EPiC Series in Computing

Volume 45, 2017, Pages 107–119

SCSS 2017. The 8th International Symposium on Symbolic Computation in Software Science 2017

A Parallel Construction of the Symbolic Observation Graph: the Basis for Efficient Model Checking of Concurrent Systems

Hiba Ouni[1][2], Kais Klai[3], Chiheb Ameur Abid[1][2], and Belhassen Zouari[2]

[1] Faculty of Sciences of Tunis, University of Tunis El Manar, 2092, Tunis, Tunisia
[2] Mediatron Lab, Higher School of Communications of Tunis, University of Carthage, Tunisia
[3] University of Paris 13, Sorbonne Paris Cité CNRS UMR 7030 LIPN

ouni.hiba@gmail.com
kais.klai@lipn.univ-paris13.fr
chiheb.abid@fst.utm.tn
belhassen.zouari@supcom.tn

Abstract

Model checking is a powerful and widespread technique for the verification of finite distributed systems. It takes as input a formal model of a system and a formal specification (formula) of a property to be checked, and states whether the system satisfies the property or not. Since it is based on state space traversal algorithms, the model checking approach suffers from the well known state space explosion problem. Indeed the space (and consequently the time) requirements increase exponentially with the size of the models. One way to deal with this problem is symbolic model checking. It aims at checking the property on a compact representation of the system by using Binary Decision Diagram (BDD) techniques. Another way is to parallelize the construction/traversal of the state space on multiple processors. In this paper, we combine the two mentioned approaches by proposing an efficient multi-threaded algorithm for the construction of the so called Symbolic Observation Graph (SOG). It is a hybrid structure where the transitions of the system are divided into observed and unobserved ones. The nodes of this graph are then defined as sets of states linked with unobserved transitions (and encoded symbolically with a BDD) and edges are labeled with observed transitions only (and represented explicitly). The basic idea is that each thread owns one part of the SOG construction. We measured the runtime of the parallel SOG construction algorithm on several models, and the obtained results are very competitive. The preliminary evaluations we have done on standard examples show that our method outperforms the sequential method which makes it attractive.

1 Introduction

Model checking [6, 7] is a powerful and widespread technique for the verification of concurrent systems. Given a (usually finite-state) formal description of the system to be analysed and a number of properties, often expressed as formulas of temporal logic, that are expected to be satisfied by the system, the model checker either confirms that the properties hold or reports that they are violated. In the latter case, it provides a counterexample: a witness run that shows that the property is violated. Such a run gives a valuable feedback and points to design errors.

At the core of model checking are algorithms that implement state space traversals. The reachable state space is traversed to find error states that violate safety properties, or to find cyclic paths on which no progress is made as counterexamples for liveness properties. The main drawback of the model checking approach is the well-known problem of combinatory state space explosion. Indeed, state space size increases exponentially with the number of components of the concurrent system. During the last three decades, numerous techniques have been proposed to cope with the state space explosion problem in order to get a manageable state space and to improve scalability of model checking. Partial order approaches (e.g., [14, 30]) exploit the fact that interleaving concurrent actions are equivalent, and only a representative interleaving needs to be explored, leading to a significant reduction of the constructed state space. Modularity is also used by limiting the exploration of state space to the parts that concern the property to check [21, 26, 29]. Symbolic techniques [4, 13, 15], on the other hand, represent the state space in a compressed manner. Indeed, transition relation and reachable states are manipulated as boolean functions. These functions can be represented compactly by decision diagrams such as BDD (Binary Decision Diagrams) [1, 3], or by their extension such as MDDs (Multi Valued Decision Diagrams). Another way to reduce the state space explosion problem is to parallelize the construction and/or the traversal of the state space. Parallel approaches can be classified into two main categories: multi-core approaches [17, 11] where parallelization is performed on several cores on the same machine, and cloud approaches [18, 19] where model checking is distributed on multiple machines in the cloud. In [12], a parallel construction of a state space for Model checking is performed by partionning it into several nodes and then merging it to achieve verification. The partitioning is performed by adopting a static scheme in order to avoid the potential communication overhead occurring in dynamic load balancing schemes through the use of an adequate hash function. In [25], a coordinator process is introduced being responsible for distribution of states and termination detection. Several techniques for reductions are also used alongside parallelization such as partial order methods or by using symbolic representation of state spaces such as BDD [31]. The latter approach is based on the parallelization of the BDD operations [33, 9] while the construction of the whole graph is kept sequential. In [32], efficient parallelization algorithms of BDDs operations as well as MDDs operations are proposed.

In this work, we propose a multi-threaded algorithm for constructing the SOG [15]. The SOG is a reduced deterministic graph where nodes represent markings linked by unobservable transitions in the formula, whereas its edges represent the firing of observable transitions that appear in the LTL/X formula. Nodes of the SOG are represented by BDDs allowing to obtain a reduced graph representation of the state space. Such an obtained graph represents an abstraction of the system on which the verification of a LTL/X property is equivalent to the verification on the original reachability graph. It has been established (see e.g.,[15, 22, 10]) that the SOG-based approach can outperform purely symbolic techniques in model checking of LTL properties. In this work, we adopt a dynamic load balancing scheme in order to balance the load on threads sharing the SOG construction task. Further, MDDs are used instead of BDD to reduce more the size of the symbolic observation. Indeed, since MDD are a generalization of BDDs to the integer domain, a node in a MDD may represent several nodes in a BDD. In general, graphs based on MDDs are more reduced than those based on BDDs [27].

This paper is organized as follows. In Section 2 , we introduce preliminary definitions as well as some useful notations. In Section 3, we formally define the symbolic observation graph. In Section 4 we present our proposed multi-threaded algorithm for the construction of the symbolic observation graph. In Section 5, we illustrate the technical aspects. In Section 6, we evaluate our method on a benchmark of well-known parameterized problems. Finally, Section 7 summarizes the main conclusions and perspectives of this work.

2 Background

The technique presented in this paper applies to different kinds of models, that can map to labeled transition systems, e.g. Petri nets. For the sake of simplicity and generality, we chose to present it for labeled transition systems (LTS for short), since the formalism is rather simple and well adapted to event-based logic which is adopted in this work.

2.1 Labeled transition system

A labeled transition system (LTS for short) consists of a set of states and a set of transitions between those states. These transitions are labelled by actions, one state is designated as the initial state and a (possibly empty) subset of states represents the final states. An example of LTS is given in Figure1 where s_0 is the initial state and s_6 and s_7 are the final states.

Definition 1 (labeled transition system)**.**
A labeled transition system is a 5-tuple $\langle \Gamma, Act, \rightarrow, I, F\rangle$ *where:*

- Γ *is a finite set of states;*
- Act *is a finite set of actions;*
- $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ *is a transition relation;*
- $I \subseteq \Gamma$ *is a set of initial states;*
- $F \subseteq \Gamma$ *is a set of final states.*

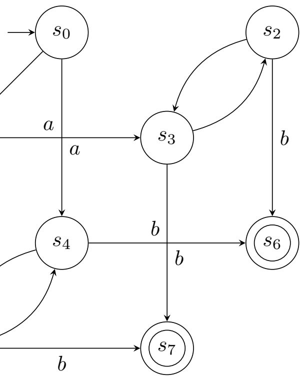

Figure 1: Example of an LTS

The LTS can be be represented either, explicitly (each state/arc is individually represented in memory), or symbolically (sets of states can share some data in memory) using BDD/MDD.

A mixed approach (hybrid) exists where states are encoded symbolically while edges are represented explicitly. The SOG is an example of the latter representation.

2.2 Binary decision diagram

Binary decision diagram (BDD) [3] is a directed acyclic graph expressing the Shannon decomposition of a Boolean function. BDDs can be used efficiently to store sets of states in symbolic model checking.

Definition 2 (Binary decision diagram).
An (ordered) BDD is a directed acyclic graph with the following properties:

1. *There is a single root node and two terminal nodes 0 and 1.*
2. *Each non-terminal node p has a variable label x_i and two outgoing edges,labeled 0 and 1; we write $lvl(p) = i$ and $p[v] = q$, where $v \in \{0, 1\}$*
3. *For each edge from node p to non-terminal node q, $lvl(p) < lvl(q)$.*
4. *There are no duplicate nodes, i.e., $\forall p \forall q (lvl(p) = lvl(q) \wedge p[0] = q[0] \wedge p[1] = q[1]) \rightarrow p = q$.*

2.3 Multi-valued decision diagram and List decision diagram

Multi-valued decision diagrams[32] (MDDs, also called multi-way decision diagrams) are a generalization of BDDs to the integer domain.

Definition 3 (Multi-valued decision diagram).
An (ordered) MDD is a directed acyclic graph with the following properties:

1. *There is a single root node and terminal nodes 0 and 1.*
2. *Each non-terminal node p has a variable label x_i and n_i outgoing edges, labeled from 0 to $n_i - 1$; we write $lvl(p) = i$ and $p[v] = q$, where $0 \leq v < ni$.*
3. *For each edge from node p to non-terminal node q, $lvl(p) < lvl(q)$.*
4. *There are no duplicate nodes, i.e., $\forall p \forall q (lvl(p) = lvl(q) \wedge \forall v, p[v] = q[v]) \rightarrow p = q$.*

List decision diagrams (LDDs) can be understood as a linked-list representation of quasi-reduced MDDs. LDDs were described in [2].

Definition 4 (List decision diagram).
A List decision diagram (LDD) is a directed acyclic graph with the following properties:

1. *There is a single root node and two terminal nodes 0 and 1.*
2. *Each non-terminal node p is labeled with a value v, denoted by $val(p) = v$, and has two outgoing edges labeled = and ¿ that point to nodes denoted by $p[x_i = v]$ and $p[x_i > v]$.*
3. *For all non-terminal nodes p, $p[x_i = v] \neq 0$ and $p[x_i > v] \neq 1$.*
4. *For all non-terminal nodes p, $val(p[x_i > v]) > 1$.*
5. *There are no duplicate nodes.*

3 Event-Based Symbolic Observation Graph

The SOG [22, 24, 15] is an abstraction of the reachability graph of concurrent systems. The construction of the SOG is guided by the set of actions occurring in the formula to be checked. Nodes of the SOG are called aggregates and may be represented and managed efficiently using decision diagram techniques (BDDs). Despite the exponential theoretical complexity of the size of a SOG (a single state can belong to several aggregates), the SOG has a very moderate size in practice. This is due to the small number of actions in a typical formula on one hand, and to the efficiency of the BDDs' structure for representing and manipulating sets of states, on the other hand. In general, the moderate size of a SOG makes the time complexity of its verification negligible compared to its building time.

In this section, we first define formally what an *aggregate* is, before providing a formal definition of a SOG associated with an LTS and a set of observed actions. An aggregate is defined as below.

Definition 5 (aggregate)**.**
Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I, F\rangle$ *be a labeled transition system with* $Act = Obs \cup UnObs$*. An* aggregate *is a tuple* $\langle S, f\rangle$ *defined as follows:*

1. S *is a nonempty subset of* Γ *satisfying:* $s \in S \Rightarrow Sat(s) \subseteq S$*;*
   $(Sat(s) = \{s' \in \Gamma \mid s \xrightarrow{*}_{UnObs} s'\})$
2. $f \in \{true, false\}$*;* f*=true iff* $S \cap F \neq \emptyset$*,*

In the following, we provide two definitions of the SOG. The first defines deterministic SOGs which are easier to understand. The second, which we will actually use later, is a useful non-deterministic generalization.

Definition 6 (Deterministic Symbolic Observation Graph)**.**
The deterministic symbolic observation graph dSOG($\mathcal{T}$) *associated with an LTS* $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F\rangle$ *is a five-tuple* $\langle \mathcal{A}, Act', \rightarrow', I', F'\rangle$ *as follows:*

1. $\mathcal{A}$ *is a finite set of aggregates with:*
   (a) *There is an aggregate* $a_0 \in \mathcal{A}$ *s.t.* $a_0.S = Sat(I)$*;*
   (b) *For each* $a \in \mathcal{A}$ *and for each* $o \in Obs$*, if* $\exists s \in a.S, s' \notin a.S \colon s \xrightarrow{o} s'$ *then* $Sat(\{s' \notin a.S \mid \exists s \in a.S, s \xrightarrow{o} s'\})$ *equals* $a'.S$ *for some aggregate* a' *and* $(a, o, a') \in \rightarrow'$*;*
2. $Act' = Obs$*;*
3. $\rightarrow' \subseteq \Gamma' \times Act' \times \Gamma'$ *is the transition relation, obtained by applying 1b;*
4. $I' = \{a_0\}$*;*
5. $F' = \{a \in \Gamma' \mid a.S \cap F \neq \emptyset\}$*.*

The deterministic SOG can be constructed by starting with the initial aggregate a_0 and iteratively adding new aggregates as long as the condition of 1b holds true.

The following more general symbolic observation graph additionally supplies a certain flexibility in the construction of aggregates. Now an aggregate can have two outgoing arcs, leading two different successors, labeled by the same observed action. Consequently, the set of 1b is replaced by disjoint subsets. Clearly, this construction is not unique. One can take advantage of such a flexibility to obtain smaller aggregates. Even if the obtained SOG would have more aggregates, it would consume less time and memory. This definition generalises the one given in [23]. The construction algorithm given in [15] is an implementation where the obtained graph is deterministic.

Definition 7 (Symbolic Observation Graph)**.**
The symbolic observation graph SOG($\mathcal{T}$) *associated with an LTS* $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I, F\rangle$ *is a five-tuple* $\langle \mathcal{A}, Act', \rightarrow', I', F'\rangle$ *such that:*

1. $\mathcal{A}$ *is a finite set of aggregates s.t.:*
   (a) *There is an aggregate* $a_0 \in \mathcal{A}$ *s.t.* $a_0.S = Sat(I)$*;*
   (b) *For each* $a \in \mathcal{A}$ *and for each* $o \in Obs$ *the set* $\{s' \notin a.S \mid \exists s \in a.S, s \xrightarrow{o} s'\}$ *is not empty if and only if it is a pairwise disjoint union of nonempty sets* $S_1 \dots S_k$ *and for* $i = 1 \dots k$*, there is an aggregate* $a_i \in \mathcal{A}$ *s.t.* $a_i.S = Sat(S_i)$ *and* $(a, o, a_i) \in \rightarrow'$*;*
2. $Act' = Obs$*;*
3. $\rightarrow' \subseteq \Gamma' \times Act' \times \Gamma'$ *is the transition relation, obtained by applying 1b;*
4. $I' = \{a_0\}$ *(s.t.* $a_0.S = Sat(I)$*);*
5. $F' = \{a \in \Gamma' \mid a.S \cap F \neq \emptyset\}$.

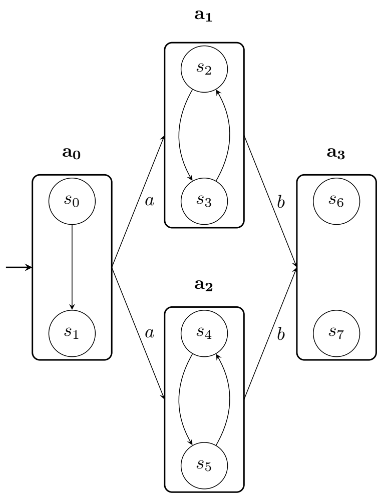

Figure 2: A SOG with Obs={a,b}

Figure 2 illustrates the SOG associated with the LTS of Figure 1. The set of observed actions contains two elements $\{a, b\}$, unlabeled edges are supposed to be labeled by non observed action. The presented SOG consists of 4 aggregates $\{a_0, a_1, a_2, a_3\}$ and 4 edges. Aggregate a_3 has two final states s_6 and s_7. Notice that states of the LTS are partitioned into aggregates which is not necessarily the case in general (i.e. a single state may belong to two different aggregates). Moreover, one can merge a_1 and a_2 within a single aggregate, or split a_3 into two different aggregates, and still respect Definition 7.

4 A parallel construction of the symbolic observation graph

4.1 Aims and Hypothesis

Our aim is to build the symbolic observation graph with a multi-threated approach. In this approach, we propose to create threads, such that each one builds a part of the SOG.

The basic idea is that each thread takes charge of the construction of some aggregates. The allocation of an aggregate with a thread is made according to its load (the number of aggregates to be processed). The number of threads is set at the beginning by the user and the same algorithm is executed by all threads. For this purpose, we propose to use a shared memory, mainly used to store the graph and identify the termination. The advantages of shared memory for all parallel threads is that there is no communication overhead. The disadvantage of this method is that concurrent threads can lead to unexpected behavior. To avoid concurrent writes on sensitive data mutual exclusions (mutexes) around critical sections must be used [8]. As perspective, we plan to perform model checking of LTL properties under the same distributed configuration. Also, we want to apply the approach of distributed verification to CTL properties.

4.2 A Multi-Threaded Algorithm

The general idea of the multi-threaded algorithm is as follows. The algorithm 1 builds the symbolic observation graph in a multi-threaded setting of a given LTS where its observable and unobservable transitions are specified. The same algorithm is executed by all threads. Each thread is identified by its identifier *idthread*. The table *Load* indexed by threads allows to store the current loads of threads. In order to distribute the work among the processes, the SOG is partitioned into several parts, using a function that computes the loads. The load of a thread is defined by the number of aggregates to be processed by the thread.

The table $Termination$, indexed by threads, is used to detect when the construction of the SOG has been terminated. Indeed, when a thread id has no aggregate to deal with, $Termination[id]$ takes the value $true$, else it takes $false$. Further, we associate with each thread id a stack $Waiting_{id}$ containing the aggregates to be processed. In the beginning, the initial thread, computes the initial aggregate (lines 3-8) from the initial marking by firing unobservable transitions and inserts it into the SOG. This aggregate is also inserted into the stack of the initial thread in order to build its successors. Thus, the load of the initial thread is incremented by one as its stack contains one element to be processed.

Then, every thread operates as a loop until the whole SOG is built, i.e. when all threads have no aggregate to deal with. In each iteration, a thread pops an aggregate from its stack and decrements its loading. It builds the successors of the popped aggregate by firing observable enabled transitions. If any successor does not exist in the SOG, it is inserted in the SOG and pushed into the stack associated with the thread having minimum load. Else, only edges connecting the popped aggregate and the existing aggregate labelled with the fired observable transition, are inserted into the SOG. It is worth noting that shared variables are locked with mutexes in order to prevent concurrent threads to update the same shared variables in the same moment.

We now prove the correctness of the parallel algorithm 1, assuming the sequential algorithm is correct [20].

```
   Data: $LTS\langle \Gamma, Obs \cup UnObs, \rightarrow, I, F \rangle$
   Result: $SOG\langle \Gamma', Obs, \rightarrow', I', F' \rangle$
1  Load: Table[1,..,M] integer;
2  Termination: Table[1,..,M] of boolean;
3  if idthread==1 then
4      $M_0$=MetaState(I); $Waiting_{idthread} = \{M_0\}$;
5      Load[idthread]←1; Termination[idthread]←false;
6  else
7      $Waiting_{idthread} = \emptyset$
8      Load[idthread]←0; Termination[idthread]←true;
9  while DetectTermination==false do
10     while $Waiting_{idthread} \neq \emptyset$ do
11         Termination[idthread]←false; Load[idthread]←Load[idthread]-1;
12         Choose M ∈ $Waiting_{idthread}$;
13         foreach $a \in Obs$ do
14             if enabled(M.S,a) then
15                 S'← succ(M.S,a); M'=MetaState(S');
16                 if ∃ M" tq M'=M" then
17                     arc(M,a,M');
18                 else
19                     $\Gamma \leftarrow \Gamma \cup \{M'\}$; arc(M,a,M');
20                     mutex lock[j]
21                     j=minCharge(); $Waiting_j \leftarrow Waiting_j \cup \{M'\}$;
22                     Load[j]←Load[j]+1;
23                     mutex unlock[j]
```

Algorithm 1: A Multi-Threaded algorithm for constructing symbolic observation graph

Proposition 1. *Let G be a Symbolic Observation Graph associated with a labeled transition system T and generated by our parallel algorithm 1. G respects definition 7.*

First, let S be the generated state space by our parallel algorithm 1. The building of state space is partitioned among the processes such that $S = \bigcup s_i, i \in \{0, .., N\}$. For each i, s_i follows directly the same arguments as in the case of sequential algorithm. On the other hand, to avoid concurrent access on sensitive data, we used mutual exclusions (mutexes) mechanisms. Also, the parallel algorithm terminates only when the parallel computation is finished and there are no more states to be explored, i.e. every stack associated with each thread is empty.

5 Technical aspects and Implementation

The SOG has been implemented using the BDD package BuDDy[1]. According to the documentation of this library, the BDDs share the common subgraphs to have more reductions. So even for the creation of the BDD associated with an aggregate, it is necessary to prevent the other threads from manipulating the BDDs.

[1]http://sourceforge.net/projects/buddy

It is worth noting that using sequential BDDs or MDDs implementation in distributed algorithms is not evident, since proposed packages use always a shared hash table to store all generated BDDs in order to provide a maximum reduction. Such a table is not thread safe, i.e. it does not allow several threads or processes to use it simultaneously.

There have been different initiatives to implement parallel BDDs on multicore machines. More recently, there are some works that have tried to propose parallel BDD or thread safe implementations. In [16], a parallel implementation of the BuDDy package is proposed but it is limited to only some operations in order to demonstrate the applicability and efficiency of Cilk++ in this domain. Also, in a thesis on JINC [28] a multi-threaded extension is described for BDDs by using several hash tables to store BDDs. Further, it does not parallelize the basic BDD operations. Sylvan[2] [32] is a parallel (multi-core) MTBDD library. Sylvan implements parallelized operations on BDDs, MTBDDs and LDDs by using a lockless one hash table. Their results are promising. Compared to BuDDy, Sylvan have better performance when using multiple workers and lower performance when using one worker [31]. Since, the results in [32] show that the majority of models especially large models, are performed up to several orders of magnitude faster using LDDs, we used and adapted the LDD extension of Sylvan to implement our algorithm. Indeed, we have mainly used the lockless hash table with sequential basic BDD operations. Such a table is thread safe and consequently well adapted to our approach.

6 Experimental results

In the current section, we present and evaluate the results of our experiments. We executed the multi-threaded algorithm 1 on various models, in order to measure the performance of the parallel construction of the SOG. All the tested examples are parameterized and the size of the reachable states space is exponential with respect to the parameter value.

The technique presented in this paper applies to various kinds of process models. But, in our implementation, we take as input Petri net models.

The measurements presented in Table 1 concern two families of Petri nets. The Petri nets of a given family are obtained by instantiating a parameter (e.g. the number of philosophers). All these examples are taken from [5].

Model	Size	Buddy	Sylvan	Th3	Th6	Th9	Th12	Sp
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
ring3	504	0,14	0,21	0,14	0,08	0,05	0,06	3,5
ring4	5136	1.04	4.76	1.88	0.91	0.78	0.79	6
ring5	53856	146.37	103.57	43.03	16.91	13.40	13.87	8
ring6	575296	1140	1981.8	757.68	339.5	255.17	256.55	7.7
philo5	1364	0,1	0,17	0,1	0,05	0,07	0,11	1.6
philo6	5778	0.32	1.00	0.43	0.22	0.20	0.25	4
philo8	103682	6,89	27,75	12,08	5,22	3,93	4,71	5.9
philo10	1.86×10^6	125,01	743.93	373,15	128,34	103,05	115,06	6,5

Table 1: Experimental results

Table 1 summarizes the results for different representative models. For each net, we give its number of reachable markings (column numbered (1)), These experiments are based on a

[2]https://github.com/utwente-fmt/sylvan

sequential BDD package (BuDDy) and a parallel package (Sylvan). First, we have measured the time in seconds consumed by the construction of SOG in a sequential way using BuDDy (2) and Sylvan(3). Then, we measured the runtime of our multi-threaded algorithm (algorithm 1) by progressively increasing the number of threads. Furthermore, we are interested in the speedup of the building of the SOG(8). The speedup is a measure for the performance gain of parallelizing an algorithm and it is calculated relative to 1 thread.

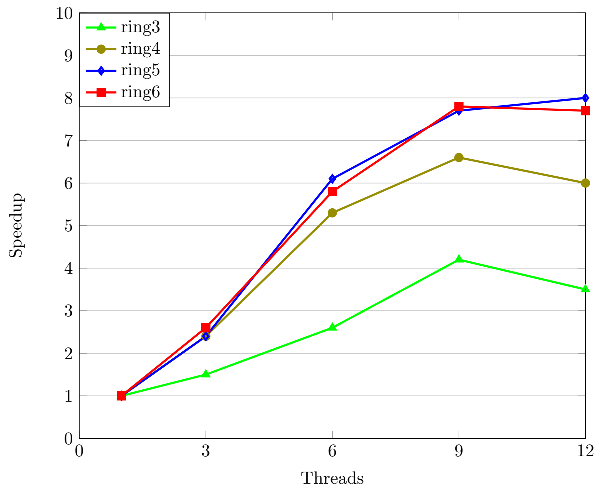

Figure 3: Speedups of construction of SOG (ring3,4,5 and 6)

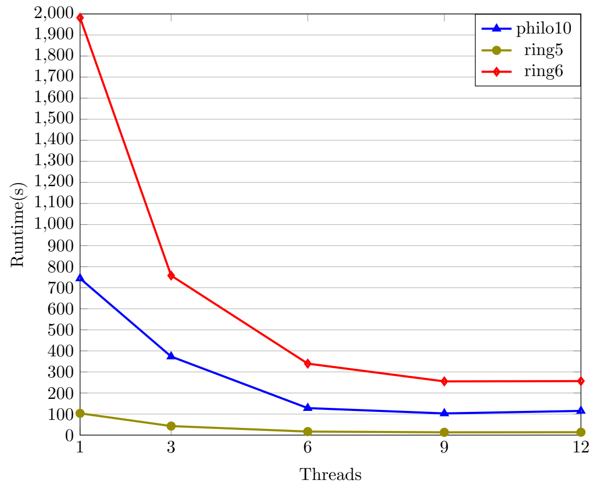

Figure 4: runtime of construction of SOG (philo10,ring5,ring6)

Figure 3 illustrates the speedups obtained in this experiment. It can be seen that the achieved speedups are largely dependent on the size of graph. This explains why the speedup obtained for the ring6 is better than for the ring3. Since, the speedup figures are closer to ideal for larger input sizes, as well as for higher number of threads.

Figure 4 shows the runtime obtained on three examples (philo10, ring5 and ring6). As it can be observed, the runtime decreases with increasing the number of threads.

7 Conclusion

In this paper, we presented an efficient multi-threaded approach of the building of the symbolic observation graph. Our approach uses multi-threading with shared memory to speed-up the computation. We have shown that the performance of the new parallel algorithm of construction of SOG scales reasonably well with increasing numbers of threads. The efficiency of our approach has been tested on various examples. Experimental results seem promising due to the obtained reduction in runtime. Experiments presented in this paper are only preliminary We plan to perform more extensive tests to get additional experimental results and to be able to give a more comrehensive view of the performance of our approach and of its scalabiliy. We are currently conducting more experiments and we aim to the verification of generic properties like deadlock freeness, liveness or specific properties that are expressed by linear-time temporal logics. Also, we plan to implement the non-deterministic version of the SOG and to measure the difference.

References

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, 100(6):509–516, 1978.

[2] Stefan Blom and Jaco Van De Pol. Symbolic reachability for process algebras with recursive data types. In *International Colloquium on Theoretical Aspects of Computing*, pages 81–95, 2008.

[3] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[4] Jerry R Burch, Edmund M Clarke, David E Long, Kenneth L McMillan, and David L Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[5] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *International Conference on Application and Theory of Petri Nets*, pages 103–122, 2000.

[6] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71, 1981.

[7] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[8] Armin B Cremers and Thomas N Hibbard. Axioms for concurrent processes. In *New Results and New Trends in Computer Science*, pages 54–68. Springer, 1991.

[9] Tom Dijk. The parallelization of binary decision diagram operations for model checking. 2012.

[10] Alexandre Duret-Lutz, Kais Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Self-loop aggregation producta new hybrid approach to on-the-fly ltl model checking. In *International Symposium on Automated Technology for Verification and Analysis*, pages 336–350. Springer, 2011.

[11] Ioannis Filippidis and Gerard J Holzmann. An improvement of the piggyback algorithm for parallel model checking. In *International SPIN Symposium on Model Checking of Software*, pages 48–57. ACM, 2014.

[12] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *International SPIN Workshop on Model Checking of Software*, pages 217–234, 2001.

[13] Jaco Geldenhuys and Antti Valmari. Techniques for smaller intermediary bdds. In *International Conference on Concurrency Theory*, pages 233–247. Springer, 2001.

[14] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *Logic in Computer Science, 1991. LICS'91., Proc. of 6th Annual IEEE Symposium on*, pages 406–415. IEEE, 1991.

[15] Serge Haddad, Jean-Michel Ilié, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In *International Symposium on Automated Technology for Verification and Analysis*, pages 196–210. Springer, 2004.

[16] Y He. Multicore-enabling a binary decision diagram algorithm, 2009.

[17] Gerard J Holzmann. Parallelizing the spin model checker. In *International SPIN Workshop on Model Checking of Software*, pages 155–171. Springer, 2012.

[18] Gerard J Holzmann. Proving properties of concurrent programs. In *International SPIN Workshop on Model Checking of Software*, pages 18–23. Springer, 2013.

[19] Gerard J Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.

[20] Kais Klai and Laure Petrucci. Modular construction of the symbolic observation graph. In *Application of Concurrency to System Design, 2008. ACSD 2008. 8th International Conference on*, pages 88–97. IEEE, 2008.

[21] Kais Klai, Laure Petrucci, and Michel Reniers. An incremental and modular technique for checking ltl\ x properties of petri nets. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 280–295. Springer, 2007.

[22] Kais Klai and Denis Poitrenaud. Mc-sog: An ltl model checker based on symbolic observation graphs. In *International Conference on Applications and Theory of Petri Nets*, pages 288–306. Springer, 2008.

[23] Kais Klai, Samir Tata, and Jörg Desel. Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. In *International Conference on Business Process Management*, pages 294–309. Springer, 2009.

[24] Kais Klai, Samir Tata, and Jörg Desel. Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. *Data & Knowledge Engineering*, 70(5):467–482, 2011.

[25] Lars M Kristensen and Laure Petrucci. An approach to distributed state space exploration for coloured petri nets. In *International Conference on Application and Theory of Petri Nets*, pages 474–483, 2004.

[26] Charles Lakos and Laure Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *Application of Concurrency to System Design, 2004. ACSD 2004. Proceedings. Fourth International Conference on*, pages 185–194. IEEE, 2004.

[27] D Michael Miller and Rolf Drechsler. Implementing a multiple-valued decision diagram package. In *Multiple-Valued Logic, 1998. Proceedings. 1998 28th IEEE International Symposium on*, pages 52–57. IEEE, 1998.

[28] Jörn Ossowski. *JINC: a multi-threaded library for higher-order weighted decision diagram manipulation.* PhD thesis, University of Bonn, 2010.

[29] Hiba Ouni, Chiheb Ameur Abid, and Belhassen Zouari. A distributed state space for modular petri nets. In *2015 7th International Conference on Modelling, Identification and Control (ICMIC)*, pages 1–6. IEEE, 2015.

[30] Antti Valmari. A stubborn attack on state explosion. In *International Conference on Computer Aided Verification*, pages 156–165. Springer, 1990.

[31] Tom Van Dijk, Alfons Laarman, and Jaco Van De Pol. Multi-core bdd operations for symbolic reachability. *Electronic Notes in Theoretical Computer Science*, 296:127–143, 2013.

[32] Tom van Dijk and Jaco van de Pol. Sylvan: Multi-core decision diagrams. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 677–691. Springer, 2015.

[33] Miroslav N Velev and Ping Gao. Efficient parallel gpu algorithms for bdd manipulation. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014.