DD1350 Logik för dataloger
Laboration 2: Modellprovning för CTL

Mattias Cederlund
19/12 – 2013

Introduktion

Denna laboration har gått ut på att konstruera en modellprovare för temporallogik i Prolog. Som indata till programmet ges en temporallogisk formel, ett tillstånd och en modell. Programmet ska utvärdera formeln med hjälp av en delmängd av reglerna i temporallogiken CTL och ge ett svar om formeln gäller i given modell och tillstånd.

Metod för modellprovning

Algoritmen för för modellprovaren utgår från den givna temporallogiska formeln och skalar rekursivt av den steg för steg. Alla nivåer av formeln matchas mot sina regler och beroende på vilken regel som appliceras evalueras formeln som korrekt eller inkorrekt. Alla nivåer behöver således vara korrekta enligt reglerna för att formeln i sin helhet ska evalueras som korrekt.

För fallet då formeln är en literal/atom, F, gäller att i det aktuella tillståndet ska given literal/atom vara sann. Detta kollas med hjälp av den inbyggda funktionen member/2 som gör en sökning i listan av variabler som gäller. Om den literal/atom som eftersöks påträffas gäller formeln.

Vid negation, neg(F), ska det negerade värdet, F, inte gälla. Detta kollas med \+ check/5 där man kollar om F gäller. För att neg(F) ska gälla ska alltså F inte gälla.

Vid and(F, G) eller or(F, G) görs kontroller på höger och vänster del av uttrycket med check/5. För and(F, G) måste båda sidor, F och G, gälla och för or(F, G) behöver minst ena sidan, F eller G gälla.

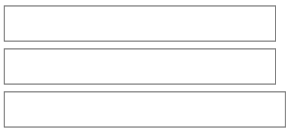
ex(F) implementeras med en sökning i listan av övergångar från det tillstånd man ska undersöka med hjälp av member/2. Enligt regeln behöver F gälla för något nästa följande tillstånd till det specificerade och man behöver alltså endast använda sig av Prologs backtracking genom member/2 för att få detta beteende.

För eg(F) finns det två regler, antingen så har man traverserat modellen i en slinga, vilket kollas med hjälp av member/2 på listan av besökta tillstånd. Blir detta sant terminerar man med resultatet att formeln gäller. Om man inte hittat en slinga så måste man vara i ett tillstånd man inte redan besökt. Detta kontrolleras med \+ member/2. Sedan så måste F gälla i det nuvarande tillståndet och det kontrolleras med check/5 på F. Sedan behöver eg(F) också gälla för något efterföljande tillstånd. Med hjälp av member/2 görs en sökning efter nästföljande tillstånd utifrån det nuvarande och man gör en check/5 på eg(F) där det nuvarande tillståndet lagts till i listan av besökta tillstånd. Även här använder man sig av Prologs backtracking av member/2 för att se till att alla möjliga tillstånd kan besökas om det behövs.

ef(F) implementeras på samma sätt som ef(F) med skillnaden att i den accepterade regeln får tillståndet inte tidigare besökts. Här kontrollerar vi även om F gäller i tillståndet. Om den första regeln inte gäller kontrollerar man den andra. Även där gäller att man inte får ha besökt det nuvarande tillståndet tidigare. Sedan görs samma sökning som för eg(F) efter nästföljande tillstånd

med member/2 och man kontrollerar ef(F) med check/5 där det nuvarande tillståndet lagts till i listan av besökta tillstånd. Backtracking används likadant som i regeln för ef(F) för att möjliggöra att besöka alla nästföljande tillstånd.

ax(F), ag(F) och af(F) har jag valt att implementera med hjälp av ekvivalenserna som fås utifrån De Morgans lagar



för att på så sätt slippa den komplexitet det skulle innebära att behöva traversera alla möjliga stigar. Denna implementation sparar även ner på kodmängden.

Modellen

Jag har valt att modellera en bankomat. Den utgörs av fem tillstånd. Starttillståndet är då bankomaten väntar på att starta en session, wait. Här är man tills användaren stoppar in sitt kort och då övergår bankomaten till tillståndet card_inserted. Då blir atomen card sann och det symboliserar att kortet är inuti bankomaten.

Nästföljande tillstånd är pin_inserted, det tillstånd bankomaten är i då användaren slagit in pinkoden. Här är fortfarande atomen card sann och även atomen auth, som symboliserar att användaren har autentiserat sig.

Nu finns möjligheten att gå till tillståndet money_withdrew, där användaren har bett om att göra ett uttag och bankomaten har slutfört transaktonen och pengarna finns redo att hämtas. Här har kortet återlämnats och atomen card är alltså falsk. Atomen auth är fortfarande sann eftersom det kan tänkas att man i ett senare skede vill ha en minneslapp vilket kräver autentisering. En atom kallad money är här också sann, vilket symboliserar att användaren har kontanter att hämta från bankomatens pengafack.

Nästa möjliga tillstånd är printed_receit vilket bankomaten hamnar i om användaren valt att skriva ut en minneslapp. Här är endast atomen receit, som symboliserar att det finns en minneslapp att ta från bankomaten. Från detta tillstånd är det endast möjligt att gå till tillståndet wait.

Från alla tillstånd är det dessutom möjligt att gå till tillståndet wait. Är kortet då instoppat kommer det att ges tillbaka från användaren.

Egenskaper

De egenskaper jag valt att specificera är:
Bankomaten kommer alltid att ge tillbaka kortet.
Det är möjligt att få ut pengar från bankomaten utan att autentisera sig.

Den andra håller självklart inte i bankomatens modell. Båda formlerna i CTL-form finns i appendix A

Resultat

Vid körning av programmet terminerar både de två egna formlerna i modellen och alla fördefinierade testfall med förväntat resultat.

Predikat

Predikat	Sant	Falskt
Verify/1	När hela givna temporallogiska formen verifierats med hjälp av check/5	Annars falskt.
Check/5	Ett predikat för varje regel i bevissystemet för CTL. Check/5 är sant om det finns någon regel som går att applicera på given temporallogiska formel.	Annars falskt.

Appendix A – Egen modell och formler

Modell

```
% Adjacency lists of LTS
[
[wait, [card_inserted]],
[card_inserted, [wait, pin_inserted]],
[pin_inserted, [money_withdrew, wait]],
[money_withdrew, [printed_receit, wait]],
[receit_printed, [wait]]
].
% Labeling of LTS
[
[wait, []],
[card_inserted, [card]],
[pin_inserted, [card, auth]],
[money_withdrew, [auth, money]],
[receit_printed, [receit]]
].
```

Valid

```
wait.
af(neg(card)).
```

Invalid

```
wait.
ef(and(money, neg(auth))).
```

Appendix B – Programkod

```
% 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, [], F) :-
      member([S, Labels], L),
      member(F , Labels).

% Negation
check(T, L, S, [], neg(F)) :-
      \+ check(T, L, S, [], F).

% 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(F)) :-
      check(T, L, S, [], neg(ex(neg(F)))).


% EX
check(T, L, S, [], ex(F)) :-
      member([S, Trans], T),
      member(NextState, Trans),
      check(T, L, NextState, [], F).

% AG
check(T, L, S, U, ag(F)) :-
      check(T, L, S, U, neg(ef(neg(F)))).
```

```
% EG
check(_, _, S, U, eg(_)) :-
      member(S,U).
check(T, L, S, U, eg(F)) :-
      \+ member(S, U),
      check(T, L, S, [], F),
      member([S, Trans], T),
      member(NextState, Trans),
      check(T, L, NextState, [S|U], eg(F)).

% EF
check(T, L, S, U, ef(F)) :-
      \+ member(S, U),
      check(T, L, S, [], F).
check(T, L, S, U, ef(F)) :-
      \+ member(S, U),
      member([S, Trans], T),
      member(NextState, Trans),
      check(T, L, NextState, [S|U], ef(F)).

% AF
check(T, L, S, U, af(F)) :-
      check(T, L, S, U, neg(eg(neg(F)))).
```