

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

In this project, we implement a symbolic LTL model checker, using the reduction to the fair-CTL verification problem. We compare between the CTL model checker and the LTL model checker and show that the LTL behaves exponentially in the size of the model. We also extend the original tableau definition to test the impact on simple formulas that use the Global and Eventually quantifiers.

2 Introduction

Throughout the course we studied LTL and CTL model checking and algorithms used to solve this problem. Though we thoroughly learned the theoretical approach to the model checking problem and the algorithms used to solve them, we did not go into detail regarding the verification tools available to solve these problems. One verification tool that we came across during the course was the SMV model checker and its expansion to LTL formulae found in [2]. Although the SMV model checker is a very useful tool for model checking, it is written in a pseudo-code language that is not very common to most programmer.

In our project we created a new model checker in *python* that allows verification of both temporal logics LTL and CTL. The model checker we implemented is symbolic in order to be efficient and uses the algorithms shown in class for model checking. Our intention was to create a model checker that is both efficient and easy to use and expand. We have developed the relevant modules to simply translate a logical model M into a symbolic representation and use this symbolic representation in order to validate LTL and CTL formulae. We tested our model checker on the snooping cache coherence protocol as found in [3].

In this paper we describe the relevant algorithms and theory needed to understand the implementation of our model checker. We also describe the main modules of the model checker and the results it achieved on the cache coherence model.

The entire implementation of this project can be found at <https://github.com/olifshitz/AutomaticVerification.git>

3 CTL and LTL Model Checking

Definition of CTL* logic syntax and the subclasses of LTL and CTL can be found in the class notes and [2]

3.1 Symbolic CTL Model Checking

The entire implementation of our project is based on the usage of reduced, ordered binary decision diagrams (ROBDDs). BDDs are a canonical form representation of boolean formulas[1].

3.2 LTL Model Checking Using Tableaus

The theoretical definition and explanation of LTL model checking using tableaux is found at the class notes and [2]

4 Implementation

In our project we implemented a symbolic CTL and LTL model checker in python using python. Throughout our entire implementation we used the python module **pyeda** which allows easy use of BDDs, as explained in the previous section. In the following sections we present the main modules in our implementation and how they are used to create the model checker.

4.1 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, \dots, 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.

4.2 Tableau construction

The next module we implemented is a module in charge of creating the tableau for a given LTL formula. 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 later 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_bdds)** (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 satisfaction 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.

4.3 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 implementation 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.

4.4 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 our project we used a symbolic algorithm for finding SCCs in a graph, taken from [4]. A detailed explanation of this algorithm can be found at that article.

4.5 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.

4.6 CTL model checking

In order to implement CTL model checking we implemented the CTL model checking algorithms shown in class when solving the find the set of nodes satisfying the temporal quantifiers *EX*, *EG* and *EU*.

When implementing these algorithm we used many of the methods already described in

the previous section; mostly the modules that are relevant to reachability analysis. For example, when solving EX we used the successor method described earlier in order to find all the successor of a given node.

5 Extending the Tableau defintion

The definition of the Tableau construction is based on the assumption the only temporal quantifiers used in the formula where X and U . That makes the definition simpler, but actual implementing this construction with those restrictions could actually effect the computation process.

In our project we extended the definition and construction of the Tableau to handle formulas with the F and G quantifiers. Using the equivalences $Fa \equiv true \ U \ a$ and $Ga \equiv \neg F \neg a$ we can obtain the new construction steps needed to extend the tableau without hurting the correctness of the model.

At the first step, we add the new elementary formulas of the form $XF a$ and $XG a$ and define:

- $el(Fa) = \{XF a\} \cup el(a)$
- $el(Ga) = \{XG a\} \cup el(a)$

Using the equivalences above we extend the sat definition and the fairness constriant set. the tableau realtion table definition stays without change (but now we need to consider the new elementary formulas):

- $sat(Fa) = sat(true \ U \ a) = sat(a) \cup (sat(true) \cap sat(X(true \ U \ a))) = sat(a) \cup sat(XF a)$
- $\frac{sat(Ga)}{sat(XF \neg a)} = \frac{sat(\neg F \neg a)}{sat(a) \cap sat(\neg XF \neg a)} = \frac{\overline{sat(F \neg a)}}{sat(a) \cap sat(X \neg a)} = \frac{\overline{sat(\neg a) \cup sat(XF \neg a)}}{sat(a) \cap sat(X \neg a)} = \frac{\overline{sat(\neg a)}}{sat(a) \cap sat(XG a)}$

Two more classes of fairness contrainsts now add to the fairness constraint set: For every subformula Fg , we add the fairness constraint

$$\{sat(\neg(gUh) \vee h) \mid gUh \text{ in } f\}$$

$$\{sat((Ga) \vee a) \mid Ga \text{ in } f\}$$

$$\{sat(\neg(Fa) \vee a) \mid Fa \text{ in } f\}$$

6 Experiments and Results

6.1 Read-Write Resource Agreement

We tested our model checker on a read-write resource management model with multiple processors. Each processor is in one of the next states at each time: *Owned* for when it can write to the resource, *Shared* for when it can only read from the resource, and *Invalid* for when the resource is not accessible for it. A processor can also be *waiting*, *snooping* or none of those.

The protocol we test assume all processors can send messages to each other processor, only one processor can send a message each time, and all processor get the messages in the order they were sent. Another way to look at this model is that each "turn" there is a single processor marked as *master* and it will be the one sends the message on that turn. The master is switched arbitrarily each turn.

There are four types of commands used in this model, each is actually a state-transition reason:

- *read – shared* - sent when a processor wants to escalate to get to the shared state
- *read – owned* - sent when a processor wants to escalate to get to the owned state
- *response* - sent when a processor has committed its changes and is ready to forfeit their owned privileges.
- *idle* - sent when the master processor wants to stall

Each processor can be described with two Kripke models, one for the turns it is the master and the command is sent by itself, and one for the turns where another processor is the master, and the command was received by the processor. Each of those models has 8 states: *shared*, *shared – waiting*, *shared – snooping*, *invalid*, *invalid – snooping*, *owned*, *owned – waiting*, and a *sink* state, that indicates something unexpected has happened.

The model for master states (send the message) is described in figure 1. The model for non-master processor is described in figure 2. In the last one, any command that is shown indicates a transition to the sink state (the master model does not need this specification because any command not shown there is just illegal).

6.2 Testing the single processor

Combining those two models using 15 states (each state has two versions - master and non-master, and another state to represent the sink state) we get a new Kripke model that represent all transitions a processor can use in its run.

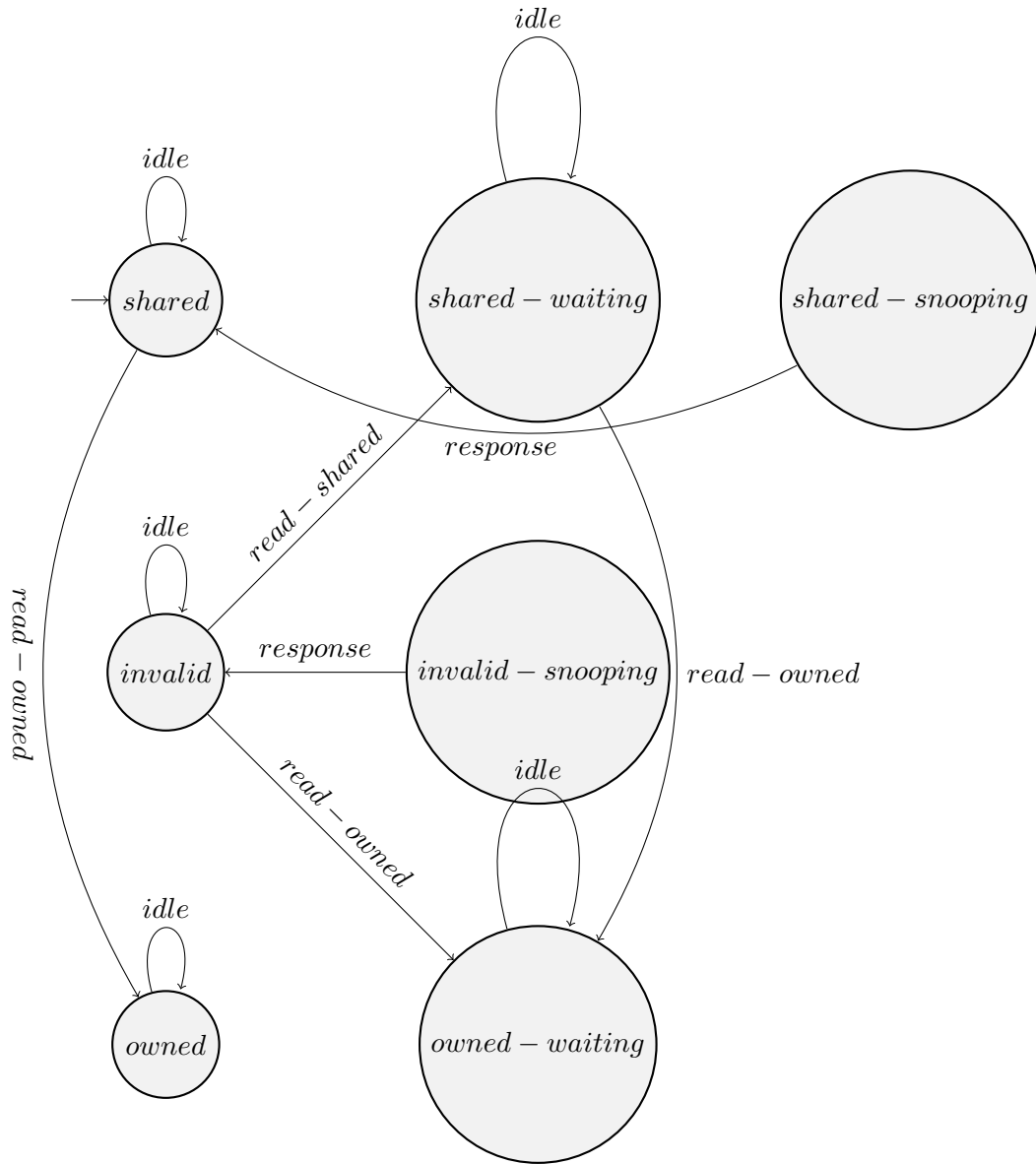


Figure 1: State transition for master processor (send a message)

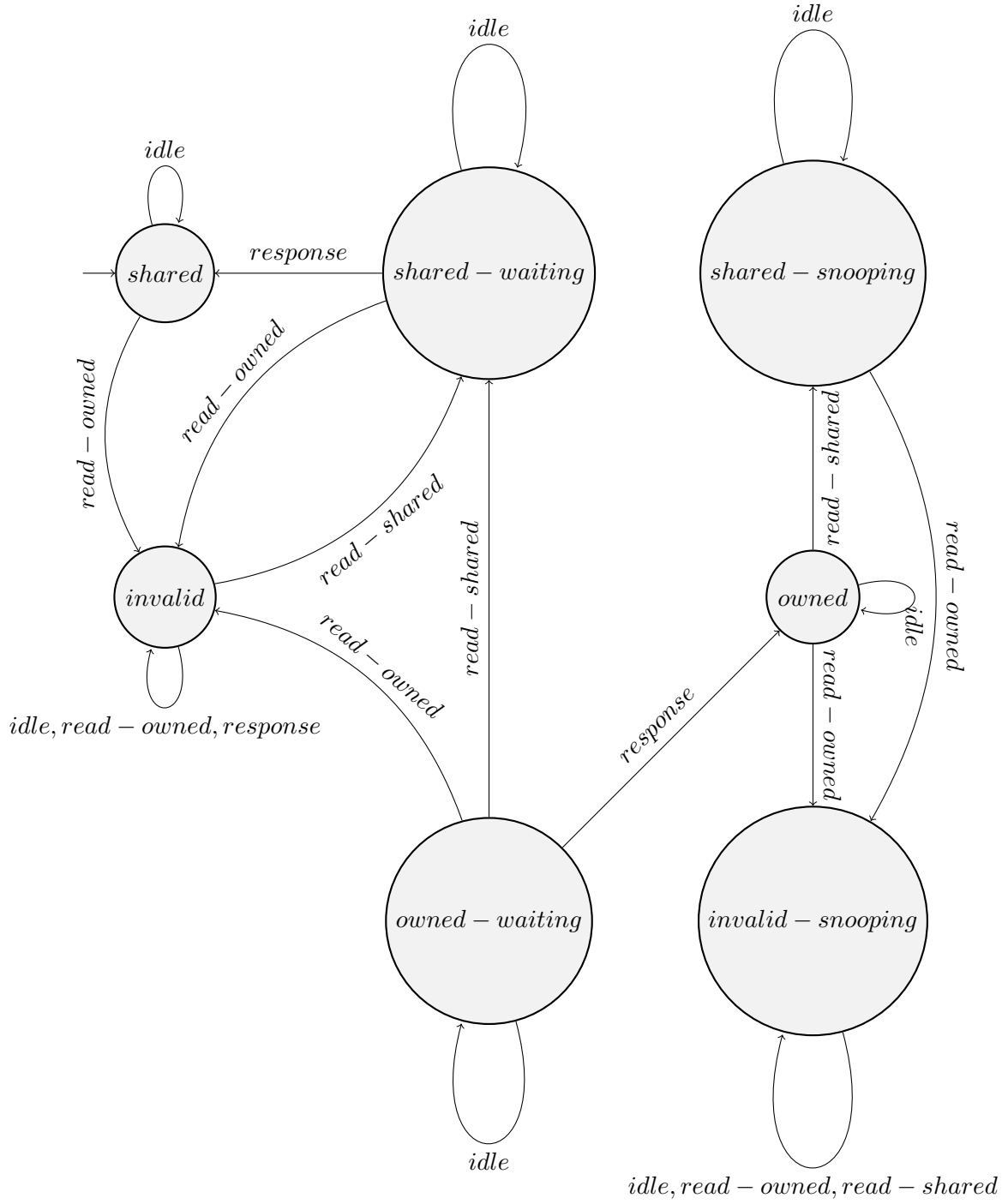


Figure 2: State transition for non-master processor (receive a message)

To make things easier we consider every processor to be in the shared state in the beginning, which means, as there must be one and only master, we have one state to consider as the initial state. At that point we started running the CTL and LTL model checkers on the next examples:

- liveness - at every state, it is possible for any model to get to a state where it owns the resource, and to a state where the resource is shared with him:

$$AG(EF(owned)EF(shared))$$

- safety - the sink state is unreachable:

$$AG(\neg sink)$$

- starvation - when a processor wants to own K , it would own eventually:

$$AG(owned_waiting \Rightarrow AF(owned))$$

$$AG(owned_waiting \Rightarrow F(owned))$$

Those three examples of formulas give us 3 CTL formulas and 2 LTL formulas. Table 2 presents the basic results obtained from running both model checkers on those 5 examples. Both checkers simplified the formulas so the LTL checker only use the U and X quantifier, and the CTL checker only uses EX , EU and EG .

The main point to notice is that the formula complexity has almost no impact on the CTL model checker. On the other hand, as the formula gets complicated, its tableau grows and the size of the model the LTL model checker has to verify grows as well. That takes much more time, as expected.

Property	Class solver	seconds	size of BDD
Liveness	CTL	0.3	80
Safety	CTL	0.32	80
Starvation	CTL	0.33	80
Safety	LTL	175	210
Starvation	LTL	435	272

Table 1: Basic results of formula checking on single processor model

6.3 Testing multiple processors

At that point, we wanted to test each of those formulas on a model of more than one processor. To generate a model that represent this protocol with more than one processor, we only need to *multiply* two single processor models - that is the model whose the state set is the cartesian product of the original models, and a relation between two states exists \Leftrightarrow *thecorrespondingrelationsexistintheoriginalmodels*.

This is almost true, except we have to make sure the following:

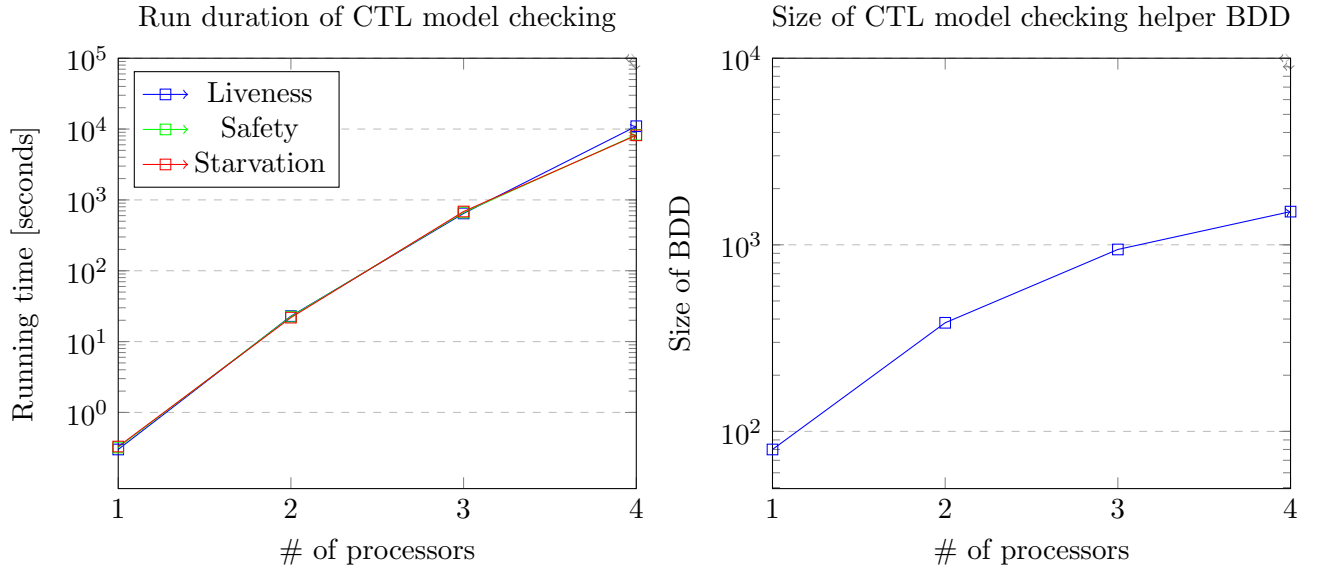


Figure 3: CTL model checking run stats on models of various sizes

- At all point there must be a single processor who is the master at that point. Using the BDD for "this state is a master state" we can restrict the new model to have a single master in every state.
- The existence of the relations in the original model is not enough for the relation in the product model to exist - they also need to have the same result for existing, which means they have to refer to the same command (the previous master sent the command, and every other processor has received it). This is done by actual labeling the transitions with a specific BDD for each command. The BDD method guarantees that when the product model is created the labels would be taken into consideration, as needed.

The initial state would be, again any state for which every state is in the shared mode (but only one is the master). Then, we can test each formula on the prespective of a single node (is it guaranteed that no node is ever starved, etc.)

Figure 3 demonstartes how the size of the model (number of processors running) impacts on the the size of the problem (obviously) and thus takes much longer to solver using the simple CTL model cheker. The graph is drawn with log axis, as we expect the size of the model to increas exponentially with the number of processes in the model. Notice that the complexity of the formula does not impact the running duration of the CTL model checker.

6.4 Testing the LTL model checker

As described in Table 1, the LTL checking computation time is significantly longer than the CTL checker. Unfortunately, it was so long we could not get any result for models with 3 processors or more. The root of the computation, the SCC finder algorithm, took many hours to iterate over the result product model. In this section we will show and briefly discuss the results for checking the models with single and two processors.

The results shown here also compare between the LTL checker with and without simplifying the LTL formulas so they only use the X and U quantifiers. Specifically, the formulas we tested on were:

- Safety
 - (1) $AG(\neg sink)$ as oppose to (1*) $A(\neg(true\ U\ \neg(\neg sink)))$
- Starvation
 - (2) $AG(owned_waiting \Rightarrow F(owned))$
as oppose to
(2*) $A(\neg(true\ U\ \neg(owned_waiting \Rightarrow true\ U\ (owned))))$

On section 4.7 we described the actual change in the algorithm needed to compute the tableau for formulas containing the G and F quantifiers. Considering the equivalences $Fa \equiv true\ U\ a$ and $Ga \equiv \neg F\neg a$, theoretically, one can change the appearance of each XF and XG elementary formulas with a XU elementary function, and every fairness constraint with the new kind of fairness constraint corresponding to the temporal quantifiers.

The conversion could help though, mostly because it demands much less computation steps - for example, each computation of sat is completed a bit faster, and even the actual formula parsing could be faster (it is much easier to analyze a unary operator string). On figure 4 we show the size of BDD needed to present the product of the formula tableau and the model with n processors. Table 2 shows the computation time needed to check the formulas with the model of 1 and 2 processors.

Another point to notice is that the size of the model (and the size of the final product as a result) does not grow exponentially perfect. This is because the size of the model is exactly $|Single\ processor\ model|^n$, as after the product processing we lose most of the nodes, as there should be one and only master at each state consider in the multiple processors model.

As mentioned above, it is clear the computation of the short versions is much faster, even though the size of the tableau stayed about the same (as expected). This mostly shows that the overhead of checking the model with the formula takes significant time of the actual computation. Additionally, The number of SCC in the product model stays exactly the same between the two versions of formulas. This is because, as established before, the tableau structure stays the same, even after using the extended form.

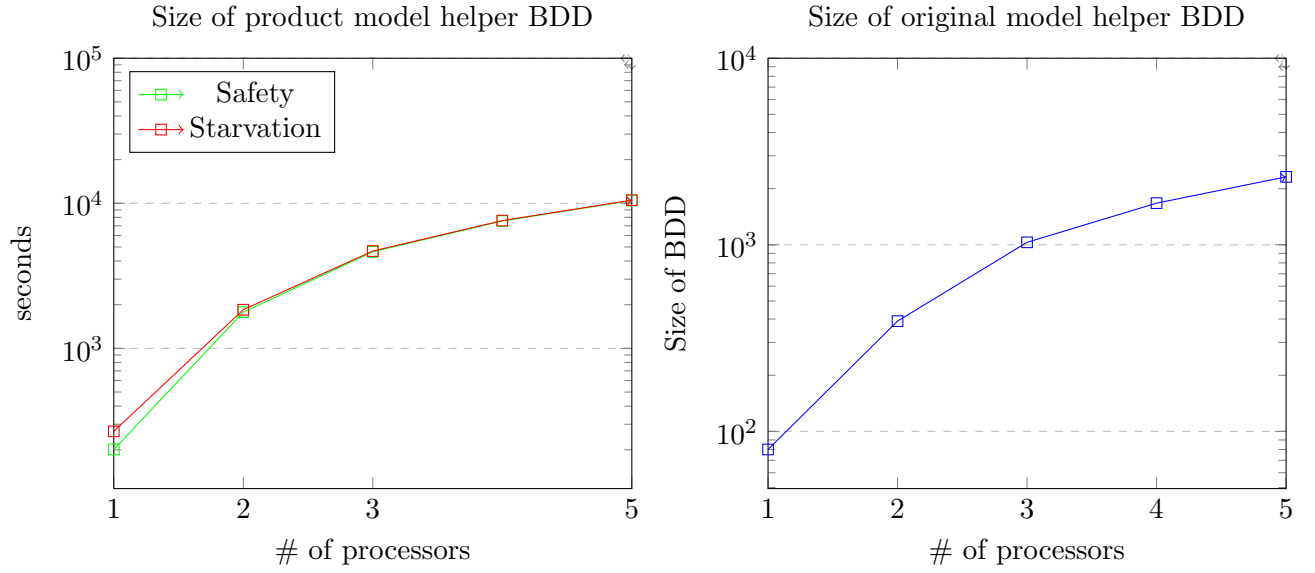


Figure 4: State transition for non-master processor (receive a message)

Formula	# of processors	seconds	size of tableau BDD	# of SCC found
Safety				
(1)	1	92	215	10
(1*)	1	175	207	10
(1)	2	7925	1768	105
(1*)	2	15459	1773	105
Starvation				
(2)	1	222	261	17
(2*)	1	435	251	17
(2)	2	16327	1844	196
(2*)	2	34098	1830	196

Table 2: Results of LTL formula checking on single and two-processor models

7 Further Work

We were surprised by that result - we expected the computation time to be about the same on the simplified. By our analysis this is caused mostly because the model building procedure has a significant amount of overhead that is reduced when the formula is less complicated. We tried to reduce that overhead using profiling and improving the points of failure in our implementation, without any significant improvements. This is also interesting because it raises the question how else can we improve the computation so that the LTL model checking is faster. On the other hand, we have established (again) that the LTL model checking is exponential with the size of the model, and that would stay the same, no matter what we find.

References

- [1] Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.
- [2] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at ltl model checking. *Formal Methods in System Design*, 10(1):47–71, Feb 1997.
- [3] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.
- [4] Aiguo. Xie and P A. Beerel. Implicit enumeration of strongly connected components. pages 37–40, Nov 1999.