CTL Model Checker

Logik för dataloger, DD1351

Ammar Alzeno

05/12 - 2023

Introduction

We will create a model checking algorithm, a formal verification technique that examines the states of a system to validate whether certain properties hold. In the domain of temporal logics, Computation Tree Logic (CTL) stands out for its expressive power to represent various temporal properties. This report will show a Prolog implementation of a CTL model checker, a tool designed to evaluate whether a given CTL formula holds in a specified model.

The core of this algorithm revolves around the interpretation of CTL operators such as EX (Exists Next), AX (For All Next), EG (Exists Globally), AG (For All Globally), EF (Exists Finally), and AF (For All Finally), alongside basic logical connectives like AND, OR, and NOT. The algorithm is designed to work with models expressed as sets of states with corresponding labeling and transitions in the form of adjacency lists. It recursively traverses these states and evaluates the CTL formulas against the model, thereby determining their validity.

Method

The Prolog CTL Model Checker will utilize a systematic approach to verify whether a Computation Tree Logic (CTL) formula is valid within a given model. The methodology is rooted in Prolog's inherent strengths in logical reasoning and recursive problem-solving. We will break down the method into the different steps of the algorithm.

Model Representation

The model is represented by a state space, where each state is associated with a set of atomic propositions (labels) that are true in that state. The transitions between states are represented as adjacency lists, showing the possible paths from one state to another.

Input Parsing

The verify(Input) predicate serves as the entry point. It reads the transition model (T), the labeling (L), the initial state (S), and the CTL formula (F) from the input file specified by Input.

Recursive Formula Evaluation

The check(T, L, S, U, F) predicate is the core of the model checker. It recursively evaluates the truth of the CTL formula F in the context of the current state S, the transition model T, and the labeling L. The list U is used to keep track of visited states, preventing infinite loops in the evaluation, especially relevant for global (G) and until (U) operators.

Handling CTL Operators

The simplest form of formula, a literal or its negation, is checked against the labels of the current state.

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

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

Logical Connectives (AND, OR): The model checker evaluates logical connectives by recursively applying the check predicate to each operand.

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

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

check(T, L, S, [], or(_, G)) :-
    check(T, L, S, [], G).
```

EX and AX (Existential/Universal Next): For EX, the model checker checks if there exists a next state where the formula holds. For AX, it checks if the formula holds for all next states.

```
% AX
check(T, L, S, U, ax(F)) :-
    findall(Next, (member([S, NextStates], T), member(Next, NextStates)),
    ↪   NextStates),
    forall(member(Next, NextStates), check(T, L, Next, U, F)).

% EX
check(T, L, S, [], ex(F)) :-
    member([S, NextStates], T),
    member(Next, NextStates),
    check(T, L, Next, [], F).
```

EG and AG (Existential/Universal Globally): These operators require checking if a formula holds on some/all paths starting from the current state globally. The model checker handles them with a combination of recursion and tracking visited states.

```
% AG
check(_, _, S, U, ag(_)) :- member(S, U).
check(T, L, S, U, ag(F)) :-
    \+ member(S, U),
    check(T, L, S, [], F),
    findall(Next, (member([S, NextStates], T), member(Next, NextStates)),
    ↪   NextStates),
    forall(member(Next, NextStates), check(T, L, Next, [S|U], ag(F))).

% EG
check(_, _, S, U, eg(_)) :-
    member(S, U).
check(T, L, S, U, eg(F)) :-
    \+ member(S, U),
    check(T, L, S, [], F),
    member([S, NextStates], T),
    member(Next, NextStates),
    check(T, L, Next, [S|U], eg(F)).
```

EF and AF (Existential/Universal Finally): EF checks if there exists a path where the formula eventually holds. AF checks if the formula holds eventually on all paths from the current state.

```
% AF
check(T, L, S, U, af(F)) :-

    \+ member(S, U),
    (
      check(T, L, S, [], F) ;
      member([S, NextStates], T),
      forall(member(Next, NextStates), check(T, L, Next, [S|U], af(F)))
    ).

% EF
check(T, L, S, U, ef(F)) :-
    \+ member(S, U),
    (
      check(T, L, S, [], F)
      ;
      findall(Next,
              (member([S, NextStates], T),
               member(Next, NextStates)),
              NextStates),
      .
      member(Next, NextStates),
      check(T, L, Next, [S|U], ef(F))
    ).
```

Path Exploration

findall and forall: These Prolog built-in predicates are used to generate and traverse all possible next states. findall gathers all next states, while forall is used to assert that a predicate holds for all elements in a list, crucial for handling universal quantifiers in CTL.

Result

With the provided skeleton code in the project document, and with the implementation of handling all the CTL operators, the CTL model checker was ready for testing. See Appendix B for the code. We will start with an example model that will be tested on the algorithm as well.

Test model

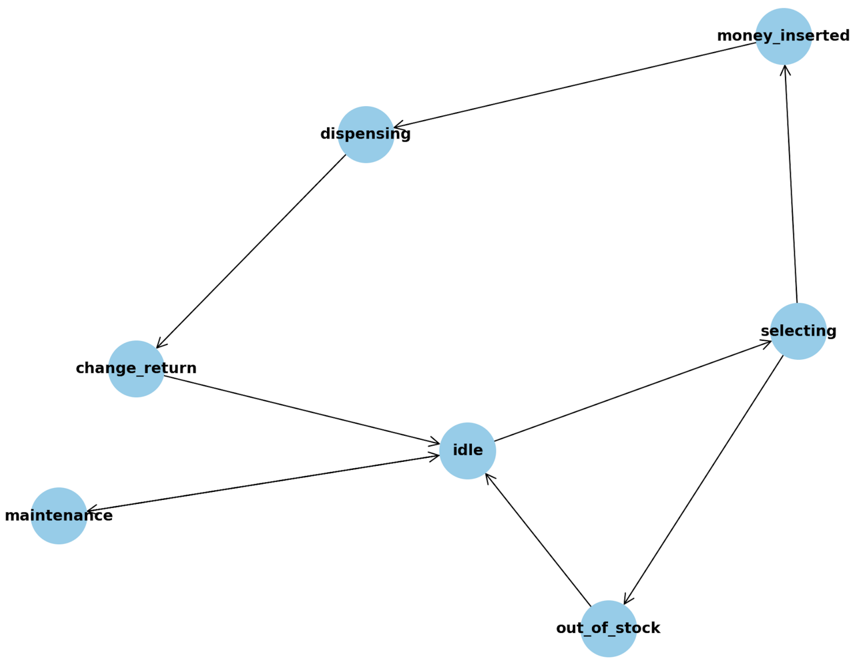


Figure 1: State graph

We will be modeling a vending machine with the following states:

idle: Waiting for customer interaction.
selecting: Customer is choosing an item.
money_inserted: Customer has inserted money.
dispensing: The selected item is being dispensed.
change_return: Change is given back to the customer.
maintenance: Vending machine is under maintenance.
out_of_stock: Selected item is unavailable.

```
[
    [idle, [selecting, maintenance]],
    [selecting, [money_inserted, out_of_stock]],
    [money_inserted, [dispensing]],
    [dispensing, [change_return]],
    [change_return, [idle]],
    [maintenance, [idle]],
    [out_of_stock, [idle]]
].

[
    [idle, [waiting]],
    [selecting, [transaction, item_selected]],
    [money_inserted, [transaction, paid]],
    [dispensing, [transaction, item_dispensed]],
    [change_return, [transaction, change_given]],
    [maintenance, [service_mode]],
    [out_of_stock, [transaction, no_stock]]
].

% Initial state
idle.
```

The following atoms were chosen:

waiting: True when the machine is idle and waiting for customer interaction.
transaction: True when there is an ongoing transaction.
item_selected: True when an item has been chosen by the customer.
paid: True when the customer has inserted money.
item_dispensed: True when the item is being dispensed.
change_given: True when the machine is returning change.
service_mode: True when the machine is in maintenance mode.
no_stock: True when the selected item is out of stock.

The valid formula checks that for every transaction where an item is selected, it will eventually either dispense the item or indicate that the item is out of stock.

```
af(or(item_selected, ef(or(item_dispensed, no_stock)))).
```

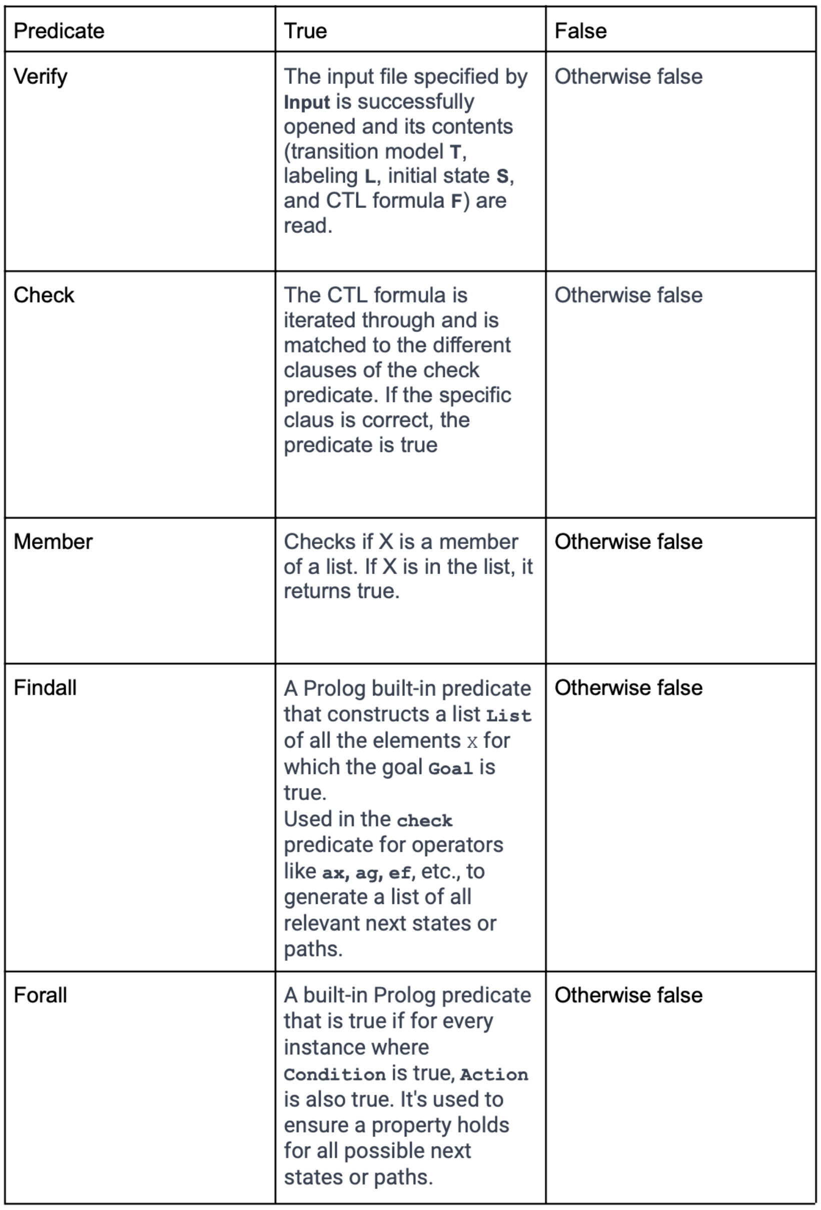
The invalid formula, on the other hand, checks for a scenario where an item is being dispensed without payment, which should not occur in a properly functioning vending machine.

```
ef(and(dispensing, neg(paid))).
```

After testing the model checker on the example above, it was then tested on all the provided valid and invalid tests, and it was successfully able to pass all of them.

Appendix A

All the predicates in the algorithm, as well as explanations for when they are true/false.



Predicate	True	False
Verify	The input file specified by **`Input`** is successfully opened and its contents (transition model **`T`**, labeling **`L`**, initial state **`S`**, and CTL formula **`F`**) are read.	Otherwise false
Check	The CTL formula is iterated through and is matched to the different clauses of the check predicate. If the specific claus is correct, the predicate is true	Otherwise false
Member	Checks if X is a member of a list. If X is in the list, it returns true.	Otherwise false
Findall	A Prolog built-in predicate that constructs a list **`List`** of all the elements `X` for which the goal **`Goal`** is true. Used in the **`check`** predicate for operators like **`ax`**, **`ag`**, **`ef`**, etc., to generate a list of all relevant next states or paths.	Otherwise false
Forall	A built-in Prolog predicate that is true if for every instance where **`Condition`** is true, **`Action`** is also true. It's used to ensure a property holds for all possible next states or paths.	Otherwise false

Figure 2: Predicates

Appendix B

The full code for the algorithm.

```
% 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.
% L - The labeling.
% S - Current state.
% U - Currently recorded states.
% 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),
    check(T, L, S, [], G).

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

% AX
check(T, L, S, U, ax(F)) :-
    findall(Next, (member([S, NextStates], T), member(Next, NextStates)),
    ↪  NextStates),
    forall(member(Next, NextStates), check(T, L, Next, U, F)).

% EX
check(T, L, S, [], ex(F)) :-
    member([S, NextStates], T),
    member(Next, NextStates),
    check(T, L, Next, [], F).

% AG
check(_, _, S, U, ag(_)) :- member(S, U).
check(T, L, S, U, ag(F)) :-
    \+ member(S, U),
    check(T, L, S, [], F),
    findall(Next, (member([S, NextStates], T), member(Next, NextStates)),
    ↪  NextStates),
    forall(member(Next, NextStates), check(T, L, Next, [S|U], ag(F))).

% EG
check(_, _, S, U, eg(_)) :-
    member(S, U).
check(T, L, S, U, eg(F)) :-
    \+ member(S, U),
    check(T, L, S, [], F),
    member([S, NextStates], T),
    member(Next, NextStates),
    check(T, L, Next, [S|U], eg(F)).

% AF
check(T, L, S, U, af(F)) :-

    \+ member(S, U),
    (
      check(T, L, S, [], F) ;
      member([S, NextStates], T),
      forall(member(Next, NextStates), check(T, L, Next, [S|U], af(F)))
    ).
% EF
```

```
check(T, L, S, U, ef(F)) :-

    \+ member(S, U),
    (
      check(T, L, S, [], F)
      ;
      findall(Next,
              (member([S, NextStates], T),
               member(Next, NextStates)),
              NextStates),
      member(Next, NextStates),
      check(T, L, Next, [S|U], ef(F))
    ).
```