Laboration 3: Modellprovning i CTL

Tom Axberg
taxberg@kth.se

Elvira Häggström
elvhag@kth.se

1. Problembeskrivning

Denna laboration går ut på att ett prologprogram ska skrivas. Det ska kontrollera ifall ett temporallogiskt bevis, skrivet med Computation Tree Locic (CTL) enligt det bevissystem som beskrivs i kurslitteraturen[1], är korrekt eller inte. Bevissystemet använder syntaktiska sekvenser av formen:

$$\mathcal{M}, s \vdash_U \phi$$

Programmet ska svara med "yes" eller "true" om det givna beviset med modellen M, med utgångspunkt i s med tillståndsmängden U blockerad, är korrekt, och annars med "no" eller "false". Ett bevis är korrekt om alla regler som används är applicerade på ett tillfredsställande sätt. En lista av reglerna finns i figur 1, appendix A.

Problemet anses löst när det skrivna programmet ges som input till det givna programmet 'run_all_tests.pl' och detta program då ger en utskrift i stdout där alla givna textfiler, med bevis i CTL, skrivs ut med strängen "passed" efteråt. "passed" indikerar att det skrivna programmet hanterar beviset på ett korrekt sätt.

2. Angreppssätt

Då ett bevissystem givits handlade det första steget om att tolka och förstå det system som beskrivs i figur 1, Appendix A. Att förstå hur detta system skulle översättas till Prologkod låg i insikten att premisserna implicerar slutsatsen. De premisser som ges i systemet är delvis de som står över strecket och dels till höger i form av en deklaration om ett tillstånd S är ett element i U eller ej.

Utöver denna insikt krävdes även en förståelse av formatet av indata och en generell förståelse av CTL som ett temporallogiskt system. Denna förståelse, tillsammans med översättandet till Prologkod, var de verktyg som behövdes för att implementera ett fullt fungerande program.

2.1 Modellprovning

Modellprovningen, alltså det Prologprogram (se appendix B), är till sin struktur uppdelat i tre delar. Predikatet check/5 tillsammans med de första raderna i programmet, vilka tar in data från textfiler (se exempel i appendix C), utgör grunden i modellprovningen. Därefter används de predikat som är implementerade i Prologprogrammet. De är alla översatta från de regler som ingår i bevissystemet i figur 1, Appendix A. Varje regel, t.ex. "AG1", har minst ett predikat kopplat till sig för att göra den möjlig att hantera. Alla predikat har dock ett och samma namn, check/5, men urskiljs med hjälp av sina inputs. Till sist används de rekursiva predikaten check_all_states/5 samt check_exists/5, som båda implementerar fall där tillståndsmängden är tom eller inte.

För att ge läsaren en överblick av programmet ges en fullständig beskrivning av predikat och dess sanningsvärden i tabell 1, sid 2, 3. Ingående förklaring av predikaten hittas i nästa avsnitt 2.2 Predikat.

[1] Michael Huth and Mark Ryan, *Logic in Computer Science*: *Modeling and reasoning about systems*, Second Edition., Cambridge Yale University Press, 2004, e-bok, definition 3.15, sid 211.

2.2 Predikat

check/5 är det predikat som används för att implementera reglerna (se Appendix A). Varje variant av check/5 hanterar en av reglerna, och kallar på andra predikat för att uppfylla kraven för respektive regel.

check_all_states/5 används för att rekursivt anropa check/5 på en lista av tillstånd, för att kontrollera en regel mot alla tillstånd i en mängd. Med hjälp av idén att blockera tillstånd, det vill säga att man applicerar en regel men bortser från en viss mängd tillstånd, gås alla tillstånd igenom för att testa för den givna regeln.

check_all_states/5 finns i tre varianter. Den första är ett basfall, och tar hand om det fall då det inte finns fler tillstånd att kontrollera. Den andra hanterar det fall då U, som är listan av blockerade tillstånd, är en icke-tom mängd. I detta fall plockas nästa tillstånd (låt oss kalla det för s_n) ut ur S, det vill säga listan med alla tillstånd, och kontrolleras mot regeln med ett anrop till check/5. Till sist anropar check_all_states/5 sig själv rekursivt, men denna gång på tillståndet s_{n+1}, och med s_n i listan U.
Den tredje och sista varianten av check_all_states/5 är det fall då U är den tomma listan, []. Detta fall är i princip endast till för att hantera regeln AX, som tar in den tomma listan i U:s plats i både sin rekursiva premisslista och i slutsatsen.

check_exist_state/5 är mycket lik check_all_states/5, men med den viktiga skillnaden att man här inte ser till att alla tillstånd i S följer regeln, utan bara ser till att det finns ett tillstånd som gör det. Precis som check_all_states/5 finns check_exist_state/5 i tre varianter. Den första är ett basfall som hanterar det fall då listan S är tom, den andra är det fall då U är icke-tom (och tillstånden gås då igenom som i check_all_states/5), och den tredje är det fall då U är den tomma listan. I detta fall är det inte för regeln AX som denna variant finns till, utan för regeln EX. Precis som AX är det den tomma listan [] som används i både premisser och slutsats.

	True when	False when
check/5 (Literal)	$f \in L(s)$	$f \notin L(s)$
check/5 (Negation)	$f \notin L(s)$	$f \in L(s)$
check/5 (And)	$f_1 \wedge f_2$	$\neg(f_1 \wedge f_2)$
check/5 (Or)	$f_1 \vee f_2$	$\neg(f_1 \vee f_2)$
check/5 (AX)	all values in neighbours of s	$\neg$(all values in neighbours of s)
check/5 (EX)	a value in neighbour of s	$\neg$(a neighbour of s applies)
check/5 (AG1)	$s \in U$	$s \notin U$
check/5 (AG2)	$(s \notin U) \wedge \varphi \wedge s'_{n+1}$ AG(φ)	$(s \in U) \vee \neg\varphi \vee \neg s'_i$ AG(φ)
check/5 (EG1)	$s \in U$	$s \notin U$
check/5 (EG2)	$(s \notin U) \wedge (s' \vee s'')$	$(s \in U) \vee \neg(s' \wedge s'')$

check/5 (EF1)	$(s \notin U) \wedge \varphi$	$(s \in U) \vee \varphi$
check/5 (EF2)	$(s \notin U) \wedge s' \text{EF}(\varphi)$	$(s \in U) \vee \neg s' \text{EF}(\varphi)$
check/5 (AF1)	$(s \notin U) \wedge \varphi$	$(s \in U) \wedge \varphi$
check/5 (AF2)	$(s \notin U) \wedge s_{n+1} \text{AF}(\varphi)$	$(s \in U) \vee \neg s'_i \text{AF}(\varphi)$
check_all_states/5	$\forall s \in S$ is true	$\exists s \in S$ is false
check_exist_state/5	$\exists s \in S$ is true	$\forall s \in S$ is false

Tabell 1: Sanningstabell

3. Delmål och Plan

Planen var att tidigt förstå hur reglerna skulle läsas för att sedan implementera dem i prolog. Delmålen var tre stycken. Först skulle teorin förstås och enklare regler implementeras. Sedan skulle implementationen av listan U över blockerade tillstånd förstås samt rekursionerna implementeras för dessa enklare regler. Sist skulle programmet färdigställas genom att implementera de resterande reglerna och rekursionerna.

Det första delmålet var att implementera de enklare reglerna så som p, neg(p), EX samt AX då det skulle ge en bas i förståelsen för hur reglerna överfördes till prologspråket. Denna bas skulle ge insikt om hur en regel implementeras samt hur ”A” och ”E” kvantifikatorerna skulle hanteras som en rekursion.

Efter detta skulle listan U över blockerade tillstånd implementeras i programmet. Denna lista skulle fyllas på med tillstånd som skulle verifieras när de rekursiva predikaten körde. Eftersom var tillstånd verifierades, lades tillstånd till i listan före rekursionen fortsatte. Detta tills listan över tillstånd att verifiera var tom, då rekursionen då ansågs färdig. Rekursionen i sig bestod av att först verifiera tillståndet, lägga till tillståndet i listan U och sedan ringa predikatet rekursivt.

När detta var på plats så återstod bara att implementera de resterande reglerna. Då insikten av hur reglerna skulle implementeras var klar färdigställdes programmet.

4. Reflektion

Tack vare föregående laboration i denna kurs har vi blivit mer vana vid att programmera i Prolog, vilket gjorde arbetet under denna labb gick smidigare. Vi har också blivit mer vana med GitHub, och detta i kombination med att vi lärt känna varandra bättre ledde till att samarbetet fungerade bättre.

Vi hade ont om tid i slutet, men tack vare att arbetet gick snabbare och smidigare än förra gången var denna labb långt ifrån lika tidskrävande.

I föregående laboration användes labbrapporten som ett verktyg under arbetets gång, men på grund av den lättare arbetsbördan samt mindre tidsåtgång så behövdes det inte i detta projekt. I stället har kommunikationen skett via Discord, där vi bollat idéer angående implementeringen.

Appendix

A. Bevissystem

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

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

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

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

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

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

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

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

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

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

Figur 1: Ett bevissystem för CTL.

B. Kod

```
% Reads input
verify(Input) :-
        see(Input),
        read(V), read(L), read(S), read(F),
        seen,
        check(V, L, S, [], F), !.

%% Literals

% And
check(V, L, S, [], and(F,G)) :-
        check(V, L, S, [], F),
        check(V, L, S, [], G).

% Or
check(V, L, S, [], or(F,G)) :-
        check(V, L, S, [], F);
        check(V, L, S, [], G).

% AX
check(V, L, S, [], ax(F)) :-
        member([S, Transitions], V),
        check_all_states(V, L, Transitions, [], F).

% EX
check(V, L, S, [], ex(F)) :-
        member([S, Transitions], V),
        check_exist_state(V, L, Transitions, [], F).

% AG1, S is in U
check(_, _, S, U, ag(_)) :-
        member(S, U).

% AG2, S is NOT in U
check(V, L, S, U, ag(F)) :-
        \+ member(S, U),
        check(V, L, S, [], F),
        member([S, Transitions], V),
        check_all_states(V, L, Transitions, [S|U], ag(F)).
```

Figur 2: Kod

```
% EG1
check(_, _, S, U, eg(_)) :-
        member(S, U).

% EG2
check(V, L, S, U, eg(F)) :-
        \+ member(S, U),
        check(V, L, S, [], F),
        member([S, Transitions], V),
        check_exist_state(V, L, Transitions, [S|U], eg(F)).

% % EF1
check(V, L, S, U, ef(F)) :-
        \+ member(S, U),
        check(V, L, S, [], F).

% EF2
check(V, L, S, U, ef(F)) :-
        \+ member(S, U),
        member([S, NextStates], V),
        delete(NextStates, S, Transitions),
        check_exist_state(V, L, Transitions, [S|U], ef(F)).

% AF1
check(V, L, S, U, af(F)) :-
        \+ member(S, U),
        check(V, L, S, [], F).

% AF2
check(V, L, S, U, af(F)) :-
        \+ member(S, U),
        member([S, Transitions], V),
        check_all_states(V, L, Transitions, [S|U], af(F)).

% p
check(_, L, S, [], X) :-
        member([S, Ls], L),
        member(X, Ls).

% neg p
check(_, L, S, [], neg(X)) :-
        member([S, Ls], L),
        \+member(X, Ls).
```

Figur 3: Kod

```
%% check_all_states calls check witch all states in in first argument.
% Fact
check_all_states(_, _, [], _, _).

% Handles cases when U is not empty.
check_all_states(V, L, [H|T], U, X) :-
        check(V, L, H, U, X),
        check_all_states(V, L, T, [H|U], X).

% Handles cases when U is empty.
check_all_states (V, L, [H|T], [], X) :-
        check(V, L, H, [], X),
        check_all_states(V, L, T, [], X).

%% check_exist_state gives true if one call to check gives true.
% Fact
check_exist_state(_, _, [], _, _) :- fail.

% Handles cases when U is not empty.
check_exist_state(V, L, [H|T], U, X) :-
        check(V, L, H, U, X);
        check_exist_state(V, L, T, [H|U], X).

% Handles cases when U is empty.
check_exist_state(V, L, [H|T], [], X) :-
        check(V, L, H, [], X);
        check_exist_state(V, L, T, [], X).
```

Figur 4: Kod

C. Exempelbevis

För att testa Prologprogrammet som skrivits skapades en modell baserad på hur ett system för smart belysning i ett hem skulle kunna se ut. Man vill då att respektive lampa ska vara tänd om någon befinner sig i ett visst rum, men inte annars.

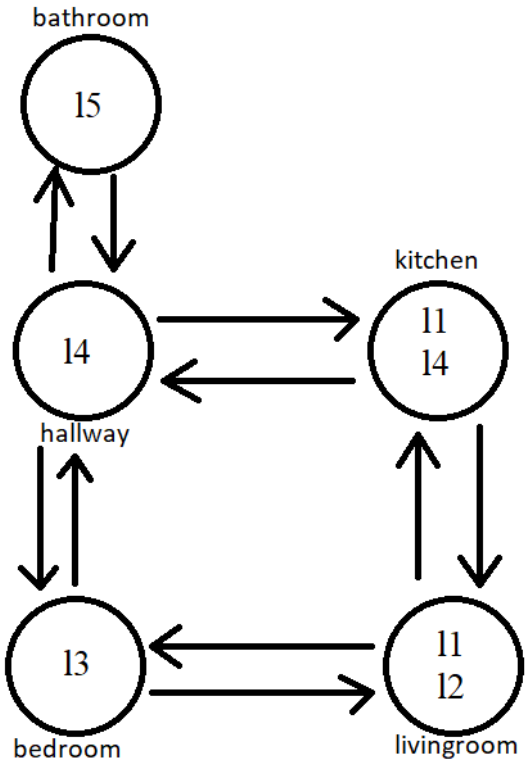

Fig. 5: Beskrivning

Den modell som skapades utgår ifrån bilden i Fig. 5 ovan. I varje rum finns en eller flera lampor, benämnda l1 - l5. Respektive lampa är tänd om dess

Två exempelfiler baserade på modellen i Fig. 5 skapades för att testa Prologprogrammet, och finns angivna nedan:

```
%% our-valid.txt %%

[[kitchen, [hallway, livingroom]],
 [bedroom, [livingroom, hallway]],
 [hallway, [bathroom, kitchen, bedroom]],
 [bathroom, [hallway]],
 [livingroom, [bedroom, kitchen]]].

[[kitchen, [l1, l4]],
 [bedroom, [l3]],
 [bathroom, [l5]],
 [livingroom, [l1, l2]],
 [hallway, [l4]]].

livingroom.

or(ax(neg(l5)), ex(ex(l5))).
```

```
%% our-invalid.txt %%

[[kitchen, [hallway, livingroom]],
 [bedroom, [livingroom, hallway]],
 [hallway, [bathroom, kitchen, bedroom]],
 [bathroom, [hallway]],
 [livingroom, [bedroom, kitchen]]].
[[kitchen, [l1, l4]],
 [bedroom, [l3]],
 [bathroom, [l5]],
 [livingroom, [l1, l2]],
 [hallway, [l4]]].

bathroom.

ax(ag(neg(l3))).
```

Figur 5: Exempelbevis