Laboration Report

Logic for Computer Scientists, DD1351
Laboration 3

2024-12-04

Group members:
Malte Berg, maltebe@kth.se
Alexander Timsäter, timsater@kth.se

1. Introduction

In this laboration a CTL model checker is implemented using SWI Prolog which is able to tell if a given model is valid or not. The model is given in the format specified below:

```
[[s0, [s2, s0]],
 [s1, [s0, s1]],
 [s2, [s1]]].

[[s0, []],
 [s1, []],
 [s2, [p]]].

s1.

af(ax(neg(r))).
```

The different sections of the file specifies the adjacencies of all given nodes, the second the labeling of all states, thirdly the initial state and finally the CTL formula to be checked. In order to accomplish this predicates checking validity of all the elementary and some of the derived operators are implemented. These are:

1.1. Sub-Questions

The laboration is split into multiple different sub questions:

Operator	Description
A	”For all paths” (Universal quantifier over paths)
E	”There exists a path” (Existential quantifier over paths)
G	”Globally” (Always, at all points along the path)
F	”Finally” (Eventually, at some point in the future along the path)
X	”Next” (In the next state along the path)
$\neg$	”Negation” (Not, opposite of the statement)
$\wedge$	”Conjunction” (And, both statements must hold)
$\vee$	”Disjunction” (Or, at least one statement must hold)

Table 1: Basic CTL Operators and Their Meanings

1. Create a Model Checker in Prolog which is able to test the validity of a given model formatted as shown in the introduction using the operators shown in table 1.
2. Design a non trivial system relevant to computer science, create a state diagram explaining the behaviour with at least five states. Create a Prolog compatible text file and test the model.
3. Write two different non trivial system properties, one valid and one invalid in a Prolog compatible text file and test the two properties using the designed Model Checker.

2. Literature Study

The live and pre recorded lectures given by Johan Karlander and Thomas Sjöland were studied in order to grasp the overlying concepts of CTL, LTL and model checking. Chapter 3 in 'Logic in Computer Science' by Michael Huth and Mark Ryan was also extensively studied for this purpose. The official SWI Prolog documentation was used get detailed information of the specifics of the built in predicates.

3. Method

3.1. Sub-Questions

The model describes an automatic locked door which can be opened either by a button or the manual handle however the button is out of reach from the manual lock. Figure 1 below shows the intended logic of the the door.

When a person approaches the door it is of course locked and closed in most cases. This is represented with the state **cl**. In **cl**, the user has the option to either press the button, which automatically opens the door and ends up in state **auto**, or physically walk up and unlock the door, since you need to unlock the door before you can open it, and reach state **cu**. From **cu** you can either lock the door again, **cl**, or open the door,

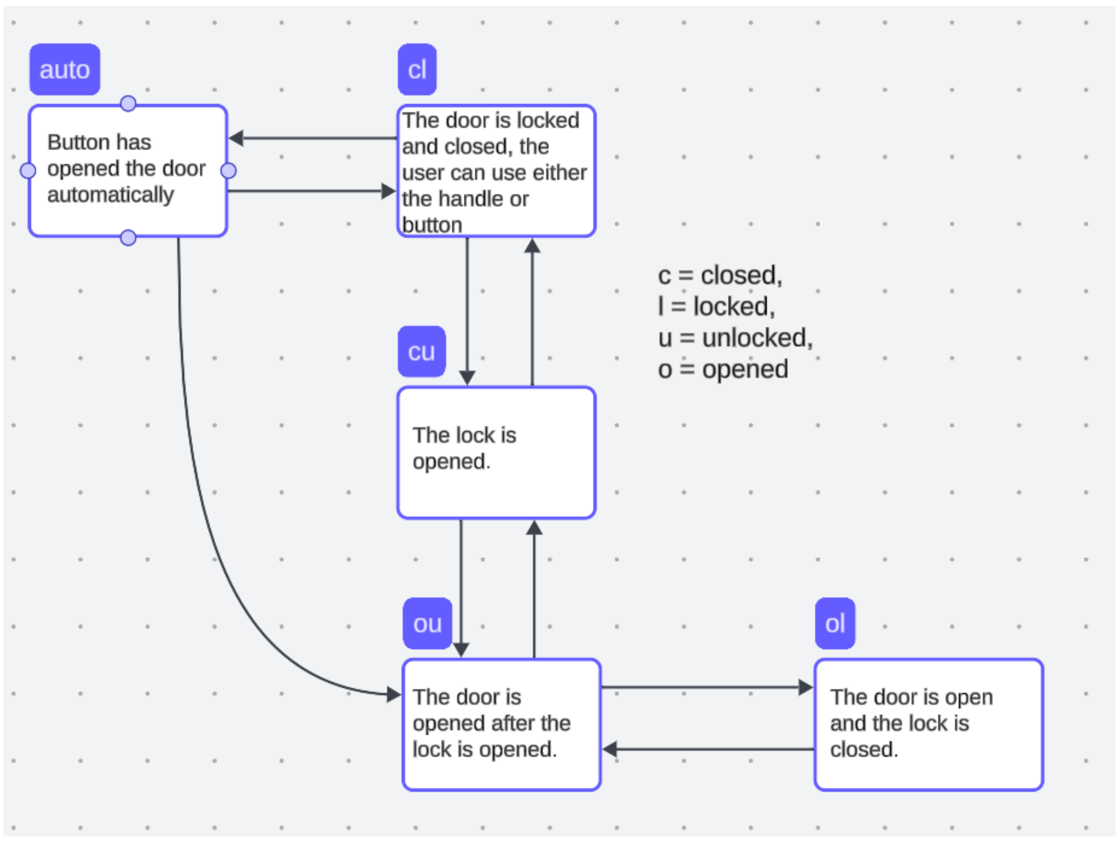
auto
Button has opened the door automatically
cl
The door is locked and closed, the user can use either the handle or button
cu
The lock is opened.
ou
The door is opened after the lock is opened.
ol
The door is open and the lock is closed.
c = closed,
l = locked,
u = unlocked,
o = opened

Figure 1: Diagram of the states and transitions of the automatic door

ou. When the door is opened you can lock the door **ol**, however you are not able to close it until you open the lock, **ou**. If you open the door using the automatic button the door unlocks and opens, **ou**. This is then translated into the format specified by the lab instructions, as a list of lists with all of the states' transitions. The atomic properties are also given, **p** = the door is open, **q** = the door is locked.

```
% c = Closed
% l = Locked
% u = Unlocked
% o = Opened
% auto = Automatic Button

[[cl, [auto, cu]],
 [cu, [cl, ou]],
 [ou, [cu, ol]],
 [ol, [ou]],
 [auto, [cl, ou]]].

[[cl, [q]],
 [auto, [q]],
 [cu, []],
 [ou, [p]],
 [ol, [p, q]]].

cl.
```

The final task was to create two system properties, one valid and one invalid and then test these using the model checker.

The valid property: EF (q $\wedge$ AG (p)) which will be formatted as:

```
ef(and(q,ag(p))).
```

The invalid property: AG (AF p $\wedge$ AG q)

```
ag(and(af(p),af(q))).
```

These were then formatted appropriately and tested using the model checker.

4. Result

The prolog code used to solve the laboration can be found in **Appendix A.1**. All of the given tests as well as the author-designed tests were passed without issue.

5. Discussion

1. **What are the main differences between the in the lab CTL and the one described in the book?**
   - The lab CTL is missing handling of $\perp$, $\top$, **U** and the derivable **W** and **R**.
2. **How were variable number of premises handled? (E.g. the rule AF)**
   - For the **A** rules, multiple next states need to be checked and be true, compared to **E** rules which only requires one next state to be true. The way this was implemented for the **A** rules were to use `forall/2`, which only comes back true if all next states are also true for the current formula or rule being checked. The predicate is dynamic as it checks all possible items in the list provided without a specific number entered.
3. **What is the upper limit of the size of models that the constructed model checker can validate?**
   - Overall this question is quite difficult to answer, in theory the bottle neck would be the stack limit. This was accidentally reached a couple of times by mistake when writing the checker. Perhaps flaws in the design have caused this implementation to have a high time complexity which causes some models to be computationally unfeasible. From research some types of specifications are also determined to be more complex than others, such as specifications containing nested complex logic per example: AG (p $\vee$ q), EF (p $\wedge$ AG (r)) on a model with a large amount of transitions and labels.

A. Appendix

A.1. Code

```
% 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, X)
    % T - The transitions in form of adjacency lists
    % L - The labeling
    % S - Current state
    % U - Currently recorded states
    % X/F - CTL Formula to check.

% ------- Literals -------
check(_, L, S, [], X) :-
    member([S, Labels], L),
    member(X, Labels).

check(_, L, S, [], neg(X)) :-
    member([S, Labels], L),
    \+ member(X, Labels).

% ----------- And -----------
check(T, L, S, [], and(F,G)) :-
    check(T, L, S, [], F), %F is found in labels at current node
    check(T, L, S, [], G). %G -||-

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

% ----------- AF -----------
check(T, L, S, U, af(X)) :-
    \+ member(S, U), % Common rule for both AF1 and AF2
    (
        check(T, L, S, [], X) % AF1
        ;
        % AF2
        member([S, Subnodes], T),
        forall(member(Snew, Subnodes), check(T, L, Snew, [S|U], af(X)))
    ).
```

```
% ----------- EF -----------
check(T, L, S, U, ef(X)) :-
    \+ member(S, U), % Common rule for both EF1 and EF2
    (
        check(T, L, S, [], X) % EF1
        ;
        % EF2
        member([S, Subnodes], T),
        member(Snew, Subnodes),
        check(T, L, Snew, [S|U], ef(X))
    ).
% ----------- AX -----------
check(T, L, S, [], ax(X)) :-
    member([S, Subnodes], T),
    forall(member(Snew, Subnodes), check(T, L, Snew, [], X)).

% ----------- EX -----------
check(T, L, S, [], ex(X)) :-
    member([S, Subnodes], T),
    member(Snew, Subnodes),
    check(T, L, Snew, [], X).

% ----------- EG -----------
check(T, L, S, U, eg(X)) :-
    (
      member(S, U) % EG1
      ;
      ( % EG2
        \+ member(S, U),
        check(T, L, S, [], X),
        member([S, Subnodes], T),
        member(Snew, Subnodes),
        check(T, L, Snew, [S|U], eg(X))
      )
    ).

% ----------- AG -----------

check(T, L, S, U, ag(X)) :-
(
    member(S, U) % AG1
    ;
    ( % AG2
        check(T, L, S, [], X),
        \+ member(S, U),
        member([S, Subnodes], T),
        forall(member(Snew, Subnodes), check(T, L, Snew, [S|U], ag(X)))
    )
).
```

Predicate	**Condition (When True)**
verify/1	Reads an input file, if the file is formatted correctly and the model is correct
Literal	The current state has the formula as a label
Negation	The formula is false for the current state
And	True when both formulas are true in current state
Or	True when either of the two formulas are true in current state
AF	The formula is true in at least one future state in every branch
EF	The formula is true in this state or in at least one future state
AX	The formula is true in all of the next states
EX	The formula is true at least one of the next states
EG	The formula is true in this state and all other states along at least one path
AG	The formula is true in current state and all states reachable from this state

Table 2: Table of predicates and their truth conditions. Predicates are `check/5` unless otherwise specified.

A.2. Table of Predicates