been visited before.
  \+ member(State, U),
  % Ensure the formula holds in the current state.
  check(Transitions, Labels, State, [], F).
check(Transitions, Labels, State, U, af(F)) :-
  % See ef/5 for furher comments.
  \+ member(State, U),
  check(Transitions, Labels, State, [State|U], ax(af(F))).

```

```
% neg
check(_, Labels, State, [], neg(F)) :-
  getList(Labels, State, Formulas),
  % Ensure the given formula is not present in the list of formulas that holds
  % in the current state.
  \+ member(F, Formulas).

% arbitrary formula
check(_, Labels, State, [], F) :-
  % Get all formulas that holds in the current state.
  getList(Labels, State, Formulas),
  % Ensure that the given formula is present in the list of formulas that
  % holds in the current state.
  member(F, Formulas).

% Iterate over all given states and ensure the given formula holds in all states.
checkAllNext(_, _, [], _, _).
checkAllNext(T, L, [State|States], U, F) :-
  check(T, L, State, U, F), !,
  checkAllNext(T, L, States, U, F).

% Get the associated list for the given state.
%
% Examples:
%
%   getList([s0, [s1]], s0, P).
%   P = [s1]
%
%   getList([s0, [r]], s0, F).
%   F = [r]
getList([[State, Paths]|_], State, Paths) :- !.
getList([_|T], State, Paths) :- getList(T, State, Paths).
```