CTL Model Checking

Konrad M. L. Claesson
konradcl@kth.se

Lab 3 – DD1351

December 11, 2019

Contents

1 Introduction

There are numerous proof-based and model-based verification techniques that can be used to verify the correctness of computer systems. Computer Tree Logic (CTL) is a branching-time logic that represents a system by a set of states and a set of possible transitions between the states. It can be used to model a system that transitions between states over time. Moreover, a model checker can be implemented to verify that the system satisfies a set of desired properties.

This report starts out by introducing a Prolog-based model checker for the following subset of CTL rules (see Figure 1 for rule definitions):

$$\phi ::= p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathrm{AX}\,\phi \mid \mathrm{AG}\,\phi \mid \mathrm{AF}\,\phi \mid \mathrm{EX}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{EF}\,\phi$$

Then, the model checker is used to verify two properties of a CTL model of an elevator.

2 Method

The model checker takes its input from a file that includes a list of state-"formula set" pairs that corresponds to a labeling function; another list of state-"state set" pairs that specifies all possible transitions from one state to another; a starting state for the model check; and a CTL formula to verify. Then, the passed in model is verified according to the rules in Figure 1. The implementation of each rule can be found in the program code in the appendix. Furthermore, tables 1, 2, 3 and 4 lists all Prolog predicates along with the conditions under which they return true.

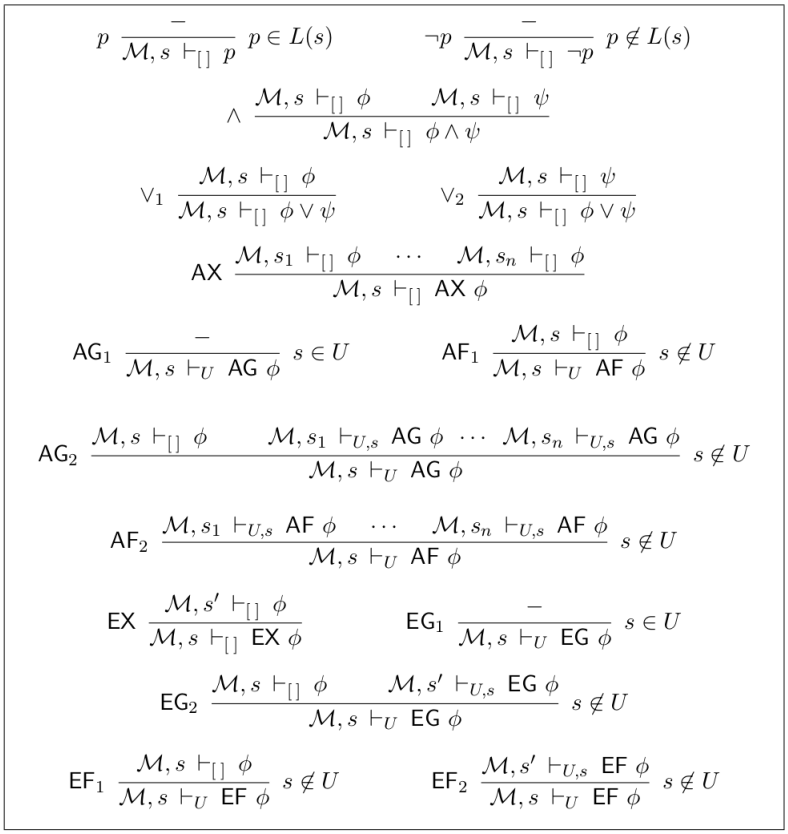

Figure 1: CTL Proof System

2.1 The Check Predicate

The `check/5` predicate takes in a transition function (called `Transitions`), a `Labeling` function, a current `State`, a list of previously visited states (called `PastStates`), and a formula to verify (called `Formula` or `X`). The term "function" is used loosely here; in the program, `Transitions` and `Labeling` are implemented as lists of pairs. `check/5` proceeds to verify the passed in formula by ensuring that it follows the CTL rules defined in figure 1.

The following table describes the `verify/1` predicate along with two helper predicates that are frequently used in the model checker.

Predicate	Parameters	Description
`verify`	`InputFileName`	`verify` returns true when the formula in the input file is verified by `check/5`.

`get_state_formulas`	`Labeling`
	`State`
	`Formulas`
		True whenever a list of the form `[State, _]` exists in `Labling`. This predicate is used to get the `Formulas` that are true in `State`.

`get_adjacent_states`	`Transitions`
	`State`
	`AdjacentStates`
		Returns true if a list of the form `[State, _]` exists in `AdjacentStates`. This predicate is used to get the states to which it is possible to transition from `State`.

Table 1: Entry Point and Helper Predicates

The CTL rules of figure 1 can be subdivided into *propositional* rules, A-rules and E-rules. Each of these categories are explored in the subsections ahead.

2.1.1 Propositional Rules

The propositional rules of CTL are $\phi ::= p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi$ and are implemented by checking whether the formula to be verified is a member of the formulas that are `true` in the current state. For example,

```
check(_, Labling, State, [], Formula) :-
    get_state_formulas(Labling, State, Formulas),
    member(Formula, Formulas).
```

verifies that `Formula` is `true`.

The below table describes under what conditions the check predicates for each of the propositional rules are true.

CTL Rule	Truth Conditions
`p`	The check predicate evaluates to true if `p` is a member of the formulas on the given state.
`neg(p)`	The check predicate evaluates to true if `p` is *not* a member of the formulas on the given state.
`and(X,Y)`	The check predicate evaluates to true if both `X` and `Y` are members of the given state.
`or(X,_)`	The check predicate evaluates to true if `X` is a member of the given state.
`or(_,Y)`	The check predicate evaluates to true if `Y` is a member of the given state.
`or(_,Y)`	The check predicate evaluates to true if `Y` is a member of the given state.

Table 2: Truth Conditions for Propositional CTL Rules

2.1.2 *A*-Rules

The CTL rules referred to as A-rules in this report are

$$\phi ::= \mathrm{AX}\,\phi \mid \mathrm{AG}\,\phi \mid \mathrm{AF}\,\phi$$

and all share the commonality that the condition that they impose must be `true` *along all paths*. Another commonality is that all A-rules leverage the `check_all/5` predicate, which has the implementation

```
check_all(_, _, [], _, _).
check_all(Transitions, Labling, [State|States], PastStates, X) :-
   check(Transitions, Labling, State, PastStates, X),
   check_all(Transitions, Labling, States, PastStates, X).
```

and checks that all states, i.e. `[State|States]` from the code, satisfy the formula `X`. The following table states under what conditions the `check/5` predicates for the A-rules return `true`.

CTL Rule	Truth Conditions
`ax(X)`	The check predicate evaluates to `true` if `X` is `true` in any state to which there exists an immediate transition from the current state.
`ag(X)`	The check predicate evaluates to `true` if `X` is `true` in all states.
`af(X)`	The check predicate evaluates to `true` if `X` is eventually true in some state along any path.

Table 3: Truth Conditions for CTL A-Rules

2.1.3 *E*-Rules

The CTL rules referred to as E-rules in this report are

$$\phi ::= \mathrm{EX}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{EF}\,\phi$$

and all share the commonality that the condition that they impose must be `true` *along some path.* Another commonality is that all E-rules leverage the `find_state/5` predicate, which has the implementation

```
find_state(Transitions, Labling, [State|States], PastStates, X) :-
   check(Transitions, Labling, State, PastStates, X);
   find_state(Transitions, Labling, States, PastStates, X).
```

and checks if there exists a state in `[State|States]` where the formula `X` is satisfied. The subsequent table states under what conditions the `check/5` predicates for the E-rules return `true`.

CTL Rule	Truth Conditions
`ex(X)`	The check predicate evaluates to `true` if `X` is `true` in some state to which there exists an immediate transition from the current state.

`eg(X)`	The check predicate evaluates to `true` if there exists a path along which `X` is `true` in all states.
`ef(X)`	The check predicate evaluates to `true` if there exists a path where `X` is `true` eventually.

Table 4: Truth Conditions for CTL E-Rules

2.2 Elevator Model

An elevator traveling between two floors was modeled and tested using the described model checker. The following atomic propositions are used in the model:

- `floor1`: The elevator is on the first floor.
- `floor2`: The elevator is on the second floor.
- `open`: The elevator doors are open.
- `closed`: The elevator doors are closed.
- `opening`: The elevator doors are opening.
- `closing`: The elevator doors are closing.
- `still`: The elevator is not moving.
- `btn1`: The elevator button to go to the first floor is toggled on.
- `btn2`: The elevator button to go to the second floor is toggled on.

The elevator was modeled with the states $S := \{s_0, s_1, \ldots, s_{13}\}$, labeling

function

$$\begin{aligned} L := \{&(s0, \{\texttt{floor1}, \texttt{open}, \texttt{still}\}),\\ &(s1, \{\texttt{floor1}, \texttt{open}, \texttt{still}, \texttt{btn2}\}),\\ &(s2, \{\texttt{floor1}, \texttt{closing}, \texttt{still}, \texttt{btn2}\}),\\ &(s3, \{\texttt{floor1}, \texttt{closed}, \texttt{still}, \texttt{btn2}\}),\\ &(s4, \{\texttt{closed}, \texttt{still}, \texttt{still}\}),\\ &(s5, \{\texttt{floor2}, \texttt{closed}, \texttt{still}\}),\\ &(s6, \{\texttt{floor2}, \texttt{opening}, \texttt{still}\}),\\ &(s7, \{\texttt{floor2}, \texttt{open}, \texttt{still}\}),\\ &(s8, \{\texttt{floor2}, \texttt{open}, \texttt{still}, \texttt{btn1}\}),\\ &(s9, \{\texttt{floor2}, \texttt{closing}, \texttt{still}, \texttt{btn2}\}),\\ &(s10, \{\texttt{floor2}, \texttt{closed}, \texttt{still}, \texttt{still}, \texttt{btn2}\}\}),\\ &(s11, \{\texttt{closed}, \texttt{down}, \texttt{btn1}\}),\\ &(s12, \{\texttt{floor1}, \texttt{closed}, \texttt{still}\}),\\ &(s13, \{\texttt{floor1}, \texttt{openning}, \texttt{still}\})\}, \end{aligned}$$

and transition function

$$\begin{aligned} \rightarrow := \{&(s_0, \{s_0, s_1\}), (s_1, \{s_0, s_1, s_2\}), (s_2, \{s_3, s_{13}\}),\\ &(s_3, \{s_4\}), (s_4, \{s_5\}), (s_5, \{s_6\}),\\ &(s_6, \{s_7\}), (s_7, \{s_7, s_8\}), (s_8, \{s_7, s_8, s_9\}),\\ &(s_9, \{s_{10}, s_6\}), (s_{10}, \{s_{11}\}), (s_{11}, \{s_{12}\}),\\ &(s_{12}, \{s_{13}\}), (s_{13}, \{s_0\})\}. \end{aligned}$$

As previously, the term "function" is used loosely here; L and $\rightarrow$ are actually sets of pairs.

The model was tested for two formulas:

- `ef(and(and(floor2, open), still))`
- `ef(and(neg(still), or(open, or(opening, closing))))`

The first formula states that "from the current state (which has to be specified), there exists a path where, eventually, `floor2` $\wedge$ `open` $\wedge$ `still` is `true`." In other words, it is possible to use the elevator to get to the second floor. The formula was be tested starting from state s_0 (an idle elevator on the first floor) and was expected to evaluate to `true`.

The second formula states that "from the current state, there exists a path where, eventually, $\neg$`still` $\wedge$ (`open` $\vee$ `opening` $\vee$ `closing`) is `true`". Phrased differently, the formula states that there exists a state where the elevator is moving while its doors are not closed. The formula was tested from state s_0 and was expected to return `false`.

3 Results

The model checker was tested with 32 unit tests that test each of the CTL rules in isolation on models with a minimal number of propositions, states and transitions. All unit tests passed. The model checker was also tested with 905 more complex integration tests. All integration tests passed too.

The formulas that were tested for the elevator model ended up evaluating to the expected boolean values. `ef(and(and(floor2, open), still))` returned `true` and `ef(and(neg(still), or(open, or(opening, closing))))` returned `false`.

The model checker along with the elevator model and tests can be found by following this link: github.com/konradcl/ctl-model-checker. The model checker and tests for the above elevator model formulas have also been appended to this report.

4 Appendix

4.1 Input File for Testing `ef(and(and(floor2, open), still))`

```
% Transition Function
[
   [s0, [s0, s1]],
   [s1, [s0, s1, s2]],
   [s2, [s3, s13]],
   [s3, [s4]],
   [s4, [s5]],
   [s5, [s6]],
   [s6, [s7]],
   [s7, [s7, s8]],
   [s8, [s7, s8, s9]],
   [s9, [s10, s6]],
   [s10, [s11]],
   [s11, [s12]],
   [s12, [s13]],
   [s13, [s0]]
].

% Labling Function
[
   [s0, [floor1, open, still]],
   [s1, [floor1, open, still, btn2]],
   [s2, [floor1, closing, still, btn2]],
   [s3, [floor1, closed, still, btn2]],
   [s4, [closed, up, btn2]],
   [s5, [floor2, closed, still]],
   [s6, [floor2, opening, still]],
   [s7, [floor2, open, still]],
   [s8, [floor2, open, still, btn1]],
   [s9, [floor2, closing, still, btn1]],
   [s10, [floor2, closed, still, btn2]],
   [s11, [closed, down, btn1]],
   [s12, [floor1, closed, still]],
   [s13, [floor1, opening, still]]
```

```
].

% Initial State
s0.

% Formula to Verify
ef(and(and(floor2, open), still)).
```

4.2 Input File for Testing `ef(and(neg(still), or(open, or(opening, closing))))`

```
% Transition Function
[
   [s0, [s0, s1]],
   [s1, [s0, s1, s2]],
   [s2, [s3, s13]],
   [s3, [s4]],
   [s4, [s5]],
   [s5, [s6]],
   [s6, [s7]],
   [s7, [s7, s8]],
   [s8, [s7, s8, s9]],
   [s9, [s10, s6]],
   [s10, [s11]],
   [s11, [s12]],
   [s12, [s13]],
   [s13, [s0]]
].

% Labeling Function
[
   [s0, [floor1, open, still]],
   [s1, [floor1, open, still, btn2]],
   [s2, [floor1, closing, still, btn2]],
   [s3, [floor1, closed, still, btn2]],
   [s4, [closed, up, btn2]],
   [s5, [floor2, closed, still]],
```

```
   [s6, [floor2, opening, still]],
   [s7, [floor2, open, still]],
   [s8, [floor2, open, still, btn1]],
   [s9, [floor2, closing, still, btn1]],
   [s10, [floor2, closed, still, btn2]],
   [s11, [closed, down, btn1]],
   [s12, [floor1, closed, still]],
   [s13, [floor1, opening, still]]
].

% Initial State
s0.

% Formula to Verify
ef(and( neg(still), or(open, or(opening, closing)) )).
```

4.3 Model Checker Implementation

```
:- (discontiguous check/5).

verify(Input) :-
   see(Input), read(Transitions), read(Labeling),
   read(State), read(Formula), seen,
   check(Transitions, Labeling, State, [], Formula).

% Recurses through Labeling to find the Formulas of State.
get_state_formulas(Labeling, State, Formulas) :-
   member([State, Formulas], Labeling).

% Recurses through Transitions to find the adjacent states to State.
% AdjacentStates may include State itself.
get_adjacent_states(Transitions, State, AdjacentStates) :-
   member([State, AdjacentStates], Transitions).

% FORMULA
check(_, Labeling, State, [], Formula) :-
   get_state_formulas(Labeling, State, Formulas),
```

```
   member(Formula, Formulas).

% NEGATION
check(_, Labeling, State, [], neg(Formula)) :-
   get_state_formulas(Labeling, State, Formulas),
   \+ member(Formula, Formulas).

% AND
check(Transitions, Labeling, State, [], and(X, Y)) :-
   check(Transitions, Labeling, State, [], X),
   check(Transitions, Labeling, State, [], Y).

% OR 1
check(Transitions, Labeling, State, [], or(X, _)) :-
   check(Transitions, Labeling, State, [], X).

% OR 2
check(Transitions, Labeling, State, [], or(_, Y)) :-
   check(Transitions, Labeling, State, [], Y).

% Checks that all states, i.e. [State|States], satisfy Formula.
check_all(_, _, [], _, _).
check_all(Transitions, Labeling, [State|States], PastStates, X) :-
   check(Transitions, Labeling, State, PastStates, X),
   check_all(Transitions, Labeling, States, PastStates, X).

% AX
check(Transitions, Labeling, State, [], ax(X)) :-
   get_adjacent_states(Transitions, State, AdjacentStates),
   check_all(Transitions, Labeling, AdjacentStates, [], X).

% AG1 and AG2
check(_, _, State, PastStates, ag(_)) :-
   member(State, PastStates).

check(Transitions, Labeling, State, PastStates, ag(X)) :-
   \+ member(State, PastStates),
```

```
   get_adjacent_states(Transitions, State, AdjacentStates),
   check(Transitions, Labeling, State, [], X),
   check_all(Transitions, Labeling, AdjacentStates,
      [State|PastStates], ag(X)).

% AF1 and AF2
check(Transitions, Labeling, State, PastStates, af(X)) :-
   \+ member(State, PastStates),
   check(Transitions, Labeling, State, [], X).

check(Transitions, Labeling, State, PastStates, af(X)) :-
   \+ member(State, PastStates),
   get_adjacent_states(Transitions, State, AdjacentStates),
   check_all(Transitions, Labeling, AdjacentStates,
      [State|PastStates], af(X)).

% Checks if there exists a state in [State|States] where Formula
% is satsified.
find_state(Transitions, Labeling, [State|States], PastStates, X) :-
   check(Transitions, Labeling, State, PastStates, X);
   find_state(Transitions, Labeling, States, PastStates, X).

% EX
check(Transitions, Labeling, State, [], ex(X)) :-
   get_adjacent_states(Transitions, State, AdjacentStates),
   find_state(Transitions, Labeling, AdjacentStates, [], X).

% EG1 and EG2
check(_, _, State, PastStates, eg(_)) :-
   member(State, PastStates).

check(Transitions, Labeling, State, PastStates, eg(X)) :-
   \+ member(State, PastStates),
   check(Transitions, Labeling, State, [], X),
   get_adjacent_states(Transitions, State, AdjacentStates),
   find_state(Transitions, Labeling, AdjacentStates,
      [State|PastStates], eg(X)).
```

```
% EF1 and EF2
check(Transitions, Labeling, State, PastStates, ef(X)) :-
   \+ member(State, PastStates),
   check(Transitions, Labeling, State, [], X).

check(Transitions, Labeling, State, PastStates, ef(X)) :-
   \+ member(State, PastStates),
   get_adjacent_states(Transitions, State, AdjacentStates),
   find_state(Transitions, Labeling, AdjacentStates,
      [State|PastStates], ef(X)).
```