A C++ Implementation Of A Model Checker

Eliyah Kilada

I. INTRODUCTION

A complexity of concurrent systems increases, former scenario-based simulations can no more guarantee correct operations. Formal verification methods based on finite state analysis are now inevitable. Computation tree logic (CTL) is a common language to formally describe the desired behavior of a concurrent system. It does not only support propositional logic but also temporal logic that describes sequential events. Model checkers are, hence, used to verify that a system meets its specification (or desired behavior) [1]–[4]. This paper presents a C++ implementation of a model checker. It accepts state graph (SG) as a model of the system, and CTL as the specification language.

Paper is organized as follows. Section II introduces the semantics of CTL used by the tool. Some implementation details (e.g., data structures and relevant functions) are described in Section III. The tool interface and supported commands are also described in the same Section. Finally, Section IV goes through a typical example where certain CTL formulas are checked against a given state graph.

II. SPECIFICATIONS SEMANTICS

The different CTL formulas supported by the tool is described below. CTL semantics are defined with respect to a state graph (SG). Let \( N \) be the set of state variables, the SG is modeled by the tuple \((S, R, L)\):

- \( S \): is the set of states.
- \( R \subseteq S \times S\): a binary relation for the state transitions.
- \( L : S \to N\): a labeling function that assigns to each state a label that is a set of the formulas that are TRUE in that state. Initially, these are the TRUE atomic formulas in the state along with the TRUE formula. Description of them will follow.

A path is an infinite sequence of states \((s_0, s_1, s_2, \ldots)\) such that \(\forall j[(s_j, s_{j+1}) \in R]\).

A. FALSE

This formula is FALSE in all the states. It has a formula ID of 0.

B. TRUE

This formula is TRUE in all the states. It has a formula ID of 1.

C. ATOMIC

The \( n \)-state variables (including inputs and outputs) are assigned \( n \) unique IDs starting from 2 to \( n+1 \). These unique IDs serve also as their atomic formula IDs. The atomic formula of a certain state variable is TRUE in a state \( s_0 \) if this state variable is 1 in \( s_0 \) (or equivalently, if the corresponding atomic formula ID of that state variable is in the label of \( s_0 \)). Formally,

\( s_0 \models f \iff f \in L(s_0) \)  \( (1) \)

D. NOT\((f)\)

This formula is TRUE in a state \( s_0 \) if formula \( f \) is FALSE in \( s_0 \). Formally,

\( s_0 \models \text{NOT}(f) \iff \neg(s_0 \models f) \)  \( (2) \)

E. AND\((f_1, f_2)\)

This formula is TRUE in a state \( s_0 \) if both formulas \( f_1 \) and \( f_2 \) are TRUE in \( s_0 \). Formally,

\( s_0 \models \text{AND}(f_1, f_2) \iff (s_0 \models f_1) \land (s_0 \models f_2) \)  \( (3) \)

F. AX\((f)\)

This formula is TRUE in a state \( s_0 \) if formula \( f \) is TRUE in every immediate successor of \( s_0 \). Formally,

\( s_0 \models AX(f) \iff \text{for all states } s_1 \text{ such that } (s_0, s_1) \in R, \quad s_1 \models f \)  \( (4) \)

G. EX\((f)\)

This formula is TRUE in a state \( s_0 \) if formula \( f \) is TRUE in one or more immediate successors of \( s_0 \). Formally,

\( s_0 \models EX(f) \iff \text{for some state } s_1 \text{ such that } (s_0, s_1) \in R, \quad s_1 \models f \)  \( (5) \)
A C++ IMPLEMENTATION OF A MODEL CHECKER

H. AU\((f_1, f_2)\)

This formula is TRUE in a state \(s_0\) if for every path starting with \(s_0\), there exists an initial prefix of that path such that \(f_2\) holds at the last state of the prefix and \(f_1\) holds at all other states along the prefix. Formally,

\[
s_0 \models AU(f_1,f_2) \iff \exists i[i \geq 0 \land s_i \models f_2 \land \forall j[0 \leq j < i \rightarrow s_j \models f_1]]
\]

I. EU\((f_1, f_2)\)

This formula is TRUE in a state \(s_0\) if for some path starting with \(s_0\), there exists an initial prefix of that path such that \(f_2\) holds at the last state of the prefix and \(f_1\) holds at all other states along the prefix. Formally,

\[
s_0 \models EU(f_1,f_2) \iff \exists i[i \geq 0 \land s_i \models f_2 \land \forall j[0 \leq j < i \rightarrow s_j \models f_1]]
\]

J. AF\((f)\)

This formula is TRUE in a state \(s_0\) if formula \(f\) holds in the future along every path from \(s_0\). Formally,

\[
AF(f) \equiv AU(TRUE,f)
\]

K. EF\((f)\)

This formula is TRUE in a state \(s_0\) if formula \(f\) holds in the future of some path from \(s_0\). Formally,

\[
EF(f) \equiv EU(TRUE,f)
\]

L. EG\((f)\)

This formula is TRUE in a state \(s_0\) if there exists some path from \(s_0\) on which \(f\) holds at every state. Formally,

\[
EG(f) \equiv NOT(AF(NOT(f)))
\]

M. AG\((f)\)

This formula is TRUE in a state \(s_0\) if formula \(f\) holds at every state on every path from \(s_0\). Formally,

\[
AG(f) \equiv NOT(EF(NOT(f)))
\]

N. OR\((f_1, f_2)\)

This formula is TRUE in a state \(s_0\) if formula \(f_1\) or formula \(f_2\) is TRUE in \(s_0\). Formally,

\[
s_0 \models OR(f_1,f_2) \iff s_0 \models f_1 \lor s_0 \models f_2
\]

O. IMP\((f_1, f_2)\)

This formula is TRUE in a state \(s_0\) if the truth of formula \(f_1\) implies the truth of formula \(f_2\) in \(s_0\). Formally,

\[
IMP(f_1,f_2) \equiv OR(NOT(f_1),f_2)
\]

III. TOOL IMPLEMENTATION

The model checker has been implemented in C++. Main data structures used are:

- SignalValues_VIV: A vector of integer vectors. For each state it holds a vector of its state variables values.
- NextStates_VIV: A vector of integer vectors. For each state it holds a vector of its successors.
- StateLabel_VIS: A vector of integer sets. For each state it holds a set of all formulas that has been proven TRUE in that state.

Following is the C++ implementation of AU and EU CTL operators. Implementation of other CTL formulas mentioned in Section II, are either based on them or straightforward.

AU implementation follows the same algorithm described in [1]. The algorithm time is \(O(\text{card}(S) + \text{card}(R))\). \(AU(f_1,f_2)\) applies the AU operator on the given two formulas (i.e., \(f_1, f_2\)) and returns a unique formula ID, \(f\), of the resulting formula.

For each state in the graph - that has not been visited by the algorithm - \(AU\) function calls another function named \(AU\_S\). \(AU\_S(f_1,f_2,s)\) checks whether \(AU(f_1,f_2)\) is TRUE in \(s\), and if so, it adds \(f\) to this state label.

To keep track of visited states, the algorithm makes use of an integer vector named Marked. Marked\([i]\) is 1 if \(s_i\) has been visited by the algorithm, otherwise it is 0, where \(0 \leq i < n\text{States}\).

To check if formula ID, \(f\), is in the label of state \(s_i\), the algorithm makes use of IsLabeled(FormulaID, \(s_i\)), where:

\[
\text{IsLabeled}(f, s_i) = 1 \iff f \in L(s_i)
\]

To add a formula \(f\) - that has been proven TRUE to a state, \(s_i\), label, the algorithm makes use of AddLabel(FormulaID, \(s_i\)). Following is a C++ code that computes \(AU(f_1,f_2)\):

```cpp
unsigned int AU(unsigned int f1, unsigned int f2) {
    unsigned int FormulaID=ProduceUniqueFormulaID();
    if (Marked[s]==1) {
        // Marked[s] is already visited; return false.
        return FormulaID;
    }
    // Label AU formula on a certain state. Returns true if AU holds in that state, otherwise false.
    unsigned int AU_S(unsigned int f1, unsigned int f2, Marked) {
        for (unsigned int n=0;n<nStates;n++) {
            if (Marked[n]==0) {
                // Call AU_S only if the state is not visited.
                if (Marked[n]==0) {
                    AU_S(f1,f2,n,Marked);
                }
                return FormulaID;
            }
        }
        // Label AU formula on a certain state. Returns 1 if AU holds in that state, otherwise 0.
        return FormulaID;
    }
    // Label AU formula on a certain state. Returns true if AU holds in that state, otherwise false.
    return IsLabeled(FormulaID, s_i);
}
```
A C++ IMPLEMENTATION OF A MODEL CHECKER

```c
if (IsLabeled(FormulaID,s)==1) return 1;
else return 0;
}
// Mark state s as visited.
Marked[s]=1;

// If f2 holds in s, then f also holds. Return 1.
if (IsLabeled(f2,s)==1) {
    AddLabel(FormulaID,s);
    return 1;
}
// If neither f2 nor f1 hold in the state, then f is FALSE. Return 0.
if (IsLabeled(f1,s)==0) {
    return 0;
}
// If f2 is FALSE and f1 is TRUE in s, then, call AU_S for each of its successors, sm. f is TRUE in s iff it is TRUE in all its successors.
for (unsigned int m=0;m<NextStates_VIV[s].size();m++) {
    if (AU_S(f1,f2,NextStates[s][m],Marked)==0){
        return 0;
    }
    AddLabel(FormulaID,s);
    return 1;
}
```

A very similar algorithm is used to implement \( \mathcal{E} \) operator. The only difference is: in case of \( f_2 \) is FALSE and \( f_1 \) is TRUE in \( s \), then, \( f \) will be TRUE in \( s \) if it is TRUE in any of its successors. This algorithm is different than the one proposed in [1]. It requires time \( O(\text{card}(S) + \text{card}(R)) \).

```c
for (unsigned int m=0;m<NextStates_VIV[s].size();m++) {
    if (EU_S(f1,f2,NextStates_VIV[s][m],Marked)==1){
        AddLabel(FormulaID,s);
        return 1;
    }
    return 0;
}
```

A small parser has been developed to read state graph files. Typical state graph file should have the following syntax:

```
SG:
STATEVECTOR:rw ap aw rp
STATES:
0:R000
1:10R0
....
EDGES:
0:R000 - rw+ -> 1:10R0
1:10R0 - aw+ -> 2:F01R
2:F01R - rw- -> 3:001R
2:F01R - rp+ -> 4:FR11
3:001R - rp+ -> 5:0R11
```

The tool loads its internal data structures with the given SG. During initialization, the tool calls `InitializeStateLabels()` to initialize the label of each state (i.e., `StateLabel_VIS`) with the atomic formulas that are TRUE in that state along with the TRUE formula. Each state label is then appended with formulas that have been proven TRUE during runtime. The tool accepts the formulas to be checked through standard command line input. To do that, flex and bison, have been used to read nested CTL formulas recursively. Following is described three types of commands a user can input on the command line:

A. Perform a CTL Formula

This command takes the following form:

```c
>> CTLFormula
```

Where `CTLFormula` could be any of the 15 formulas mentioned in Section II or any recursion of them. The command is executed as follows - this will be done recursively for a nested CTL:

1) Tool accepts the CTL formula from the user through flex and bison.
2) Then, it generates a unique formula ID for it.
3) It checks this formula in each state in the graph.
4) For each state where this formula holds, it will add this formula ID to the state label.
5) It returns the unique formula ID.

B. Check a CTL Formula

This command takes the following form:

```c
>> CheckSG(CTLFormula)
```
or, equivalently,

```c
>> CheckSG(FormulaID)
```

where `FormulaID` identifies the same `CTLFormula`. The model checker will print whether this formula is TRUE in all the states or if it is FALSE in some states and what these states are.

C. Variable Assignment

This command takes the following form:

```c
>> VariableName=CTLFormula
```
or, equivalently,

```c
>> VariableName=FormulaID
```

where `FormulaID` identifies the same `CTLFormula`. This command is very useful in case a certain formula will be used again in other subsequent formulas. In this case, a variable that holds the formula ID could be used directly instead of computing the `CTLFormula` again.

IV. EXAMPLE

As an example, several CTL formulas will be checked against the state graph of Fig. 1. Following is the tool run log:

```c
input: CheckSG(NOT(FALSE))
The given formula is TRUE in all the states.
```

```c
input: CheckSG(NOT(rw))
The given formula is FALSE in the following state(s):
1 2 3 4 6 11
```

```c
input: CheckSG(NOT(rw))
The given formula is FALSE in the following state(s):
1 2 3 4 6 11
```

```c
input: v1=NOT(rw)
Setting Formula ID of v1 to 8
To check this formula truth, write CheckSG(v1)
```

```c
input: CheckSG(v1)
The given formula is FALSE in the following state(s):
1 2 3 4 6 11
```

```c
input: g=IMP(NOT(ap),AU(NOT(ap),aw))
Setting Formula ID of g to 12
To check this formula truth, write CheckSG(g)
```

```c
input: CheckSG(g)
The given formula is FALSE in the following state(s):
```
A C++ implementation of a protocol model checker has been presented. Essential data structures used as well as relevant functions have been described. The tool interface and the supported commands have been illustrated through a typical example. The presented model checker supports 15 different CTL formulas or any recursion of them. It has a standard input command line interface. Algorithms used to compute CTL formulas require, at most, time $O(\text{card}(S)+\text{card}(R))$.

V. Conclusion

A C++ implementation of a protocol model checker has been presented. Essential data structures used as well as relevant functions have been described. The tool interface and the supported commands have been illustrated through a typical example. The presented model checker supports 15 different CTL formulas or any recursion of them. It has a standard input command line interface. Algorithms used to compute CTL formulas require, at most, time $O(\text{card}(S)+\text{card}(R))$.

REFERENCES


