DD1350 Logik för dataloger
Laboration 1: Modellprovning för CTL

Jarl Silvén och Simon Hellberg

14 december 2016

Predikat-tabell

Här presenteras reglerna som används till modellprövaren, som på grund av det givna kodskelettet och enkelheten i implementationen tog upp större delen av labben. Dessa översattes sen till koden i appendix B, och används för att verifiera validiteten hos modeller såsom klockmodellen i appendix A.

Variablernas namn för tabell, med förklaringar:

$\mathcal{M}$ = Modell, består av T och L.
s = Nuvarande state.
T = Transitioner, möjliga vägar ifrån nuvarande state s.
L = Labels, sanningsvärden i nuvarande state s.
U = Lista av hittills besökta states.
$[]$ = Tom lista.
ϕ, ψ = Formler sammansatta av nedanstående regler.

Predikat		Sant om och endast om:
Literal	$\mathcal{M}, s \vdash_{[]} p$	$p \in L(s)$
Neg	$\mathcal{M}, s \vdash_{[]} \neg\phi$	$\mathcal{M}, s \nvdash \phi$.
And	$\mathcal{M}, s \vdash_{[]} \phi \wedge \psi$	$\mathcal{M}, s \vdash_{[]} \phi$ och $\mathcal{M}, s \vdash_{[]} \psi$.
Or_1	$\mathcal{M}, s \vdash_{[]} \phi \vee \psi$	$\mathcal{M}, s \vdash_{[]} \phi$.
Or_2	$\mathcal{M}, s \vdash_{[]} \phi \vee \psi$	$\mathcal{M}, s \vdash_{[]} \psi$.
AX	$\mathcal{M}, s \vdash_{[]} \text{AX } \phi$	$\mathcal{M}, s_1 \vdash_{[]} \phi \ldots \mathcal{M}, s_n \vdash_{[]} \phi$, där $s_1 \ldots s_n \in T(s)$.
EX	$\mathcal{M}, s \vdash_{[]} \text{EX } \phi$	$\mathcal{M}, s' \vdash_{[]} \phi$ där $s' \in T(s)$.
AG_1	$\mathcal{M}, s \vdash_{U} \text{AG } \phi$	$s \in U$.
AG_2	$\mathcal{M}, s \vdash_{U} \text{AG } \phi$	$\mathcal{M}, s \vdash_{[]} \phi$ och $\mathcal{M}, s_1 \vdash_{[U\|s]} \text{AG } \phi \ldots \mathcal{M}, s_n \vdash_{[U\|s]} \text{AG } \phi$ där $s_1 \ldots s_n \in T(s)$.
EG_1	$\mathcal{M}, s \vdash_{U} \text{EG } \phi$	$s \in U$.
EG_2	$\mathcal{M}, s \vdash_{U} \text{EG } \phi$	$\mathcal{M}, s \vdash_{[]} \phi$ och $\mathcal{M}, s' \vdash_{[U\|s]} \text{EG } \phi$ där $s_1 \ldots s_n \in T(s)$.
AF_1	$\mathcal{M}, s \vdash_{U} \text{AF } \phi$	$s \notin U$ och $\mathcal{M}, s \vdash_{[]} \phi$.
AF_2	$\mathcal{M}, s \vdash_{U} \text{AF } \phi$	$s \notin U$ och $\mathcal{M}, s_1 \vdash_{[U\|s]} \text{AF } \phi \ldots \mathcal{M}, s_n \vdash_{[U\|s]} \text{AF } \phi$ där $s_1 \ldots s_n \in T(s)$.
EF_1	$\mathcal{M}, s \vdash_{U} \text{EF } \phi$	$s \notin U$ och $\mathcal{M}, s \vdash_{[]} \phi$.
EF_2	$\mathcal{M}, s \vdash_{U} \text{EF } \phi$	$s \notin U$ och $\mathcal{M}, s' \vdash_{[U\|s]} \text{EF } \phi$ där $s' \in T(s)$.

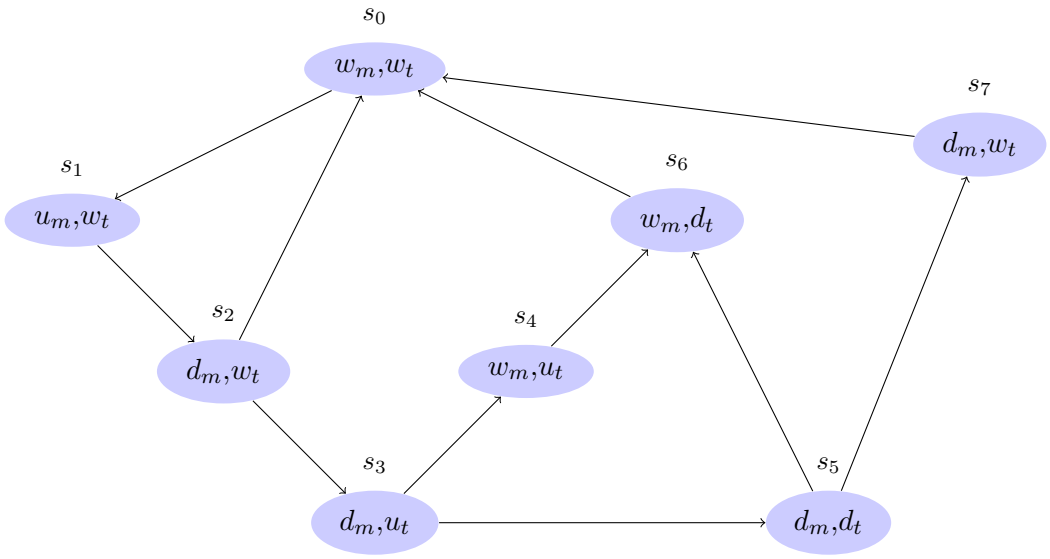
De regler som har fler än ett predikat är sanna om endera av dess predikat är sant.

Reglerna är nästan samma som de som gavs i labinstruktionerna, med undantag från negation, som i instruktionerna bara fick gälla på labels, dvs $\mathcal{M}, s \vdash \neg p$ är sant om $p \notin L(s)$.

Att utföra negationen även på hela formler är sunt därför att negation kan användas på vad som helst som kan ge ett sant eller falskt värde, och invertera detta - inte bara när det gäller literaler. Med denna ändringen blir reglerna för modellprövaren lite mer flexibla.

Appendix A - Modell och formler

Exempellmodellen baseras på en klocka. Varje state har två labels, en för minuter och en för timmar, och båda labels kan befinna sig i tillstånden waiting (w), updating (u) och done (d).
Uppdateringen av tillstånden antas också inte ta längre än en minut.
Detta ger grafen nedan:



Grafen kollas mot två formler;
$\phi = \text{EG } \neg(u_t \vee d_t)$
$\psi = \text{AG EF } (w_m \wedge \text{ EX } u_t)$
där den första kan översättas till:

> ”Det finns alltid en väg där timmen aldrig uppdateras.”

Detta motsvaras av vägen $s_0 \to s_1 \to s_2 \to s_0 \to \ldots$ som upprepas till 60 minuter har gått. Formeln godtas därmed av modellprövaren.

> ”Från var som helst kan man alltid hitta ett framtida state där minuten väntar, och i nästa state uppdateras timmen”

Vilket utifrån antagandet att uppdateringen av siffrorna är kortare än 60 sekunder är helt falskt. En timme kan aldrig börja uppdateras när en minut inte har gått. Formeln godtas därför *inte* av modellprövaren!

```
% Klockmodellens Transitions
[
        [s0,    [s1]],
        [s1,    [s2]],
        [s2,    [s0,s3]],
        [s3,    [s4,s5]],
        [s4,    [s6]],
        [s5,    [s6,s7]],
        [s6,    [s0]],
        [s7,    [s0]]
].
```

```
% Klockmodellens Labels
[
        [s0,    [wm,wt]],
        [s1,    [um,wt]],
        [s2,    [dm,wt]],
        [s3,    [dm,ut]],
        [s4,    [wm,ut]],
        [s5,    [wm,dt]],
        [s6,    [dm,dt]],
        [s7,    [dm,wt]]
].
```

Appendix B - Källkod

```
% Load model, initial state and formula from file.
% To execute: consult('your_file.pl'). verify('input.txt').
verify(Input) :-
see(Input), read(T), read(L), read(S), read(F), seen,
check(T, L, S, [], F).

% check(T, L, S, U, F).
% Should evaluate to true iff the sequent below is valid.
% (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.

% Literals
check(_,L,S,[],X) :-
        find_state(S,L,P),
        in_list(X,P),!.
% Neg
check(T,L,S,[],neg(X)) :-
        check(T,L,S,[],X),!,fail.
check(_,_,_,[],neg(_)) :- !.
% And
check(T,L,S,[],and(F,G)) :-
        check(T,L,S,[],F),
        check(T,L,S,[],G),!.
% Or
check(T,L,S,[],or(F,_)) :-
        check(T,L,S,[],F),!.
check(T,L,S,[],or(_,G)) :-
        check(T,L,S,[],G),!.
% AX
check(T,L,S,[],ax(X)) :-
        find_state(S,T,P),
        for_all(T,L,P,[],X),!.
% EX
check(T,L,S,[],ex(X)) :-
        find_state(S,T,P),
        exists(T,L,P,[],X),!.
% AG
check(_,_,S,U,ag(_)) :-
        in_list(S,U),!.
check(T,L,S,U,ag(X)) :-
        check(T,L,S,[],X),
        find_state(S,T,P),
        for_all(T,L,P,[S|U],ag(X)),!.
```

```
% EG
check(_,_,S,U,eg(_)) :-
        in_list(S,U),!.
check(T,L,S,_,eg(X)) :-
        find_state(S,T,[]),!,
        check(T,L,S,[],X).
check(T,L,S,U,eg(X)) :-
        check(T,L,S,[],X),
        find_state(S,T,P),
        exists(T,L,P,[S|U],eg(X)),!.
% AF
check(_,_,S,U,af(_)) :-
        in_list(S,U),!,fail.
check(T,L,S,_,af(X)) :-
        check(T,L,S,[],X),!.
check(T,_,S,_,af(_)) :-
        find_state(S,T,[]),!,fail.
check(T,L,S,U,af(X)) :-
        find_state(S,T,P),
        for_all(T,L,P,[S|U],af(X)),!.
% EF
check(_,_,S,U,ef(_)) :-
        in_list(S,U),!,fail.
check(T,L,S,_,ef(X)) :-
        check(T,L,S,[],X),!.
check(T,L,S,U,ef(X)) :-
        find_state(S,T,P),
        exists(T,L,P,[S|U],ef(X)).

% for_all(T,L,P,U,X)
% Verify that all states in P satisfies formula X.
for_all(_,_,[],_,_).
for_all(T,L,[Head|Tail],U,X) :- check(T,L,Head,U,X),for_all(T,L,Tail,U,X).

% exists(T,L,P,U,X)
% verify that at least one state of P satisfies formula X.
exists(_,_,[],_,_) :- fail.
exists(T,L,[Head|_],U,X) :- check(T,L,Head,U,X),!.
exists(T,L,[_|Tail],U,X) :- exists(T,L,Tail,U,X).

% find_state(S,L,R)
% Find state S in list L, if it exists, and unify P with S's Property.
find_state(_,[],_) :- fail.
find_state(S,[[S,P]|_],P) :- !.
find_state(S,[_|Tail],P) :- find_state(S,Tail,P).

% in_list(X,L)
% Verify that element X is present in list L.
in_list(_,[]) :- fail.
in_list(X,[X|_]) :- !.
in_list(X,[_|Tail]) :- in_list(X,Tail).
```