Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

Rapport lab 3

Inledning

Den här rapport behandlar konstruktionen av en modellprovare i prolog och skapandet av en exempelmodell och dess representation i ett prolog-kompatibelt format. Modellprovarens uppgift är att kontrollera om en temporallogisk formel gäller för ett visst tillstånd i en given modell för reglerna i CTL.

Allmän algoritmbeskrivning

Predikatet verify tar in indatan från filen som läses in där vi kallar predikatet check. I check, kommer vi först kommer kolla om current state och sitt sanningsvärde finns med i listan med med sanningsvärden och states. Här kollas även om det vi skickar in är sanningsvärdet av current state, om inte failar den och går vidare till de andra checks som vi gör.

Beroende på om alla eller något av vägarna kollas kommer predikatet allaA (om exempelvis AX(p) skickas in) eller allaE(om exempelvis EX(p) skickas in). Där kommer det kollas vilka states som current state leder till, och denna lista skickas sedan vidare till antingen a_check eller e_check.

På a_check och e_check delas listan av states upp till head och tail. Head skickas tillbaka till check och behandlas som en “current state”. Sedan så kallar vi rekursivt på a_check/e_check igen men med vår tail, som fortsätts brytas ner head för head som tidigare nämnt. Detta tills vi når basfallet av att Tail listan blir tom. En liten men signifikant skillnad mellan a_check och e_check är att på a_check så måste alla checks gälla, alltså att alla current states ska ge ett sant värde, medan på e_check räcker det med att enbart en av dom ger oss ett sant värde. Detta är implementerat i metoden med hjälp av ett semicolon.

Modell

Vår exempelmodell simulerar en bankapplikation där klienten kan logga in för att sedan gå till olika sidor däribland göra en betalning. I varje state symboliserar sanningsvärdet vad som behöver gälla i det statet för att det ska vara sant.

Vår modell beskrivs följande:

Atomer:
n = network connection
i = inloggad med rätt kod

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

f = false code
r = recipient
q = sending account
a = amount
s = sufficient funds
is = insufficient funds

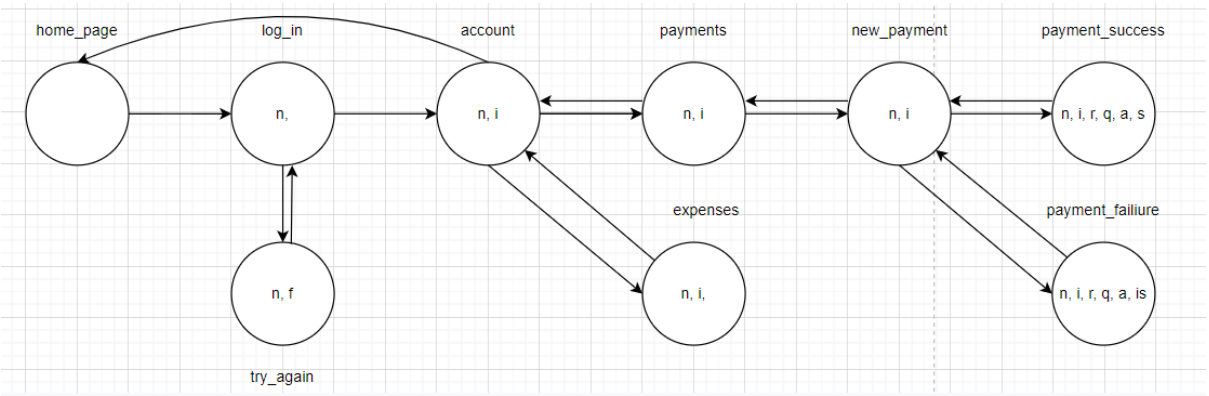
M = (S, →, L), där
S =(home_page, log_in, try_again, account, payments, expenses, new_payment, payment_failiure, payment_success)

→ = {(home_page, log_in), (log_in, try_again, account), (try_again, log_in), (account, home_page, payments, expenses), (expenses, account), (payments, new payment, account), (new_payment, payments, payment_success, payment_failiure), (payment_failiure, new_payment), (payment_success, new_payment)}

L(home_page) = {}
L(log_in) = {n}
L(try_again) = {n, f}
L(account) = {n, i}
L(expenses) = {n, i}
L(payments) = {n, i}
L(new_payment) = {n, i}
L(payment_success) = {n, i, r, q, a, s}
L(payment_failiure) = {n, i, r, q, a, is}

atomer = {n, i, f, r, q, a, s, is}

Nedan är tillståndsdiagrammet vi skapade för modellen:



Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

Vi kan se ovan att för alla states förutom home page kräver nätverksuppkoppling. För att komma till account behöver man vara inloggad vilket görs när koden är korrekt. Om man skriver in fel kod skickas man till try_again där man kan gå tillbaka till att logga in och försöka skriva in koden igen. För att detta ska ske behöver man nätverksuppkoppling och fel kod måste anges. När man väl loggat in kommer man att vara inloggad tills att man loggar ut (vilket man kan göra i account stadiet). Sedan kan man gå till payments eller expenses. Från new_payment kan man göra en betalning, där det för att genomföra betalningen korrekt behövs utöver nätverksuppkoppling och inloggning att man fyller i r (recipient), q (sending account), a (amount), s (sufficient funds). Om alla dessa förutom s uppfylls och istället uppfylls is (insufficient funds), i vilket fall som helst går man sedan tillbaka till new_payment.

Prolog-kompatibel version av modellen

För att göra modellen kompatibel med prolog-koden skapade vi två fall där en var korrekt och en inkorrekt. Själva modellen var densamma, där vi i den första listan beskriver vilka länkar varje state har. Modellen startar alltid i home_page som vi kan se nedan där modellen är.

```
 [[home_page, [log_in]],
  [log_in, [try_again, account]],
  [try_again, [log_in]],
  [account, [home_page, expenses, payments]],
  [expenses, [account]],
  [payments, [account, new_payment]],
  [new_payment, [payments, payment_success, payment_failure]],
  [payment_failure, [new_payment]],
  [payment_success, [new_payment]]].



 [[home_page, []],
  [log_in, [n]],
  [try_again, [n, r]],
  [account, [n, i]],
  [expenses, [n,i]],
  [payments, [n, i]],
  [new_payment, [n, i]],
  [payment_failure, [n, i, r, q, a, is]],
  [payment_success, [n, i, r, q, a, s]]].


 home_page.
```

Det första första fallet som är sant som vi provade är:
ex(n).
Fallet stämmer eftersom det exister ett nästa state som har n som sanningsvärde, vilket är log_in statet. Detta stämmer överens i vår exekvering av programmet.

Det andra fallet vi provade som inte gäller är:
ag(r).

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

Fallet gäller inte eftersom det inte för alla states stämmer att r är ett sanningsvärde, till exempel har inte start statet (home_page) r som sanningsvärde.

Predikattabell

Predikat	**Sann/falsk**
verify	Läser in inputfilen och kallar på check, sann om check blir sann.
check	Det finns många olika check funktioner som används i olika tillfällen. Vi har till en början checks som kollar vad för label/sanningsvärde vi har vid vår startnod. Beroende på vilken check som kallas kommer olika predikat att kallas. Om vi t.ex. skickar in ax(X) så kommer vi att vid den checken att ta funktionen innanför ax, alltså X, och skicka den som vår "formula to check, F" till funktionen allaA. Samma sak gäller för t.ex. eg(X), då man får skicka X till allaE. eg är även en egen check. Här kollar vi ifall vårt state inte är en del av våra "forbidden states, U", och sedan skickar vi X från eg(X) tillbaka till check, som kommer fungera som tidigare nämnt. Om allt stämmer kommer vi att sedan kalla på, i detta fall, allaE.
allaA	Med hjälp av member funktionen hittar vi adjacency listan till vår startnod/current node, och skickar sedan denna till a_check. Vi skickar även F som vi fick in från check till a_check. Det denna metod i princip gör är att hitta alla next states och skickar vidare dessa (t.ex. [s0, [s2, s0], då skickar vi s2,s0 vidare), samt att vi skickar vidare det som finns inuti vår funktion, t.ex. ax(X), då skickas X vidare till a_check. Detta beror dock på vilken funktion vi behandlar. Ifall vi t.ex. behandlar AF, då skickas hela af(X) vidare. Det beror helt på ifall vi har någon regel som kräver att vi sparar våra tidigare states, t.ex. AX är inte beroende av detta,

	medan AG är det.
allaE	Med hjälp av member funktionen hittar vi adjacency listan till start/current node, och skickar sedan denna till e_check. Metoden fungerar i princip som allaA, bara att den skickar till e_check istället för a_check
a_check	Här tar vi emot en lista av next states och det X från t.ex. ax(X). Med dessa kallar vi på check, och skickar in head av vår next state lista samt vår nya "formula to check", alltså X från t.ex. ax(X). Sedan så kallar vi rekursivt på a_check med tail av vår lista av next states, då vi redan bearbetat head. Sedan finns det ett basfall för att avsluta funktionen när vår next state lista är tom.
e_check	Här tar vi emot listan av next states. Vi skickar in head av denna lista till check, och även vår CTL formula to check, som kommer att vara nerskalad. Efter denna check så kallar vi rekursivt på e_check och skickar in resten av listan (Tail) tillbaka till e_check, och då görs samma process om igen tills den listan är tom.

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

Appendix

$$p\ \frac{-}{\mathcal{M}, s \vdash_{[]} p}\ p \in L(s) \qquad \neg p\ \frac{-}{\mathcal{M}, s \vdash_{[]} \neg p}\ p \notin L(s)$$

$$\wedge\ \frac{\mathcal{M}, s \vdash_{[]} \phi \qquad \mathcal{M}, s \vdash_{[]} \psi}{\mathcal{M}, s \vdash_{[]} \phi \wedge \psi}$$

$$\vee_1\ \frac{\mathcal{M}, s \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{[]} \phi \vee \psi} \qquad \vee_2\ \frac{\mathcal{M}, s \vdash_{[]} \psi}{\mathcal{M}, s \vdash_{[]} \phi \vee \psi}$$

$$\mathsf{AX}\ \frac{\mathcal{M}, s_1 \vdash_{[]} \phi \quad \cdots \quad \mathcal{M}, s_n \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{[]} \mathsf{AX}\,\phi}$$

$$\mathsf{AG}_1\ \frac{-}{\mathcal{M}, s \vdash_U \mathsf{AG}\,\phi}\ s \in U \qquad \mathsf{AF}_1\ \frac{\mathcal{M}, s \vdash_{[]} \phi}{\mathcal{M}, s \vdash_U \mathsf{AF}\,\phi}\ s \notin U$$

$$\mathsf{AG}_2\ \frac{\mathcal{M}, s \vdash_{[]} \phi \qquad \mathcal{M}, s_1 \vdash_{U,s} \mathsf{AG}\,\phi \quad \cdots \quad \mathcal{M}, s_n \vdash_{U,s} \mathsf{AG}\,\phi}{\mathcal{M}, s \vdash_U \mathsf{AG}\,\phi}\ s \notin U$$

$$\mathsf{AF}_2\ \frac{\mathcal{M}, s_1 \vdash_{U,s} \mathsf{AF}\,\phi \quad \cdots \quad \mathcal{M}, s_n \vdash_{U,s} \mathsf{AF}\,\phi}{\mathcal{M}, s \vdash_U \mathsf{AF}\,\phi}\ s \notin U$$

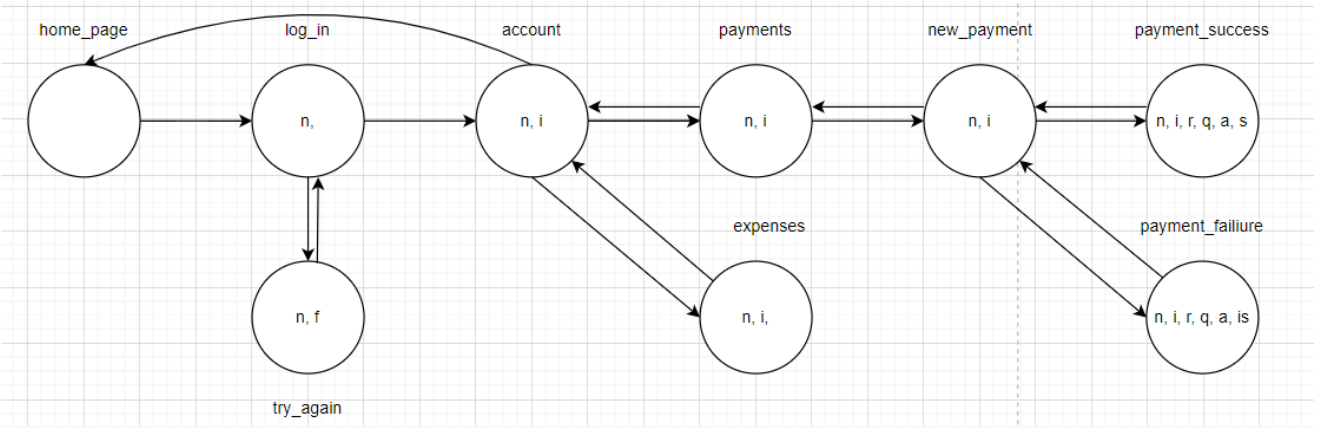
$$\mathsf{EX}\ \frac{\mathcal{M}, s' \vdash_{[]} \phi}{\mathcal{M}, s \vdash_{[]} \mathsf{EX}\,\phi} \qquad \mathsf{EG}_1\ \frac{-}{\mathcal{M}, s \vdash_U \mathsf{EG}\,\phi}\ s \in U$$

$$\mathsf{EG}_2\ \frac{\mathcal{M}, s \vdash_{[]} \phi \qquad \mathcal{M}, s' \vdash_{U,s} \mathsf{EG}\,\phi}{\mathcal{M}, s \vdash_U \mathsf{EG}\,\phi}\ s \notin U$$

$$\mathsf{EF}_1\ \frac{\mathcal{M}, s \vdash_{[]} \phi}{\mathcal{M}, s \vdash_U \mathsf{EF}\,\phi}\ s \notin U \qquad \mathsf{EF}_2\ \frac{\mathcal{M}, s' \vdash_{U,s} \mathsf{EF}\,\phi}{\mathcal{M}, s \vdash_U \mathsf{EF}\,\phi}\ s \notin U$$

Figur 1: Ett bevissystem för CTL.

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08



Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

EgenModellValid

```
[[home_page, [log_in]],
 [log_in, [try_again, account]],
 [try_again, [log_in]],
 [account, [home_page, expenses, payments]],
 [expenses, [account]],
 [payments, [account, new_payment]],
 [new_payment, [payments, payment_success, payment_failure]],
 [payment_failure, [new_payment]],
 [payment_success, [new_payment]]].



[[home_page, []],
 [log_in, [n]],
 [try_again, [n, r]],
 [account, [n, i]],
 [expenses, [n,i]],
 [payments, [n, i]],
 [new_payment, [n, i]],
 [payment_failure, [n, i, r, q, a, is]],
 [payment_success, [n, i, r, q, a, s]]].


home_page.

ex(n).
```

EgenModellInvalid

```
[[home_page, [log_in]],
 [log_in, [try_again, account]],
 [try_again, [log_in]],
 [account, [home_page, expenses, payments]],
 [expenses, [account]],
 [payments, [account, new_payment]],
 [new_payment, [payments, payment_success, payment_failure]],
 [payment_failure, [new_payment]],
 [payment_success, [new_payment]]].



[[home_page, []],
 [log_in, [n]],
 [try_again, [n, r]],
 [account, [n, i]],
 [expenses, [n,i]],
 [payments, [n, i]],
 [new_payment, [n, i]],
 [payment_failure, [n, i, r, q, a, is]],
 [payment_success, [n, i, r, q, a, s]]].


home_page.

ag(r).
```

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

```
% 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 hela första lista
   % L - The labeling hela andra listan
   % S - Current state bara startnoden
   % U - Currently recorded states listda med blocks
   % F - CTL Formula to check. formeln längst ner
   %
   % 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, [], X) :-
   %write('cet'),
      member([S, Lname],L), % kolla i label listan vilken label vår state har
      %write('cet2'),
      member(X, Lname),!. % kolla om X, vilket i detta fall antingen är formula to
check elr formeln nerskalad
   %litneg
   check(_, L, S, [], neg(X)) :-
   %write('rup'),
       member([S, Lname],L),
           %write('rup2'),
       \+ member(X, Lname).



   % And

   check(T, L, S, [], and(F,G)) :- % skicka dom två variablerna i "and" för att
checkas, båda ska gälla
   %write('and ayay'),
       check(T, L, S, [], F),
       check(T, L, S, [], G).
```

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

```
   % Or

   check(T, L, S, [], or(F,G)) :- % skicka båda variablerna i "or" för att checkas,
minst en ska gälla för att få true
   %write('or'),
       check(T, L, S, [], F);
       check(T, L, S, [], G).

   %AX VI BEHÖVER EN HJÄLP FUNKTION

   check(T,L, S,[], ax(X)):- % eftersom att AX innebär att X ska hålla i nästa
state för alla paths så skickar vi enbart X till allaA
       %write('AX innan allaA'),
       allaA(T,L,S,[], X).



   % EX

   check(T,L, S,[], ex(X)):- % samma som AX
       %write('ord'),
       allaE(T,L,S,[],X).

   % AG1
   check(_,_,S,U,ag(_)):- % om den state vi är på är med i forbidden states så bilr
detta som ett basfall
   member(S,U).

   %AG2
   check(T,L,S,U,ag(X)):- % AG vill att alla paths ska hålla för en subsequent path
       %write('AG gang'),
       \+member(S,U), % kolla att vi inte är på en forbidden state
       check(T,L,S,[],X), % kolla så att subsequent paths hold, stripa bort ag from
af(X)
       allaA(T,L,S,[S|U], ag(X)). %när vi är klara med check så lägger vi till den
staten vi är i till forbidden listan, och kallar allaA

   %EG1

   check(_,_,S,U,eg(_)):-
   member(S,U).   % kolla ifall vår state är med i forbidden states, basfall.

   %EG2
   check(T,L,S,U, eg(X)):- % ska finnas en path som hållller för en subsequent path
   %write('EG yeye'),
```

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

```
   \+member(S,U), % om vårt state inte är i forbidden states
   check(T,L,S,[],X), % kolla så subsequent paths hold, alltsa strip bort eg from
eg(x)
   allaE(T,L,S,[S|U], eg(X)). % efter ovannämnda check lägger vi till vår state
till forbidden states och kallar allaE

   %AF1
   check(T,L,S,U,af(X)):- % basfall
   \+member(S,U), % så länge current state inte är medlem i forbidden states
   check(T,L,S,[],X). % denna kommer köras tills vi antingen hittar en state som
gälker eller så kommer vi att komma ur den och komma till  AF2

   %AF2
   check(T,L,S,U,af(X)):- % AF vill att alla paths kommer att eventually hold
   \+member(S,U), % behövdes för att vi inte bara ska komma in i denna metod.
   allaA(T,L,S,[S|U], af(X)). %lägg till current state till forbidden states och gå
till allaA

   %EF1
   check(T,L,S,U,ef(X)):-
   \+member(S,U), % så länge current state inte är medlem i forbidden states
   check(T,L,S,[],X). % denna kommer köras tills vi antingen hittar en state som
gälker eller så kommer vi att komma ur den och komma till  EF2

   %EF2
   check(T,L,S,U,ef(X)):- % EF vill att en path eventually kommer att hold
   \+member(S,U), % behövdes för att vi inte bara ska komma in i denna metod.
   allaE(T,L,S,[S|U], ef(X)). %lägg till current state till forbidden states och gå
till allaE




   %-------------
%TAFRAMGRANNAR
   allaA(T,L,S,U,F) :-
       %write('woria ah ah'),
       member([S,Tail], T), % hitta allt vår state leder till, t.ex. [s0, [s2,
s0]], s0 leder till s2 och s0, tail blir s2,s0
       a_check(T,L,U,F,Tail),!. % skicka vidare tail, alltså denna del:    --------

   %
   a_check(_,_,_,_,[]). %när Tail listan från a_check blir tom
   a_check(T,L,U,F,[H|Tail]):- % dela upp tidigare tail listan till en head och ny
tail
```

Sam Khosravi
Robert Minasyan
CINTE
2021-12-08

```
   %write('innan check a check'),
       check(T,L,H,U,F), % skicka första elementet i tail som start state, och
kolla check på den
       %write('efter check a check'),
       a_check(T,L,U,F,Tail),!.  % rekursivt kalla på a_check, med Tail som nya
listan, tills den tar slut, då kickar basfallet in



   allaE(T,L,S,U,F):-
       %write('woria e e'),
       member([S,Tail], T), % hitta allt vår state leder till,lagras i tail, som
allaA exemplet
       e_check(T,L,U,F,Tail),!. % skicka tail till e_check

   %e_check(_,_,_,_,[]). %när Tail listan från e_check blir tom
   e_check(T,L,U,F,[H|Tail]):- % dela upp tidigare tail listan till en head och ny
tail
       check(T,L,H,U,F); % skicka head som vår current state
       e_check(T,L,U,F,Tail),!. % rekursivt kalla på e_check, med Tail som nya
listan, tills den tar slut och då kickar basfallet in
```