Labb 3 - CTL

Mehrdad Vafaee och Naveed Rahman - CINTE

2023-12-07

Introduktion

Denna rapport beskriver uppbyggnaden av en modellprovning i prolog för temporallogikens, CTL, regler och även ett exempel på en icke-trivial modell av en kaffemaskin. Syftet med en modellprovare är att kunna kontrollera om en temporallogisk formel vid ett specifikt tillstånd av den modellen i fråga är valid eller icke-valid med hjälp av de definierade reglerna för CTL.

Beviskontroll

Algoritmen för detta program sker på följande sätt; den får in data som innehåller tillstånden och dess grannar, varje tillstånds atommängder, nuvarande tillstånd, och CTL formeln som skall kontrolleras. Denna data skickas vidare senare mot predikatet *check*, som då ska kontrollera och verifiera om den givna CTL-formeln är valid eller icke-valid. Den givna CTL-formeln, följer den specifika CTL-reglerna som på appendix C, på så sätt går det att identifiera vilken regel som skall verifieras. Verifieringen sker med respektive krav som måste uppfyllas för den specifika regeln.
För hindring av oändlig loop vid exekvering, som kan ske vid övergångar där ett tillstånd övergår till sig själv, används en tom lista som sparar de tillstånd som kontrollen redan har varit hos. Vi använder oss av rekursivitet för att kontrollera alla premisser och få med listan med kontrollerade tillstånd.

Predikat	**Sanningskriterie**
check(-, L, S, [], X)	X är sann i tillståndet S.
check(-, L, S, [], neg(X))	X är inte sann i tillståndet S.
check(T, L, S, [], and(X,Y))	X och Y är sanna.
check(T, L, S, [], or(X,Y))	X eller Y är sanna.
check(T, L, S, [], ax(X))	X är sann i alla grannar av S.
check(T, L, S, [], ex(X))	X är sann i någon granne av S.
check(-, -, S, U, ag(-))	S finns i listan U.
check(T, L, S, U, ag(X))	S finns inte i listan U, X är sann i tillståndet S, X är sann i alla tillstånd man kan nå från S.
check(-, -, S, U, eg(-))	S finns i listan U.
check(T, L, S, U, eg(X))	S finns inte i listan U, X är sann i tillståndet S, X är sann i någon tillstånd man kan nå från S.
check(T, L, S, U, af(X))	S finns inte i listan U, X gäller i tillståndet S.
check(T, L, S, U, af(X))	S finns inte i listan U, af(x) gäller i alla tillstånd man kan nå från S.
check(T, L, S, U, ef(X))	S finns inte i listan U, X gäller i tillståndet S.
check(T, L, S, U, ef(X))	S finns inte i listan U, ef(x) gäller i någon tillstånd man kan nå från S.
checkA(-, -, [], -, -).	-
checkA(T, L, [S—Tail], U, X)	X är sann i tillstånd X, X är sann i alla tillstånd man kan nå från X.
checkE(T, L, [S—Tail], U, X)	X är sann i tillstånd X, X är sann i någon tillstånd man kan nå från X.

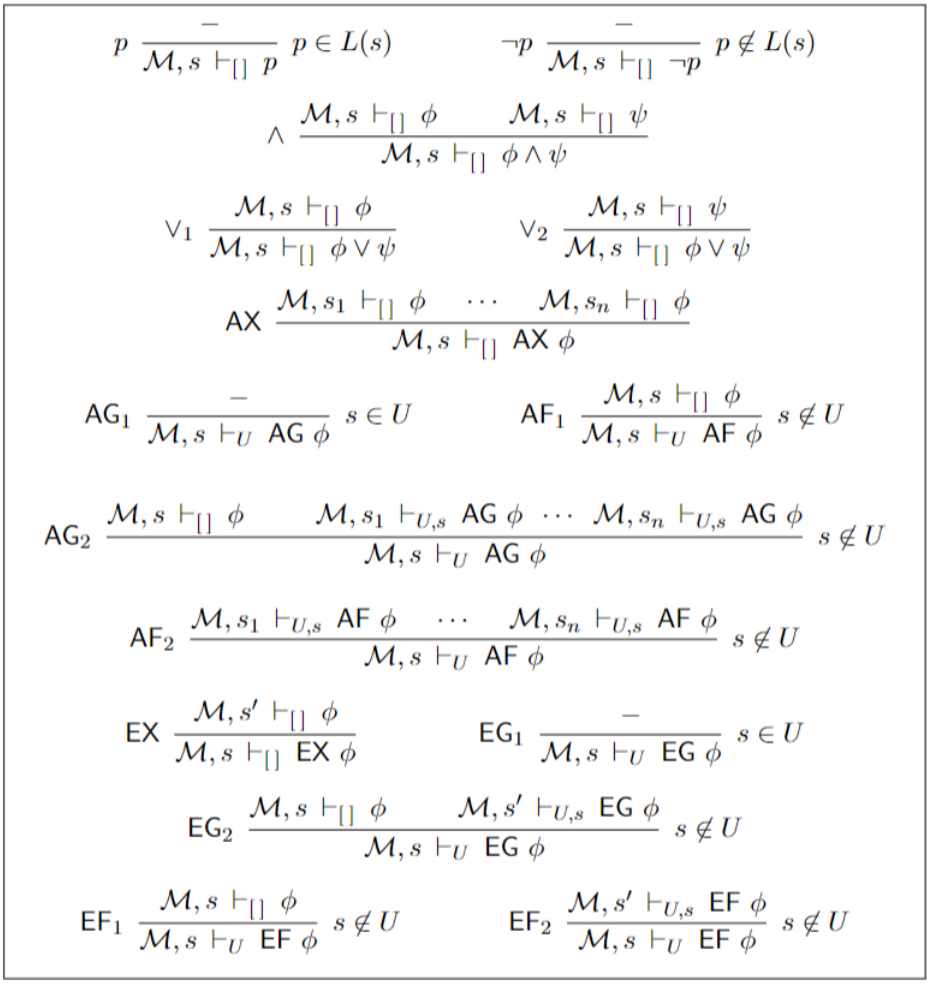

Figure 1: Ett bevissystem för CTL.

Modellering

Vår modell beskriver en variant av kaffemaskin, där slutprodukten självfallet ska bli kaffe. Modellens start sker alltid vid start av maskinen, som innehåller en tom mängd atomer. Efter första tillstånd kan övergången ske till två olika tillstånd; *full eller empty*, där den vägen går mot fylld den innehåller atomen *water*, det vill säga om maskinen har vatten. Om maskinen inte innehåller vatten går vägen mot tillståndet *empty* med atommängden *noWater*. Från tillståndet *empty*, behöver kaffemaskinen då fyllas på med vatten men förens man har gjort det kommer tillstånd vara kvar på samma tillstånd. Efter att det finns vatten i maskinen, då blir nästa steg att sätta in kapsylen med kaffet, som är tillståndet *ready* med atomerna *capsule och water*. När man har lagt in kapsylen finns det ett steg kvar som är knapptryckningen, då hamnar vi på tillståndet *brew*, som innehåller atomerna *button, capsule och water*. Efter detta tillstånd är det klart och hamnar på tillståndet med slutprodukten; *done* och atomen *coffee*.

Modellen och semantiken

M = (S, →, L)

S = {start, empty, full, ready, brew, done}.

→ = {start, empty), (start, full),(empty, empty), (empty, full), (full, full), (full, ready), (ready, ready), (ready, brew), (brew, done), (done, start)}

L = {start: {}, empty: {noWater}, full: {water}, ready: {water, capsule}, brew: {water, capsule, button}, done: {coffee}}

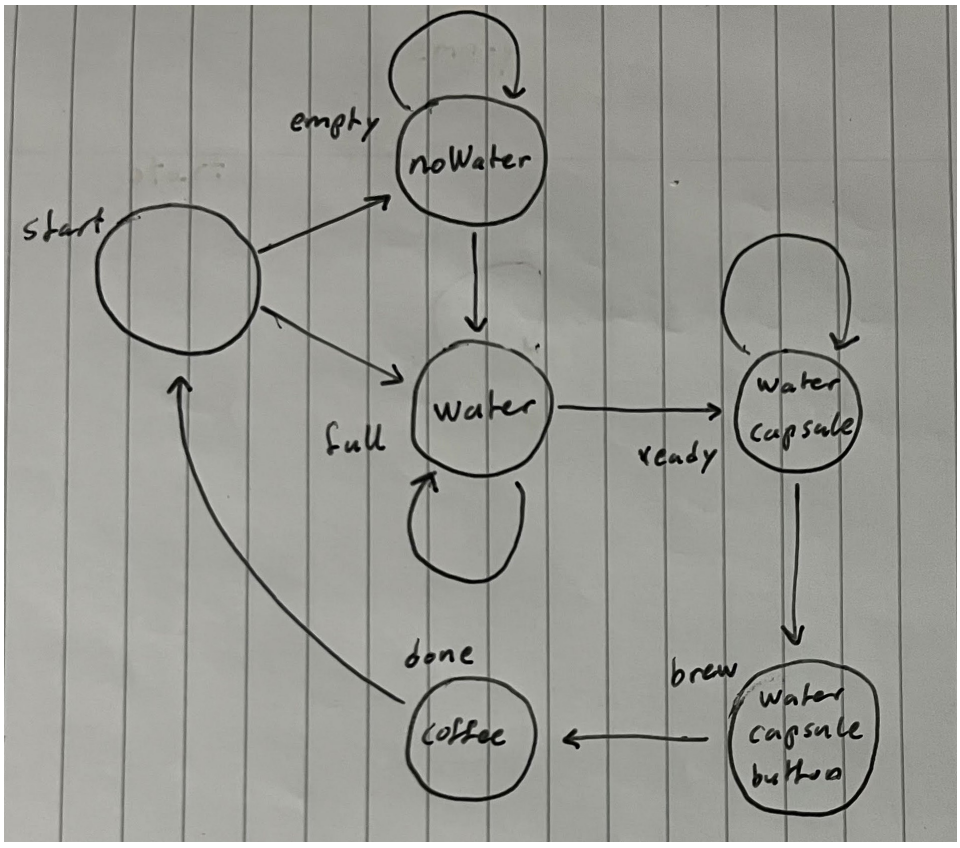

Figure 2: Modell av en kaffemaskin.

Diskussion

Figur 3 visar bokens definition av CTL formlerna som är betydligt mer komplett i jämförelse med vår implementation. Bokens definition innehåller t.ex. implikation och motsägelse som vår definition inte täcker. Modellprovaren behöver utökas med flera regler för att täcka bokens definition.

Definition 3.12 We define CTL formulas inductively via a Backus Naur form as done for LTL:

$$\phi ::= \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid \mathrm{AX}\,\phi \mid \mathrm{EX}\,\phi \mid \mathrm{AF}\,\phi \mid \mathrm{EF}\,\phi \mid \mathrm{AG}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{A}[\phi\ \mathrm{U}\ \phi] \mid \mathrm{E}[\phi\ \mathrm{U}\ \phi]$$

where p ranges over a set of atomic formulas.

Figure 3: Kursbokens definition av CTL.

Vårt program har klarat av alla tester. Vi tror det som begränsar hur stora modeller och formler den klarar av att verifiera, är vår definition av CTL och minne samt tid tillgänglig för programmet.

A Programkod

```
verify(Input) :-
    see(Input),
    read(T), % T - The transitions in form of adjacency lists
    read(L), % L - The labeling
    read(S), % S - Current state
    read(F), % F - CTL Formula to check
    seen,
    check(T, L, S, [], F),!.

% X true for all
checkA(_, _, [], _, _).
checkA(T, L, [S|Tail], U, X) :-
    check(T, L, S, U, X),
    checkA(T, L, Tail, U, X).

% X true for some
checkE(T, L, [S|Tail], U, X) :-
        check(T, L, S, U, X);
        checkE(T, L, Tail, U, X).

% P
check(_, L, S, [], X) :-
    member([S,Tail], L),
    member(X, Tail).

% Neg(P)
check(_, L, S, [], neg(X)) :-
    \+check(_, L, S, [], X).

% And
check(T, L, S, [], and(X,Y)) :-
    check(T, L, S, [], X),
    check(T, L, S, [], Y).

% Or
check(T, L, S, [], or(X,Y)) :-
    check(T, L, S, [], X);
    check(T, L, S, [], Y).

% AX - För alla nästa tillstånd gäller X
check(T, L, S, [], ax(X)) :-
    member([S,Tail], T),
```

```
    checkA(T, L, Tail, [], X).

% EX - För något nästa tillstånd gäller X
check(T, L, S, [], ex(X)) :-
    member([S,Tail], T),
    checkE(T, L, Tail, [], X).

% AG1 - X atomen gäller för alla vägar
check(_, _, S, U, ag(_)) :- member(S, U).
% AG2
check(T, L, S, U, ag(X)) :-
    \+member(S, U),
    check(T, L, S, [], X),
    member([S, Tail], T),
        checkA(T, L, Tail, [S|U], ag(X)).

% EG1 - Det finns en väg där X atomen alltid gäller
check(_, _, S, U, eg(_)) :- member(S, U).
% EG2
check(T, L, S, U, eg(X)) :-
    \+member(S, U),
    check(T, L, S, [], X),
    member([S, Tail], T),
        checkE(T, L, Tail, [S|U], eg(X)).

% AF1 - På alla vägar gäller X atomen någongång
check(T, L, S, U, af(X)) :-
    \+member(S, U),
    check(T, L, S, [], X).
% AF2
check(T, L, S, U, af(X)) :-
        \+member(S, U),
        member([S, Tail], T),
        checkA(T, L, Tail, [S|U], af(X)).

% EF1 - Vid någon väg gäller X atomen någongång
check(T, L, S, U, ef(X)) :-
    \+member(S, U),
    check(T, L, S, [], X).
% EF2
check(T, L, S, U, ef(X)) :-
    \+member(S, U),
    member([S, Tail], T),
    checkE(T, L, Tail, [S|U], ef(X)).
```

B Modell

```
% States = {start, empty, full, ready, brew, done}
% Atoms = {noWater, water, capsule, button, coffee}

% Adjacency lists
[[start, [empty, full]],
[empty, [empty, full]],
[full, [full, ready]],
[ready, [ready, brew]],
[brew, [done]],
[done, [start]]].

% Labeling
[[start, []],
[empty, [noWater]],
[full, [water]],
[ready, [water, capsule]],
[brew, [water, capsule, button]],
[done, [coffee]]].

%current state
start.

% valid
ef(and(coffee, neg(and(capsule, and(button, water))))).

% invalid
ef(and(button, noWater))
```