Automatic Verification of Systems
Final Project - Symbolic Model Checker

Omri Lifshitz
205490675
omrilifshitz@mail.tau.ac.il

Idan Berkovits
205404130
berkovits@mail.tau.ac.il

1 Abstract

2 Introduction

3 CTL and LTL Model Checking

Let us begin by describing the temporal logic CTL*, and from there continue to describe both Computation Tree Logic (CTL) and Linear Temporal Logic(LTL). CTL* contains two types of formulae: state formulae which hold for specific states and path formulae which hold along a specific path. State formulae are given by the following rules [**?**]:

- If $p \in AP$ (Atomic propositions) then p is a state formula.
- If $f,\ g$ are state formulae then $\neg f, f \vee g$ are state formulae.
- If f is a path formula, then Ef ("for some computation path") is a state formula.

Path formulae are defined by the following rules:

- If f is a state formula, it is also a path formula.
- If $f,\ g$ are path formulae then $\neg f,\ f \vee g,\ Xf$ and fUg are also path formulae.
- If f is a path formula, then Ef ("for some computation path") is a state formula.

Let $M = (S, R, L)$ be a Kripke structure where S is the set of states, R is the transition relation (we assume there is a total transition relation) and L be the labeling function.

Definition 1 *A path in Kripke structure M is an infinite sequence of states $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$.*

The notation $M, s \models f$ is used to state that f holds for state s in the Kripke structure M, and similarly for $M, \pi \models f$ for path π in M.

Definition 2 *Let f_1, f_2 be state formulae and g_1, g_2 path formulae. The relation $\models$ is defined recursively as follows:*

$$
\begin{aligned}
M, s \models p &\iff p \in L(s) && (1)\\
M, s \models \neg f_1 &\iff M, s \not\models f_1 && (2)\\
M, s \models f_1 \vee f_2 &\iff M, s \models f_1 \, or M, s \models f_2 && (3)\\
M, s \models E g_1 &\iff \text{there is a path } \pi \text{ starting at } s \text{ such that } M, \pi \models g_1 && (4)\\
M, \pi \models f_1 &\iff s \text{ is the first state of } \pi \text{ and } M, s \models f_1 && (5)\\
M, \pi \models \neg g_1 &\iff M, \pi \not\models g_1 && (6)\\
M, \pi \models g_1 \vee g_2 &\iff M, \pi \models g_1 \text{ or } M, \pi \models g_2 && (7)\\
M, \pi \models X g_1 &\iff M, \pi^1 \models g_1 && (8)\\
M, \pi \models g_1 U g_2 &\iff \exists k \geq 0 \text{ s.t. } M, \pi^k \models g_2 \text{ and } \forall 0 \leq j \leq k \; M, \pi^j \models g_1 && (9)
\end{aligned}
$$

Definition 3 *CTL is the subset of CTL* obtained by specifying the path formulae using the following rules[**?**]:*

- *If f, g are state formulae, then Xf and fUg are path formulae.*
- *If f is a path formulae, then so is $\neg f$.*

From the definition we can see that CTL is a subset of CTL* that only permits branching operators. Therefore, in this new logic, all linear-time operators can only appear if they are immediately preceded by path quantifiers.

Definition 4 *LTL is a subset of CTL restricted to all formulae of the form Af where f is a path formula in which the only state sub-formulae are atomic propositions[**?**]. More precisely, path formulae here are defined in the following recursive manner:*

- *An atomic proposition.*
- *If f, g are path formulae, then $\neg f$, $f \vee g$, Xf and fUg are also path formulae.*

4 Symbolic Model Checking - BDDs

5 Implementation

In our project we implemented a symbolic CTL and LTL model cheker in python using python. Throughout our entire implementation we used the python module **pyeda** which

allows easy use of ROBDDs, as explained in the previous section. In the following sections we present the main modules in our implemenation and how they are used to create the model checker.

5.1 Logical model parsing

The first module we implemented is the **SymbolicModel** class; a python class used in order to represent a the symbolic version of a given logical model. The class' conststructor receives an integer specifying the number of states in the model, and uses that in order to create the variables $v_1, \ldots, v_n$ representing the states in the states. It also has two interface methods, *add_atomic*, *add_relation* used to specify the atomic propositions that hold in each state and the transition relation between the states in the model. As explained in the previous sections, both the labeling and the transition relation of the model are also given as BDDs. The SymbolicModel is used to represent the model that we wish to check, and thus this class is used throught the entire project.

5.2 Tableau construction

The next module we implemented is a module in charge of creating the tableau for a given LTL formual. In order to create this module we began by implementing a formula parser in charge of parsing the given formula for later uses such as parsing the elementary formulae for the tableau or iterating over subformulae.

The next step of implementing the tableau was to create a list of all elementary formulae. This was done using the **get_elementary_formulas** method we implemented. This method uses the aforementioned formula parser in order to iterate over all parts of the formula and for each quantifier found in the formula it creates the relevant elementary formulae. For each of the elementary formulae we create a BDD variable that will late be used to indicate whether this elementary formula is satisfied.

The next part of creating the tableau is to find its initial states. As explained in the previous section, the initial states of the tableau are all states that satisfy the given formula. In order to find these states, we implemented the **sat** module. This module implements the method **get_sat(formula, el_bbds)** (where el_bdds is the set of BDDs explained above) and returns a new BDD using logical operands on the received BDDs according to the construction of the given formula.

The elementary BDDs and the **sat** module described above are then used in order to create the transition relation of the tableau itself. Again, in order to make this a symbolic model checker and not explicitly describe all of the states and transitions, the transition relation is also given as a BDD. The fairness constraints of the tableau are also given as a BDD and constructed in a similar manner to the statisfaction BDD.

All the methods described above are encapsulated inside a **Tableau** class object that we created. This object receives the formula and its atomic propositions as parameters for the

constructor and uses all the methods and modules described above in order to create the tableau.

5.3 Product structure construction

Using the tableau construction above we now implemented the product of the tableau and the given logical model. We implemented this as a method of the tableau class described above. This method receives as input the symbolic model described earlier and using its relations, atomic propositions and the relations and states of the tableau creates the product model.

The use of BDDs makes this operation very simple, as we can see in the implemenation presented below.

Algorithm 1 Product model creation

procedure PRODUCT(self,model)

assert isinstance(model, SymbolicModel)

product_relations = model.relations & self.relations & model.atomic & model.atomic.compose(global_compose)

return Graph(global_compose, product_relations)

The use of global_compose is in order to create a new set of variables used for the product model. The Graph object used in the code above is a data structure we created in order to create a symbolic representation of the product including both its vertices and the edges between them based on the relation BDD.

5.4 Finding maximal SCCs

As taught in class, in order to assert that there is a fair path in the product structure, we need to find a strongly connected component (SCC) that is reachable from the initial state and that intersects with each one of the fairness constraints.

In this section we will present the algorithm use to find the SCCs in a given model using BDDs. Our algorithm is based the algorithm explained in [**?**]. The algorithm presented in the paper, and implemented in our project uses reachability analysis in order to find the set of SCCs in a graph. This idea will also be useful late when we analyse the number of reachable states in our product structure.

We first begin by creating two very crucial methods allowing use to find the successor and predecessor states of a set of states in the graph when searching for them int a given bound. These two methods are presented below.

Algorithm 2 Predecessor

procedure PREDECESSOR(base, bound, relation, other_compose)

result = ignore_prims(relation & base.compose(other_compose), other_compose.values())

return result & bound

Algorithm 3 Successor

procedure SUCCESSOR(base, bound, relation, other_compose)

result = ignore_prims(relation & base, other_compose.values()).compose(other_compose)

return result & bound

Next, use reachability analysis in order to find the backward_set and forward_set of each node in the graph. The backward_set is the set of all states in the graph from which we have a path that reaches our desired and node. The forward_set is defined in a similar manner and contains all the nodes to which there exists a path from the specified base node.

Besides the forward and backward sets described above, the algorithm also introduces finite maximal distance (FMD) predecessors. These predecessors to a given node which can traverse through a finite distance path on the way to the node. These FMDs are used in order to find SCCs, as each FMD predecessor is not part of an SCC connected the node that it is a predecessor to. Below we present the algorithm found in [**?**] which we used in order to find all SCCs in the graph.

Algorithm 4 SCC Decomposition

procedure SCC_DECOMP(N,V)

$V' \leftarrow V$

while $V' \neq 0$ **do**

$v \leftarrow random\_take(V')$

$B(v) \leftarrow backward_set(v, V', N)$

$SCC\_DECOMP\_RECUR(v, B(v), N)$

$V' \leftarrow V' \wedge \overline{v \vee B(v)}$

Algorithm 5 Finite maximum distance predecessors

procedure FMD_PRED(W, U, N)

$pred \leftarrow 0$

$front \leftarrow W$

$bound \leftarrow U$

while $front \neq 0$ **do**

$x \leftarrow \exists_Y front(Y) \wedge N(X,Y) \wedge bound(X)$

$y \leftarrow \exists_Y bount(Y) \wedge N(X,Y) \wedge bound(X)$

$front \leftarrow x \wedge \overline{y}$

$pred \leftarrow pred \vee front$

$bount \leftarrow bounts \wedge \overline{front}$

return *pred*

Algorithm 6 Recursive method to find SCCs

procedure SCC_DECOMP_RECUR(v, B(v), N)

$F(v) \leftarrow forward\_set(v, B(v), N)$

if $F(v) \neq 0$ **then**

report $F(v)$ an SCC

else

report v non- SCC

$x \leftarrow F(v) \vee v$

$R \leftarrow B(v) \wedge \overline{x}$

$y \leftarrow$ FMD_PRED(x, R, N)

report y non-SCC

$R \leftarrow R \wedge \overline{y}$

$IP \leftarrow \exists_Y (y \vee x)(Y) \wedge N(X,Y) \wedge R(X)$

while $R \neq 0$ **do**

$v \leftarrow random\_take(IP)$

$B(v) \leftarrow backward_set(v, R, N)$

$SCC\_DECOMP\_RECURE(v, B(v), N)$

$R \leftarrow R \wedge \overline{v \vee B(v)}$

$IP \leftarrow IP \wedge \overline{v \vee B(v)}$

5.5 Finding fair paths

As explained in the previous section, in order to check the model, we need to assert that there is a fair path in the product. In order to do so, we need to check that each of the fairness constraints is satisfied in one of the strongly connected components we found earlier. This will tell us if there is a fair SCC in the graph. If no such SCC exists, we can conclude that no fair path exists in the graph. Otherwise, we continue to check that this SCC is reachable from one of the initial states in the graph.

The logic described above is implemented in the module **FairPathFinder**. We use the method **find_fair_path** that returns a python generator with possible initial states from which there exist fair paths.

6 Experiments and Results

7 Further Work