Lab-3
CTL

Tor Arvill
arvill@kth.se

Mateo Olofsson
mateoof@kth.se

11 december 2020

Contents

1 CTL Model Checker

1.1 General approach

Our model checker will evaluate all the rules as defined in the lab by checking if F holds true in the current state. The model checker uses pattern matching to identify the different rules defined in the program. As we have chosen to write all the rules inside the predicate "check", it will recursively find all the rules defined in the rule F, by decomposing F. Every rule uses its own version of the check-predicate to evaluate if F holds true in the current rule, if so, the program will return true and indicate that the formula that has been inputted holds true for S=phi. In most cases a helper predicate handles the internal recursion, for example ensuring that AG holds true for all subsequent states. Some simple functions used throughout were put as separate predicates, such as translateLabels\3 that converts a state to a list of all its linked states and binding the result to an empty variable. The code for the model-checker can be found in Appendix 2.

1.2 Predicates

- **check(_T,L,S,_U,neg(F))**: The predicate is true when F does not exist in the current state (S), else the predicate will return false. It represents the rule neg.
- **check(T, L, S, U, and(F,G))**: The predicate is true when both F and G is true, else the predicate will return false. It represents the rule AND.
- **check(T, L, S, U, or(F,_)) & check(T, L, S, U, or(_,G))**: The first predicate will return true when F is true, else it will return false and the second predicate will return true when G is true, else it will return false. It represents the rule OR.
- **check(T,L,S,U,ax(F))**: The predicate will return true when the states following S holds true for F, else it will return false. It represents the rule AX.
- **check(T,L,S,U,ex(F))**: The predicate will return true when at least one of the states following S holds true for F, else it will return false.It represents the rule EX.

- **check(T,L,S,U,af(F))**: The predicate will return true if S holds true for F, but also when all paths reachable from S eventually reaches a state that holds true for F.It represents the rule AF.
- **check(T,L,S,U,ef(F))**: The predicate will return true if S holds true for F, but also when at least one path reachable from S eventually reaches a state that holds true for F. It represents the rule EF.
- **check(T,L,S,U,ag(F))**: The predicate will return true if S holds true for F and all states reachable from S hold true for F. It represents the rule AG.
- **check(T,L,S,U,eg(F))**: The predicate will return true if S holds true for F and that at least one path where all states reachable from S holds true for S. It represents the rule EG.
- **check(_,L,S,_,F)**: The predicate will return true when the current State (S) holds true for F and that F is an atom. It represents the rule P.
- **ax_checker(T,L,[HLinks|TLinks],U,F)**: This helper predicate helps AX to determine if all reachable states hold true for F.
- **ef_checker(T,L,[HLinks|TLinks],U,F)**: This helper predicate helps EF to determine if at least one path reachable from S eventually reaches a state that holds true for F.
- **eg_checker(_T,_L,[HLinks|_TLinks],U,_F)**: This helper predicate helps EG to determine if S holds for F and that at least one path where all states holds true for F.
- **af_checker(T,L,[HLinks|TLinks],U,F)**: This helper predicate helps AF to determine if S holds true for F and if all paths reachable from S eventually reaches a state that holds true for F.
- **ex_checker(T,L,Links,U,F)**: This helper predicate checks if the current state (S) have any reachable states and if so it calls the helper predicate, ex_checker_sub\5 else it fails.
- **ex_checker_sub(T,L,[HLinks|TLinks],U,F)**: This helper predicate helps EX determine if at least one of the states following S holds true for F.

- **f_checker(F)**: This helper predicate evaluates if F is an atom, by returning true if a rule is matched else it returns false and confirms that F is an atom.
- **translateLinks(T,Sn,Links)**: This helper predicate takes the current state (Sn) and returns the states that are reachable from S.
- **translateLabel(L,Sn,Atom)**: This helper predicate takes the current state(Sn) and returns the atom present in the state.

1.3 Testing

Throughout the design-process we used the included test-suite extensively and also used modified versions of these tests to isolate the problems. We also used a small set of tests of our own that were intended to be as simple as possible to ensure that the predicates behaved correctly in the most basic scenario. These tests were designed to target one predicate each. Our model of a turnstile was instead tested through our program as we at this point were fairly confident in its correctness.

2 CTL Model

2.1 Our model

Our model describes a classic subway turnstile, where a card is blipped before passage is allowed. The basic idea is that it defaults to a “Ready” state from which a card can be scanned to allow passage. The process goes as follows; the turnstile is “Ready” (s0), a card is scanned (s1), a request to some server (ticket/subway-card validation) is sent (s2) after which the result is returned (s4), the gate is then opened (s3) before eventually returning to ready(s0). The gate is “locked” in all states except open(s3). Note that if for example the request can’t be sent or a payment failed it will go to a fail-state (s5), which returns the machine to “Ready” (s0). The atoms we have chosen to include are locked (when “locked” holds, the gate is closed), payment processed (a successful request has been received, note that this only pertains to the processing and is not dependent on if the payment went through.) and various modes for the LEDs and speaker. The gate being open is represented by the lack of the atom “locked”.

2.2 Figure 1

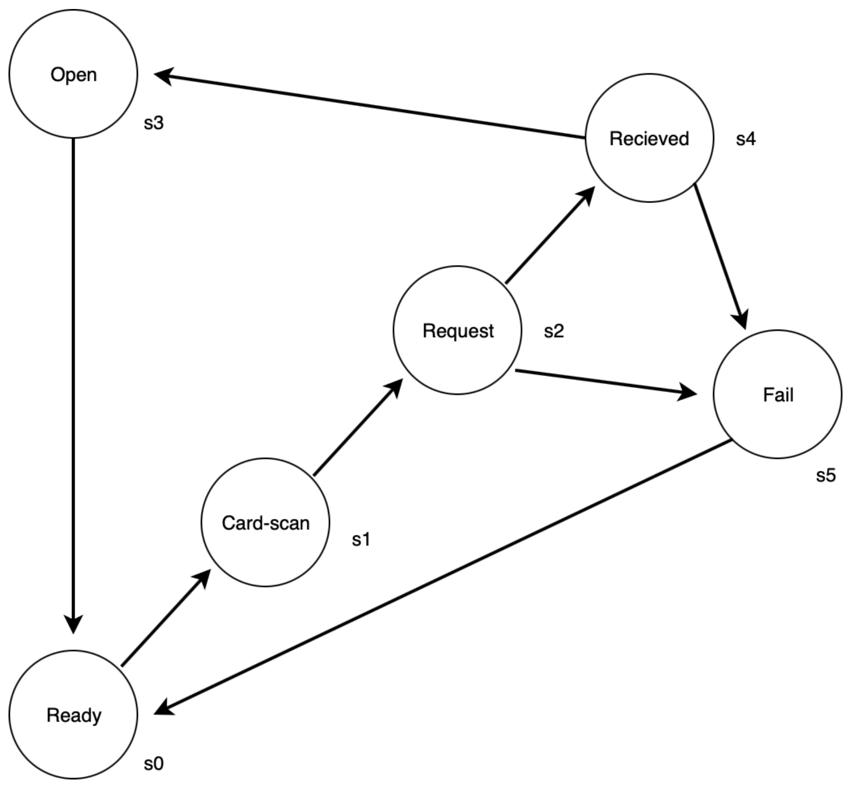
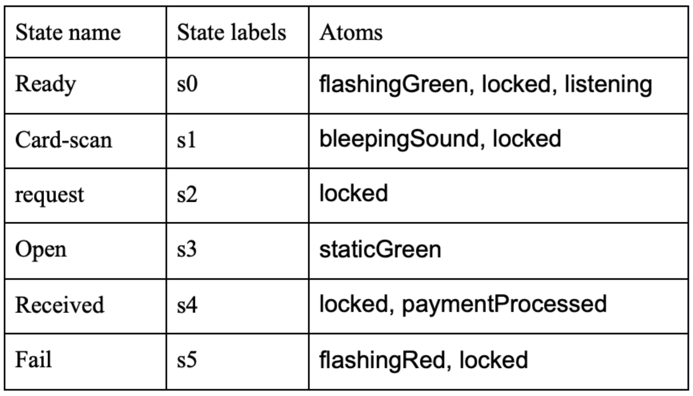


Figure 1: *shows relations between states in the turnstile-model.*

2.3 Table 1



State name	State labels	Atoms
Ready	s0	flashingGreen, locked, listening
Card-scan	s1	bleepingSound, locked
request	s2	locked
Open	s3	staticGreen
Received	s4	locked, paymentProcessed
Fail	s5	flashingRed, locked

Figure 2: *displays the atoms that hold true in each state.*

2.4 System attributes

The two main functions of the system we have chosen to describe are that the system will always be able to open and return to ready and that there is no way in which the turnstile is opened without processed payment. The first aspect is described more formally as follows:

```
s0, AG(AND(EF(NEG(locked)),EF(listening))).
```

What we mean is that there is no such state that the gate cannot be eventually opened and that there is no such state that it "locks up" and does not eventually accept inputs again. Without looking at the actual meaning we say that for every state reachable from s0 there is always a way to reach a state of neg(locked) (meaning open) and there is also always a way to transition to a state where listening holds true (coincidentally s0). In the model used the choice of starting state is arbitrary for this expression as long as it's among the defined states.

The second aspect we wanted to describe was the fact that there is no way to transition to neg(locked) (meaning open) from any other state without processing payment. This is described as an expression that seeks to prove a way for this to exist and failing. What we attempt to prove is that all states in the system either only link to locked states or if it links to unlocked (no locked atom), then it needs to not have "paymentProcessed". There is admittedly a naivety to our approach in that we assume that the "paymentProcessed" atom is adjacent to an unlocked state. This is however true in our model and the rule is therefore sound.

We define the rule as for all states reachable from s0 (ready) is one of three options: adjacent to at least one unlocked state and without "paymentProcessed", locked and adjacent to only locked states or unlocked itself. The NEG(locked) option is not strictly necessary, but is present to display that this only strictly applies about locked states and its holding true for the unlocked state is mere happenstance.

```
s0, AG(OR(OR(AND(EX(NEG(locked)),NEG(paymentProcessed)),
AND(locked,AX(locked))),NEG(locked))).
```

Both formulas and the model can be found in a prolog-friendly format in Appendix 1.

3 Questions

3.1 Lab-CTL vs Book-CTL

We can see that the book's CTL handles more rules, such as union (A and E versions), implication or strictly false (floor) or true, which the lab does not include. To make use of these we would have to implement the rules in our model checker. Also while not intentional, our implementation might be slightly more naive than the complete version described in the book due to the risk of human error in an actual implementation.

3.2 Varying number of premises

Our model checker handles the different number of variables in the premises by using a help predicate "translateLinks\3" that can identify the states transitioning from S by assigning the list of transitioning states to a variable called "Links ", before calling the help predicate to the specific rule for example, ax_checker\5. This enables the ax_checker\5 to handle the list of states and recursively look at each state once by calling the head of the list and returning the tail. Multiple atoms is not a problem either as we always handle the atoms of a state as a list with member\2

3.3 Maximum model size

Our implementation does not have an exact max-size and in theory it can work on humongous models. However, as rules such as AG require a lot of recursion and branching, the complexity of the operation can get out of hand especially with nested rules.

4 Apendix 1

Prolog compatible versions of model and formulas

4.0.1 Valid formula

```
[[s0,[s1]],
  [s1,[s2]],
  [s2,[s4,s5]],
  [s3,[s0]],
  [s4,[s3,s5]],
  [s5,[s0]]].
```

```
 [[s0,[flashingGreen,locked,listening]],
  [s1,[bleepingSound,locked]],
  [s2,[locked]],
  [s3,[staticGreen]],
  [s4,[locked,paymentProcessed]],
  [s5,[flashingRed,locked]]].

 s0.

 ag(and(ef(neg(locked)),ef(listening))).
```

4.0.2 Invalid formula

```
[[s0,[s1]],
  [s1,[s2]],
  [s2,[s4,s5]],
  [s3,[s0]],
  [s4,[s3,s5]],
  [s5,[s0]]].

 [[s0,[flashingGreen,locked,listening]],
  [s1,[bleepingSound,locked]],
  [s2,[locked]],
  [s3,[staticGreen]],
  [s4,[locked,paymentProcessed]],
  [s5,[flashingRed,locked]]].

 s0.

ag(or(or(and(ex(neg(locked)),neg(paymentProcessed)),
and(locked,ax(locked))),neg(locked))).
```

5 Apendix 2

The code for the model checker

```
% LAB-3 Tor Arvill och Mateo Olofsson.
% version 2020-12-11
% including scraps.
verify(Input) :-
see(Input), read(T), read(L), read(S), read(F), seen,
check(T, L, S, [], F).


verify(Input) :-
see(Input), read(T), read(L), read(S), read(F), seen,
check(T, L, S, [], F).

%neg
check(_T,L,S,_U,neg(F)):-
  translateLabel(L,S,Atom), not(member(F, Atom)).

% And
check(T, L, S, U, and(F,G)) :-
check(T, L, S, U, F),
check(T, L, S, U, G),!.

% Or
check(T, L, S, U, or(F,_)):-
check(T, L, S, U, F).

check(T, L, S, U, or(_,G)):-
check(T, L, S, U, G),!.
% AX
check(T,L,S,U,ax(F)):-translateLinks(T,S,Links), ax_checker(T,L,Links,U,F).

% EX
check(T,L,S,U,ex(F)):-
  translateLinks(T,S,Links),
  ex_checker(T,L,Links,U,F),!.

%AF
```

```
check(T,L,S,U,af(F)):- not(member(S,U)), translateLinks(T,S,Links),
af_checker(T,L,Links,[S|U],F), !.
check(T,L,S,U,af(F)):- not(member(S,U)),check(T,L,S,U,F),!.

%EF
check(T,L,S,U,ef(F)):- not(member(S,U)), translateLinks(T,S,Links),
\+ef_checker(T,L,[S|Links],U,F).
check(T,L,S,U,ef(F)):-not(member(S,U)), check(T,L,S,U,F).
%AG
check(T,L,S,U,ag(F)):-
  \+member(S,U), translateLinks(T,S,Links), ag_checker(T,L,Links,[S|U],F),
  check(T,L,S,[],F),!.
check(_T,_L,S,U,ag(_F)):- member(S,U).

%EG
check(T,L,S,U,eg(F)):-
  (member(S,U); (\+member(S,U), translateLinks(T,S,Links),
  eg_checker(T,L,Links,U,F),check(T,L,S,U,F))),!.

%p
check(_,L,S,_,F):- f_checker(F), translateLabel(L,S,Atom),member(F,Atom),!.

%helper to Ax.
ax_checker(_,_,[],_,_).
ax_checker(T,L,[HLinks|TLinks],U,F):- check(T,L,HLinks,U,F),
ax_checker(T,L,TLinks,U,F).

%helper to Ex.
ex_checker(T,L,Links,U,F):- (Links=[]; (\+Links=[],
ex_checker_sub(T,L,Links,U,F))).
ex_checker_sub(_,_,[],_,_):-!,false.
ex_checker_sub(T,L,[HLinks|TLinks],U,F):-!, (check(T,L,HLinks,U,F);
(\+check(T,L,HLinks,U,F), ex_checker_sub(T,L,TLinks,U,F))).

%helper to AG.
ag_checker(_,_,[],_,_).
ag_checker(T,L,[HLinks|TLinks],U,F):- check(T,L,HLinks,U,ag(F)),
ag_checker(T,L,TLinks,[HLinks|U],F).

%helper to EG.
eg_checker(_,_,[],_,_):-false.
```

```
%eg_checker(T,L,[HLinks|TLinks],U,F):- \+[HLinks|TLinks]=[],
translateLinks(T,HLinks,NextLinks),member(HLinks,NextLinks),
check(T,L,HLinks,U,F).
eg_checker(_T,_L,[HLinks|_TLinks],U,_F):- member(HLinks,U).
eg_checker(T,L,[HLinks|TLinks],U,F):- not(member(HLinks,U)),
translateLinks(T,HLinks,NextLinks),
(check(T,L,HLinks,U,F), eg_checker(T,L,NextLinks,[HLinks|U],F));
eg_checker(T,L,TLinks,U,F).



%helpers to AF
af_checker(_,_,[],_,_):-false.
af_checker(T,L,[HLinks|TLinks],U,F):- member(HLinks,U),
af_checker(T,L,TLinks,U,F).
af_checker(T,L,[HLinks|TLinks],U,F):- not(member(HLinks,U)),
    translateLinks(T,HLinks,NextLinks),!,
    (check(T,L,HLinks,U,F);af_checker(T,L,NextLinks,[HLinks|U],F)),
    !,(TLinks=[];af_checker(T,L,TLinks,[HLinks|U],F)),!.

%helpers to EF

  ef_checker(_,_,[],_,_).
  ef_checker(T,L,[HLinks|TLinks],U,F):- member(HLinks,U),
  ef_checker(T,L,TLinks,U,F).
  ef_checker(T,L,[HLinks|TLinks],U,F):- not(member(HLinks,U)),
    translateLinks(T,HLinks,NextLinks),
    \+check(T,L,HLinks,[],F),ef_checker(T,L,NextLinks,[HLinks|U],F),
    ef_checker(T,L,TLinks,[HLinks|U],F).

%General helpers
translateLinks(T,Sn,Links):-
    member([Sn|[FoundLinks]],T),
    Links=FoundLinks,!.

translateLabel(L,Sn,Atom):-
    member([Sn|[FoundAtom]],L),
    Atom=FoundAtom,!.

f_checker(F):- \+F=and(_), \+F=neg(_), \+F=or(_), \+F=ax(_), \+F=ex(_),
\+F=ag(_), \+F=eg(_), \+F=ef(_), \+F=af(_).
```