Vérification parallèle de systémes concurrents en utilisant le Graphe d'Observation Symbolique

Hiba Ouni

Université Paris 13 **Université de Tunis El Manar**

Thèse

Présentée par

Hiba Ouni

pour obtenir le grade de Docteur d'université

Spécialité :

Informatique

Parallel verification of concurrent systems using the Symbolic Observation Graph

Soutenue le 20 décembre 2019 devant un jury composé de:

Directeurs de thèse :

Kais Klai	LIPN, Université Paris 13
Belhassen Zouari	Mediatron, Sup'Com Tunis

Rapporteurs :

Jaco van de Pol	Aarhus University
Alexendre Duret-lutz	LRDE, l'Epita

Examinateurs :

Hanna Klaudel	IBISC, Université d'Evry
Laure Petrucci	LIPN, Université Paris 13

Co-encadrant :

Chiheb Ameur Abid	Mediatron, Université de Tunis ElManar

Declaration of Authorship

I, Hiba Ouni, declare that this thesis titled, “ Parallel verification of concurrent systems using the Symbolic Observation Graph ” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

Abstract

An efficient way to cope with the combinatorial explosion problem induced by the model checking process is to compute the Symbolic Observation Graph (SOG). The SOG is defined as a condensed representation of the state space based on a symbolic encoding of the nodes (sets of states). It has the advantage to be much reduced comparing to the original state space graph while being equivalent with respect to linear time properties. Aiming to go further in the process of tackling the state space explosion problem, we present in this thesis three different approaches to parallelize the construction of the SOG. A multi-threaded approach based on a dynamic load balancing and a shared memory architecture, a distributed approach based on a distributed memory architecture and a hybrid approach that combines the two previous approaches. Experiments show that parallel approaches can improve drastically the performances of the SOG computation regarding a sequential construction.

We exploit the strengths of the parallel reachability to design an on-the-fly model checker for $LTL \setminus X$ logic based on the event-state based SOG. We implemented the proposed model checking algorithms within a C++ prototype and compared our preliminary results with the state of the art model checkers.

Keywords. Symbolic model checking, Parallel model checking, Symbolic Observation Graph, Shared memory, Distributed memory.

Résumé

Un moyen efficace pour résoudre le problème d'explosion combinatoire induit par le processus de vérification du modèles consiste à calculer le Graphe d'Observation Symbolique (SOG). Le SOG est défini comme une représentation condensée de l'espace d'états basée sur un codage symbolique des noeuds (ensembles d'états). Il présente l'avantage d'être très réduit par rapport a l'espace d'état initial tout en étant équivalent par rapport aux propriétés temporelles linéaires.

Dans le but d'aller plus loin dans le processus de résolution du problème de l'explosion combinatoire de l'espace d'états, nous présentons dans cette thèse trois approches différentes pour paralléliser la construction du SOG. Une approche multi-thread basée sur un équilibrage dynamique de charge et une architecture à mémoire partagée, une approche distribuée basée sur une architecture à mémoire distribuée et une approche hybride combinant les deux approches précédentes. Les expériences montrent que les approches parallèles peuvent améliorer considérablement les performances de la construction du SOG par rapport à la construction séquentielle.

Nous proposons, par la suite, un modèle checker basé sur la construction parallèle du SOG, qui permet la vérification à la volée de propriétés de la logique $LTL\ setminusX$. Nous avons implémenté les algorithmes de 'model checker' proposés dans un prototype C ++ et comparé nos résultats préliminaires aux 'model checkers' de l'état de l'art.

Mots clés : Model checking symbolique , Model checking paralléle, Graphe d'Observation Symbolique, Mémoire partagée, Mémoire distribuée.

Acknowledgements

First, I would like to express my sincere gratitude to my advisor Dr. Kais KLAI for the continuous support of my Ph.D study and related research, for his patience, motivation and immense knowledge. His supervision and crucial contribution made him a backbone of this research.

Namely, i am deeply indebted to my supervisor Dr. Belhassen ZOUARI for his guidance, advices and encouragement throughout the course of my doctoral program.

I would also wish to express my gratitude to my co-supervisor Chiheb Ameur ABID. He has provided positive encouragement and a warm spirit to finish this thesis. It has been a great pleasure and honor to have him as co-supervisor.

I am grateful to Jaime Arias. I want to acknowledge and appreciate his help and transparency during my research. I am also so thankful to Sami Evangelista and Camille Cotti for their valuable suggestions and comments on my research works. I thank all members of LIPN laboratory for making it such a nice environment full of interesting people.

I also thank the committee members for agreeing to participate in my PhD committee, and reviewing this work. Thank you Alexandre Duret-lutz and Jaco Van de pol for investing time and providing interesting and valuable feedback.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue. They are the ultimate role models. Most importantly, I wish to thank my loving and supportive husband and my three wonderful sisters who provide unending inspiration. Last, I would like to thank my friends and all the good people who helped me in keeping my sanity and focus, and made the time generally enjoyable.

Contents

Declaration of Authorship i

Acknowledgements v

1 Introduction 1
- 1.1 Model Checking . . . 2
- 1.2 Existing parallel and distributed model checking methods . . . 4
- 1.3 Aims and contributions of this work . . . 7
- 1.4 Overview and Reading Guide . . . 8

2 Background and Preliminaries 11
- 2.1 Labeled transition system . . . 11
- 2.2 Kripke structure . . . 12
- 2.3 Labeled Kripke Structure . . . 13
- 2.4 Decision Diagrams . . . 14
  - 2.4.1 Binary Decision Diagrams . . . 14
  - 2.4.2 Multi-valued Decision Diagrams . . . 15
  - 2.4.3 List decision diagrams . . . 15
- 2.5 Linear Temporal Logic . . . 16
- 2.6 Symbolic Observation Graph . . . 17
  - 2.6.1 Event-based SOG . . . 18
  - 2.6.2 State-based SOG . . . 20
- 2.7 Event-state based SOGs for LTL model checking . . . 21
  - 2.7.1 Revisiting SOG for Hybrid LTL . . . 22
  - 2.7.2 Checking stuttering invariant properties on SOGs . . . 25

3 Multi-threaded approach for the parallel generation of the SOG 30
- 3.1 Introduction . . . 30
- 3.2 Multi-threaded algorithm for constructing the SOG . . . 31
  - 3.2.1 Aims and Hypothesis . . . 31
  - 3.2.2 Description of the algorithm . . . 32
- 3.3 Technical aspects and Implementation . . . 34

3.3.1 Multi-core decision diagram packages . . . 34
3.3.2 Symbolic encoding of the SOG . . . 35
3.3.3 Adaptation of Sylvan to the parallel implementation of the SOG . . . 35
3.4 Experimental results . . . 37
3.4.1 Comparison BuDDy - Sylvan . . . 37
3.4.2 Results of the multi-threaded algorithm . . . 37
3.5 Conclusion . . . 39

4 Distributed approach for the construction of the SOG 41
4.1 Introduction . . . 41
4.2 Distributed algorithm for constructing the SOG . . . 42
4.2.1 Aims and Hypothesis . . . 42
4.2.2 Description of the algorithm . . . 43
4.2.3 Termination Detection . . . 44
4.3 Technical aspects and implementation . . . 45
4.4 Experiments . . . 45
4.5 Conclusion . . . 46

5 Hybrid approach for generating the SOG 48
5.1 Introduction . . . 48
5.2 A hybrid approach for constructing a SOG . . . 49
5.2.1 Aims and hypothesis . . . 49
5.2.2 Description of the algorithms . . . 50
5.2.3 Termination detection . . . 53
5.2.4 Correctness proof . . . 53
5.3 Experiments . . . 55
5.3.1 Results of the hybrid approach . . . 55
5.3.2 Comparative analysis . . . 57
5.4 Conclusion . . . 59

6 Reducing time and/or memory consumption of the SOG construction 61
6.1 Introduction . . . 61
6.2 Canonicalization algorithm . . . 62
6.3 Experiments . . . 65
6.3.1 Results of the canonicalization algorithm . . . 65
6.3.2 Comparative analysis . . . 69
6.4 Conclusion . . . 70

7 PMC-SOG : Parallel Model checker based on the SOG **74**
7.1 Introduction 74
7.2 Multi-Core LTL Model Checking 77
7.2.1 Description of the multi-core approach 77
7.2.2 Parallelization at the level of aggregates 77
7.2.3 Parallelization at the level of decision diagrams operations . 79
7.2.4 Implementation 80
7.2.5 Experiments 81
7.3 A hybrid approach for parallel LTL Model Checking 84
7.3.1 Description of the algorithm 85
7.3.2 Implementation 88
7.3.3 Experiments 88
7.4 Conclusion 89

8 Conclusion **91**

Bibliography **97**

Chapter 1

Introduction

In recent years, computer systems have played an important role in various areas, such as health, transport, economy, communications, etc. With the technological evolution, the complexity of these systems is undergoing considerable development, and is set to make further progress in the future. Given the risks of their failures, the issue of their safety and their proper functioning is of major importance. Especially for critical systems whose failures could cause serious problems. Indeed, the consequences of a dysfunction or an abnormal behavior is sometimes irreparable. For instance, the Intel Pentium bug in the division algorithm had led to devastating economic consequences for the company [CKZ96]. On January 17, 1995, Intel announced a pre-tax charge of 475 million of dollars against earnings, ostensibly the total cost associated with replacement of the flawed processors. Although the evaluations carried out by independent organizations, they had shown the low importance of the consequences of the bug and its negligible effect in most uses of Intel. In this context, it is inevitable, when studying any system, to specify and precisely define its behavior. It is therefore a real challenge to be answered by powerful verification techniques: using them as early as possible in the design of a system avoids the costs that would be involved in detecting a serious error of this system during the operation phase.

It is necessary to provide designers with effective means of verification and validation. Such means exist, but it is imperative that they can support very large systems. Indeed, the major IT projects are measured today in millions of lines of code and often involve interactions between hundreds of components. Thus, it is not enough just to provide new verification techniques, but they should be usable for real systems. We focus on concurrent systems that have multiple components interacting and running in parallel. They are increasingly widespread architectures, but their non-determinism makes them complex systems to check. The verification of such systems can take several forms.

Testing is the simplest and most common verification mode. Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Different aspects of system behavior can be tested. It is about writing test sets that are designed to verify that a system meets a set of properties by running it in a test environment. They describe the expected behaviors of the components of the tested system. The major drawback of this technique is its non-exhaustiveness. Indeed, tests rarely cover the entire system behaviors. This means that even if the tests are running correctly, it is possible that it exists a behavior of the system which is not covered by the tests, such that it does not respect the desired properties. Also, testing is an expensive part in project budget.

Theorem proving is a technique by which both the system and its desired properties are expressed as formulas in some mathematical logic. It consists of describing a computer system in the form of a set of axioms. The properties are expressed using theorems that must be proved in order to validate the architecture of the system. Proofs can be constructed by hand or by machine-assisted theorem proving [Mar96]. This method makes possible to parametrically check a system: a verified property remains true as long as the constraints between parameters are verified. Moreover, it allows the verification of infinite systems. However, this is a difficult technique to implement and requires strong skills, and therefore remains difficult to disseminate.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. In contrast to theorem proving, Model checking is not a parametric verification technique. However, it is a simpler to implement. The ease of use of model checking comes from the fact that it is an automatic process.

1.1 Model Checking

Model checking, invented by Clarke and Emerson [CE81] and Queille and Sifakis [QS82], is a powerful formal verification method that can be used to improve the safety of concurrent systems. In fact, model checking techniques offer an automatic solution to check whether a model of the system meets its requirements. Indeed, the growing interest of such techniques is that they do not only help in the task of finding errors early, but they can also provide counterexamples when the system model violates any of its requirements.

Given a (usually finite-state) formal description of the system to be analyzed 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 counterexamples: a sample run inducing to the violation of the property. Such a run gives a valuable feedback and allows to point to design errors.

If we verify a system using a model checking procedure, we have to model the system at first. Various formalism can be used to do so. In this work we are interested in both Labeled Transition System and Kripke structure. In general, such a formalism should be used to model the system as faithfully as possible. Note that unless the model is inaccurate, we can conclude the correctness of the system from the correctness of the model.

The system properties have to be transformed into an appropriate formalism as well. Various temporal logics are used for this purpose. The expression power of most temporal logics is high enough to express all the interesting and generally important system properties such as deadlock, livelock, or reachability of a given good or bad situation. The temporal logics may express even more complex and structural properties like, for instance, whenever a situation A occurs then a situation B occurs eventually. Temporal logics come in two variants, linear time and branching time. Linear time properties are concerned with properties of paths. So a state in the model is said to satisfy a linear time property if all the paths emanating from the state satisfy the property. On the other hand, branching time logics describe properties that depend on the branching structure of the model. Unfortunately, model checking procedures differ significantly for both kind of logics. This thesis deals only with the linear time logic model checking problem.

It is important for efficiency of a model checking procedure that the model is given implicitly. However, in such a case the procedure performs, in fact, two tasks simultaneously. It does the verification of the model as well as it solves the state space generation problem. This allows the algorithm to reveal an error in the model under consideration before the explicit representation of the model is completely generated. Model checking procedures that perform the model checking task simultaneously with the state space generation are generally referred to on-the-fly model checking procedures.

At the core of model checking are algorithms that implement state space exploration. The reachable state space is explored 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 combinatorial state space explosion [Cla+01]. 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 such as partial order techniques or symbolic model checking.

Partial order approaches [GW91; Val90; VAM96] 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. For example, a system can go through a succession of states to invariably lead to the same state. If the intermediate states are not relevant to the verification of the desired properties, then they are not preserved and thus the place they would have taken in memory is saved.

Symbolic techniques [Bur+94; GV01; HIK04], 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) [Ake78; Bry92].

The Symbolic Observation Graph (SOG) [HIK04; KP08a] is an other type of approach aiming the same goal (reducing the state space explosion problem). It is an abstraction of the reachability state graph of concurrent systems. A SOG is a graph whose construction is guided by a set of observed actions (e.g. atomic propositions involved in a linear time formula). The nodes of a SOG are aggregates hiding a set of local states which are connected with non observed actions, and are compactly encoded using Binary Decision Diagram techniques (BDDs). The arcs of a SOG are exclusively labeled with observed actions which allows on-the-fly verification of $LTL \setminus X$ (Linear-time Temporal Logic minus the next operator) formulas through the synchronized product with the corresponding Büchi automaton. The verification of these latter on the SOG is equivalent to that on the original systems.

Recently, there has also been increased interest in parallel and distributed verification approaches [Bar+18]. A distributed approach does not alleviate the explosion problem, however it offers more available memory for storage of the state space and has the potential to speed-up the process of verification in time. Indeed, a distribution of requirements on space and computational power can help with solving problems that have demands which cannot be met by using a single computer.

1.2 Existing parallel and distributed model checking methods

In this section, we give a non exhaustive presentation of some works in the literature aiming to parallelize the model checking approach. Parallel and distributed state-space exploration and model checking have been active areas of research for over a

decade and several attempts have been proposed. These attempts can be classified into two main categories: parallel approaches using multiple processors in shared-memory machines and distributed approaches where the state space construction is distributed on multiple machines in some network (cluster, cloud). A number of solutions target shared memory machines (e.g [Cas+94; AKH97; BBR09; Hol12; DP16]).

Allmaier et al. [Cas+94; AKH97] were among the first to implement a parallel state space construction algorithm for shared memory systems. They avoid the consistency problem of shareable spaces by using locking variables to protect the shared storage structure.In [IB02], an other parallel algorithm based on a work stealing scheduling paradigm was proposed to provide dynamic load balancing without a blocking phase. The data structure used to store already visited states is a global hash table. The work resulted in reasonable speedups, but the method is inherently unsuitable to support liveness algorithms, because the correctness of such algorithms depends strongly on the precise state counting arguments.

In [BBR09; Bar+10] a parallel model checker DiVinE for multi-processor systems has been presented. The state space is partitioned into parts, each thread maintains its assigned part and its own hash table. Their results were promising, but the algorithms exhibit limited scalability on multi-core systems. DiVinE recently also implemented compression [Bar+13]. A difference in the DiVinE implementation of tree compression is the choice for n-ary trees with resizing hash tables [Šti13].

Several efforts have been made to parallelize the Spin model checker [Hol04]. As described in [HB07; Hol08], Holzmann altered a parallel version of Spin with a multi-core algorithm. It uses a shared hash table protected by a fine-grained locking technique where only the parts that contain a newly generated state of the hash table are locked. Later, this work has been extended [LPW10] with a lockless shared hash table where locks are emulated with atomic primitives in a separated array. In [Hol12], Holzman proposed another multicore version of SPIN takes off where a previous version [HB07] left off by increasing the performance of parallel breadth-first search, while decreasing the complexity of the algorithm.

In [Hol12], Holzmann extends the Spin model checker with a multi-core algorithm. The states are statically partitioned over the worker threads, so the algorithm can be improved with separate local hash tables, instead of one shared hash table. Holzmann shows that the performance of the parallel breadth-first search algorithm scales reasonably well with increasing the numbers of cpu-cores.

Several techniques for reductions are also used alongside parallelization such as partial order methods or by using symbolic representation of state spaces such as BDDs [VDLVDP13]. The latter approach is based on the parallelization of the

BDD operations [VG14; Dij12] while the construction of the whole graph is kept sequential.

In [DP15; DP16], an efficient parallelization algorithms of BDDs operations as well as MDDs operations [Kam98] are proposed. Sylvan [DP15; DP16] uses work-stealing and scalable parallel data structures to provide parallelization of algorithms on decision diagrams. It currently supports BDDs and a variation of MDDs called List Decision Diagrams (LDDs). It has been designed as an extensible framework with custom BDD operations in mind and features parallel garbage collection.

Moreover, a distributed memory state space generation algorithm was proposed in [CGN98]. The authors presented a static partition function based on a state hashing technique. It performs the breadth first traversal of the graph. If a new state is generated, it is processed and its successors are generated locally, otherwise, it is sent over network to the appropriate workstation and its local processing is omitted. Similar approaches were presented in [CCM01; Rod+06].

Also, authors of [KP04] described an approach to conduct distributed state space exploration for Colored Petri Nets. This distribution is based on the introduction of a coordinator process and a number of worker processes. The coordinator process is responsible for the distribution of states and the termination detection and the worker processes are responsible for the storage of states and the computation of successor states. The architecture used is based on a distributed system consisting of several machines and a coordinator node whose role is to initiate treatment, to distribute states, and to detect the end of exploration.

In [GMS01], a parallel construction of a state space for Model checking is performed by partitioning it into several nodes. The partitioning is performed by adopting a static load balancing scheme in order to avoid the potential communication overhead occurring in dynamic load balancing schemes through the use of an adequate hash function. In [KP04], a coordinator process is introduced. This coordinator is the responsible for the distribution of states construction and termination detection. Furthermore, a distributed approach to tree logic formulas verification is described [Bel+13]. This approach exploited a parametric state space builder, designed to ease the adoption of big data platforms.

A multi-core state space exploration algorithm is proposed in [EKP13]. The algorithm is based on state compression and state reconstruction to reduce memory consumption. Also, it requires a little inter-thread synchronization making it highly scalable.

BDD-based reachability algorithms targeting compute clusters are proposed in [ODP17]. The proposed algorithms are based on a distributed hash table, cluster-based work stealing algorithms, and several caching structures that all utilize the

newest networking technology. Also, this paper present the first BDD package that targets both distributed and multi-core architectures. This approach is evaluated on a collection of models, it has been shown that the larger models benefit from the extra memory available on compute clusters and from all available computational resources.

1.3 Aims and contributions of this work

This thesis contains several contributions related to the parallel and distributed state space generation. This section summarizes these contributions:

As a first contribution, we exploit the efficiency of the SOG representation of state space and its moderate size in order to cope with the state explosion problem. At the same time, we adopt parallel algorithms to benefit from additional speedups and performance improvement in execution time.

For this purpose, we investigate three approaches to parallelize the SOG construction using three different algorithms. The key idea of our approaches is to build simultaneously several nodes (aggregates) of the symbolic graph. The first proposed algorithm targets shared memory architectures by using threads to build nodes of a SOG. 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 SOG. Indeed, since MDDs [Kam98] are a generalization of BDDs to the integer domain, a node in an MDD may represent several nodes in an BDD. In general, graphs based on MDDs are more reduced than those based on BDDs.

The second approach targets distributed memory architecture by using processes. Each process has its own local memory, and communication between processes is performed by using the Message Passing Interface MPI [Wal94]. This approach has the advantage to provide more memory space to manage the built SOG. Distributing aggregates is performed through the computation of a hash function in order to cope with communication overhead.

Aiming to find the best way to parallelize the SOG construction, we propose a hybrid technique which combines the two previous approaches. The SOG construction is then shared among a number of processes, each creates a set of threads that run on (typically) the same number of CPUs of a processor. Among these threads of each process, one (the coordinator) is responsible of the communication (including performing overlapped asynchronous message passing), while the others (workers) are responsible for the construction of the SOG nodes. The hybrid (shared and distributed) architectures provides significant performance benefits for

the enumerative model checking performed on clusters (currently the most common high-performance computers).

The second contribution of this thesis is the canonicalization approach. Basic idea behind is to reduce the size of states set associated with each aggregate that can lead to important memory savings. For this reason, we propose a specific and symbolic algorithm that allows to reduce the size of an aggregate. Then, we apply such a reduction within the three previous parallel approaches and we measure the impact of the use of the canonicalization on the construction of the SOG. The canonicalization allows us not only to efficiently decrease the memory consumption, but also to improve the scalability of both the distributed and the hybrid approaches (especially the distributed one). As expected, the canonicalization decreases the communication cost by reducing the size of the communicated aggregates (sets of states) between the processes involved in the SOG construction. Indeed, instead of sending the set of all states that forms an aggregate, we propose to use a reduced canonic representation that is sufficient to rebuild it.

Finally, we exploit the strengths of the parallel reachability to create a parallel model checking based on the parallel construction of the SOG, where both event-based and state-based properties can be expressed, combined, and verified. Instead of composing the whole system with the Buchi automaton representing the negation of the formula to be checked, we make the synchronization of the automaton with an abstraction of the original reachability graph of the system. Two different approaches are proposed to design an on-the-fly model-checker for $LTL \setminus X$ logic based on the event-state based SOG. The first one targets shared memory architectures. We have proposed two versions using different techniques of parallelization. The second approach is dedicated to the cluster architectures. Thus, we create a set of processes each uses a set of threads in order to build the SOG. One of these processes, called model checker process, computes the synchronized product and then performs the emptiness check between the automaton modeling the negation of the LTL formula with the LKS corresponding to the SOG.

1.4 Overview and Reading Guide

This report is organized as follows:

Chapter 1 deals with the problem of the combinatorial explosion problem and its content focuses on the parallel and distributed model checking as solution. Chapter 2 provides more general background information that is presented in its following chapters.

Parts of this thesis consist of conference papers, that have been published in the following publications:

Chapter 3 is mostly based on this paper: [Oun+17a], "A Parallel Construction of the Symbolic Observation Graph: the Basis for Efficient Model Checking of Concurrent Systems". In: The 8th International Symposium on Symbolic Computation in Software Science 2017 this paper, presented at SCSS 2017, is about the multi-threaded construction of the SOG. The basic idea is that each thread owns one part of the SOG construction. Also, in this chapter, MDDs have been used instead of BDDs for the aggregates (the nodes of the SOG) encoding.

Chapter 4 is based on the paper[Oun+17b] "Parallel Symbolic Observation Graph" which was published at: the IEEE International Symposium on Parallel and Distributed Processing with Applications. This paper presents two parallel algorithms to build the SOG. The first algorithm is dedicated for shared memory architectures, and is based on the distribution of the SOG construction on several threads using a dynamic load balancing scheme. The second algorithm is proposed for distributed memory architectures, and distributes the SOG construction on processes using a static load balancing scheme. These two algorithms are implemented and their performances are studied and compared against each other as well as against the sequential construction of the SOG.

Chapter 5 is mostly based on the paper: "Towards parallel verification of concurrent systems using the Symbolic Observation Graph" [Oun+19], which was published at: the 19th International Conference on Application of Concurrency to System Design. We propose in this chapter to parallelize the construction of the SOG using an hybrid (distributed-shared memory) approach. Doing so, we take advantage of the recent advances in computer hardware, by distributing the construction process over a large number of multi-core processors. We studied the performance of our new approach comparing against both distributed and shared memory approaches.

Chapter 6 is based on the paper [Oun+18] "Reducing Time and/or Memory Consumption of The SOG construction in a Parallel Context". This paper presented at 2018 IEEE International Symposium on Parallel and Distributed Processing with Applications. In this chapter we go a step forward in improving the SOG construction process by reducing, on the fly, the size of its aggregates. We proposed a Multi-valued decision diagrams (MDDs) based algorithm to determine a single representative for each strongly connected component in every aggregate allowing to remove from memory a consequent number of states which are no more necessary for the construction process. Then, we present a study of the impact of such an optimization in the parallel construction of the SOG using the three proposed approaches.

We present, in chapter 7, parallel and hybrid approach for checking linear time

temporal logic properties of finite systems combining on-the-fly and symbolic approaches. Based on previous algorithms for the parallel construction of the SOG, we propose a parallel on-the-fly model checker targeting shared-memory architectures by using two different techniques. One uses threads such that the parallelization is performed at the level of nodes building of a SOG. The second technique performs parallelization at the level of MDD operations by using a work-stealing framework. Also, we propose a model checker that targets distributed memory architectures. The latter uses processes and threads to parallelize the construction of the SOG while performing model checking simultaneously. We have implemented the different proposed approaches in a software tool and we have conducted experiments to compare them against the parallel model checker LTSmin [Kan+15; VDBL13; LPW11; BPW10].

Finally, Chapter 8 concludes the thesis with a reflection of what has been achieved, and some promising directions for future work.

Chapter 2

Background and Preliminaries

In this chapter, we briefly establish some basic definitions and results that are needed throughout the thesis. The technique presented in this thesis applies to different kinds of models, that can map to Labeled Transition Systems (LTS) which are well adapted to event-based logic, Kripke Structures (KS) which are adapted to a state-based reasoning. Also, we present it for Labeled Kripke Structures (LKS), since this formalism is suitable for both event and state based formalism.

2.1 Labeled transition system

Labeled Transition System (LTS) belong to a specific class of automata, called the Finite State Automata (FSA). It is a graph-like structure that shows the different states that a system can be in, and possible transitions between them. These transitions are labeled by actions, one state is designated as the initial state and a (possibly empty) subset of states represents the final states.

Definition 1 (labeled transition system) *A labeled transition system is a 4-tuple* $\langle \Gamma, Act, \rightarrow, I \rangle$ *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;*

We distinguish LTS observed actions, denoted by a subset Obs, from unobserved actions, denoted by the subset $UnObs$ with $Obs \cup UnObs = Act$ and $Obs \cap UnObs = \emptyset$.

An example of LTS is given in Figure 2.1 where s_0 is the initial state. The set of observed actions contains two elements $\{a, b\}$, unlabeled edges are supposed to be labeled by non observed actions.

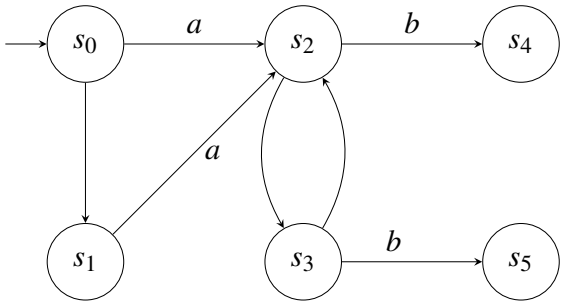


FIGURE 2.1: Example of LTS

2.2 Kripke structure

Definition 2 (Kripke structure)
Let AP *be a finite set of atomic propositions. A Kripke structure (*KS for short*) over* AP *is a 4-tuple* $\langle \Gamma, L, \rightarrow, s_0 \rangle$ *where:*

- Γ is a finite set of states *;*
- $L : \Gamma \rightarrow 2^{AP}$ is a labeling (or interpretation) function *;*
- $\rightarrow \subseteq \Gamma \times \Gamma$ is a transition relation *;*
- $s_0 \in \Gamma$ is the initial state *;*

Notations

- Let $s, s' \in \Gamma$. We denote by $s \rightarrow s'$ that $(s, s') \in \rightarrow$,
- Let $s \in \Gamma$. $s \not\rightarrow$ denotes that s is a dead state (i.e. $\nexists s' \in \Gamma$ such that $s \rightarrow s'$),
- $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ is used to denote paths of a Kripke structure and $\bar{\pi}$ denotes the set of states occurring in π,
- A finite path $\pi = s_1 \rightarrow \ldots \rightarrow s_n$ is said to be a *circuit* if $s_n \rightarrow s_1$. If $\bar{\pi}$ is a subset of a set of states S then π is said to be a circuit of S.
- Let $\pi = s_1 \rightarrow \ldots \rightarrow s_n$ and $\pi' = s_{n+1} \rightarrow \ldots \rightarrow s_{n+m}$ be two paths such that $s_n \rightarrow s_{n+1}$. Then, $\pi\pi'$ denotes the path $s_1 \rightarrow \ldots \rightarrow s_n \rightarrow sn+1 \rightarrow \ldots \rightarrow s_{n+m}$.
- $\forall s, s' \in \Gamma, s \xrightarrow{*} s'$ denotes that s' is reachable from s (i.e. $\exists s_1, \ldots, s_n \in \Gamma$ such that $s_1 \rightarrow \ldots \rightarrow s_n \wedge s = s_1 \wedge s' = s_n$). $s \xrightarrow{+} s'$ denotes the case where $n > 1$ and $s \xrightarrow{*} S^{s'}$ (resp. $s \xrightarrow{+} S^{s'}$) stands when the states $s_1 \rightarrow \ldots \rightarrow s_n$ belong to some subset of states S.

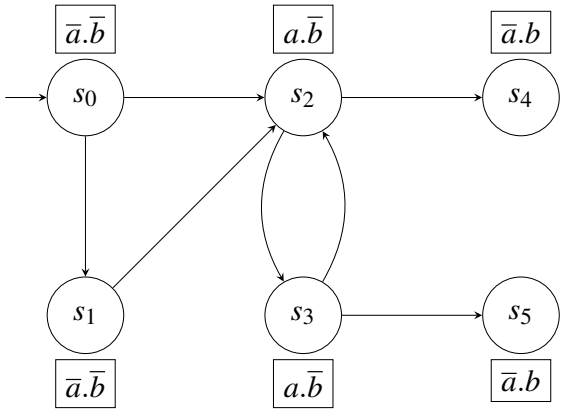


FIGURE 2.2: Example of Kripke Structure

Figure 2.2 illustrates an example of Kripke Structure. The set of atomic propositions contains two elements $\{a,b\}$ and each state of the KS is labeled with the values of these propositions.

2.3 Labeled Kripke Structure

A Labeled Kripke Structure (LKS) is a mix of the two previous models which can be used to represent the behavior of a system.

Definition 3 (Labeled Kripke structure)
Let AP *be a finite set of atomic propositions and let Act be a set of actions. A Labeled Kripke structure (*LKS for short*) over* AP *is a 5-tuple* $\langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ *where:*

- $\langle \Gamma, Act, \rightarrow, s_0 \rangle$ *is an LTS;*
- $\langle \Gamma, L, \rightarrow, s_0 \rangle$ *is a KS;*

Figure 2.3 illustrates an example of *LKS* over $AP = \{a,b\}$.

An LTS (resp. KS and LKS) can be represented either, explicitly (each state/arc is individually represented in memory), or symbolically (sets of states can share some data in memory) using decision diagrams such as BDDs. A mixed approach (hybrid) exists where states are encoded symbolically while edges are represented explicitly. The SOG is an example of the latter representation.

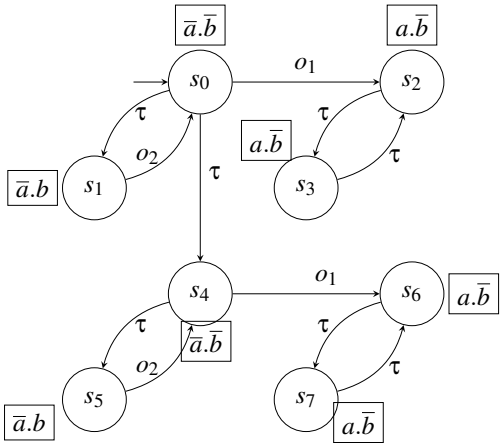


FIGURE 2.3: A Labeled Kripke Structure

2.4 Decision Diagrams

2.4.1 Binary Decision Diagrams

The nodes of the Symbolic Observation Graph (aggregates) are encoded symbolically using Binary decision diagrams (BDDs) in [HIK04; KP08a], and Multi-valued decision diagrams (MDDs) in [Oun+17a; Oun+17b].

A binary decision diagram (BDD) is a binary tree in the form of a directed acyclic graph which represents one or more Boolean functions $\mathbb{B}^N \to \mathbb{B}$. Each level in this binary tree represents an input variable of the Boolean function. The terminal nodes in this tree represent true and false, respectively. We use 0 to denote false and 1 to denote true, also, we use $f_{x=v}$ to denote a Boolean function f where the variable x is given value v. When the BDD is ordered such that input variables occur in ascending level order and is reduced so that no two nodes represent the same function, one obtains a (reduced, ordered) binary decision diagram (ROBDD) [Bry86]. A ROBDD is a rooted directed acyclic graph with leaves 0 and 1. Each internal node has a variable label x_i and two outgoing edges labeled 0 and 1, called the "low" and the "high" edges.

Definition 4 (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.,* $\forall p \forall q(lvl(p) = lvl(q) \wedge p[0] = q[0] \wedge p[1] = q[1]) \rightarrow p = q$.

2.4.2 Multi-valued Decision Diagrams

Multi-valued decision diagrams (MDDs) are a generalization of BDDs to the integer domain. Like BDDs, MDDs can be used to represent sets of states. BDDs represent functions $\mathbb{B}^N \rightarrow \mathbb{B}$, MDDs are typically used to represent functions on integer domains $(\mathbb{N}_{<v})^N \rightarrow \mathbb{B}$. Rather than two outgoing edges, each internal MDD node with variable x_i has n_i labeled outgoing edges labeled from 0 to $n_i - 1$.

Definition 5 (Multi-valued decision diagram)
An (ordered) MDD 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* n_i *outgoing edges, labeled from 0 to* $n_i - 1$*; we write* $lvl(p) = i$ *and* $p[v] = q$*, where* $0 \leq v < n_i$.
3. *For each edge from node p to non-terminal node q,* $lvl(p) < lvl(q)$.
4. *There are no duplicate nodes, i.e.,* $\forall p \forall q(lvl(p) = lvl(q) \wedge \forall v, p[v] = q[v]) \rightarrow p = q$.

2.4.3 List decision diagrams

List Decision Diagrams (LDDs) [BVDP08; DP15; DP17], are a form of MDDs. They represent sets of integer vectors, such as sets of states in model checking. They are called list decision diagrams because instead of having one node with many edges, we have a linked list. Like MDDs, LDDs encode functions $(\mathbb{N}_{<v})^N \rightarrow \mathbb{B}$. Each internal node has a value v and two outgoing edges called the *right* and the *down* edge. Along the right (resp. down) edge, values v are in ascending order. Also, it is not possible to duplicate nodes. Figure 2.4 shows examples of MDD and LDD representing the same set of integer pairs where we hide edges to 0 to improve the readability. LDD nodes have a property called a level (and its dual, depth), which is defined as follows: the root node is at the first level, the nodes along right edges stay at the same level, while down edges lead to the next level. The depth of an LDD node is the number of down edges leading to leaf 1. All maximal paths from an LDD node have the same depth. Compared to MDDs, LDDs have numerous advantages [BVDP08; DP15]. In fact, valuations that lead to 0 simply do not appear in the LDD.

Definition 6 (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,* $val(p[x_i > v]) > v$.
5. *There are no duplicate nodes.*

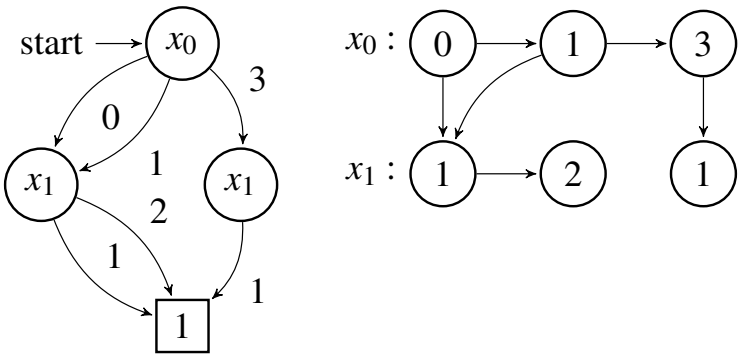


FIGURE 2.4: MDD (left) and LDD (right) representing the set $\{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 3,1\rangle\}$ (For simplicity, we hide paths to 0 for MDD and paths to 0 and 1 for LDD)

2.5 Linear Temporal Logic

Propositional linear time logic (LTL) is the basic prototypical logic used in formal verification. LTL is widely used for verification of properties of several concurrent systems (for example, safety and liveness), especially software systems.

Definition 7 (Hybrid LTL) *Given a set of atomic propositions AP and a set of actions Act, an LTL formula is defined inductively as follows:*

- *each member of* $AP \cup Act$ *is a formula,*
- *if* ϕ *and* ψ *are LTL formulae, so are* $\neg\phi$, $\phi \vee \psi$, $X\phi$ *and* $\phi U \psi$.

Other temporal operators e.g., F (eventually) and G (always) can be derived as follows: $F\phi = true \cup \phi$ and $G\phi = \neg F \neg \phi$.

An interpretation of an LTL formula is an infinite run $w = x_1 x_2 x_3 \ldots$ (of some *LKS*), assigning to each state a set of atomic propositions and a set of actions that are satisfied within that state. An atomic proposition is satisfied by a state s if it belongs to its label ($L(s)$) while an action is said to be satisfied within a state if it occurs from this state (in the current path). In our case (interleaving model of concurrency), where a single action can occur at a time, at most one action can be assigned to a state of a run. We write w^i for the suffix of w starting from x_i and $p \in x_i$, for $p \in AP \cup Act$, when p is satisfied by x_i. The hybrid LTL semantics is then defined inductively as follows:

- $w \models p$ iff $p \in x_0$, for $p \in AP \cup Act$,
- $w \models \phi \vee \psi$ iff $w \models \phi$ or $w \models \psi$,
- $w \models \neg\phi$ iff not $w \models \phi$,
- $w \models X\phi$ iff $w^2 \models \phi$, and
- $w \models \phi U \psi$ iff $\exists i \geq 1$; $w^i \models \psi$ and $\forall 1 \leq j < i$, $w^j \models \phi$.

An *LKS* K satisfies an LTL formula φ, denoted by $K \models \varphi$ iff all its runs satisfy φ.

It is well known that LTL formulae without the *next operator* (X) are invariant under the so-called *stuttering equivalence* [CGP00]. We use this equivalence relation to prove that event- and state-based SOGs preserves $LTL \setminus X$ properties. Stuttering occurs when the same atomic propositions hold on two or more consecutive states of a given path.

Checking an LTL formula over a *KS* is performed by analyzing its maximal paths.

Definition 8 (maximal paths)

Let T be Kripke structure and let $\pi = s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$ be a path of T. Then, π is said to be a maximal path if one of the two following properties holds:

- $s_n \not\rightarrow$,
- $\pi = s_1 \rightarrow \ldots \rightarrow s_m \rightarrow \ldots \rightarrow s_n$ *and* $s_m \rightarrow \ldots \rightarrow s_n$ *is a circuit.*

2.6 Symbolic Observation Graph

The Symbolic Observation Graph [KP08b; KTD11; HIK04] is an abstraction of the reachability graph of concurrent systems. The construction of a SOG is guided by the

set of atomic propositions (*AP*) occurring in the LTL formula to be checked. These *AP* are called observed atomic propositions while the others are unobserved.

Nodes of the SOG are called aggregates, and each of them may represent a set of states that are encoded efficiently using decision diagram techniques. Despite the exponential theoretical complexity of the size of a SOG (a single state can belong to several aggregates). We recall, in the following the definition of the SOG for both event- and state-based versions. The difference between the event- and the state-based versions is the aggregation criterion. In event-based version, observed atomic proposition corresponds to some actions of the system, so that an aggregate contains states that are connected by unobserved actions. In state-based version, observed atomic propositions are Boolean state-based ones, so that an aggregate regroups states with the same truth values of the observed atomic propositions.

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/MDDs structure for representing and manipulating sets of states, on the other hand.

It has been proven in [KP08b] that observation graphs can be used for model checking using an equivalence relation called *stuttering equivalence*. It is well known that LTL formulas without the next operator are invariant under the stuttering equivalence. Stuttering occurs when the same atomic propositions (label) holds on two or more consecutive states of a given path. We recall the definition of stuttering equivalence between two paths.

Definition 9 (Stuttering equivalence)
Let T and T' be two Kripke structures over an atomic proposition set AP and let $\pi = s_0 \rightarrow s_1 \rightarrow \ldots$ and $\pi' = r_0 \rightarrow r_1 \rightarrow \ldots$ be respectively paths of T and T'. π and π' are said to be stuttering equivalent, written as $\pi \sim_{st} \pi'$, if there are two sequences of integers $i_0 = 0 < i_1 < i_2 < \ldots$ and $j_0 = 0 < j_1 < j_2 < \ldots$ s.t. for every $k \geq 0, L(s_{i_k}) = L(s_{i_{k+1}}) = \ldots = L(s_{i_{k+1}} - 1) = L'(r_{j_k}) = L'(r_{j_{k+1}}) = \ldots = L'(r_{j_{k+1}} - 1)$.

2.6.1 Event-based SOG

Considering an LTS $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I \rangle$ and an $LTL \setminus X$ formula that represents the property to check, actions of the LTS are partitioned into two subsets ($Act = Obs \cup UnObs$). Subset *Obs* represents the actions that appear in the considered $LTL \setminus X$ formula. These actions are called observed actions. Subset *UnObs* consists of the other actions that are called unobserved ones. The building of a SOG is based on the idea that hiding the identities of unobserved actions does not alter the verification results [KP08a].

In the following, we define formally what an *aggregate* is, before providing a formal definition of a SOG associated with an LTS and a set of observed actions. Intuitively, an aggregate regroups states that are linked by unobserved transitions.

Definition 10 (Event-based aggregate)
Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I \rangle$ *be a labeled transition system with* $Act = Obs \cup UnObs$. *An* aggregate *is a tuple* $\langle S, d, l \rangle$ *defined as follows:*

1. S *is a nonempty subset of* Γ *satisfying:* $s \in S \Leftrightarrow Sat(s) \subseteq S$;
   $(Sat(s) = \{s' \in \Gamma \mid s \xrightarrow{*}_{UnObs} s'\}$ *i.e. the set of states that are reachable from* s *by unobserved action sequences only)*

2. $d \in \{true, false\}$; $d = true$ *iff* $\exists s \in S \mid s \nrightarrow$.

3. $l \in \{true, false\}$; $l = true$ *iff* S *contains an unobserved cycle (involving unobserved actions only);*

In the following, we provide a definition of the deterministic event-based SOG which is easier to understand.

Definition 11 (Deterministic event-based SOG)
The deterministic symbolic observation graph dSOG($\mathcal{T}$) *associated with an LTS* $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I \rangle$ *is an LTS* $\langle \mathcal{A}, Act', \rightarrow', I' \rangle$ *where:*

1. $\mathcal{A}$ *is a finite set of aggregates such that:*

   (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* $o \in Obs$, $(\exists s \in a.S, s' \in \Gamma \mid s \xrightarrow{o} s') \Leftrightarrow (\exists a' \in \mathcal{A} \mid a'.S = Sat(\{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{o} s'\}) \wedge (a, o, a') \in \rightarrow')$,

2. $Act' = Obs$,

3. $\rightarrow' \subseteq \Gamma' \times Act' \times \Gamma'$ *is the transition relation, obtained by applying 1b,*

4. $I' = \{a_0\}$,

A 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 (see [HIK04] for a possible construction algorithm).

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 generalizes the one given in [KTD09]. The construction algorithm given in [HIK04] is an implementation where the obtained graph is deterministic.

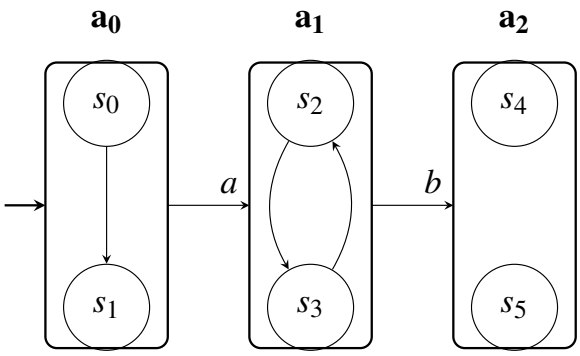


FIGURE 2.5: An event-based SOG with Obs={a,b}

Figure 2.5 illustrates the SOG associated with the LTS of Figure 2.1. The obtained SOG has a_0 as initial aggregate. The set of observed actions contains two elements $\{a,b\}$, unlabeled edges are supposed to be labeled by non observed actions.

2.6.2 State-based SOG

In the following, we give the definition of an aggregate and a state-based SOG structure according to a given KS.

Definition 12 (State-based aggregate)

Let $T = \langle \Gamma, L, \rightarrow, s_0 \rangle$ be a KS over an atomic proposition set AP . An aggregate a of T is a non empty subset of Γ satisfying $\forall s, s' \in a, L(s) = L(s')$.

We introduce three particular sets of states and two predicates. Let a and a' be two aggregates of T.

- Out (a) = $\{s \in a | \exists s' \in \Gamma\ a, s \rightarrow s'\}$
- Ext(a) = $\{s' \in \Gamma\ a | \exists s \in a, s \rightarrow s'\}$
- In (a,a') = $\{s' \in a'\ a | \exists s \in\ a, s \rightarrow s'\}$ (i.e. $I(a,a') = Ext(a) \cap a'$)
- Dead (a) = ($\exists s \in a$ s.t. $s \not\rightarrow$)
- $Live(a) =$ ($\exists \pi$ a circuit of a)

Let us describe informally, for an aggregate a, the meaning of the above notations: $Out(a)$ denotes the set of output states of a i.e. any state of a having a successor outside of a. $Ext(a)$ contains any state, outside of a, having a predecessor in a. Given an aggregate a', $In(a,a')$ denotes the set of input states of a' according to the predecessor a, notice that $In(a,a') = Ext(a) \cap a'$. Finally the predicate $Dead(a)$ (resp. $Live(a)$) holds when there exists a dead state (resp. a circuit) in a.

Definition 13 (State-based SOG)

Let $T = \langle \Gamma, L, \rightarrow, s_0 \rangle$ be a KS over an atomic proposition set AP. A symbolic observation graph of T is a 4-tuple $G = \langle \Gamma', L', \rightarrow', a_0 \rangle$ where :

1. $\Gamma' \subseteq 2^{\Gamma}$ is a finite set of aggregates
2. $L' : \Gamma' \rightarrow 2^{AP}$ is a labeling function satisfying $\forall a \in \Gamma'$, let $s \in a, L'(a) = L(s)$.
3. $\rightarrow' \subseteq \Gamma' \times \Gamma'$ is a transition relation satisfying:
   (a) $\forall a, a' \in \Gamma'$, *such that* $a \rightarrow' a'$,
      i. $a' \neq a \Rightarrow In(a,a') \neq \emptyset$ and a is compatible with $In(a,a')$
      ii. $a' = a \Rightarrow$ there exists a circuit π of a such that a is compatible with $\bar{\pi}$ and $\forall E \in In_G(a)$ there exists a circuit π_E of a such that a is compatible with $\bar{\pi}_E$ and $\exists e \in E, c \in \bar{\pi}_E$ satisfying $e \xrightarrow{*} a^c$
      Where $In_G(a) = \{E \subseteq a | E = \{S_0\} \vee \exists a' \in \Gamma' \ \{a\}, a' \rightarrow' a \wedge E = In(a,a')\}$
   (b) $\forall a \in \Gamma', Ext(a) = \bigcup_{a' \in \Gamma', a \rightarrow' a'} In(a,a')$.
4. $a_0 \in \Gamma'$ is the initial aggregate and is compatible with $\{s_0\}$

Example:

Figure 2.6 illustrates the SOG associated with the KS of Figure 2.2. The obtained SOG consists of 3 aggregates $\{a_0, a_1, a_2\}$ where a_0 is the initial one. Each aggregate is indexed with a triplet $(d, l, L(a))$. The states of the KS are partitioned into aggregates but this is not necessary the case in general.

2.7 Event-state based SOGs for LTL model checking

We propose to adapt the symbolic observation graphs [HIK04] in order to abstract systems' behavior while preserving hybrid LTL formulae. The Symbolic Observation Graph [KP08b; KTD11; HIK04] is an abstraction of the reachability graph of concurrent systems. The construction of a SOG is guided by the set of atomic propositions occurring in the LTL formula to be checked. Such atomic propositions are

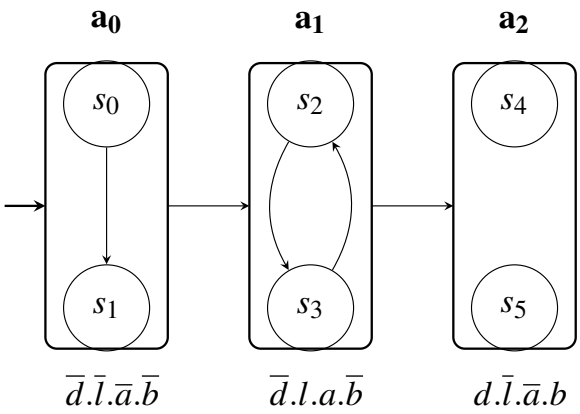


FIGURE 2.6: A state-based SOG, with $AP = \{a, b\}$

called observed while the others are unobserved. Nodes of the SOG are called aggregates, each of them is a set of states encoded efficiently using decision diagram techniques. Despite the exponential theoretical complexity of the size of a SOG (a single state can belong to several aggregates), its size is much more reduced than the original reachability graph.

The difference between the event- and the state-based versions of the SOG ([HIK04] and [KP08b; KTD11] respectively) is the aggregation criterion. In event-based version, observed atomic proposition correspond to some actions of the system and an aggregate contains states that are connected by unobserved actions. In state-based version, observed atomic propositions are Boolean state-based conditions and an aggregate regroups states with the same truth values of the observed atomic propositions.

In this section, we propose to define an event-state based SOG preserving hybrid LTL formulae (i.e., both state and action-based atomic propositions can be used within a same formula). The modeling framework consists of Labeled Kripke structures (*LKS*). The construction of the SOG depends on a set of actions *Act* and state variables appearing as atomic propositions *AP* involved by the formula to be checked.

2.7.1 Revisiting SOG for Hybrid LTL

The adaption of the *SOG* to hybrid LTL leads to a new aggregation criterium: (1) two states belonging to a same aggregate have necessarily the same truth values of the state-based atomic propositions of the formula, (2) For any state s in the aggregate, any state s', having the same truth values of the atomic propositions and being reachable from s by the occurrence of an unobserved action, belongs necessarily to the same aggregate, and (3) for any state s in the aggregate, any state s' which is

reachable from s by the occurrence of an observed action is necessarily not a member of the same aggregate (even if it has the same label as s), unless it is reachable from an other state s'' of the aggregate by unobserved action.

Definition 14 (Event-state based aggregate) *Let $\mathcal{K} = \langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ be an LKS over a set of atomic propositions AP and let $Obs \subseteq Act$ be a set of observed actions of $\mathcal{K}$. An aggregate a of $\mathcal{K}$ w.r.t. Obs is a triplet $\langle S, d, l \rangle$ satisfying:*

- $S \subseteq \Gamma$ *where:*
  - $\forall s, s' \in S,\ L(s) = L(s')$;
  - $\forall s \in S,\ (\exists (s', u) \in \Gamma \times (Act \setminus Obs) \mid L(s') = L(s) \wedge s \xrightarrow{u} s') \Leftrightarrow s' \in S$;
  - $\forall s \in S,\ (\exists (s', o) \in \Gamma \times Obs \mid s \xrightarrow{o} s') \wedge (\nexists (s'', u) \in S \times (Act \setminus Obs) \mid L(s'') = L(s') \wedge s'' \xrightarrow{u} s') \Leftrightarrow s' \notin S$.
- $d \in \{true, false\}$; $d = true$ *iff S contains a dead state.*
- $l \in \{true, false\}$; $l = true$ *iff S contains an unobserved cycle (i.e., with unobserved actions).*

Before defining the event- and state-based SOG, let us introduce the following operations:

- $SAT_{AP}(S)$: returns the set of states that are reachable from any state in S, by a sequence of unobserved actions and which have the same value of the atomic propositions as S. It is defined as follows:

  $SAT_{AP}(S) = \{s'' \in \Gamma \mid \exists s \in S, \exists \sigma \in UnObs^*, s \xrightarrow{\sigma} s'' \wedge \forall s' \in \Gamma, \forall \beta\ prefix\ of\ \sigma, s \xrightarrow{\beta} s' \Rightarrow L(s) = L(s')\}$.

- $Out(a,t)$: returns, for an aggregate a and a action t, the set of states outside of a that are reachable from some state in a by firing t. It is defined as follows:

  $$Out(a,t) \begin{cases} if\ t \in Obs & \{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{t} s'\} \\ if\ t \in UnObs & \{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{t} s' \wedge L(s) \neq L(s')\} \end{cases}$$

- $Out_\tau(a)$: returns, for an aggregate a, the set of states whose label is different from the label of any state of a, and which is reachable from some state in a by firing unobserved actions. It is defined as follows:

  $Out_\tau(a) = \bigcup_{t \in UnObs} Out(a,t)$.

- $Part_{AP}(S)$: returns, for a set of states S, the set of subsets of S that define the smallest partition of S according to the labeling function L. It is defined as follows:

$Part_{AP} : 2^{\Gamma} \longrightarrow 2^{2^{\Gamma}}$
$Part_{AP}(S) = \{S_1, S_2, ..., S_n\} \Leftrightarrow S = \bigcup_{i=1}^{n} S_i \wedge \forall i \in \{1..n\}, \forall s, s' \in S_i, L(s) = L(s') \wedge$
$\forall s \in S_i, \forall s' \in S_j, j \neq i, L(s) \neq L(s')$.

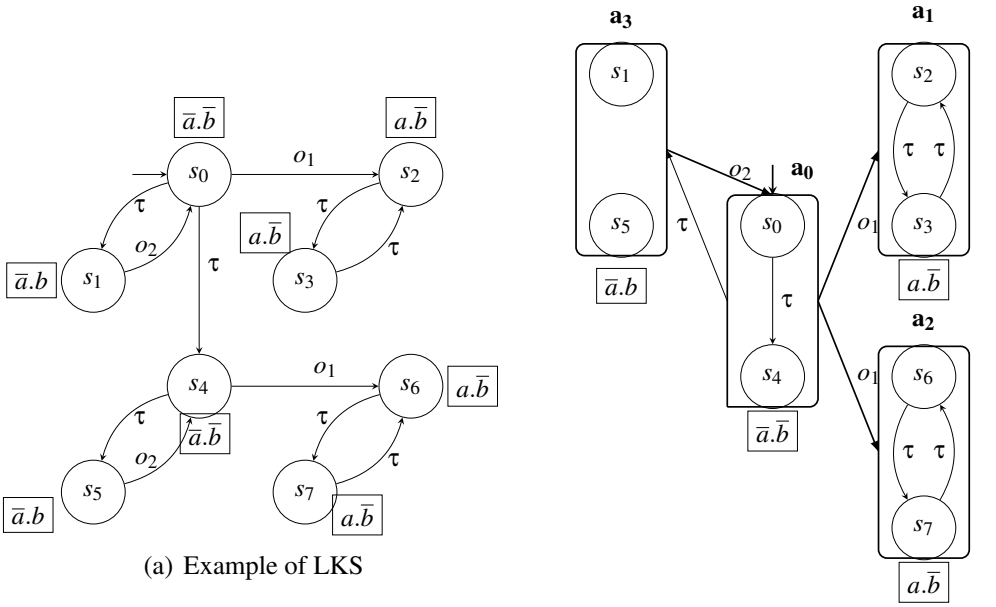
Definition 15 *Let $\mathcal{K} = \langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ be an LKS over a set of atomic propositions AP and let $Obs \subseteq Act$ be a set of observed actions of $\mathcal{K}$. The SOG associated with $\mathcal{K}$, over AP and Obs, is an LKS $\mathcal{G} = \langle A, Obs \cup \{\tau\}, L', \rightarrow', a_0, \Omega \rangle$ where:*

1. *A is a non empty finite set of aggregates satisfying :*
   - $\forall a \in A,\ \forall t \in Obs, \forall o_i \in Part(Out(a,t)), \exists a' \in A\ s.t.\ a' = SAT_{AP}(o_i)$
   - $\forall a \in A,\ \forall o_i \in Part(Out_{\tau}(a)), \exists a' \in A\ s.t.\ a' = SAT_{AP}(o_i)$
2. *$L' : A \rightarrow 2^{AP}$ is a labeling (or interpretation) function s.t. $L'(a) = L(s)$ for $s \in a.S$;*
3. *$\rightarrow \subseteq A \times Act \times A$ is the action relation where:*
   - $((a,t,a') \in \rightarrow') \Leftrightarrow ((t \in Obs) \wedge (\exists o_i \in Part(Out(a,t))\ s.t.\ SAT_{AP}(o_i) = a')$
   - $((a,\tau,a') \in \rightarrow') \Leftrightarrow (\exists o_i \in Part(Out_{\tau}(a))\ s.t.\ SAT_{AP}(o_i) = a')$
4. *a_0 is the initial aggregate s.t. $s_0 \in a.S$.*

The finite set of aggregates A of the SOG is defined in a complete manner such that the necessary aggregates are represented. The labeling function associated with a SOG gives to any aggregate the same label as its states. Point (3) defines the action relation: (1) there exists an arc, labeled with an observed action t (resp. τ), from a to a' iff a' is obtained by saturation (using SAT_{AP}) on a set of equally labeled reached states $Out(a,t)$ (resp. $Out_{\tau}(a)$) by the firing of t (resp. any unobserved action) from $a.S$. The last point of Definition 15 characterizes the initial aggregate.

Figure 2.7(b) illustrates an event-state based SOG corresponding to the LKS of Figure 2.7(a). The presented SOG consists of 4 aggregates $\{a_0, a_1, a_2, a_3\}$ and 4 edges. The initial aggregate a_0 is obtained by adding any state reachable from the initial state s_0 of the *LKS*, by unobserved sequences of actions only, and labeled similarly to s_0. For this reason, the initial aggregate contains the state s_4. State s_2, which is reachable from s_0 by an observed action o_1, is excluded from a_0 and belongs to a_1. The same holds for s_6 which is reachable from s_4 by o_1 and belongs to the aggregate a_2. s_3 (resp. s_7) is added to a_1 (resp. a_2) since it is reachable from s_2 (resp. s_6) by an unobserved action and since it is labeled similarly. Note that one can merge a_1 and a_2 since they have the same label.

According to Definition 15, the SOG associated with an *LKS* is unique. It can also be non deterministic since, for instance, an aggregate can have several successors with τ (when the reached states, by τ, have different labels).



(a) Example of LKS

(b) A corresponding SOG: $AP = \{a, b\}$ and $Obs = \{o_1, o_2\}$

FIGURE 2.7: An LKS and its SOG

2.7.2 Checking stuttering invariant properties on SOGs

The equivalence between checking a given stuttering invariant formula (e.g., $LTL \setminus X$ formula) on the new adapted SOG and checking it on the original reachability graph is ensured by the preservation of maximal paths (finite paths leading to a dead state and infinite paths). First, the SOG preserves the observed traces of the corresponding model which allows to preserve infinite runs involving infinitely often observed transitions. Then, the truth value of the state-based atomic propositions occurring in the formulae are visible on the SOG by labeling each aggregate with the atomic propositions labeling (all) its states. Finally, the d and l attributes of each aggregate allows to detect deadlocks and livelocks (unobserved cycles) respectively. Note that the detection of the existing of dead states and cycles inside an aggregate is performed using symbolic operations (decision diagram-based set operations) only.

In conclusion, the following result establishes that an *LKS* satisfies an $LTL \setminus X$ formula iff the corresponding SOG does.

Theorem 16 *Let $\mathcal{K}$ be an LKS and let $\mathcal{G}$ be the corresponding SOG over Obs and AP. Let φ be an $LTL \setminus X$ formula on a subset of $Obs \cup AP$. Then $\mathcal{K} \models \varphi \Leftrightarrow \mathcal{G} \models \varphi$*

Proof of Theorem 16

To prove Theorem 16, we will prove that the SOG preserves the maximal paths of the corresponding *LKS*. We recall that maximal paths (of finite systems) are any path π satisfying one of the following requirements:

1. $\pi = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} s_n$ such that s_n is a dead marking
2. $\pi = s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_l} s_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} s_n$ such that $s_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} s_n$ is a circuit.

Before giving the proof of the preservation of maximal paths, let us present two lemmas about the correspondence between paths of $\mathcal{K}$ and those of $\mathcal{G}$.

Lemma 17 *Let $\pi = s_1 \xrightarrow{t_2} s_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n$ be a path of $\mathcal{N}$ and a_1 be an aggregate of $\mathcal{G}$ such that $s_1 \in a_1$. Then, there exists a path $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t'_l} a_l$ of $\mathcal{G}$ and a strictly increasing sequence of integers $i_1 = 1 < i_2 < \cdots < i_{l+1} = n+1$ satisfying $\{s_{i_k}, s_{i_k+1}, \cdots, s_{i_{k+1}-1}\} \subseteq a_k$ for all $1 \leq k \leq l$.*

Proof 18 *We proceed by induction on the length of π. If $n = 1$, knowing that $s_1 \in a_1$ concludes the proof. Let $n > 1$ and assume that $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t'_{l-1}} a_{l-1}$ and $i_1, \cdots, i_l$ correspond to the terms of the lemma for the path $s_1 \xrightarrow{t_2} s_2 \xrightarrow{t_3} \cdots \xrightarrow{t_{n-1}} s_{n-1}$. Then, $s_{n-1} \in a_{l-1}$. Let us distinguish two cases.*

(i) If $t_n \in UnObs \wedge L(s_{n-1}) = L(s_n)$ then, by definition of aggregates, $s_n \in a_{l-1}$. Thus both the path $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t'_{l-1}} a_{l-1}$ and the sequence $i_1, \cdots, i_l$ for the path of length $n-1$ stand for the path of length n as well.

(ii) If $t_n \in Obs \vee L(s_{n-1}) \neq L(s_n)$ then, since $s_{n-1} \xrightarrow{t_n} s_n$, there exists (by definition of the SOG) an aggregate a_l such that $a_{l-1} \xrightarrow{t_n} a_l$ and $s_n \in a_l$. As a consequence, the path $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t_{l-1}} a_{l-1} \xrightarrow{t'_l = t_n} a_l$ and the sequence $i_1, \cdots, i_l, i_l + 1$ satisfy the proposition.

The next lemma shows that the inverse also holds.

Lemma 19 *Let $\pi = a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} a_n$ be a path of $\mathcal{G}$. Then, there exists a path $s_1 \xrightarrow{\sigma_1} s'_1 \xrightarrow{t_2} s_2 \xrightarrow{\sigma_2} s'_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n \xrightarrow{\sigma_n} s'_n$ of $\mathcal{N}$ s.t., $\forall i = 1 \ldots n$, $\sigma_i \in UnObs^*$, s_i, s'_i belong to $a_i.S$ and all the traversed states from s_i and s'_i by firing σ_i have the same label (the same truth value of state-based atomic propositions).*

Proof 20 *Let $\pi = a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} a_n$ be a path of $\mathcal{G}$. First, let us define the four following functions related to π and any aggregate a_i of π, for $i = 1 \ldots n$.*

- $In_\pi(a_i) = \begin{cases} \{s' \in a_i.S \mid \exists s \in a_{i-1} : s \xrightarrow{t_i} s'\} & \text{if } i \neq 1; \\ Out_\pi(a_i) & \text{otherwise.} \end{cases}$
- $Out_\pi(a_i) = \begin{cases} \{s \in a_i.S \mid s \xrightarrow{t_i}\} & \text{if } i \neq n; \\ In_\pi(a_i) & \text{otherwise.} \end{cases}$

- $Candidate_\pi(a_i) = \begin{cases} \{s \in In_\pi(a_i) \mid \exists s' \in Candidate'(a_i), \exists \sigma \in UnObs^* : s \xrightarrow{\sigma} s'\} & \text{if } i \neq 1; \\ Candidate'_\pi(a_i) & \text{otherwise.} \end{cases}$
- $Candidate'_\pi(a_i) = \begin{cases} \{s \in Out_\pi(a_i) \mid \exists s' \in Candidate(a_{i+1}), s \xrightarrow{t_{i+1}} s'\} & \text{if } i \neq n; \\ Candidate_\pi(a_i) & \text{otherwise.} \end{cases}$

Informally, $In_\pi(a_i)$ (for $i = 2 \ldots n$) represents the set of input *states of aggregate a_i that are immediately reached by firing t_i from states in a_{i-1}. All the states of a_i are then obtained by adding the successors of this set of states by unobservable sequences, while having the same truth values of the state-based atomic propositions. For the first aggregate, $In_\pi(a_1)$ is the set of* output *states $Out_\pi(a_1)$ i.e., the states in $a_1.S$ enabling t_2. The same holds for $Out_\pi(a_i)$, for any $i = 1 \ldots n-1$ i.e., it contains states enabling action t_{i+1}. For a_n, $Out_\pi(a_n) = In_\pi(a_n)$.*

Sets $Candidate_\pi(a_i)$ and $Candidate'_\pi(a_i)$, for $i = 1 \ldots n$ represent sets of states from which the states s_i and s'_i could be chosen, respectively, in order to build the path $s_1 \xrightarrow{\sigma_1} s'_1 \xrightarrow{t_2} s_2 \xrightarrow{\sigma_2} s'_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n \xrightarrow{\sigma_n} s'_n$ in a reverse order (starting from the end). In fact, $s'_n = s_n$ can be first chosen from $Candidate_\pi(a_n) = In_\pi(a_n)$. Then s'_{n-1} is obtained from $Candidate'_\pi(a_{n-1})$ i.e. states in $a_{n-1}.S$ enabling t_n and thus leading to $Candidate_\pi(a_n)$. s_{n-1} can be chosen from $Candidate_\pi(a_{n-1})$ i.e. to reach $Candidate'_\pi(a_{n-1})$ by unobservable actions only (and without traversing a differently labeled state), ... and so on.

We are now in position to study the correspondence between maximal paths to prove Theorem 16 through two new lemmas.

Lemma 21 *Let $\pi = s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_n$ be a maximal path of $\mathcal{K}$. Then, there exists a maximal path $\pi' = a_0 \xrightarrow{t'_1} \cdots \xrightarrow{t'_l} a_l$ of $\mathcal{G}$ such that there exists a sequence of integers $i_0 = 0 < i_1 < \cdots < i_{l+1} = n+1$ satisfying $\{s_{i_k}, s_{i_k+1}, \cdots, s_{i_{k+1}-1}\} \subseteq a_k$ for all $0 \leq k \leq l$.*

Proof 22 *If s_n is a dead marking then knowing that $s_0 \in a_0$ and using Lemma 17, we can construct a path $\pi' = a_0 \xrightarrow{t'_1} a_1 \cdots \xrightarrow{t'_l} a_l$ and the associated integer sequence corresponding to π. Because the last visited state of π belongs to a_l, the dead attribute of a_l is necessarily equal to true and π' is then a maximal path of the SOG.*

Now, if s_n is not a dead marking then, one can decompose π as follows: $\pi = \pi_1 \pi_2$ s.t. $\pi_1 = s_0 \xrightarrow{t_1} s_1 \rightarrow \cdots \xrightarrow{t_{k-1}} s_k$ and $\pi_2 = s_k \xrightarrow{t_{k+1}} \cdots \xrightarrow{t_n} s_k$ s.t., π_2 is a circuit. Once again, applying Lemma 17 from s_0, one can construct a path $\pi'_1 = a_0 \xrightarrow{t'_1} a_1 \xrightarrow{t'_2} \cdots a_m$ corresponding to π_1. The path in $\mathcal{G}$ associated with π'_2 can be also constructed applying the same lemma. However, this path must be constructed from a_m to which belongs s_k. π_1 and π_2 can be chosen as follows:

1. *If π_2 involves unobserved actions and all traversed states (by π_2) have the same truth value of state-based atomic propositions, then π_2 is a cycle inside aggregate a_m and the* live *attribute (cycle) of a_m is necessarily set to true.*

2. *otherwise (i.e. either π_2 involves observed actions, or unobserved actions that change the truth value of state-based atomic propositions), then we chose the subpaths such that t_{k+1} as an observed action, or an unobserved one s.t. $(L)(s_k) \neq (L)(s_{k+1})$. By definition of the SOG, since $s_k \in a_m$, there exists an aggregate a_{o_1} successor of a_m by action t_{k+1}. By using Lemma17, and the definition of the SOG, let $\pi'_1 = a_0 \xrightarrow{t'_1} a_1 \xrightarrow{t'_2} \cdots a_m$ and $\pi'_2 = a_{o_1} \xrightarrow{t_{p_1}} \ldots \xrightarrow{t_l} a_{q_1}$ with $s_k \in a_{q_1}.S$. If $a_{q_1} \xrightarrow{t_{k+1}} a_{o_1}$, then π'_2 is a circuit of $\mathcal{G}$ and $\pi'_1\pi'_2$ is a maximal path of $\mathcal{G}$ satisfying the proposition. Otherwise, by construction of the SOG, there exists an other successor of a_{q_1} containing s_k. Applying again Lemma 17 from this aggregate, we can construct a new path in $\mathcal{G}$ corresponding to π_2. Let $a_{o_2} \xrightarrow{t'_{p_1}} \ldots a_{q_2}$ be this path. If we can deduce a circuit of $\mathcal{G}$ from this path (if $a_{q_2} \xrightarrow{t_{k+1}} a_{o_2}$), this concludes the proof. Otherwise, we can construct a new path corresponding to π_2 starting from a successor of a_{q_2}. Because the number of aggregates in $\mathcal{G}$ is finite, in particular the number of aggregates to which belongs s_k is bounded by 2^N (where N is the number of state in the original LKS), a circuit will be necessarily obtained.*

Notice that for all the previous cases above, a sequence of integers can be easily constructed from the ones produced by Lemma 17.

Lemma 23 *Let $\pi' = a_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} a_n$ be a maximal path of $\mathcal{G}$. Then, there exists a maximal path $s_0 \xrightarrow{\sigma_1} s'_1 \xrightarrow{t_2} s_2 \xrightarrow{\sigma_2} s'_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n \xrightarrow{\sigma_n} s'_n$ of $\mathcal{K}$ s.t., $\forall i = 1 \ldots n$, $\sigma_i \in UnObs^*$ and s_i, s'_i belong to $a_i.S$.*

Proof 24 *Let π' be a maximal path reaching an aggregate a_n such that $a_n.d = true \vee a_n.l$ (either the dead or the livelock attribute (cycle) is true). First, let us notice that the proof is trivial if the path π' is reduced to a single aggregate because dead state (resp. a state containing a circuit of a_0) is necessarily reachable from s_0.*

Otherwise, using the same principle of Lemma 19 proof, one can demonstrate the existence of the maximal path in $\mathcal{K}$. We have just to define $In_\pi(a_0)$ as the singleton $\{s_0\}$ and $Out_\pi(a_n)$ as the dead state (if $a_n.d = true$) or the set of states forming a cycle in a_n (if $a_n.l = true$).

Now, if neither $a_n.d$ nor $a_n.l$ is true, then by construction of the SOG, $\pi' = a_0 \xrightarrow{t_1} \cdots \xrightarrow{t_l} a_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} a_n$ with $a_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} a_n$ a circuit of $\mathcal{G}$ i.e., $a_n = a_l$. Here also, we can use the same scheme as for the proof of Lemma 19 by defining $In_\pi(a_0)$ as the singleton $\{s_0\}$ and $Out_\pi(a_n)$ as the set of states in a_n enabling t_{l+1}. Thus, starting

from these states, and using the functions $Candidate_\pi()$ *and* $Candidate'_\pi()$ *as defined in Lemma 19, one can build by backtracking the maximal path in* $\mathcal{K}$ *satisfying the terms of Lemma 23.*

Chapter 3

Multi-threaded approach for the parallel generation of the SOG

Contents

1.1 Model Checking 2
1.2 Existing parallel and distributed model checking methods . . . 4
1.3 Aims and contributions of this work 7
1.4 Overview and Reading Guide 8

3.1 Introduction

A symbolic observation graph (SOG) provides the advantage to represent the state space in a condensed representation based on the use of binary decision diagrams (BDD). Indeed, a SOG represents an abstraction of the system on which the verification of a considered LTL/X property is equivalent to the verification on the original reachability graph. In practice, such a SOG has a very moderate size allowing to tackle efficiently the explosion state problem. However, computing a SOG requires an important time that is can be considered as a bottleneck. As a solution, we propose to a parallelize the computation of a SOG. For this purpose, we propose in this chapter a multi-threaded approach that exploits multi-core machines [Oun+17a]. The basic idea is to parallelize the building of a SOG by using several threads, such that every thread builds a part of the graph. A dynamic load balancing scheme is used in order to balance the load on threads sharing the SOG construction task. Further, we propose to enhance the condensation of a SOG by using Multi-valued Decision Diagrams (MDDs) instead of BDDs [MD98].

3.2 Multi-threaded algorithm for constructing the SOG

3.2.1 Aims and Hypothesis

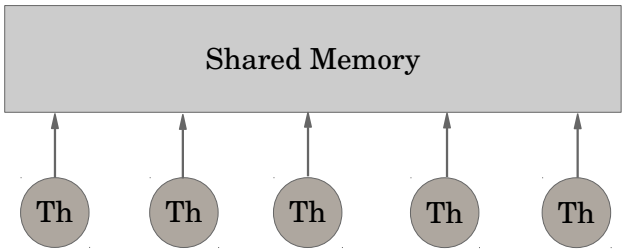


FIGURE 3.1: Shared memory multi-core

Considering an LTS and an LTL property to be checked, our aim is to build the symbolic observation graph of the LTS with a multi-threaded approach. In such setting, parallel treatments (threads) are assumed to be executed in the same machine (see Figure 3.1). Thus, threads can share the same memory space ensuring no communication overhead. The main advantage that shared memory architecture offers over distributed memory is that it provides a shareable memory space for concurrent manipulation, thus avoiding the need of passing messages among the processors. As a consequence, there is no more need for a slicing function to partition the state space because the storage structure is shared among the processors. However, it imposes other difficulties related to data consistency and synchronization operations to manipulate the shared data. Data consistency is mandatory to ensure that any processor is accessing the most recent update of the global data. Indeed, concurrent threads can lead to unexpected behavior. In order to avoid concurrent writes on sensitive data, mutual exclusions (mutexes) around critical sections must be used [CH91].

The basic idea of our proposed approach, for the construction of a SOG, is that each thread takes charge of the construction of some aggregates that will be stored in the shared memory space. The distribution of the aggregates on different threads should be well balanced. In order to maximize the number of treatments to be executed simultaneously by different threads, and consequently, obtaining a better speedup.

The decision to allocate the construction of an aggregate to one thread is made after comparing current loads (the number of aggregates to be processed) of individual threads. The number of threads is set at the beginning by the user and the same

algorithm is executed by all threads.

3.2.2 Description of the algorithm

```
   Data: LTS⟨Γ, Obs ∪ UnObs, →, I⟩
   Result: SOG⟨Γ′, Obs, →′, I′⟩
1  Load : Table[1,..,M] integer;
2  Termination: Table[1,..,M] of boolean;
3  Waiting_idthread; /* a stack associated with the thread having as id 'idthread' */
4  if idthread==1 then
5      A_0=Aggregate(I);
6      Waiting_idthread = {A_0};
7      Load[idthread] = 1;
8      Termination[idthread] = false;
9  else
10     Waiting_idthread = ∅
11     Load[idthread] = 0;
12     Termination[idthread] = true;
13 while DetectTermination == false do
14     while Waiting_idthread ≠ ∅ do
15         Termination[idthread] = false;
16         mutex lock[idthread];
17         Choose A ∈ Waiting_idthread;
18         Load[idthread] = Load[idthread] − 1;
19         mutex unlock[idthread];
20         foreach a ∈ Obs do
21             if enabled(A,a) then
22                 S′ = succ(A,a);
23                 A′ = Aggregate(S′);
24                 if ∃A″ such that A′ == A″ then
25                     Insert the arc(A,a,A′) in the SOG;
26                 else
27                     Γ = Γ ∪ {A′};
28                     Insert the arc(A,a,A′) in the SOG;
29                     int j = minCharge();
30                     mutex lock[j];
31                     Waiting_j = Waiting_j ∪ {A′};
32                     Load[j] = Load[j] + 1;
33                     mutex unlock[j];
```

Algorithm 1: A Multi-Threaded algorithm for constructing the SOG

Algorithm 1 builds the symbolic observation graph in a multi-threaded setting of a given LTS where its observable and unobservable actions are specified. The

same algorithm is executed by all threads. Each thread is identified by its identifier *idthread*. The table *Load* indexed by *idthread* 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 *idthread*, 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 3-11) from the initial states by executing unobservable actions 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 executing observable enabled actions. Each 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 labeled with the executed observable action, 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 at the same moment.

We now justify informally the correctness of Algorithm 1, assuming the sequential algorithm is correct [KP08a]. Algorithm 1 follows directly the same arguments as in the case of sequential algorithm. Let G be a Symbolic Observation Graph associated with a labeled transition system T and generated by Algorithm 1. G respects definition 11. On the other hand, to avoid concurrent access on sensitive data, we use mutual exclusions (mutexes) mechanisms.

Also, Algorithm 1 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. The function $DetectTermination$ returns $true$ if:

$$\begin{cases} \forall i \in [0,M], Termination[i] = true \\ \forall i \in [0,M], Load[i] = 0 \end{cases}$$

3.3 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.

It is worth noting that using sequential BDDs or MDDs implementation in distributed algorithms is not evident, since proposed packages use always a global 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. Since BuDDy is not based on a thread safe hash table we had to either intervene at the library level and to protect the hash table or re implement the SOG based on another multi-core package.

3.3.1 Multi-core decision diagram packages

There have been different initiatives to implement parallel BDDs on multi core machines. More recently, there are some works that have tried to propose parallel BDD or thread safe implementations.

We were interested in the used hash table by each approach. The hash table in BDD packages needs to support concurrently execution of one key operation, which returns the key if it already exists and inserts the key otherwise.

In [He09], 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. Therefore, they do not provide public download and evaluation because they did not develop a parallel version supporting full functionality of BuDDy.

In a thesis on JINC [Oss10], a multi-threaded extension is described for BDDs by using several hash tables to store BDDs but JINC does not parallelize the basic BDD operations. JINC uses spin locks [And90] instead of using mutex to avoid concurrent access by several threads for every unique-table. Spin locks is a lock which causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Then, the thread remains active but without performing a useful task. Indeed, searching for a node in the unique-table can be implemented without any locking.

Sylvan[2] [DP15] is a parallel (multi-core) MTBDD library. Sylvan implements parallelized operations on BDDs, MTBDDs and LDDs by using a lockless one hash

[1] http://sourceforge.net/projects/buddy
[2] https://github.com/utwente-fmt/sylvan

table. The hash table in Sylvan is based on the hash table, described in [LPW10], that is designed to store visited states in model checking. The structure of this hash table has been modified several times and new versions have adhered to it [VDLVDP13; DP15; DP16]. The hash tables store fixed-size decision diagram nodes (16 bytes for each node) and strictly separates lookup and insertion of nodes from a stop-the-world garbage collection phase, during which the table may be resized. Garbage collection is essential for manipulating decision diagrams. Most operations on a subgraph of decision diagrams implies the modification of all ancestors in that structure and most operations on decision diagrams continuously create new nodes in the nodes table. Therefore, unused nodes should be deleted to free space for new nodes. The implementation of stop-the-world garbage collection in Sylvan is based on the Work-stealing framework Lace.

Their results are promising. Compared to BuDDy, Sylvan have better performance when using multiple workers and lower performance when using one worker [VDLVDP13]. We have chosen to use Sylvan for the implementation of the parallel version of SOG instead of BuDDy.

3.3.2 Symbolic encoding of the SOG

Since, the results in [DP15] show that the majority of models especially large models, were processed to several orders of magnitude faster using LDDs, we used and adapted the LDD extension of Sylvan to implement our algorithm. An advantage of the LDD is a reduction in memory accesses needed to evaluate it compared with the BDD from which it was derived. Indeed, grouping n binary inputs together to form a single LDD variable reduces computation time by a factor of n.

Assuming that we have a set of two states $s1$ and $s2$, identified by three variables $v1$, $v2$ and $v3$ whose values are (2,2,3) and (2,2,0). Figure 3.2 illustrates the representation of $s1$ and $s2$ using BDDs and LDDs. Like MDDs, LDDs are a generalization of BDDs to the integer domain. A node in a LDD may represent several nodes in a BDD.

3.3.3 Adaptation of Sylvan to the parallel implementation of the SOG

We have chosen to use Sylvan package to implement our approach, since this package allows to manipulate BDDs and LDDs while providing a thread safe hash table. Therefore, the table can be shared between several concurrent threads. Indeed, Sylvan package provides an implementation of multi-thread operations for manipulating LDDs and BDDs.

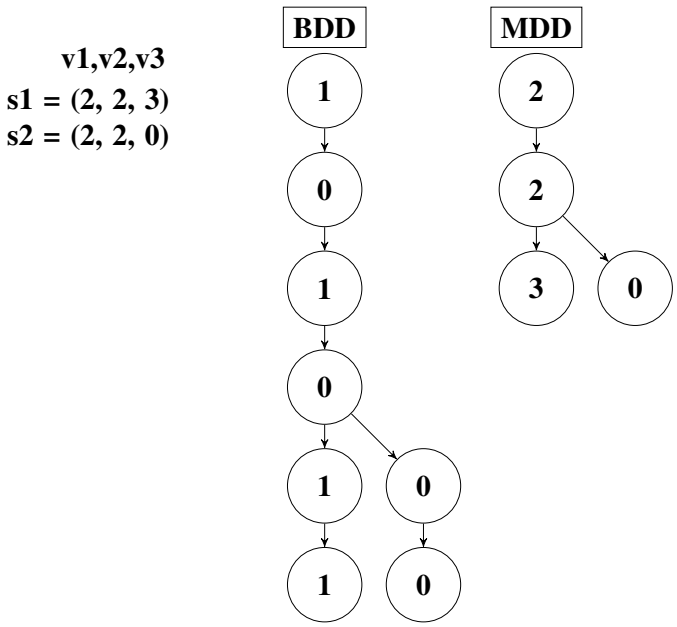


FIGURE 3.2: Example of BDD and MDD

The level of parallelization in the implementation provided by Sylvan package is lower than what we propose in our approach for the building of a SOG, i.e. in Sylvan parallelization is done at the level of elementary operations manipulating BDD (or LDD) structures, whereas, in our approach, the parallelization is to be performed at the level of the building of nodes of a SOG. Then, in addition to the implementation of specific functions in order to deal with Petri net models that are considered by our experiments, it was necessary to rewrite some routines of Sylvan package in order to take into account aforementioned aims.

We have experimentally tested the implementation and we have made a simple comparison between the SOG based on Buddy (using BDD encoding) and the SOG based on Sylvan (LDD encoding). Results are reported in the subsection 3.4.1.

Afterward, we have moved to the parallelization of the new version of SOG using the proposed multi-threaded algorithm. The issue in this case is that we can not use the garbage collector in the same way as used by Sylvan. In fact, to initiate garbage collection, Sylvan use a feature in the Lace framework that suspends all current work and starts a new task tree. This task suspension is a cooperative mechanism. Workers regularly check whether the current task tree is being suspended, either explicitly by calling a method from the parallel framework, or implicitly when creating or synchronizing on tasks. Since threads created by Pthread library are not controlled by Lace, then we have to trigger the garbage collection by one of the threads. During garbage collection we have to prevent, by mutexes, other threads to insert nodes in the hash table, and also to protect temporary results from being erased.

3.4 Experimental results

In the current section, we present and evaluate the results of our experiments.

3.4.1 Comparison BuDDy - Sylvan

Table 3.1 presents the results of the construction of the SOG using two different decision diagram packages BuDDy and Sylvan. All the tested examples are parameterized and the size of the reachable states space is exponential with respect to the parameter value (e.g. the Dining Philosophers, a well known academic example; is parameterized with the number of philosophers). The philosophers, ring and fms examples are taken from [CLS00], while the RobotManipulation, CircularTrains and ERK examples are taken from the Model Checking Contest benchmark[3].

The RobotManipulation model can be given in several instances parameterized by scaling parameters. It may not be very realistic in terms of modeling, but it scales up well. In this model, processes manipulate robots following a simple protocol.

CircularTrains model was presented in [BRR06]. On a circular railroad divided in S sections, $S/3$ trains circulate in the same direction. For security reasons, a segment may never contains more than one train at a time. Traffic lights manage the access to each sections.

ERK is a short name for "RKIP/MEK-ERK signalling pathway". The RKIP inhibited ERK pathway discussed as three related Petri net models in [HDG10].

For each model (column 1), we give the number of observed actions used for the construction (2), its number of reachable markings (3) and the number of the aggregates (4) and the arcs (5) of the SOG. We have measured the time in seconds of the SOG construction on sequential way using Buddy (6) and Sylvan (7). Results show that BuDDy is faster than Sylvan for some models and Sylvan is faster for others. What was unexpected was the very important difference between the runtime of the SOG construction based on BuDDy and Sylvan for the models robots and trains. Sylvan in this case has a better performance compared to BuDDy.

3.4.2 Results of the multi-threaded algorithm

The implementation of the multi-threaded approach is available in the PMCSOG project[4].

[3] https://mcc.lip6.fr/models.php
[4] https://depot.lipn.univ-paris13.fr/PMC-SOG/thread-sog

Model (1)	Obs (2)	Size (3)	Agg (4)	Arcs (5)	Buddy (6)	Sylvan (7)
ring4	8	5136	304	1280	2.18	2.15
ring5	10	53856	1632	8320	85.8	43.46
ring6	12	575296	3805	20698	609.75	870.87
philo6	12	5778	64	384	0.21	0.47
philo8	16	103682	256	2048	4.42	11.94
philo10	20	1.86×10^6	1024	10240	73.98	311.10
fms4	4	438600	266	830	260.77	208.11
fms5	4	2.89×10^6	93280	519972	1622.89	2088.75
robot4	6	48620	2574	11649	40.37	7.10
robot5	6	184756	6006	28600	226.47	36.16
robot6	6	587860	12376	61061	1043.26	135.09
robot7	6	1.63×10^6	23256	117776	3995.65	505.90
train2	4	86515	81	188	468.43	42.49
train2	6	86515	178	448	323.18	29.64
train2	8	86515	1660	6696	1136.14	26.73
erk10	4	47047	550	1803	38.72	46.63
erk20	8	1.69×10^6	21230	15297	1223.68	957.68

TABLE 3.1: Comparison between the SOG construction based on BuDDy and Sylvan

We have executed the multi-threaded algorithm 1 on various models, in order to measure the performance of the parallel construction of the SOG. We have implemented the algorithm 1 using C++. The implementation of the algorithm uses the Pthread library, that provides a mutual exclusion mechanism to avoid critical runs on shared variables between threads.

We made scaling experiments on the Magi cluster[5] of Paris 13 university. This cluster has 12 processors each with 12 cores (two Xeon X5670 at 2.93GHz), 24GB of RAM and they are connected by an InfiniBand network.

Table 3.2 summarizes the results for different representative models. These experiments are based on a parallel package Sylvan. First, we have measured the time in seconds consumed by the construction of SOG in a sequential way using Sylvan (4). Then, we have measured the runtime of our multi-threaded algorithm (algorithm 1) by progressively increasing the number of threads (5)-(10). Furthermore, we are

[5]http ://www.univ-paris13.fr/calcul/wiki/

Model (1)	Obs (2)	Size (3)	Seq (4)	Th2 (5)	Th4 (6)	Th6 (7)	Th8 (8)	Th10 (9)	Th12 (10)	SpMax (11)
ring4	8	5136	2.15	1.17	0.65	0.50	0.46	0.43	0.42	5.1
ring5	10	53856	43.46	22.20	11.28	8.54	6.80	6.04	5.71	7.6
ring6	12	575296	870.87	437.61	227.32	172.73	137.93	109.09	87.70	9.9
philo6	12	5778	0.47	0.29	0.19	0.15	0.12	0.11	0.11	4.3
philo8	16	103682	11.94	6.18	3.53	2.77	2.29	2.19	1.89	6.3
philo10	20	1.86×10^6	311.10	160.74	88.97	61.35	52.49	49.78	36.67	8.5
fms4	4	438600	208.11	90.15	58.75	28.18	24.66	22.03	21.85	10.5
fms5	4	2.89×10^6	2088.75	933.15	476.36	381.50	282.70	229.80	222.82	9.3
robot4	6	48620	7.10	3.21	1.77	1.35	1.23	1.01	0.92	7.7
robot5	6	184756	36.16	16.26	8.62	6.81	5.48	5.21	4.86	7.4
robot6	6	587860	135.09	66.94	35.72	26.63	22.46	19.64	17.75	7.61
robot7	6	1.63×10^6	505.90	229.06	118.81	91.27	76.42	67.69	61.97	8.1
train2	4	86515	42.49	26.31	14.87	14.29	13.10	13.08	12.58	3.5
train2	6	86515	29.64	18.22	9.90	7.78	6.37	6.28	5.83	5.1
train2	8	86515	26.73	13.75	7.15	5.35	4.11	3.48	2.96	9.0
erk10	4	47047	46.63	28.06	14.18	9.78	7.82	6.23	5.11	9.1
erk20	8	1.69×10^6	957.68	474.09	241.48	162.33	123.89	101.24	86.44	11.0

TABLE 3.2: Experimental results of the multi-threaded algorithm

interested in the speedup of the building of the SOG (11). The speedup is a measure for the performance gain of parallelizing an algorithm and it is calculated relative to one thread.

Figure 3.3 shows the obtained runtime . As it can be observed, for all the testes examples, the runtime decreases by increasing the number of threads.

Figure 3.4 illustrates the obtained speedups for the examples ring4, ring5 and ring6. It can be seen that the achieved speedups are largely dependent on the size of the graph. This explains why the speedup obtained for the ring6 example is better than for the ring4. Since, the speedup figures are closer to ideal for larger input sizes, as well as for higher number of threads.

3.5 Conclusion

In this chapter, we have 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 this algorithm of construction of the SOG scales reasonably well with increasing the numbers of threads. The efficiency of our approach has been tested on various

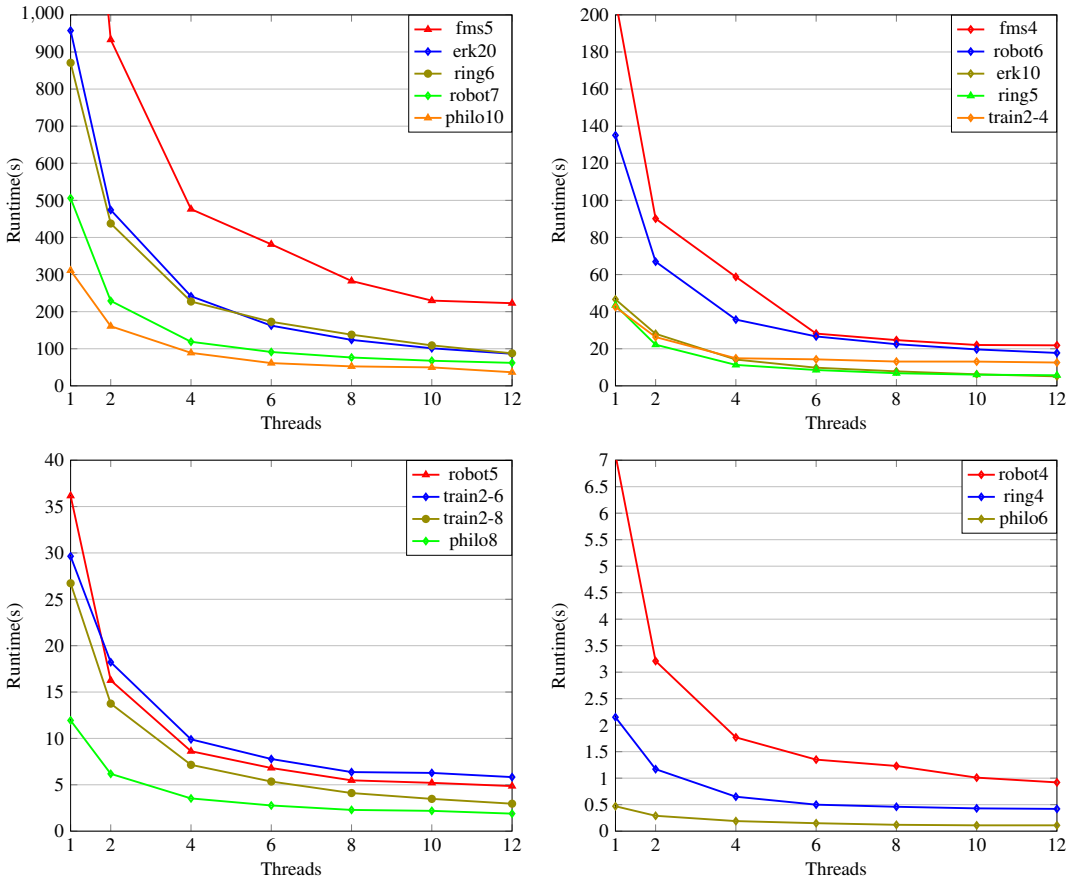


FIGURE 3.3: Runtime of the construction of the SOG

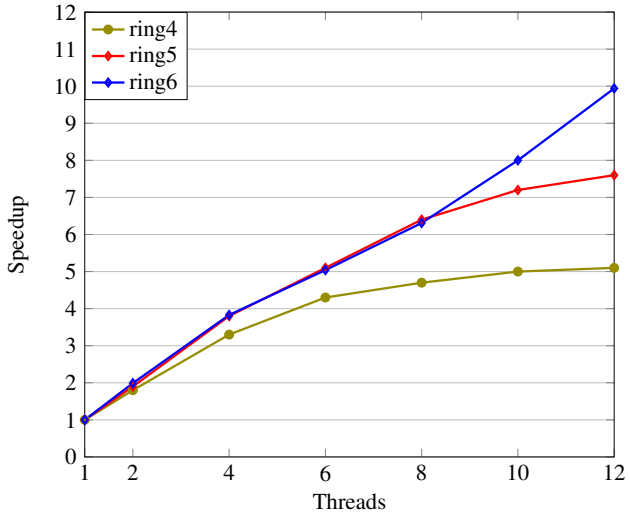


FIGURE 3.4: Speedup of the multi-thread algorithm

examples. Experimental results seem promising due to the obtained reduction in runtime.

Chapter 4

Distributed approach for the construction of the SOG

Contents

2.1 Labeled transition system 11

2.2 Kripke structure 12

2.3 Labeled Kripke Structure 13

2.4 Decision Diagrams 14

2.4.1 Binary Decision Diagrams 14

2.4.2 Multi-valued Decision Diagrams 15

2.4.3 List decision diagrams 15

2.5 Linear Temporal Logic 16

2.6 Symbolic Observation Graph 17

2.6.1 Event-based SOG 18

2.6.2 State-based SOG 20

2.7 Event-state based SOGs for LTL model checking 21

2.7.1 Revisiting SOG for Hybrid LTL 22

2.7.2 Checking stuttering invariant properties on SOGs 25

4.1 Introduction

In the previous chapter, we have proposed a multi-threaded algorithm for computing SOG. Comparing to sequential algorithm, results show interesting performance improvement in execution time. However, multi-threaded algorithms are based on a shared memory platform. This is can be considered as a limitation, since producing shared-memory machines with increasing number of processors is expensive

and difficult, as they require complex hardware cache controllers [AB86]. In this work, we propose a distributed algorithm that is based on distributed memory platform [Oun+17b]. We compare and evaluate the three algorithms (sequential, multi-threaded and distributed) by applying them on a set of well-known parameterized problems.

4.2 Distributed algorithm for constructing the SOG

4.2.1 Aims and Hypothesis

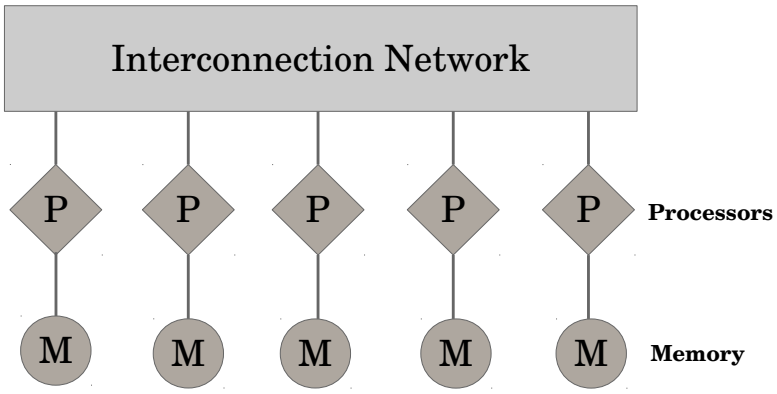


FIGURE 4.1: Distributed memory multi-processor

We consider a distributed memory multi-processor with N processors numbered from 0 to $N-1$ and communicating via message passing interface (MPI) [Wal94; Gro+96].

In a distributed-memory multiprocessor, each memory module is associated with a processor as shown in fig. 4.1. Any processor can directly access its own memory. A message passing (MP) mechanism is used in order to allow a processor to get information from other memory modules associated with other processors. Message passing interface (MPI) is a message-passing library specification, designed to ease the use of message passing by end users, library writers, and tool developers.

The partitioning of the construction of the symbolic observation graph is performed according to aggregates. In contrast, the load balancing is performed statically. Each process executes an instance of the algorithm 2 and constructs a part of the SOG. The distribution and the storing of aggregates by each process are determined using a static partition function $h : S \rightarrow [0, N-1]$ returning the identity of

the process to which an aggregate is assigned, where S is the set of states associates with the aggregate. We associate with each process i a stack $waiting_i$ containing the aggregates to be processed. An advantage of distributed memory is that data locality is fully exploited, since processes can only access local memory.

4.2.2 Description of the algorithm

In our proposed algorithm 2, the major faced problem is the determination of the identity of process that must deal with an aggregate. For this purpose, we use a hash function that associates a unique process identity number to each aggregate. The initial process generates the initial aggregate and computes, using the hash function h, the identity of the process that must store this aggregate. Eventually, the function $SendAggregate()$ is used to send the aggregate to another process. Each process i executes an instance of Algorithm 2. In each iteration, it builds the successors of the popped aggregate by executing observable enabled actions and associates with each aggregate a *key* value. Then, for an aggregate A', if $h(A') = i$, the aggregate A' and the arc linking A to A' are stored locally as well. Otherwise, A' is sent to the process $h(A')$ via calls to SendAggregate(). Thus, the process i stores only the key value associated with A' and the arc linking A to A'. When a worker i receives an aggregate A by calling ReceiveAggregate() it stores it in its local memory, it pushes it into the local stack and pursues the generation of successor aggregates. When process i finishes constructing its local aggregates, it waits for more successors from other processes. When all processes have finished their work the construction has completed.

Since an aggregate A can be generated by several predecessors, it must be ensured that all predecessors send A to the same process. To address this problem, the *key* value of the aggregate A is exploited in the hash function $h(A)$ to specify the destination. The hash function has the form $h(A) = key(A) mod [N]$.

The Partition function that takes a state and returns the identifier of the process to which it belongs must depend exclusively on the aggregate itself. For this reason, the hash function used in the algorithm (line 13) depends on the key value associated with the aggregate to be processed. Indeed, the key value is computed by an internal hash function that is applied to the aggregate (LDD) when it is stored in the hash table.

The correctness of Algorithm 2 follows directly the same arguments as in the case of sequential [KP08a]. Let G be a Symbolic Observation Graph associated with a labeled transition system T and generated by Algorithm 2. G respects definition 11. Further, Algorithm 2 terminates only when the parallel computation is finished and there are no more states to be explored (see Termination Detection 4.2.3). Also, the

hash function is globally known, agreed upon by all processes, and stable over time, otherwise different processes would send the same aggregate to different processes, leading to exploring aggregates more than once.

```
   Data: LTS⟨Γ, Obs ∪ UnObs, →, I⟩
   Result: SOG⟨Γ', Obs, →', I'⟩
1  if h(I) == i then
2      Waiting_i = {I};
3      A_0 = Aggregate(I);
4  else
5      Waiting_i = ∅
6  while DetectTermination==false do
7      while ∃A ∈ Waiting_i do
8          Waiting_i = Waiting_i \ {A}
9          forall a ∈ Obs do
10             if enabled(A, a) then
11                 S' ← succ(A, a);
12                 A'= Aggregate(S');
13                 if h(A') ≠ i then
14                     SendAggregate(A', h(A'));
15                 else
16                     if ∃A'' tq A' == A'' then
17                         arc(A, a, A');
18                     else
19                         Γ = Γ ∪ {A'};
20                         arc(A, a, A');
21                         Waiting_i = Waiting_i ∪ {A'};
22     Waiting_i = Waiting_i ∪ ReceiveAggregate();
```

Algorithm 2: A distributed algorithm for constructing the SOG

4.2.3 Termination Detection

In order to detect the termination of the parallel generation of the SOG, we use a virtual ring-based algorithm inspired by [Mat87]. The global termination is reached when all local computations are finished (stacks of aggregates are empty)

The principle of the termination detection algorithm used is the following. [GMS01] All processes are assumed to be on an unidirectional virtual ring that connects every process i to its successor process $(i+1) mod [N]$. Every time the initiator process finishes its local computations, it checks whether global termination has been reached by generating two successive waves of Receive and Send messages on the virtual ring to collect the number of messages received and sent by all processes, respectively. In

practice, to reduce the number of termination detection messages, each process propagates the current wave only when its local computations are finished. The initiator process checks whether the total number of messages sent is equal to the total number of messages received. If this is the case, it will inform the other processes that termination has been reached by sending a termination message on the ring. Otherwise, the initiator concludes that termination has not been reached yet and will generate a new termination detection waves later. According to [GMS01], this distributed termination detection scheme seems to use less messages than the centralized termination detection schemes used in the parallel versions of Spin [LS99] and Murφ [SD97].

4.3 Technical aspects and implementation

We have implemented the algorithm using C++. As communication platform for the implementation of algorithm 2, we used the MPI [Wal94; Gro+96]. Communications can create a large overhead and the code granularity often has to be large to minimize the latency. In this work, the granularity of the parallelization is an aggregate (set of states). We encode the marking that allow to rebuild the aggregate. We send the message that contains this encoding (of type String) to the 'owner' process of this aggregate. The owner must receive and decode the message to rebuild the aggregate. Since each aggregate is identified by a key value, the sender keeps this key. This can help us easily find the recipient and to avoid the re-send, in case of the multiple construction of the aggregate.

4.4 Experiments

In the current section, we show an empirical evaluation of the distributed algorithm (algorithm 2) by comparing its absolute performance and scalability with that of the multi-threaded one. The presented benchmark setup is the same used in the previous chapter.

All the results are taken for executions of the Algorithm 2. For each net, we have measured the time in seconds consumed by the construction of SOG in a sequential way. Then, we measured the runtime of our Distributed algorithm by progressively increasing the number of process. Figure 4.2 shows the run times of only three models (ring5, ring6, philo10) using respectively Algorithm 2. The speedups are displayed in figure 4.3. The speedup is a measure for the performance gain of parallelizing an algorithm and it is done by normalizing performance gain with regard to the sequential run. We found in Figure 4.3) the speedups achieved by our examples.

Model	Seq	8	16	24	32	40	SpMax
ring4	2.15	0.92	0.84	0.87	0.65	0.95	3.3
ring5	43.46	15.14	13.60	14.57	11.39	16.29	3.8
ring6	870.87	490.64	323.76	452.14	347.36	463.04	3.6
philo6	0.47	0.29	0.34	0.29	0.30	0.25	2
philo8	11.94	8.00	5.63	7.12	6.69	9.23	2.1
philo10	311.10	219.53	209.75	149.07	176.83	238.44	1.85
fms4	208.11	181.01	164.13	171.55	172.36	179.85	1.3
fms5	2088.75	1699.14	1733.08	1633.87	1614.88	1779.79	1.3
robot4	7.10	4.26	3.70	4.60	3.93	4.91	2.3
robot5	36.16	22.67	18.26	19.64	16.94	24.75	2.1
robot6	135.09	105.8	88.41	90.05	79.53	99.85	1.9
train2-4	42.49	30.99	31.30	36.34	39.09	40.04	1.4
train2-6	29.64	18.89	18.11	19.21	18.69	20.26	1.8
train2-8	26.73	11.16	10.73	11.63	10.74	11.16	2.5
erk10	46.63	28.98	19.87	19.70	23.69	28.97	2.4
erk20	957.68	587.35	468.12	397.23	511.26	589.38	2.4

TABLE 4.1: Experimental results of the distributed-memory algorithm

It can be seen that for all examples, the measured speedup is not so good, 40 processor runs can achieve speedups of at best 4.5. We conclude from our experiments that the shared memory is more efficient than the distributed memory architectures, probably because the cost of the processes communication.

4.5 Conclusion

In this chapter, we have presented an approach for the parallel construction of the SOG proposed for distributed memory architectures. It uses a static load balancing scheme, since it allows to avoid supplementary communication between processes. We have implemented this algorithm and studied their performances. We have compared the results obtained by this approach against the multi-threaded approach and a non parallel construction of the SOG. Experiments show that shared-memory algorithm is more efficient than distributed-memory algorithm. This is can be explained by the communication overhead between processes. Since, communication overhead plays an important role in scalability of distributed-memory algorithms.

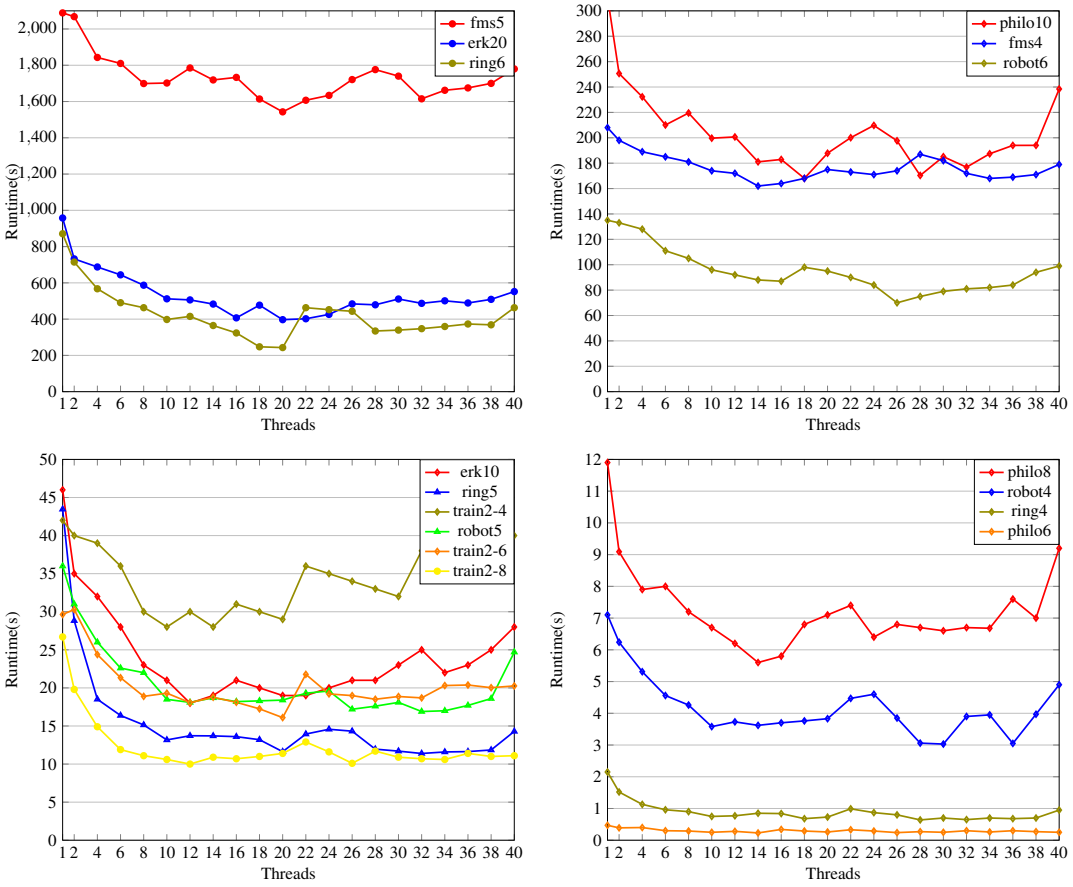


FIGURE 4.2: Runtime of distributed-memory algorithm

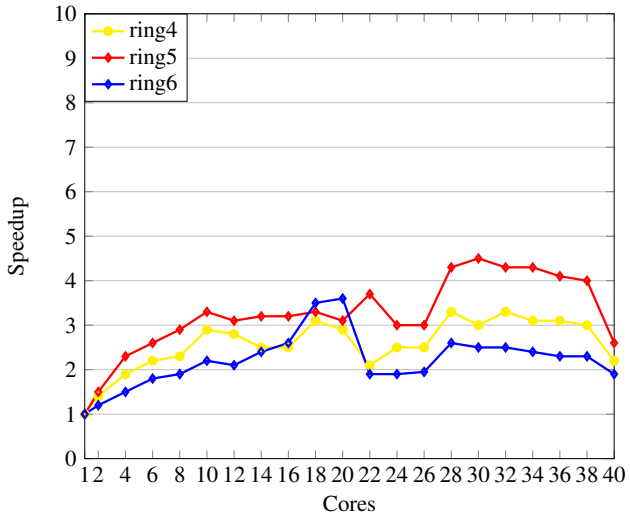


FIGURE 4.3: Speedup of the distributed-memory algorithm

Chapter 5

Hybrid approach for generating the SOG

Contents

3.1 Introduction 30

3.2 Multi-threaded algorithm for constructing the SOG 31

3.2.1 Aims and Hypothesis 31

3.2.2 Description of the algorithm 32

3.3 Technical aspects and Implementation 34

3.3.1 Multi-core decision diagram packages 34

3.3.2 Symbolic encoding of the SOG 35

3.3.3 Adaptation of Sylvan to the parallel implementation of the SOG 35

3.4 Experimental results 37

3.4.1 Comparison BuDDy - Sylvan 37

3.4.2 Results of the multi-threaded algorithm 37

3.5 Conclusion 39

5.1 Introduction

In order to reduce the building time of a SOG, we have proposed a parallel (shared memory) and distributed (message passing) construction algorithms in [Oun+17a; Oun+17b]. The first one is a multi-threaded algorithm dedicated to shared memory architectures (presented in chapter 3). Comparing to sequential algorithm, results show interesting performance improvement in execution time. The second algorithm is based on a distributed memory architecture (see chapter 4). Thus, each process has its own local memory, and communication between processes is performed by

using the Message Passing Interface MPI [Wal94]. Experiments show that, in general, shared-memory algorithm outperforms the distributed-memory algorithm (this is mainly due to communication overhead).

In line with our previous work, and aiming to find the best way to parallelize the SOG construction, we present in this chapter a hybrid technique which combines the two previous approaches [Oun+19]. The SOG construction is then shared among a number of processes, each creates a number of threads that run on (typically) the same number of CPUs. Among the threads of each process, one (the coordinator) is responsible of the communication (including performing overlapped asynchronous message passing), while the others (workers) are responsible for the construction of the SOG nodes. The hybrid Shared and Distributed architectures provides significant performance benefits for the enumerative model checking performed on clusters (currently the most common high-performance computers).

5.2 A hybrid approach for constructing a SOG

5.2.1 Aims and hypothesis

Our proposed approach considers an LTS where its observable and unobservable actions are specified, in order to build its SOG in a parallel and distributed setting. This is performed by running several processes where each process consists of several threads. Therefore, the partitioning of the building of a SOG is performed according two levels. At the first level, the partitioning of the building of the SOG is performed at the process level. The load balancing between processes is performed statically through the use of a hash function. Note that this function should have a homogeneous distribution to ensure load balancing, therefore, we have chosen MD5 [Dob96] as it is known to provide a good distribution for any kind of input. At the second level the partitioning of each part of the graph associated with a process is performed at the threads level. In order to increase the number of treatments to be executed simultaneously by different threads, and consequently, to obtain a better speed up, we use a dynamic load balancing approach. The decision to allocate the construction of an aggregate to one thread is made after comparing current loads. Threads are executed on the same machine and can share the same space memory that can lead to unexpected behavior. To avoid concurrent writes on sensitive data, mutual exclusions (mutexes) around critical sections are used.

For this purpose, we propose to create communicating processes via Message Passing Interface (MPI). As it is illustrated in Fig. 5.1, each process creates two types of threads:

- The first type consists of one thread, called coordinator thread. The main purpose of this thread is to manage communication between processes, and to detect the termination of the construction.
- The second type of threads, called worker threads, allows to build jointly a part of the SOG.

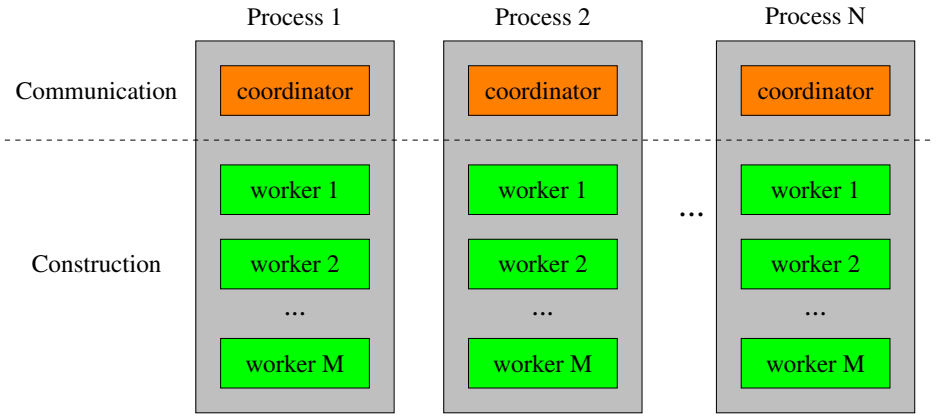


FIGURE 5.1: Architecture of the hybrid approach

5.2.2 Description of the algorithms

We consider a distributed memory with N processes numbered from 0 to $N-1$ and communicating via message passing interface (MPI). Process number 0 is called a master process. It launches the construction of the SOG from the initial state of the considered LTS, and it will launch the detection of the termination. Each process constructs a part of the SOG using several threads. Each thread is identified by its identifier stored in its local variable *idthread*. Threads having 0 as identifier in a given process are the coordinator threads.

A process i can send a message m to process j by invoking function $Send(j,m)$, and can receive a message through the call of function $Receive(m)$. The latter function returns a Boolean value indicating whether a message has been received or not. Each process creates one coordinator thread and several *worker* threads. We associate with the coordinator thread a stack to store the messages to be sent to other processes. Further, we associate with each worker thread two stacks. The first one contains messages received by the coordinator from other processes through static load balancing and followed by a dynamic load balancing performed by the coordinator thread. Indeed, these messages correspond to aggregates that must be constructed and stored by the thread. The second contains the aggregates to be built by the worker and they are assigned through a dynamic load balancing within the parent process.

The array *Load* indexed by *idthread* allows to store the current loads of the worker threads. The load of a thread is defined by the number of aggregates to be processed by the thread.

Data: $LTS\langle\Gamma, Obs \cup UnObs, \rightarrow, I\rangle$
Result: $SOG\langle\Gamma', Obs, \rightarrow', I'\rangle$
Load: Array[1,..,M] integer;
$Waitmsg_i$: a stack to store the messages;
if $id_{th} == 0$ **then**
    $A_0 = Aggregate(I)$;
    if $(h(A_0) == 0)$ **then**
        $Waitmsg_1 = \{A_0\}$;
    else
        SendAggregate$(A_0, h(A_0))$;
while $DetectTermination == false$ **do**
    if $(id_{th} == 0)$ **then**
        while *TAG*==*Aggregate* **do**
            /* The message is tagged with the tag value Aggregate */
        A=ReceiveAggregate();
        j=minCharge();
        $Waitmsg_j = Waitmsg_j \cup A$;
        mutex lock$(Load_j)$;
        $Load_j \longleftarrow Load_j + 1$;
        mutex unlock$(Load_j)$;
        while $\exists msg \in Waitmsg_0$ **do**
            $Waitmsg_0 = Waitmsg_0 \setminus msg$;
            SendMsg();

Algorithm 3: A hybrid algorithm for constructing the SOG (coordinator thread)

Each process executes an instance of the algorithms 1 and 2 in order to construct a part of the SOG. Algorithm 3 is executed by the coordinator thread during the distributed construction of the SOG. We distinguish the coordinator thread belonging to the master process. This thread initiates the building of the SOG by generating the initial aggregate. Then, it determines if the current process is responsible (owner) of this aggregate. If not, it sends it to the process determined by the hash function $h : A \rightarrow \{0, 1, \cdots, N-1\}$ used to distribute aggregates between processes. Then, it operates a loop, as any other worker thread, for receiving and sending messages. If the received message is an aggregate (the message is tagged with the TAG value Aggregate i.e. it is not a message for termination detection), the coordinator pushes this message into the stack of messages $Waitmsg_i$ of the worker thread having minimum load. While the stack of messages $Waitmsg_0$ of the coordinator is not empty, in each

iteration, it pops a message and sends it to the appropriate process by calling the SendAggregate() function.

```
   Data: LTS⟨Γ, Obs∪UnObs, →, I⟩
   Result: SOG⟨Γ', Obs, →', I'⟩
1  Load: Array[1,..,M] integer;
2  Waitmsg_i: a stack to store the aggregates;
3  Waiting_i: a stack to store the messages;
4  while DetectTermination == false do
5      while ∃A ∈ Waiting_i do
6          Waiting_i = Waiting_i \ {A}
7          forall a ∈ Obs do
8              if enabled(A,a) then
9                  S' ← succ(A,a);
10                 A'= Aggregate(S');
11                 if h(A') ≠ i then
12                     Waitmsg_0 = Waitmsg_0 ∪ msg_to_send;
13                 else
14                     if ∃A'' tq A' == A'' then
15                         arc(A,a,A');
16                     else
17                         Γ = Γ ∪ {A'};
18                         arc(A,a,A');
19                         j=minCharge();
20                         mutex lock(Waiting_j);
21                         Waiting_j = Waiting_j ∪ {A'};
22                         mutex unlock(Waiting_j);
23                         mutex lock(Load_j);
24                         Load_j = Load_j + 1;
25                         mutex unlock(Load_j);
26     while ∃msg ∈ Waitmsg_i do
27         mutex lock(Waitmsg_i);
28         Waitmsg_i = Waitmsg_i \ msg;
29         mutex unlock(Waitmsg_i);
30         S'=DecodingMsg(msg);
31         A'= Aggregate(S');
```

Algorithm 4: A hybrid algorithm for constructing the SOG (worker i)

Algorithm 4 is executed by each of the workers. At each iteration, a worker thread i pops an aggregate from its stack of aggregates $Waiting_i$ and decrements its loading. It builds the successors of the popped aggregate by executing observable enabled actions. It computes, using the hash function h, the identity of the processes that must store the successors (aggregate). If a newly built successor must be stored

locally and does not exist in the SOG, it is inserted in the SOG and pushed into the stack associated with the thread having the minimum load. Otherwise, it pushes this aggregate into the stack of the coordinator thread that is responsible of sending an aggregate to the "owner" process. Next, the workers runs in a loop exploring aggregates received by the coordinator. In each iteration, it pops a message from its stack of messages $Waitmsg_i$ and decrements its loading. It stores the received aggregate in its local memory, pushes it into the local stack and pursues the generation of successor aggregates.

5.2.3 Termination detection

The termination detection for the hybrid approach is done into two phases: a local termination detection and a global termination detection.

Local termination detection: It is the responsibility of the coordinator thread (within each process). At each process, the local termination occurs when all worker threads have finished the construction of their local aggregates and when no load balancing operations are in progress. In this case, the process moves towards a state in which it expects global termination detection or more aggregates to process.

Global termination detection: It is detected through the use of a virtual ring-based algorithm inspired by [Mat87] on which the termination probing message is exchanged only between neighbors. The token message chain can be started only by the master process when it finishes its local computation (when it detects local termination). Since every process updates the global sent and received message counting on the token before forwarding it, if the master process finds the two counters to match then the parallel computation is over. In fact, this implies that all the threads are inactive and all messages that have been sent have also been received. It is worth noting that the global detection is canceled if there is a process that does not detect its local termination. In a such case, the master process has to re-initiate the global termination when its local termination detection is still valid.

5.2.4 Correctness proof

In General, to prove the correctness of a parallel program, it is necessary in the first place, to demonstrate the correctness of its components (tasks) that are sequential programs. Further, additional properties related to safety and liveness must be proved [Lam77]. Safety properties stated by sufficient conditions on the assertions to guarantee the coherence of the subroutine (mutual exclusion, deadlock-free), and liveness properties stating conditions which guarantee termination. In addition, from graph construction point of view, we prove that any maximal path in the SOG built

sequentially is a maximum path in the SOG built by our Algorithm (see the following proofs).

In the following, we prove that each maximal path in the SOG built sequentially is built by our hybrid parallel algorithm and vice versa.

Theorem 25 *Let $\mathcal{T}$ be an LTS and $\mathcal{G}_s$ the corresponding sequential SOG [HIK04]. Let $\mathcal{G}_p$ the SOG Obtained by our Hybrid algorithm. Let $\pi = a_0 \xrightarrow{o_1} a_1 \xrightarrow{o_1} \cdots \xrightarrow{o_n} a_n$ be a path linking aggregates $a_0 \ldots a_n$ then the following holds:*

π is a maximal path in $\mathcal{G}_s \Leftrightarrow \pi$ is a maximal path in $\mathcal{G}_p$

Proof.

We proceed by induction on the length of π.

- If $\pi = a_0$, then a_0 contains either a dead state or a cycle. Note that the initial aggregate is the same in $\mathcal{G}_a$ and $\mathcal{G}_p$. In fact, for the parallel construction, a given thread (belonging to a given process) is responsable of the construction of this aggregate using a sequential construction and the same saturation algorithm as given by the definition of an aggregate. The existence of dead states and/or cycles within this aggregate is then determined similarly in sequential and parallel construction algorithms.

- Assume that the theorem is correct for any path of length n, and let $\pi = a_0 \xrightarrow{o_1} a_1 \xrightarrow{o_1} \cdots \xrightarrow{o_n} a_n$ be a path of length $n+1$. Let t be the identifier of the thread that is responsible of the building of a_{n-1}. Since it is the same aggregate as in the sequential version, thread t will compute its successor by action o_n, before sending this successor aggregate to the owner process (using the hash function). Lets consider the two following cases:

  - a_n contains a deadlock (resp. a cycle): again, such a detection will be done in both cases (sequential and parallel) in the same way, and the maximum path belongs to both SOGs versions.

  - a_n is identified an aggregate belonging to the current path (i.e. we are in the case of a maximal path with an infinite number of observable actions' occurrences). Let $l \leq n$ be the index of the aggregate a_l s.t., $a_l = a_n$. Once the thread t has computed a_n, the owner process of a_n will be computed by the hash function (proved to give unique value for identical aggregates). Thus, the owner will be the same process that already computed a_l which is saved in the local memory of that process. The current path is then detected as a maximal one in $\mathcal{G}_p$.

The proof of the other direction in the two previous cases is trivial.

In the following, we discuss correctness issues of our algorithm regarding the satisfaction of safety and liveness properties (mutual exclusion, deadlock-freeness and termination):

- There are no critical runs when the threads access the shared variables. We avoid critical runs by using proper and correct mutual exclusion mechanisms provided by the PThread library.

  At the level of processes, the hash function is globally known, agreed upon by all workers, and stable over time (otherwise different processes would add the same state to different owners). We used the hash function MD5 (Message Digest) [Dob96] designed by Ron Rivest as a strengthened version of MD4. The surety of this function was widely studied by several authors [Dob96].

- The correctness about this issue comes directly from the fact that the threads running in parallel have to synchronize only when there are no aggregates in the stack, thus the worker thread has to wait for the communication thread to receive some aggregates and push them in its associated stack. However, the communication thread never waits for the worker thread as its main purpose is to send some aggregates to processes or to receive aggregates from processes that popped (resp. pushed) from (resp. into) dedicated stacks. When no aggregates are received and there are no aggregates to send, there exist two cases. If the communication thread belongs to a master process, it will initiate termination. Else, surely, it will receive a termination message. For the two cases, termination has occur as aforementioned in the termination detection proof.

- We used the termination algorithm in [Mat87], which is known to be correct. Termination is reached when all local computations are finished (i.e., each process i has no remaining states to explore and all sent states have been received).

5.3 Experiments

5.3.1 Results of the hybrid approach

Extensive experiments with a variety of different models were performed to measure the performance of the proposed hybrid algorithm.

Net	Obs	states	Agg	Arcs	Seq	2	4	6	8	10	12	SpMax
ring4	8	5136	304	1280	2.15	0.57	0.68	0.70	0.70	0.77	0.71	3.7
ring5	10	53856	1632	8320	43.46	4.39	4.13	4.28	4.25	3.84	3.98	11.3
ring6	12	575296	3805	20698	870.87	71.34	37.16	28.51	23.44	21.05	18.66	46.6
philo6	12	5778	64	384	0.47	0.17	0.23	0.26	0.24	0.26	0.37	2.7
philo8	16	103682	256	2048	11.94	1.07	0.94	1.11	1.14	1.21	1.29	12.7
philo10	20	1.86×10^6	1024	10240	311.10	22.62	19.23	17.18	14.42	13.91	14.58	22.3
fms4	4	438600	266	830	208.11	22.84	23.15	23.88	26.89	26.14	31.34	9.1
fms5	4	2.89×10^6	93280	519972	2088.75	502.95	473.68	406.51	343.22	481.73	524.64	6.0
robot4	6	48620	2574	11649	7.10	1.14	1.19	2.31	3.08	3.42	3.37	6.2
robot5	6	184756	6006	28600	36.16	4.06	3.28	4.63	4.77	5.98	5.73	11.0
robot6	6	587860	12376	61061	135.09	36.47	34.72	23.66	21.12	17.52	20.45	7.7
robot7	6	1.63×10^6	23256	117776	505.90	48.87	32.89	37.52	43.27	49.66	53.15	15.4
train2	4	86515	81	188	42.49	8.45	7.99	8.39	9.41	9.97	10.86	5.3
train2	6	86515	178	448	29.64	4.12	4.23	4.51	4.26	5.17	4.55	7.2
train2	8	86515	1660	6696	26.73	3.23	3.51	3.50	3.45	3.35	3.47	8.2
erk10	4	47047	505	1803	46.63	3.95	3.48	3.16	3.08	3.64	4.00	15.1
erk20	8	1.69×10^6	21230	15297	957.68	62.15	58.33	51.71	53.67	57.32	60.84	18.5

TABLE 5.1: Scalability of the Hybrid approach of the construction of the SOG

We made scaling experiments on the Magi cluster[1] of Paris 13 university. This cluster has 12 processors each with 12 cores (two Xeon X5670 at 2.93GHz), 24GB of RAM and they are connected by an InfiniBand network.

In table 5.1, we provide some results of the Hybrid approach. For each model, we have measured the time in seconds consumed by the building of the SOG in a sequential way. Then,we set the number of threads for each process at 12 and we increased the number of processes from 2 to 12. We analyzed the runtime efficiency and we calculated the maximum obtained speedup. We can notice that the hybrid approach scales well, but we can see that the increase in the number of processes does not necessarily imply an increase in the efficiency of the algorithm.

In order to improve the performance of the parallel generation of the SOG, it is essential to achieve a good load balancing between the processes. Meaning that all processes should have an equal number of aggregates during the generation of the SOG. As indicated in Section 4, for the hybrid approach, we adopted a static partition scheme (we have chosen MD5) to ensure, at the first level, load balancing among the involved processes. Fig. 5.2 shows the distribution of the aggregates on 12 processes for different examples. Overall, we interpret a good performance of the partition function in the distribution of workload between processes. We can also mention that the load balancing is well performed for models having more size and

[1] http ://www.univ-paris13.fr/calcul/wiki/

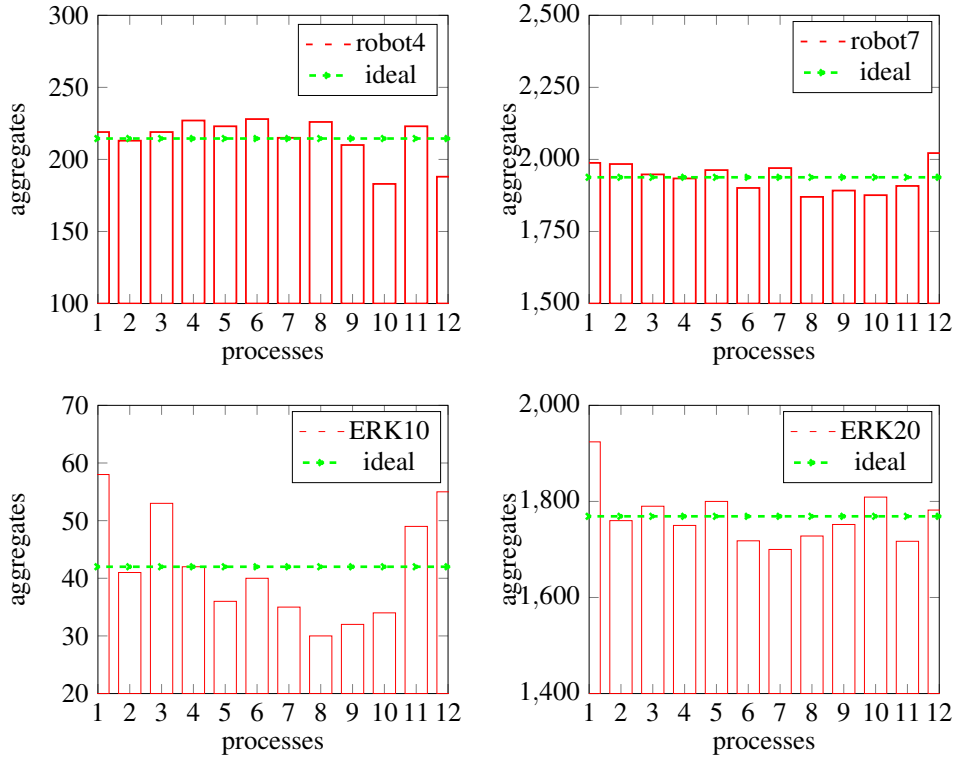


FIGURE 5.2: Distribution of the aggregates on 12 processes

more number of aggregates.

At the second level, for each process, we assigned the workload to the worker threads using a dynamic load balancing scheme. The allocation of the construction of an aggregate to one worker thread is made after comparing current loads (the number of aggregates to be processed) of individual threads. Even this method shows that the load is balanced evenly between threads (the number of aggregates is nearly equal for the worker threads). We aim to make various optimizations like to make the comparison between the current loads of the threads using the number of the LDD nodes instead of the number of aggregates also the sum of message sizes to process instead of the number of messages. Indeed, there is a difference in size from one aggregate to another. Therefore, if we have the same number of aggregates, it does not mean that we have the same load for each thread.

5.3.2 Comparative analysis

In the Table 5.2, we compare the proposed approach performances with those presented in chapters 3 and 4 (shared and distributed-memory algorithms to construct the SOG). Table 5.2 illustrates the differences in terms of minimal runtime and maximal speedup. We experimented the distributed approach with 40 processes. The maximum speedups for this approach are detected for at the most 30 processes. Then, we

Model						Distributed		Hybrid		Multi-thread	
Model	Obs	states	Agg	Arcs	Seq	runtime (s)	SpMax	runtime (s)	SpMax	runtime(s)	SpMax
ring4	8	5136	304	1280	2.15	0.64	3.3	0.57	3.7	0.42	5.1
ring5	10	53856	1632	8320	43.46	11.39	3.8	3.98	11.3	5.71	7.6
ring6	12	575296	3805	20698	870.87	243.13	3.6	18.66	46.6	87.71	9.9
philo6	12	5778	64	384	0.47	0.28	1.6	0.17	2.7	0.12	3.9
philo8	16	103682	256	2048	11.94	5.63	2.1	0.94	12.7	1.89	6.3
philo10	20	1.86×10^6	1024	10240	311.10	168.03	1.8	13.91	22.3	36.67	8.5
fms4	4	438600	266	830	208.11	162.74	1.3	22.84	9.1	21.85	9.5
fms5	4	2.89×10^6	93280	519972	2088.75	1543.52	1.3	406.50	5.1	222.82	9.3
robot4	6	48620	2574	11649	7.10	3.03	2.3	1.14	6.2	0.92	7.7
robot5	6	184756	6006	28600	36.16	16.94	2.1	3.28	11.0	4.86	7.4
robot6	6	587860	12376	61061	135.09	70.67	1.9	17.52	7.7	17.75	7.6
robot7	6	1.63×10^6	23256	117776	505.90	311.78	1.6	32.89	15.4	61.97	8.1
train2	4	86515	81	188	42.49	28.88	1.4	7.99	5.3	12.58	3.3
train2	6	86515	178	448	29.64	16.10	1.8	4.12	7.2	5.83	5.0
train2	8	86515	1660	6696	26.73	10.73	2.5	3.23	8.2	2.86	9.0
erk10	4	47047	505	1803	46.63	18.84	2.4	3.08	15.1	5.11	9.1
erk20	8	1.69×10^6	21230	15297	957.68	397.23	2.4	51.71	18.5	86.44	11.0

TABLE 5.2: The best execution times using multi-threaded, distributed and hybrid approaches

stopped the evaluation at 40 processes because the speedups have been decreased for all the tested models by increasing the number of processes more than 30 (mainly due to the communication overhead). Moreover, because of material limitations (a cluster containing 12 nodes with 12 cores for each node), the multi-threaded approach is limited to 12 cores. Therefore, the hybrid approach will essentially allow to exploit such architecture of clusters, leading for most examples (as shown in the table) to better speedup.

We analyze the execution times in Figure 5.3 of our algorithms measured on different architectures. It can be seen that the results of the hybrid and multi-threaded approach are competitive, whereas the distributed approach is less efficient.

We illustrate the gain of our approaches in terms of speedup in Figure. 5.4. Comparing sequential runs against the hybrid approach, the speedup curves are moving up as the number of cores increases indicating that this approach scales to some degree. It achieves a maximum speedup of 46 when 144 cores CPU are used (for the ring6 example). Up to 12 cores, we note that the multi-threaded approach scales better than the hybrid and the distributed approach.

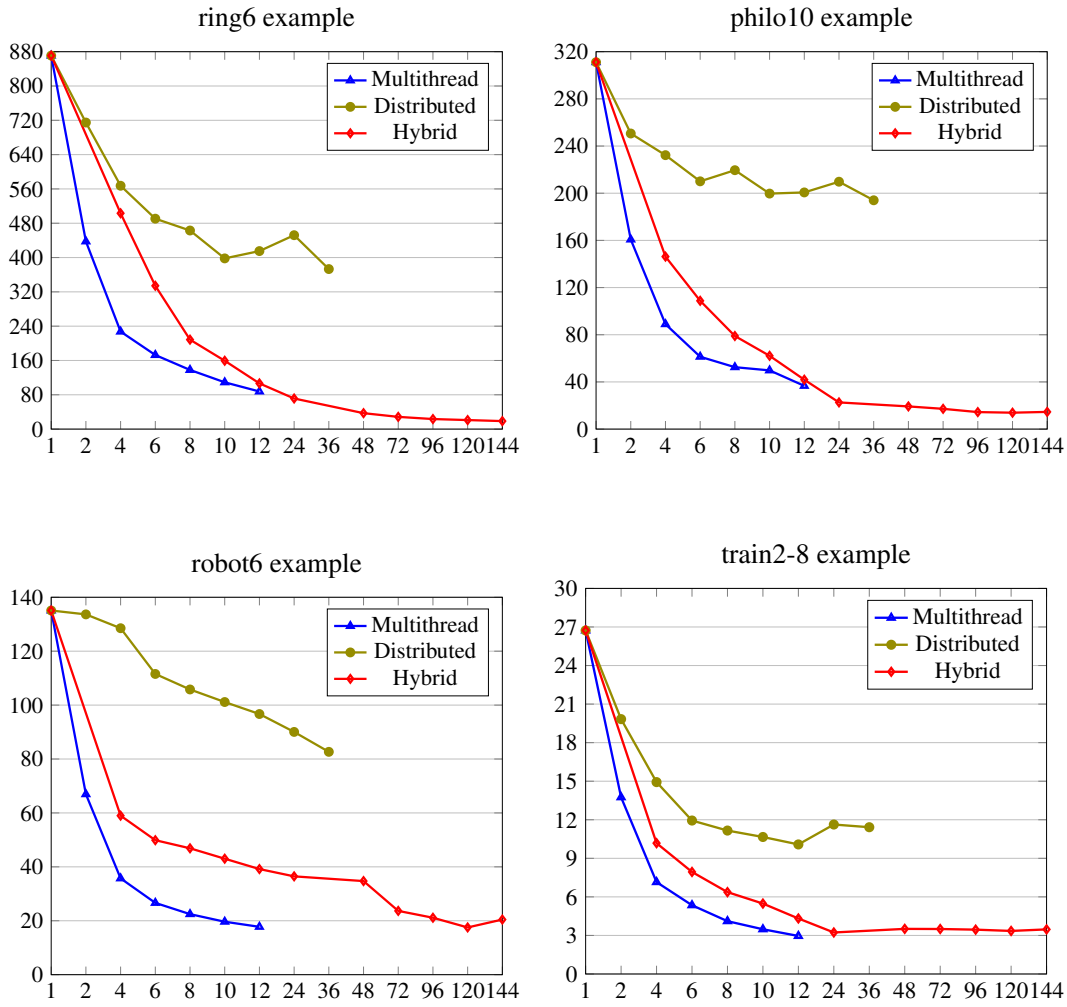


FIGURE 5.3: Comparison between the multi-threaded, distributed and hybrid approaches in term of the obtained runtime. On the X axis are the numbers of cores and on the Y axis are execution times in seconds.

5.4 Conclusion

We have proposed in this chapter a hybrid MPI-thread approach for the construction of the Symbolic Observation Graph. Then, we have analyzed the benefits of this approach in terms of memory and runtime performances. To validate this approach, we have performed experiments and we have evaluated the presented approach on a benchmark of well-known parameterized problems. The experimental results are encouraging, and confirm that the hybrid algorithm scales well while obtaining high speedups and utilizing the available computational power to its full extent. We have also compare the hybrid approach with the two previous approaches that we have proposed in the previous chapters. In all test cases, we observe a better global scalability of the parallel construction, although the maximization of the number of cores

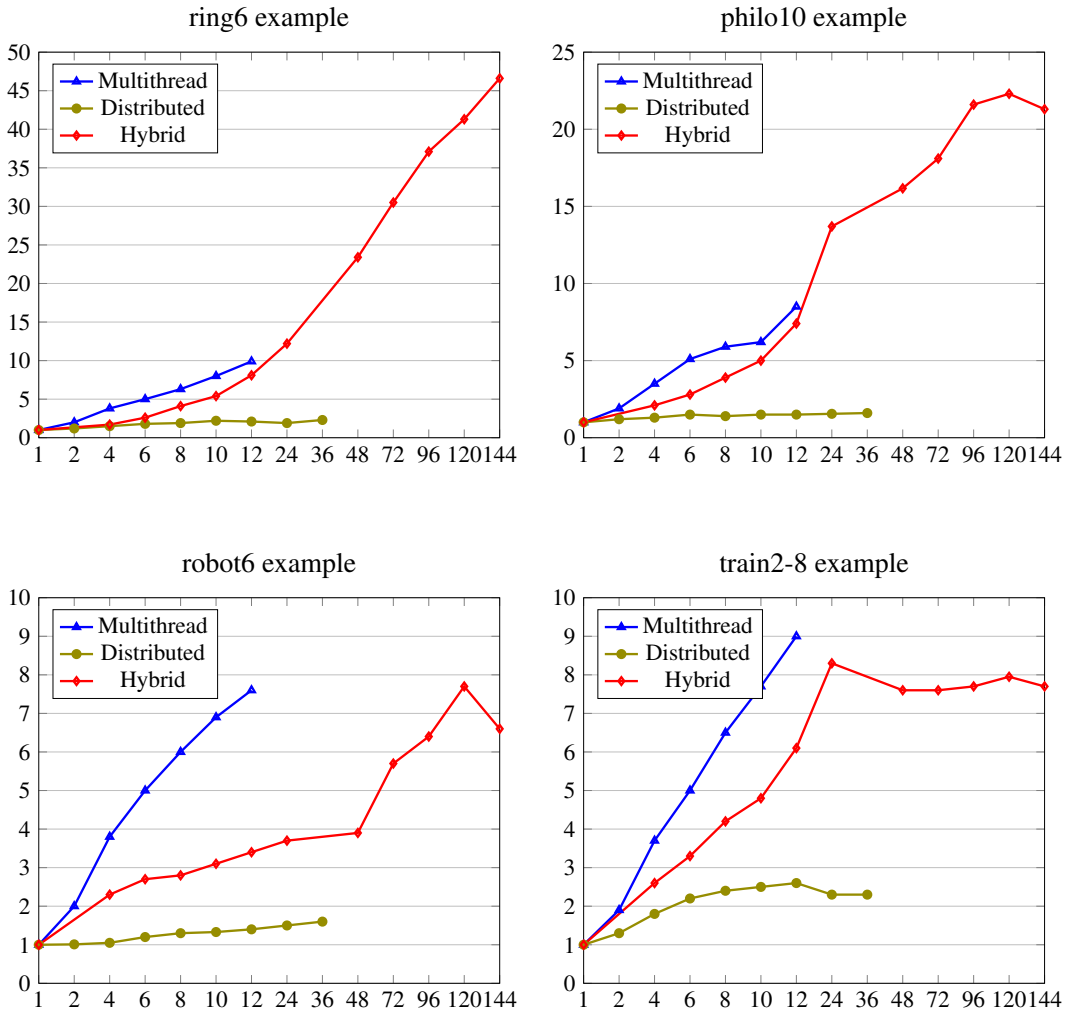


FIGURE 5.4: Comparison between the multi-threaded, distributed and hybrid approaches in term of the achieved speedups. On the X axis are the numbers of CPU cores and on the Y axis are the speedups

do not necessarily improve the scalability of the hybrid algorithm.

Chapter 6

Reducing time and/or memory consumption of the SOG construction

Contents

4.1 Introduction 41

4.2 Distributed algorithm for constructing the SOG 42

4.2.1 Aims and Hypothesis 42

4.2.2 Description of the algorithm 43

4.2.3 Termination Detection 44

4.3 Technical aspects and implementation 45

4.4 Experiments 45

4.5 Conclusion 46

6.1 Introduction

In the previous chapters, we have proposed new algorithms to parallelize the construction of the symbolic observation graph (SOG). Each node of a SOG, called an aggregate, represents a set of explicit states that is encoded symbolically by a BDD, or by any extension such as an MDD. We have proposed three different approaches, in order to benefit from additional speedups and performance improvements in execution time. A multi-threaded approach based on a dynamic load balancing and a shared memory architecture , a distributed approach based on a distributed memory architecture and a hybrid (shared-distributed memory) approach that combines the two previous approaches. Experiments in chapter 5 showed that the distributed-memory approach is less efficient than the multi-threaded and the hybrid approaches. This can be explained by the communication overhead between processes. In this Chapter, we aim to improve the distributed algorithm proposed in chapter 4 by reducing the size of the communicated aggregates (sets of states) between the processes

involved in the SOG construction. Indeed, instead of sending the set of all states that forms an aggregate, we propose to use a reduced canonical representation that is sufficient to rebuild it. Further, such a representation has to be unique in order to identify the represented aggregate, and also to retrieve the process responsible for its construction and storing. In order to obtain such a representation, we replace the corresponding set of states by the subset consisting of one representative per strongly connected component (SCC) of the subgraph spanned by this set. Since, the computation of initial SCCs is a critical step with respect to the time complexity [XB00; ZC10], we propose a specific and symbolic algorithm that allows to reduce the size of an aggregate. Then, we apply such a reduction within the three previous parallel approaches and we measure the impact of the use of the canonicalization on SOG construction.

6.2 Canonicalization algorithm

Our aim is to reduce the size of the set of states associated with each aggregate. In addition to reducing the size of aggregates that can lead to important memory savings, the canonicalization allows us to develop a more efficient distributed algorithm by reducing the communication cost. This reduction is performed by determining a canonic representative subset of states for each aggregate. Indeed, taking the unobserved events as edges, an aggregate may be viewed as a graph and it is sufficient to extract one representative per each initial strongly connected component (SCC) of this graph in order to preserve the observed behavior starting from the aggregate. The computation of SCCs is known to be a critical task with respect to the time complexity. Indeed, a standard symbolic search algorithm of initial SCCs of a graph may have a bad time complexity.

In [HIK04], a BDD-based approach has been developed which takes advantage of the parallelism of the system under observation. For the parallel systems, this approach outperforms the standard approaches [BGS06; XB00] and its efficiency has been shown by standard examples. For this purpose, in this chapter, we propose an LDD-based approach inspired by the BDD-based approach proposed in [HIK04].

Algorithm 5 determines a representative LDD (subset of states) for a specified aggregate. It is a recursive function that considers the LDD representing all states of the given aggregate, the set of the unobserved events and the level (depth) i in the specified LDD, in order to compute the canonic representative of the given aggregate (LDD).

In the first call, i is initialized to 0, in order to start the computation from the root node of the LDD. This function divides the considered LDD S into two subsets

```
1  CANONIZER(LDD S, Events UnObs, int i)
2  LDD S1, S2, Front, Reach, Repr;
3  S1 = R_i ∪ Down(R_i);
4  /* R_i is the root node of the LDD S */
5  /* Down(R_i) returns the down edges R_i */
6  S2 = S \ S1;
7  if S1 ≠ ∅ and S2 ≠ ∅ then
8      Front = S2;
9      Reach = S2;
10     repeat
11         Front = Img(Front, UnObs) \ Reach;
12         /*Img returns the set of immediate successors of the states of
             Front by the occurrence of UnObs.*/
13         Reach = Reach ∪ Front
14         S1 = S1 \ Front
15     until Front = ∅ or S1 = ∅;
16 if S1 ≠ ∅ and S2 ≠ ∅ then
17     Front = S1;
18     Reach = S1;
19     repeat
20         Front = Img(Front, UnObs) \ Reach;
21         Reach = Reach ∪ Front;
22         S2 = S2 \ Front;
23     until Front = ∅ or S2 = ∅;
24 Repr = ∅;
25 if size(S2) ≤ 1 then
26     Repr = Repr ∪ S2;
27 else
28     Repr = Repr ∪ CANONIZER(S2, UnObs, i);
29 if size(S1) ≤ 1 then
30     Repr = Repr ∪ S1;
31 else
32     Repr = Repr ∪ CANONIZER(S1, UnObs, i + 1);
33 return Repr;
```

Algorithm 5: Canonizer algorithm

$S1$ and $S2$. The first one contains the root node of the LDD and the 'down' edges of this node (line 3), while the second is the rest of the input LDD (line 6). If it is not possible to partition S we pass to the next level. We remove from the first subset $S1$ all the states which are in the forward closure of the second subset $S2$, i.e. the elements of $S1$ that are reachable from some elements in S2 (lines 7-15). Such deleted states either do not belong to an initial SCC or their representatives of their SCCs are already present in the second subset ($S2$). Now, $S1$ contains states that are

not reachable from $S2$. We now eliminate the states of $S2$ that are reachable from $S1$, since they do not belong to an initial SCC (lines 16-23). After this double reduction, if $S1$ (resp. $S2$) is a singleton (one state), then $S1$ (resp. $S2$) contains a representative state that must be added to set 'Repr', which is the eventual result. Otherwise, we execute recursively the function Canonizer on $S1$ (resp. $S2$). Both subsets may be independently analyzed in order to find the representatives of the initial SCCs. When states can no longer be reduced by the Canonizer algorithm, the set of states 'Repr' is the canonic representation of the input set of states (line 33).

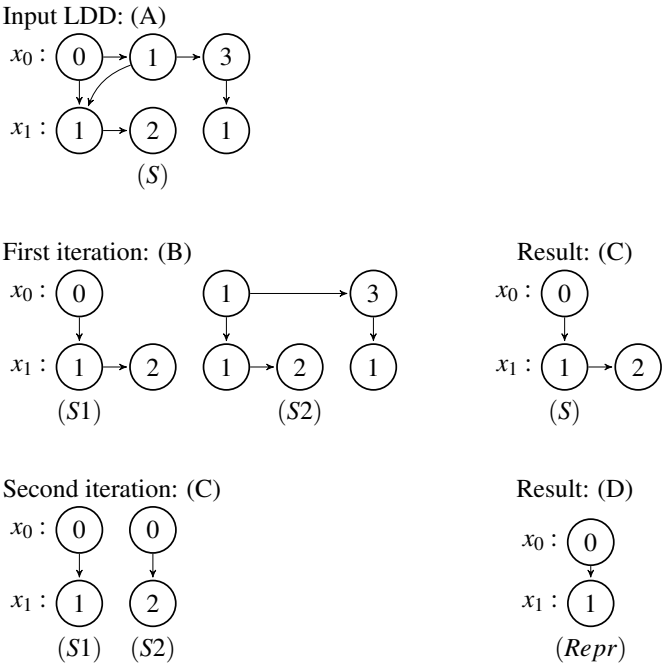


FIGURE 6.1: Example of canonicalization

Let us detail the run of the Canonizer algorithm on an illustrative example. Figure 6.1 shows the LDD that represent the set of states S. We suppose:
$\langle 0,1\rangle \longrightarrow \langle 0,2\rangle$ (this means that $\langle 0,2\rangle$ is reachable from $\langle 0,1\rangle$)
$\langle 0,1\rangle \longrightarrow \langle 1,1\rangle$
$\langle 0,2\rangle \longrightarrow \langle 1,2\rangle$
$\langle 0,1\rangle \longrightarrow \langle 3,1\rangle$

Figure 6.1 (B), gives the obtained two subsets S1 and S2 by partitioning S according to the level 0 (root level). By applying the reductions on $S1$ and $S2$, we obtain the set $S1$ illustrated by fig. 6.1 (C). As we can see through this example, each step of the algorithm simultaneously removes several states. Indeed, the canonicalization is performed in two iterations.

At each step the current set of states is split into two subsets. The first iteration reduces all the states that belong to S2 and are reachable from S1 (S1=$\{\langle 0,1\rangle, \langle 0,2\rangle\}$

, S2= $\{\langle 1,0\rangle, \langle 1,2\rangle, \langle 3,1\rangle\}$). At the second iteration, the set of states is split into two subsets by passing to the next level. Finally, $\{\langle 0,1\rangle\}$ is the canonical representative of S.

$S = \{\langle 0,1\rangle, \langle 0,2\rangle\}$ is split into two subsets by passing to the next level. In this case, $S1 = \{\langle 0,1\rangle\}$ and $\{\langle 0,2\rangle\}$.

6.3 Experiments

6.3.1 Results of the canonicalization algorithm

Model	**Without canonicalization**		**Using canonicalization**	
Net	lddNodes	Size (bytes)	lddNodes	Size (bytes)
ring4	29052	464832	9728	155648
ring5	200440	3207040	65280	1044480
ring6	773383	12374128	187320	2997120
philo6	5232	83712	2268	36288
philo8	28224	451584	12288	195840
philo10	142080	2273280	61440	983040
fms4	41983	671728	8904	142464
fms5	115633	1850128	19339	309424
robot4	615472	9847552	38610	617760
robot5	2457884	39326144	90090	1441456
robot6	8236228	131779648	185640	2970240
robot7	24110012	385760192	348840	5581440
train2-4	2860268	45764288	3888	62208
train2-6	2838971	45423536	8544	136704
train2-8	1849162	29586592	79680	1274880
erk10	307404	4918464	11555	184880
erk20	12936794	206988704	461929	7390864

TABLE 6.1: Size of the SOG before and after the Canonicalization

We report on the results of performance measurements for all approaches. We have extensively tested all approaches with a variety of different models in order to identify the achieved speedups. The machine used for these tests has 12 processors, each with 12 cores. We have run the multi-threaded algorithm on 12 cores (one thread per core). For the experiments of the distributed algorithm we used 40 cores. We recall that all the tested examples are parametrized.

Model	**Without canonicalization**			**Using canonicalization**		
Net	TimeSeq	TimeMin	Sp	TimeSeq	TimeMin	Sp
ring4	2.15	0.64	3.3	4.56	0.67	6.8
ring5	43.46	11.39	3.8	94.42	10.73	8.8
ring6	870.87	243.13	3.6	2100.41	226.06	9.3
philo6	0.47	0.28	1.6	0.95	0.21	4.5
philo8	11.94	5.63	2.1	23.32	3.92	5.9
philo10	311.10	168.03	1.8	586.03	75.82	7.7
fms3	11.08	7.12	1.5	29.37	6.89	4.2
fms4	208.11	162.74	1.3	559.83	103.10	5.4
fms5	2088.75	1543.52	1.3	7433.16	1168.79	6.3
robot4	7.10	3.03	2.3	14.60	2.72	5.3
robot5	36.16	16.94	2.1	80.80	12.53	6.4
robot6	135.09	70,67	1.9	312.74	49.05	6.3
robot7	505.90	311.78	1.6	1005.91	166,99	6.0
train2-4	42.49	28.88	1.4	101.28	26.33	3.8
train2-6	29.64	16.10	1.8	82.59	14.63	5.6
train2-8	26.73	10.73	2.5	41.64	7.18	5.8
erk10	46.63	18.84	2.4	264.80	34.02	7.7
erk20	957.68	397.23	2.4	5978.54	657,47	9.1

TABLE 6.2: Experimental results of the distributed-memory algorithm with and without canonicalization

In Table 6.1, for each model, we have computed the number of the used LDD nodes as well as the overall size of the SOG. We observe that the canonicalization leads to important memory savings and reduces the number of LDD nodes. Notably, by using the canonicalization, the memory consumption for the SOG construction is consequently decreased.

In Table 6.2, we have measured the time (in seconds) consumed by the construction of the SOG in a sequential way. It can be seen that the canonicalization slow down the computation time in this case. Then, we show the execution times of the distributed algorithm by progressively increasing the number of cores. Finally, we calculated the maximum speedup achieved by each model. We describe the same measurements for the multi-threaded algorithm in Table 6.3 and for the hybrid approach in Table 6.4.

Compared to the non canonical version of the SOG, for the distributed approach, the construction of the canonical SOG consumes less memory and less execution

Model	**Without canonicalization**			**Using canonicalization**		
Net	TimeSeq	TimeMin	Sp	TimeSeq	TimeMin	Sp
ring4	2.15	0.42	5.1	4.56	0.84	5.4
ring5	43.46	5.71	7.6	94.42	15.49	6.1
ring6	870.87	87.71	9.9	2100.41	429.58	4.8
philo6	0.47	0.12	3.9	0.95	0.31	3.0
philo8	11.94	1.89	6.3	23.32	4.56	5.1
philo10	311.10	36.67	8.5	586.03	97.07	6.0
fms4	208.11	21.85	9.5	559.83	159.53	3.5
fms5	2088.75	222.82	9.3	7433.16	1270.56	5.8
robot4	7.10	0.92	7.7	14.60	2.00	7.3
robot5	36.16	4.86	7.4	80.80	9.66	8.3
robot6	135.09	17.75	7.6	312.74	38.97	8.0
robot7	505.90	61.97	8.1	1005.91	136,96	7.3
train2-4	42.49	12.58	3.3	101.28	26.21	3.8
train2-6	29.64	5.83	5.0	82.59	12.57	6.5
train2-8	26.73	2.86	9.0	41.64	5.53	7.5
erk10	46.63	5.11	9.1	264.80	27,43	9.6
erk20	957.68	86.44	11.0	5978.54	567,82	10.5

TABLE 6.3: Experimental results of the multi-threaded algorithm with and without canonicalization

time. This is due to the reduction of the size of the set of states that represent the aggregate to be sent/received by the involved processes, which allows to reduce the communication overhead.

Results in Table 6.3 show that the canonicalization slowdowns the execution time of the multi-threaded algorithm. Nevertheless, the multi-threaded algorithm achieves a maximum speedup of 10 when 12 cores are used for the construction of the canonicalized SOG. It can be seen in Table 6.4 that the canonicalization improves the scalability of the hybrid approach. We notice that it increases the speedup for the tested examples (except ring6 example).

Table 6.5 illustrates the differences in terms of size of messages exchanged before and after the application of the canonicalization process. We can remark that the size of the exchanged messages has been decreased drastically by using this approach. Doing that, we reduce the duration required by the communication between the involved processes (sent and received messages).

In fig. 6.2, we display the time execution consumption of the philo10 example in the case of the distributed memory. The green curve denotes the time required

Model	**Without canonicalization**			**Using canonicalization**		
Net	TimeSeq	TimeMin	Sp	TimeSeq	TimeMin	Sp
ring4	2.15	0.57	3.7	4.56	0.38	12.0
ring5	43.46	3.98	11.3	94.42	2.96	35.6
ring6	870.87	18.66	46.6	2100.41	77.55	28.8
philo6	0.47	0.17	2.7	0.95	0.20	4.7
philo8	11.94	0.94	12.7	23.32	1.27	18.3
philo10	311.10	13.91	22.3	586.03	15.82	37.0
fms4	208.11	22.84	9.1	559.83	46.27	12.1
fms5	2088.75	406.50	6.0	7433.16	315.16	23.6
robot4	7.10	1.14	6.2	14.60	0.68	21.4
robot5	36.16	3.28	11.0	80.80	3.21	25.1
robot6	135.09	17.52	7.7	312.74	7.72	40.5
robot7	505.90	32.89	15.4	1005.91	22.4	45.4
train2-4	42.49	7.99	5.3	101.28	14.19	7.1
train2-6	29.64	4.12	7.2	82.59	6.64	12.4
train2-8	26.73	3.23	8.2	41.64	1.94	23.6
erk10	46.63	3.08	15.1	264.80	12.10	21.9
erk20	957.68	51.71	18.5	5978.54	94.86	63.0

TABLE 6.4: Experimental results of the hybrid algorithm with and without canonicalization

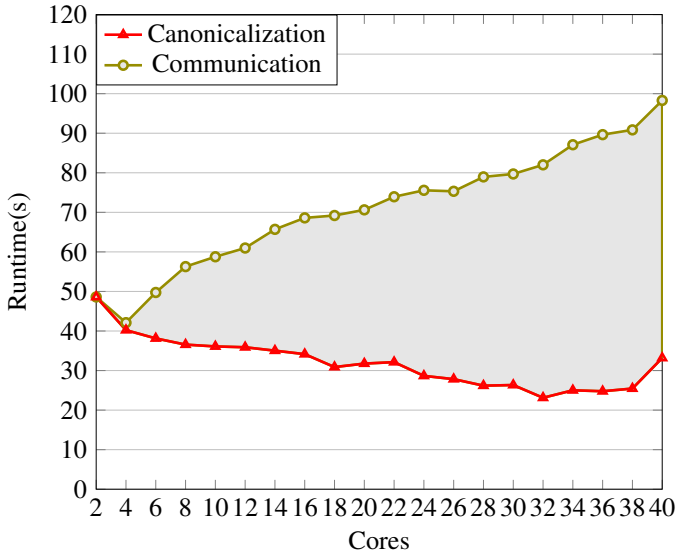


FIGURE 6.2: Runtime of philo10 example using distributed-memory algorithm

Model	**Size Of Messages without canonicalization**	**Size Of Messages using canonicalization**
ring4	281 022	16 286
ring5	4 961 128	114 408
ring6	63 188 686	217 390
philo8	5 076 252	33 900
philo10	118 986 364	176 824
robot4	4 694 292	92 457
robot5	22 755 426	221 391
robot5	88 153 511	466 106
robot6	155 362 177	835 567
train2-4	6 694 448	5600
train2-6	6 684 946	12 850
train2-8	4 807 590	75 000
erk10	8 676 325	16 773
erk20	181 569 699	1 081 666

TABLE 6.5: Size (in bytes) of the exchanged messages between 4 processes

by the communication between processes without using canonicalization algorithm, whereas, the red curve denotes the time consumed by the canonicalization itself. As it can be seen, when we include the use of the canonicalization in our process, the time dedicated to the communication between processes decreases. That allows us to obtain good results in the global runtime of the whole building process (the gray area).

6.3.2 Comparative analysis

As we have explained in chapter 5, we have experimented the distributed approach with 40 processes because the maximum speedups for this approach are detected for at the most 30 processes. Then, we have stopped the evaluation at 40 processes because the speedups have been decreased for all the tested models by increasing the number of processes more than 30 (mainly due to the communication overhead).

Moreover, because of material limitations (a cluster containing 12 processors with 12 cores for each processor), the multi-threaded approach is limited to 12 cores.

In fig. 6.3 we analyze the execution times of our algorithms measured on different architectures using the canonicalization algorithm. We notice that all approaches bring an improvement in term of runtime performance. But, it can be seen that the hybrid approach outperforms the other approaches and scales better. Indeed, the runtime continues to decrease by increasing the number of cores.

Figure 6.4 shows the speedups of the parallel algorithms for different model sizes. Compared hybrid approach against sequential runs, the speedup curves are moving up as the number of cores increases indicating that this approach scales better than the other approaches. For the ERK20 example, it achieves a maximum speedup of 63 when 144 cores CPU are used. Comparing the hybrid approach against the multi-threaded and the distributed approaches, using up to 12 cores, we note that the difference between all approaches is relatively small. More than 12 cores, we compare the hybrid approach against the distributed one, we can see that the distributed approach is less efficient. Indeed, the curve of speedup slows down by increasing the number of processes while it continue to increase for the hybrid approach.

We have achieved another way to improve the execution time of the hybrid approach: we applied the canonicalization only to the aggregates to be sent. In this case, the computed maximum speedup is relative to the sequential construction of the SOG without the use of the canonicalization. Compared to the parallel construction of the SOG without canonicalization in Table I, the partial application of the canonicalization allows to reduce the runtime for the majority of cases. Indeed, the impact of the canonicalization appears mostly when the size of aggregates is considerable, and then the size of the message to be sent is sizable. Also, we have noticed that, for some examples having a larger size, the execution process often hangs without the use of the canonicalization.

6.4 Conclusion

In this chapter, we have proposed a symbolic (LDD-based) algorithm to canonically reduce the size of the SOG aggregates. The algorithm is implemented within the previous parallel approaches (proposed in chapters 3, 4 and 5) for the SOG construction. The first one is dedicated to distributed memory multiprocessors using the message passing paradigm, the second is a multi-threaded algorithm for shared memory architecture and the third one is an algorithm dedicated to hybrid (shared-distributed memory) architectures. Experiments show that canonicalization allows to efficiently decrease memory consumption in all approaches. Also, it allows to improve the

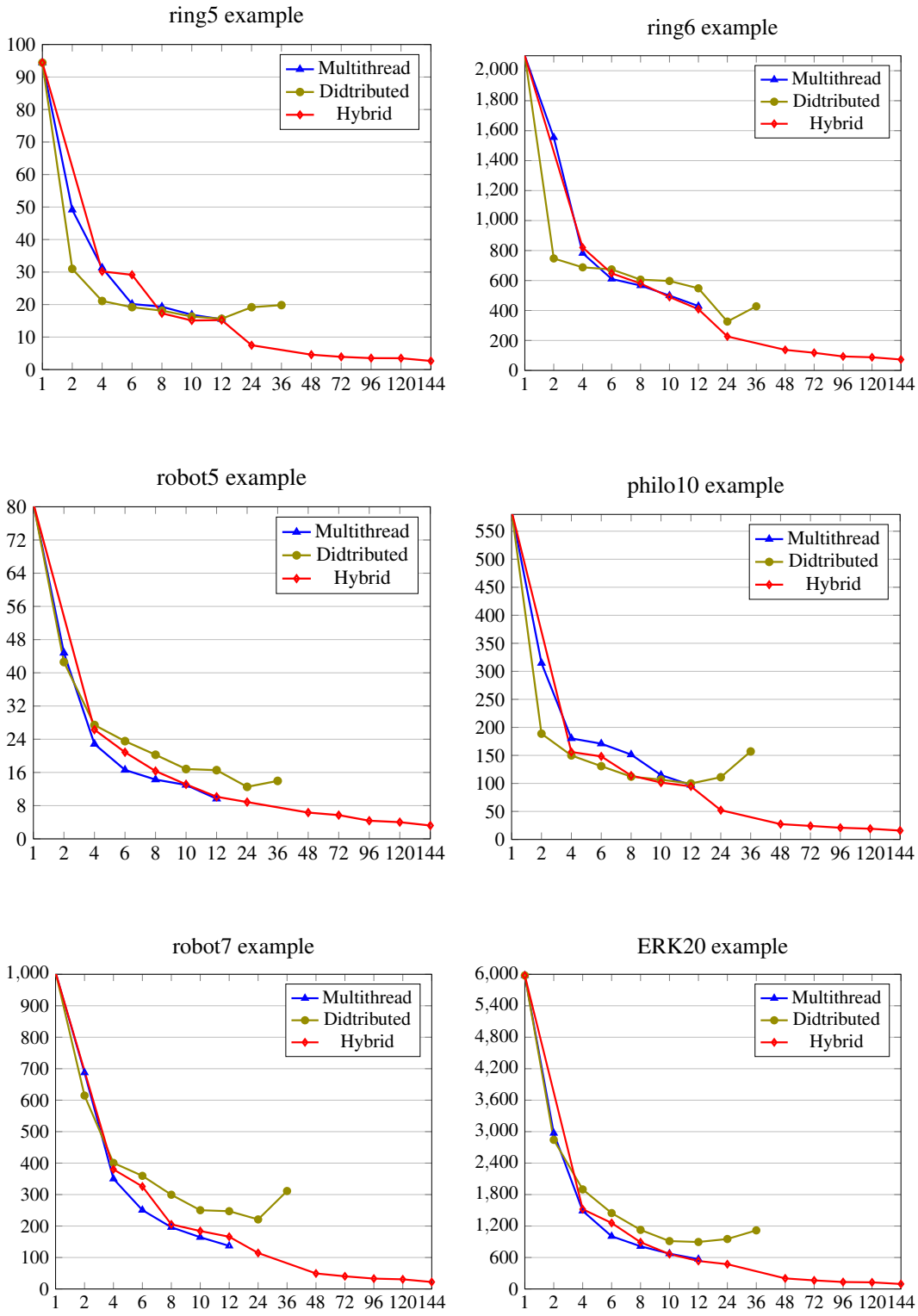


FIGURE 6.3: Comparison between the multi-threaded, distributed and hybrid approaches in term of the obtained runtime using the canonicalization algorithm.

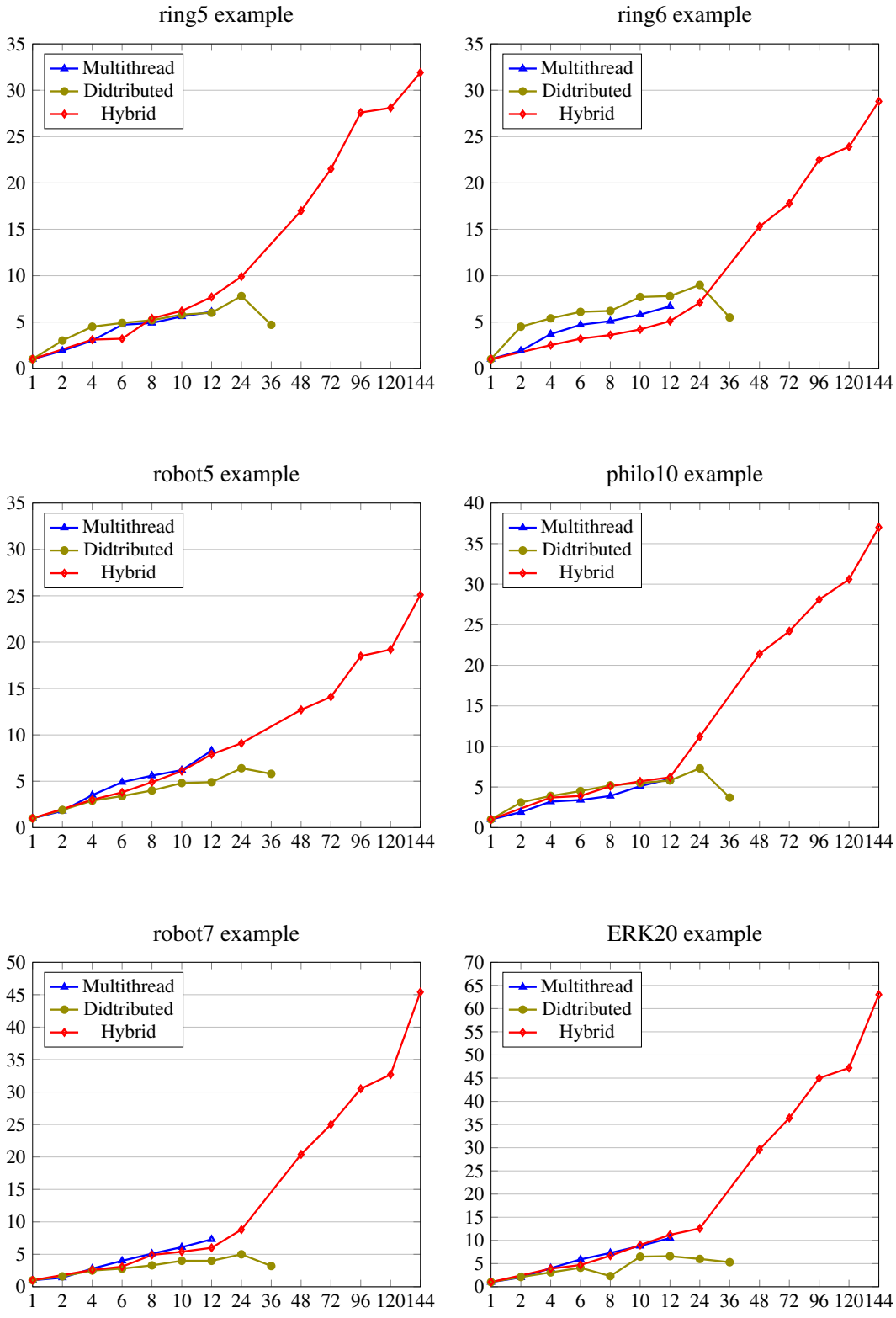


FIGURE 6.4: Comparison between the multi-threaded, distributed and hybrid approaches in term of the achieved speedup using the canonicalization algorithm

scalability for both distributed and hybrid approaches. This is due to the fact that the communication time between processes has been drastically reduced. However, in the case of the shared memory approach the construction process becomes a bit slower although it consumes less memory.

Chapter 7

PMC-SOG : Parallel Model checker based on the SOG

Contents

5.1 Introduction 48

5.2 A hybrid approach for constructing a SOG 49

5.2.1 Aims and hypothesis 49

5.2.2 Description of the algorithms 50

5.2.3 Termination detection 53

5.2.4 Correctness proof 53

5.3 Experiments 55

5.3.1 Results of the hybrid approach 55

5.3.2 Comparative analysis 57

5.4 Conclusion 59

7.1 Introduction

The use of distributed processing increases the speedup and scalability of model checking by exploiting the cumulative computational power and memory of a cluster of computers. Such approaches have been studied in various contexts leading to different proposed solutions for both symbolic and explicit model checking (Chapter 1).

In the previous chapters, we have investigated three approaches to parallelize the SOG construction using three different algorithms to benefit from additional speedups and performance improvement in execution time. The key idea of our approaches is to build simultaneously several nodes (aggregates) of the symbolic graph.

The first proposed algorithm is based on a shared memory architecture. We adopt a dynamic load balancing scheme in order to balance the load on threads sharing the SOG construction task. [Oun+17a; Oun+17b]. The second approach is based on a distributed memory architecture. Therefore, each process has its own local memory, and communication between processes is performed by using the Message Passing Interface (MPI). Distribution of aggregates is performed through the computation of a hash function in order to cope with communication overhead. Finally, the third is a hybrid technique which combines the two previous approaches. The SOG construction is shared among a set of processes, where each creates a number of threads that run on (typically) the same number of CPUs of a processor.

In the current chapter, we exploit the strengths of the parallel generation of the reachable state space to propose a parallel model checking based on the parallel construction of the SOG, where both event-based and state-based properties can be expressed, combined, and verified. Instead of composing the whole system with the Büchi automaton representing the negation of the formula to be checked, we make the synchronization of the automaton with an abstraction of the original reachability graph of the system: an event/state-based SOG.

The event-based and state-based logics for properties expression are interchangeable. An event-based logic can be encoded as a change in state variables. Also, it is possible equip a state with different events to reflect different values of its internal variables. However, converting from one representation to the other often leads to a significant enlargement of the state space due to the increased size of the formula. Typically, event-based semantic is adopted to compare systems according to some equivalence or pre-order relation (e.g., [TBD95; KV92]), while state-based semantics is more suitable to model-checking approaches [GV01]. Being able to use both events and state-based atomic propositions within a same LTL formula allows to express properties in an intuitive and an easy way leading to short formulas. Also, the design of an event-state based model checker allow to compare our approach with both state- and event-based model checkers.

Aiming to improve the efficiency of the SOG-based model checking approach, we propose in this chapter model checking algorithms built on our parallel construction of the SOG. Our proposed model checker is based on the automata theoretic approach to LTL model checking (see Figure 7.1). It takes as input an LTL formula φ and the description of the model. The model checking problem is reduced to an on-the-fly emptiness check processed on the synchronized product of the Büchi automaton $A_{\neg\varphi}$ corresponding to the negation of the formula, and the automaton A_{SOG} of the SOG (checking whether the language $L(\neg\varphi \otimes A_{SOG}) = \varnothing$ or not). If the emptiness check returns true, then the formula is proved to be satisfied by the Model.

Otherwise, a counterexample (a possible run that violates φ) is supplied to the user.

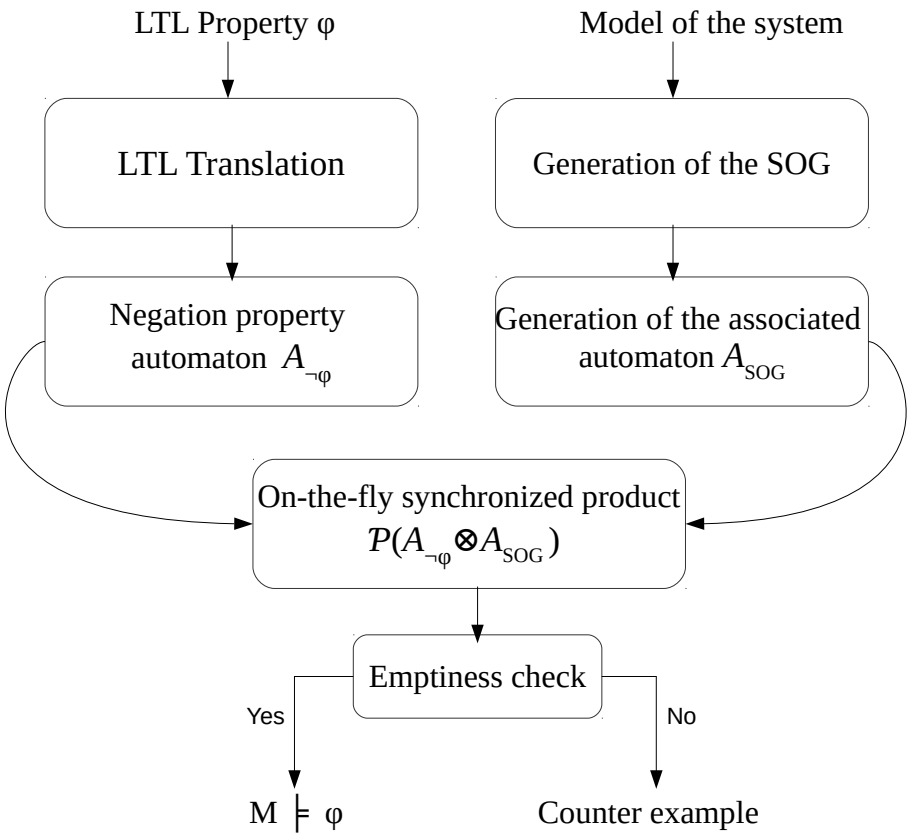


FIGURE 7.1: Our Model Checker process

The main contributions of this chapter are:

- The proposal of a parallel model checker based on a parallel construction of the SOG. Such model checker aims to verify LTL properties on-the-fly using different parallelization techniques. This allows in some cases to achieve the model checking without a complete construction of the SOG.
- Both event-based and state-based properties can be expressed and verified.
- The implementation and the evaluation of an on-the-fly $LTL \setminus X$ parallel model-checker.

We have implemented different proposed algorithms for model checking using Spot [DL+16], an object-oriented model checking library written in C++. We have then evaluated the performances of our approach according to experimental results obtained on different benchmarks. Further, we have compared these results with other solutions proposed in the literature.

7.2 Multi-Core LTL Model Checking

In this section, we present an on-the-fly multi-core LTL model checking approach based on the Symbolic Observation Graph. Model checker thread allows to compute the synchronized product of the automaton modeling the LTL formula to be checked with the SOG of the model to be checked. Thus, the parallelization is basically in the construction phase of the SOG (see algorithm 6) while the emptiness check process is sequential.

7.2.1 Description of the multi-core approach

We propose two algorithms using different techniques depending on the granularity of parallelization task. The first one splits the SOG construction over a number of threads according to a dynamic load balancing by following the same method presented in Chapter 3. In the second algorithm, parallelization is performed at the level of the decision diagrams operations by using lock-less data structures and a work-stealing scheduling strategy.

```
   Data: agg, dest : Aggregate;
   s: Stack;
   obs_tr : Set of actions;
1  obs_tr=getFirableObservablelactions(agg);
2  for every action t ∈ obs_tr do
3      s.push(t);
4      get_successor(agg, t);
5  /*builder threads : parallel section*/
6  while s is not empty do
7      t=s.pop();
8      dest=ComputeAggregate;
9      if dest does not exist in the SOG then
10         dest.div = isDiv(DEST);
11         dest.deadlock = isDeadlock(DEST);
12         Insert into the SOG the node dest;
13         Insert into the SOG the arc (agg, t, dest));
14     else
15     Insert into the SOG the arc (agg, t, dest)); ;
```

Algorithm 6: Computing an aggregate

7.2.2 Parallelization at the level of aggregates

We propose an LTL model checker approach that is based on a number of threads using the same technique, described in chapter 3, for the construction of a SOG. In

addition to threads dedicated to the construction of the SOG (builder threads), we use a supplementary thread, called model checker thread that is executed in parallel with builder threads.

Steps of this approach are illustrated by the UML state-chart of fig.7.2.

As described in chapter 3, we recall that builder threads cooperate simultaneously in order to build a SOG by adopting a dynamic load balancing. We associate with each thread a stack to store the aggregates to be processed. A newly generated aggregate is pushed into the stack of the builder thread having the minimum load. The minimum load is determined from a table load that stores the number of aggregates to be processed for every thread.

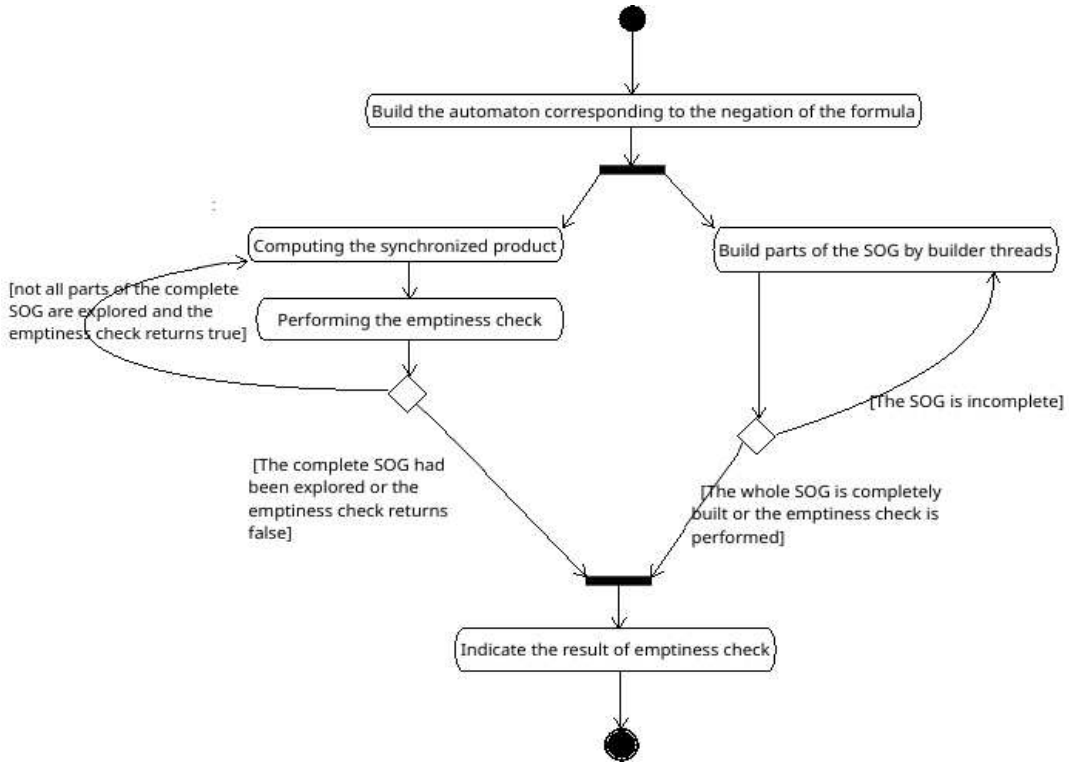


FIGURE 7.2: UML statechart diagram of the LTL model checker based on POSIX threads

Since the computation of the synchronized product and the building of the SOG are performed simultaneously, it is possible that the model checker thread tries to reach nodes of the SOG that are not yet built. For this reason, we add a Boolean attribute to every aggregate to indicate whether its successors are built, or not. By checking the value of this attribute of an aggregate, the model checker thread can check if it is possible to reach the successors of the aggregate, otherwise it has to wait until the attribute of the considered aggregate is updated.

The termination of the model checking algorithm is determined by the model checker thread. It is performed when the emptiness check process is finished. When the construction of the SOG is not finished whereas the model checker thread terminates, in this case threads are forcibly terminated by the model checker thread. The property is then proved to be unsatisfied by the system. If threads finish the construction of the SOG before the computation of the synchronized product is finished, only the builder threads terminate. In this cas, the conclusion is that the original property is satisified by the system.

7.2.3 Parallelization at the level of decision diagrams operations

For this algorithm, a finer granularity level of parallelization is proposed for the model checking. We exploit parallel LDD operations already implemented in the Sylvan library[DP15; DP16] which is used in the implementation of the SOG construction. The difference between the current and the previous algorithm is parallelism granularity. Indeed, in the previous algorithm, parallelization concerns aggregates construction, whereas the current algorithm performs parallelization at the level of LDD operations through the use of recursive functions. In the previous algorithm, only one thread is responsible of the construction of an aggregate while, in the second, different threads can cooperate in the construction of a unique aggregate.

Sylvan is a multi-core decision diagrams library based on the work-stealing framework Lace [DP14]. Like the majority of work-stealing frameworks, Lace implements task-based parallelism by creating tasks (spawn) and waiting for their completion (sync) to use the results. Parallelization of LDD operations is performed by using lock-less data structures and work-stealing scheduling strategy. The basic idea behind Work-stealing [BL99] is to break down a calculation into small tasks. Independent subtasks are stored in queues (work-pools) and idle processors steal tasks from the queues of busy processors. This will allow a processor to always have tasks to perform. The data structures used by Sylvan library are based on the lock-less paradigm, which ensures mutual exclusion and depends on atomic operations.

Consider an LTL property and an LKS system. The synchronized product, between the automaton modeling the negation of the property with the SOG of the considered LKS, is computed sequentially, however building an aggregate is performed in parallel. When the model checker requests to get the successors of an aggregate, we perform the construction of an aggregate successors using LDD parallel operations. Algorithm 7 describes the construction of an aggregate successors using Sylvan library. We start by computing enabled observable actions from the considered aggregate. Then, using $SPAWN$, for every enabled observable action t,

```
   Data: agg,dest : Aggregate;
   s: Stack;
   obs_tr : Set of actions;
1  obs_tr=getFirableObservablelactions(agg);
2  for every action t ∈ obs_tr do
3      s.push(t);
4      SPAWN(ComputeAggregate,get_successor(agg,t));
5  while s is not empty do
6      t=s.pop();
7      dest=SYNC(ComputeAggregate);
8      if dest does not exist in the SOG then
9          dest.div = isDiv(DEST);
10         dest.deadlock = isDeadlock(DEST);
11         Insert into the SOG the node dest;
12         Insert into the SOG the arc (agg,t,dest));
13     else
14     Insert into the SOG the arc (agg,t,dest)); ;
```

Algorithm 7: Computing an aggregate successors using Lace

we create a task for the recursive function $ComputeAggregate$ to compute the aggregate obtained by firing t. Results are retrieved by calling SPAWN. It is worth noting that $ComputeAggregate$ is implemented recursively using basic LDD parallel operations. Further, since tasks are stored in queues with a LIFO (Last In First Out) order, then last created task will be the first one to deliver its results. When a new aggregate is inserted in the SOG, we compute whether it contains a divergence (unobserved cycle)or deadlock state.

7.2.4 Implementation

The implementation of the multi-core model checker is based on Spot library [DL+16]. It is an object-oriented model checking library written in C++ that offers a set of building blocks that allow to develop LTL model checkers based on the automata theoretic approach.

The automaton class used by Spot to represent $\omega - automata$ is called action-based $\omega - automaton$ ($T\omega A$ for short). As its name implies, the $T\omega A$ class supports only action-based acceptance, but it can emulate state-based acceptance using action-based acceptance by ensuring that all actions leaving an aggregate have the same acceptance set membership. In addition, there is a class, named *kripke* that can be used to represent an LKS. During model checking, we translate built parts of the SOG to the LKS structure.

The checking algorithm visits the synchronized product of the $\omega - automaton$ corresponding to the negation of the formula and the LKS corresponding to the SOG (see figure 7.1). The translation of an LTL formula into an $\omega - automaton$ is proposed by Spot and it is dedicated to different formalism for the representation of the system to be checked.

Three abstract classes must be specialized to represent the $\omega - automata$. The first abstract class defines a state, the second allows to iterate on the successors of a given state and the last one represents the whole $\omega - automaton$. In our context, we have derived these classes for implementing a multi-core model checker based on the SOG. It is important to notice that the effective construction of the SOG is driven by the emptiness check algorithm of Spot and it can be be managed on-the-fly. The construction of aggregates depends on places and actions appearing as atomic proposition in the checked formula. In our proposed approach, we distribute the construction of the SOG on a number of threads. Two techniques of parallelization are proposed:

- Using POSIX threads
  We create using Pthread library a number of threads which are responsable of the construction of the SOG (builder threads). Each aggregate will be constructed by a unique thread.

- Using Lace framework
  We exploit the parallel decision diagrams operations already implemented in Sylvan library for the construction of the SOG. Thus, several threads may cooperate to construct one aggregate and the balancing of the load between threads is controlled by Lace.

In a given state, an atomic proposition associated with a place is satisfied if the place contains at least one token. In this case, the complete set of states corresponding to an aggregate is obtained by applying, until saturation, the action relation limited to the actions which do not modify the truth value of atomic propositions. Instead of checking this constraint explicitly, we statically restrict the set of Petri net actions to be considered to the ones which do not modify the state of places used as atomic propositions.

7.2.5 Experiments

We implemented our model checker using the C++ language with Pthreads, then, we implemented it using Task-based parallelism. The experimental results presented in

this section were obtained on Magi cluster[1] of Paris 13 university. This cluster has 12 processors each with 40 cores (two Xeon X5670 at 2.93GHz), 24GB of RAM and they are connected by an InfiniBand network. A total of 20 models from the Model Checking Contest benchmark[2] have been used in the experiments, which are shown in Table 7.1 (we filtered out models which were too small to be interesting, or too big to fit into the available memory). In this chapter, we have added three examples from Model Checking Contest: CloudObsManagement, Swimming Pool and Token Rings examples.

We exploit for these experiments a shared memory architecture with 40 cores. We have measured the time in seconds consumed by the verification of 10 formulas (a selection of random LTL formulas) by progressively increasing the number of cores (10, 20 and 40). Every run was repeated at least three times, to exclude any accidental fluctuation in the measurements. For each number of thread, we calculated the average of the obtained runtime. Then we collect the minimum time reached by each technique.

The results of our experiments are reported in 7.1. We perform a simple comparison between our multi-core model checker (Lace version and Pthread version) with the LTSmin model checker [Kan+15]. Since LTSmin is a state based model checker, we considered only the atomic propositions based on the states. LTSmin is an $LTL/CTL/\mu-calculus$ model checker, it started out as a generic toolset for manipulating (LTS)s. It connects a sizeable number of existing verification tools as language modules, enabling the use of their modeling formalisms. For sake of simplicity, we choose the ordinary place/transition Petri nets in PNML format. We adopt the DFS algorithm for all approaches. We keep all the parameters across the different model checkers the same. Using these parameters on a per-model basis could give faster results than presented here. It would, however, say little about the scalability of the core algorithms. Therefore, we decided to leave all parameters the same for all the models. We avoid resizing of the state storage in all cases by increasing the initial hash table size enough for all benchmarked input models.

We observe that the version with parallel LDD operations outperforms the one based on POSIX threads in most cases. We note that the minimum runtime is not necessarily achieved when using 40 cores for the two versions. A first remark concerning the obtained results in terms of time (minimum obtained runtime for the verification of 10 formulas) is that no model checker has an absolute advantage on the other for all the experiments: our Model checker is the faster for some models

[1] http ://www.univ-paris13.fr/calcul/wiki/
[2] https://mcc.lip6.fr/models.php

Net	States	PMC-SOG Pthread	PMC-SOG Sylvan	LTSmin
Cloud5by2	1.61×10^{6}	4,94	4,12	3,54
Cloud10by5	1.07×10^{10}	527,65	238,18	1376,23
Cloud20by10	–	2429,88	1014,69	13073,75
philo10	59049	5,23	3,82	0,28
philo20	3.48×10^{9}	1011,91	852,56	108,42
robot5	184756	0,26	0,14	0,49
robot10	2.00×10^{7}	3,72	2,03	50,79
robot20	4.10×10^{9}	10,77	10,83	35,17
robot50	–	29,19	30,72	–
SPool2	89621	2,87	3,45	7,07
SPool3	3408031	34,87	19,07	21,26
SPool4	3.22×10^{7}	105,77	91,7	152,13
Spool5	5.91×10^{8}	267,95	103,21	336,36
TRing10	58905	112,59	42,13	0,18
TRing15	3.53×10^{7}	1015,66	924,87	206,17
train24	86515	2,82	2,23	0,34
train48	2.39×10^{10}	18,23	18,55	267,62
kanban5	2.54×10^{6}	9,19	10,43	7,52
kanban10	1.00×10^{9}	1055,27	845,65	370,09

TABLE 7.1: Comparison in terms of minimum runtime between the multi-core PMC-SOG using POSIX threads, the multi-core PMC-SOG using Sylvan and LTSmin

while the LTSmin performs better for other examples. We notice that the robot50 example have no results for the LTSmin approach. The execution of this example was interrupted, displaying the message "tree leafs table full!". This can be explained by the fact that size of a SOG is smaller than the explicit representation of state space used by LTSmin, which makes the available memory space sufficient for our model checker while insufficient for LTSmin.

Figure 7.3 shows a selection of our experimental results. The performances are presented using a logarithmic scale. Each point represents an experiment, a model and formula pair. Roughly speaking, for the speedup figure, our approach performs better than LTSmin for all the experiment appearing below the diagonal while the opposite stands for the experiment above the diagonal (and it is the reverse for the runtime figure). We did not represent the robot50 example since we did not have a result when using LTSmin. We represented a total of 72 formulas including 36 verified formulas and 36 violated formulas. We put in blue the verified formulas and in red the violated formulas. The formulas considered include a selection of random LTL formulas with different sizes, which were filtered to be able to compare with a state based model checker. As shown by the figure, the two methods often have very similar performances in terms of minimum obtained runtime (right) and maximum achieved speedup (left).

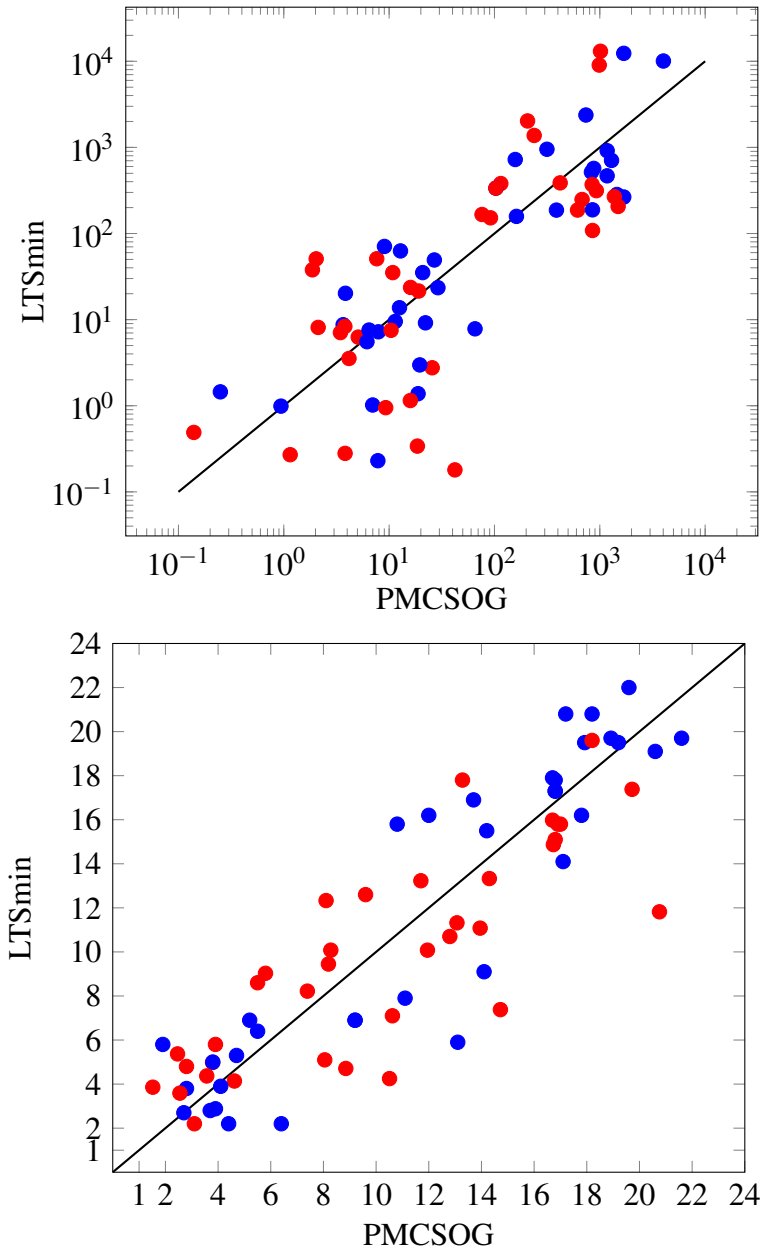


FIGURE 7.3: Comparison of the PMCSOG-Sylvan against LTSmin in minimum runtime (above) and maximum speedup (below) for verified formulas (blue) and violated formulas (red)

7.3 A hybrid approach for parallel LTL Model Checking

We present in this section a distributed LTL model checker for a distributed memory platform. Such a model checker allows to check event-based and state based $LTL \setminus X$ formulas on a SOG following the same hybrid technique, for the construction of a SOG, described in chapter 5. Indeed, we create a set of processes such that every

process uses a set of threads in order to build the SOG.

7.3.1 Description of the algorithm

The basic idea of our algorithm is to propose an LTL parallel model checker for a distributed memory setting. As it is described in chapter 5, the building of the SOG is performed by running several processes where each process consists of several threads. Our aim is to compute the synchronized product (the model checking) simultaneously with the construction of the SOG. For this purpose, we add a process, called model checker process, to compute the synchronized product and then performs the emptiness check between the automaton modeling the negation of the LTL formula with the LKS corresponding to the SOG.

Since every process has its own memory, the model checker process will then, during the computation of the synchronized product, request from the builder processes for an aggregate whether it contains the divergence (unobserved cycle) or a deadlock state, and in the memory of which process its successors are stored. Model checker process does not need to receive the LDD structure from the builders processes, it requires only a unique identifier for every aggregate. This allows to reduce the size of exchanged messages between the model checker process and the builder ones, as the size of an aggregate can be huge. For this purpose, we use the hash function MD5 [Dob96]. It is the same one used to decide where an aggregate will be stored during construction of the SOG, i.e. the same function used to statically balance the load of aggregates construction on builder processes.

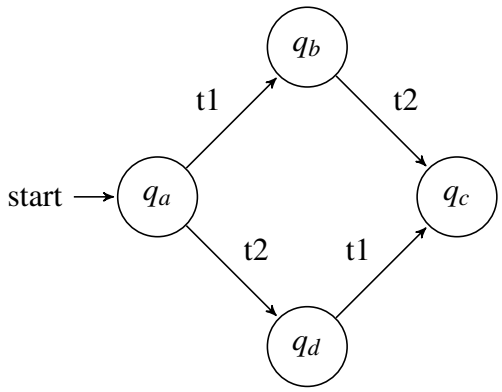


FIGURE 7.4: A sample example of the SOG

In order to illustrate how the model checker process retrieves information about a SOG from builder processes, we consider the SOG sample described in fig. 7.4. Let us assume that we have three builder processes. Assume that the static load balancing produces three graphs illustrated in fig. 7.5. In this figure, a dotted node of a graph in a process i indicates an aggregate such that its LDD structure is not stored by process

i. Process i stores only its MD5 value (its unique identifier) and the identity of the process that should store it. For instance, in the graph built by process 1, there is only one aggregate that is stored with its LDD structure by process 1 which is q_a. For the two others aggregates q_b and q_d only their MD5 values are stored. Indeed, q_b is stored in memory of process 2, while q_d and q_c are stored in memory of process 3.

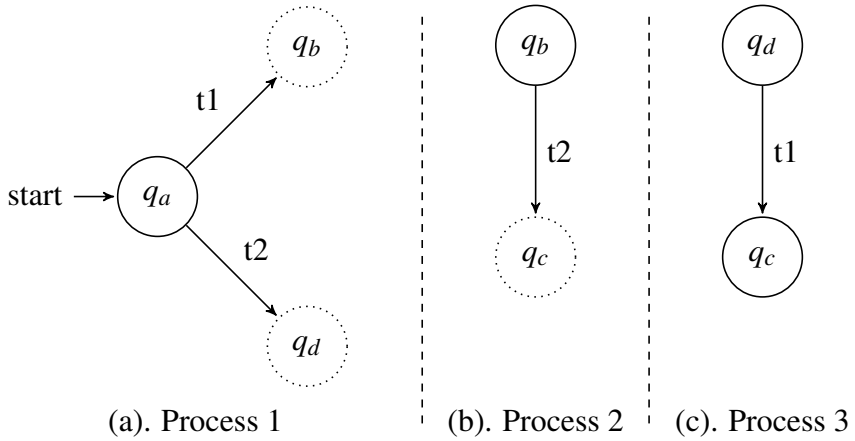


FIGURE 7.5: A distributed SOG

When performing model checking, model checker process starts by asking process 1 the initial aggregate. The builder process gets as an answer its MD5 value. When the model checker asks the successors of an aggregate, it sends its request to the process storing the aggregate. For example, for the SOG of fig.7.5, when the builder process wants to explore successors of the initial aggregate, its sends a request to the builder process 1. As an answer to this request, the model checker should get the MD5 values of aggregates q_b and q_d and the identities of processors storing these successors. By this way, in order to get the successors of q_b, the model checker process will send its request to process 2.

The termination is managed totally by the model checker process. Indeed, the builder process must exist even they terminate the SOG construction, as the model checker process may request information from them about SOG parts that are not yet being explored.

Algorithm 8 illustrates different messages that can be received by a builder process from the model checker process. We use function $Receive(TAG, message, 0)$ to receive a message from process with ranking 0 such that the parameter TAG indicates the type of message. Function $Send(TAG, message, 0)$ allows to send a message to process with ranking 0 and TAG indicates the type of message. A builder process can receive the following types of messages from a model checker process:

```
   Data: agg : Aggregate ;
   message,md5 : string;
   TAG : integer;
   list_succ : array of Aggregate;
1  Receive(TAG,message,0);
2  switch TAG do
3      case ASK_INITIAL do
4          agg=getInitialAggregate();
5          /*Build a string representing the message to send containing the
             MD5 value of the initial aggregate and the process storing it */
6          message=BuildMessage(agg.md5,agg.storingProcess);
7          Send(ACK_INTIAL,message,0);
8      case ASK_SUCCESSORS do
9          /* Extract the MD5 value that identifies the aggregate to get its
             successors list*/
10         md5=decodeMessage(message);
11         agg=findAggregate(md5);
12         list_succ=getSuccessors(agg);
13         /* The message to send consists of the list of the successors of agg,
             and Boolean attributes that indicate whether agg contains a
             divergence or a deadlock state */
14         message=BuildMessage(list_succ,agg.isDiv(),agg.isDeadlock());
15         Send(ACK_SUCESSORS,message,0);
16     case TERMINATE do
17         Terminate this process;
```

Algorithm 8: Messages received by a builder process from a model checker process

- Message *ASK_INITIAL* corresponds to a request from the model checker process to get the MD5 value and the identity of the process storing the aggregate. This message is only received by the master builder process.

- Message *ASK_SUCCESSORS* corresponds to a request from the model checker process to get the list of successors of the aggregate having the MD5 value specified in the message. Also, information about divergence and deadlocks related to the same aggregate are concerned.

- Message $TERMINATE$ is sent by the model checker process in order to terminate builder processes. This message is sent when the model checker process had already performed the emptiness check.

7.3.2 Implementation

We consider, for the hybrid approach, a cluster architecture (cluster of CPUs). We create a number of processes communicating via message passing interface (MPI). The master process is dedicated to the model checking while the others are dedicated to the construction of the SOG.

The main goal of this approach is to improve the scalability of the implementation. In a distributed computation, every process has simply its own memory which it fully manages. However, the main drawback of this approach is the communication overhead between processes which causes a delay in the execution time.

Like the previous section, the implementation of the hybrid algorithm is based on Spot library. Our parallel implementation of the SOG-based model checker using all approaches is available on line at PMC-SOG[3] project (open source).

7.3.3 Experiments

Net	States	PMCSOG Sylvan	LTSmin	PMCSOG hybrid
Cloud5by2	1.61×10^{6}	4,12	3,54	4,36
Cloud10by5	1.07×10^{10}	238,18	1376,23	344,15
Cloud20by10	–	1014,69	13073,75	1457,62
Cloud40by20	–	–	–	16921,48
philo10	59049	3,82	0,28	13,47
philo20	3.48×10^{9}	852,56	108,42	923,56
robot5	184756	0,14	0,49	0,15
robot10	$2{,}00\times10^{7}$	2,03	50,79	1,62
robot20	4.10×10^{9}	10,83	35,17	4,23
robot50	–	30,72	–	12,7
SPool2	89621	3,45	7,07	3,54
SPool3	3408031	19,07	21,26	24,62
SPool4	3.22×10^{7}	91,7	152,13	110,56
Spool5	5.91×10^{8}	103,21	336,36	176,98
TRing10	58905	42,13	0,18	32,80
TRing15	3.53×10^{7}	924,87	206,17	653,36
TRing20	$2.44\text{E}\times10^{10}$	–	–	1384,07
train24	86515	2,23	0,34	2,19
train48	2.39×10^{10}	18,55	267,62	25,43
train96	$2{,}59\times10^{21}$	–	–	1178,65
kanban5	2.54×10^{6}	10,43	7,52	8,87
kanban10	1.00×10^{9}	845,65	370,09	614,77
kanban50	8.05×10^{11}	–	–	3694,42

TABLE 7.2: Experimental results of the PMC-SOG and LTSmin

[3] https://depot.lipn.univ-paris13.fr/PMC-SOG/mc-sog

The distributed memory computation is performed on a cluster using several processes that communicate via MPI. Each process runs several threads sharing the same memory space. As noticed in chapter 5, the cluster architecture allowed us to reduce the runtime of a SOG building for the majority of our examples.

The experimental results are given in Table 7.2. The reported table describes experimental data related to different parameterized models.

For each tested problem, we have executed the hybrid algorithm. We set the number of threads for each process at 12 and we increased the number of processes from 3 to 6 then to 12. We analyzed the runtime efficiency, and we kept the minimum obtained runtime. Then, we compare it against the multi-core approach based on the SOG and the LTSmin model checker. We can notice that the results of the hybrid approach is close to the multi-core one. The increase in the number of machines does not necessarily imply an increase in the efficiency of the algorithm, the performance drops when we increase the number of the used machines.

It is worth noting that we have the possibility to execute larger examples using the cluster environment, while the execution stops due the lack of available memory for both the multi-core model checking based on the SOG and the LTSmin model checker. This is because a distributed approach provides a total memory space that is equal to the sum of memory spaces allocated to each created process by the operating system. Thus, for the hybrid approach, we have performed the execution of larger examples that generate up to $2.5E^{+21}$ states. However, a major drawback, of the distributed model checker approach, is the number of messages exchanged through the network. Indeed, communication increases the execution time of the model checker.

The main reason for the development of a distributed model checking algorithm is typically to increase the available memory space, although this can impact the runtime negatively due to communication cost.

7.4 Conclusion

In this chapter, we have proposed two on-the-fly parallel model-checker approaches for $LTL\setminus X$ logic based on event-based and state-based symbolic observation graphs. First approach targets shared memory architecture. We have proposed two versions using different techniques of parallelization. One is based on the POSIX threads, and the other is based on the work-stealing framework LACE. Second approach is dedicated to a distributed-shared memory architecture by creating several processes including threads.

The emptiness check is performed on-the-fly during the construction of the SOG allowing to reduce the runtime of the model checking process. Moreover, when the

SOG is visited entirely during the model checking (i.e. the property holds), it can be reused for the verification of another formula (at the condition that the set of atomic propositions is included in the one used by the SOG). Experiments show that our approach is competitive in comparison with the LTSmin parallel model checker even we have used a DFS approach for the exploration of the SOG during the computation of a synchronized product. Indeed, DFS exploration is not well suited with our approaches for the construction of a SOG. For this reason, we expect better results by using BFS (Breadth-First Search).

Further, the distributed-shared approach show the ability to manage larger examples than any multi-core approach despite a performance loss. The loss is primarily induced by the communication between processes.

Chapter 8

Conclusion

In this thesis, we have dealt with model checking of concurrent systems. Model checking is a powerful formal verification method that can be used to improve the safety of concurrent systems. Given a formal specification, model checking algorithms analyze the system behavior by exhaustively searching all reachable state space. 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 limitation of the model checking approach is the well-known problem of combinatorial state space explosion.

As a solution to cope with the explosion state problem, the symbolic observation graphs (SOG) provide a compressed version of state space. A SOG is a graph whose construction is guided by a set of observable atomic propositions involved in a linear time formula. Such atomic propositions can represent events or actions (event-based SOG) or state-based properties (state-based SOG). The nodes of a SOG are aggregates hiding a set of local states which are equivalent with respect to the observable atomic propositions, and are compactly encoded using Binary Decision Diagram techniques (BDDs). The arcs of an event-based SOG are exclusively labeled with observable actions. It has been proven that both event and state-based SOGs preserve stutter-invariant LTL formulas.

The use of SOG requires more computations and consequently the runtime of the construction of a SOG can become a bottleneck in order to complete the model checking process. In order to overcome the latter drawback while keeping an efficient approach to tackle the state space explosion, we have investigated the parallelization of model checking based on SOGs. Our work is performed mainly in two steps.

In a first step, we have investigated the parallelization of the construction of a SOG. For this purpose, we have proposed three approaches. First one targets multi-core architectures. Its key idea is to build simultaneously several nodes (aggregates) of the symbolic graph through the use of several threads and by adopting a dynamic

load balancing scheme in order to balance the load on threads sharing the SOG construction task. The second approach targets distributed memory architectures. Parallelization is performed through the creation of processes that communicate by using the Message Passing Interface MPI. Load balancing is ensured statically through the use of a hash function. Third approach is hybrid one that combines multi-threading and multi-processing. Indeed, we create several processes such that every process creates a set of threads. Then, load balancing is performed in two levels: statically at processes level and dynamically at threads level. In addition to parallelization, we have improved the reduction of the SOG. Since MDDs are a generalization of BDDs to the integer domain, a node in MDD may represent several nodes in BDDs. For this reason, we have used LDDs instead of BDDs to represent aggregates of a SOG. Also, we have proposed algorithm to determine a single (canonical) representative for each strongly connected component in every aggregate allowing to remove from memory a consequent number of states which are no more necessary for the construction process. Experiments for the three approaches have shown a more reduced runtime comparing to a sequential construction. The better speed-up was obtained with the multi-core approach. However, the canonical representation of aggregates have drastically improved the hybrid approach due reduction in communication cost.

In second step, we have proposed an on-the-fly parallel model-checker based on two approaches for LTL $\setminus$ X logic using event-based and state-based symbolic observation graphs. First one, called multi-core approach, is mainly dedicated for shared memory architectures. We have performed parallelization using two different techniques. One is based on the POSIX threads, and the other is based on a work-stealing framework. Second approach, called hybrid approach, is dedicated for distributed-shared memory architectures by using processes such that every one creates a set of threads. For these two approaches, model checking (the emptiness check) is performed simultaneously with the construction of a SOG. Experiments show competitive results for multi-core approach comparing it to the LTSmin parallel model checker. However, the hybrid approach compensates its runtime by allowing to perform model checking on larger models for which the other approaches fail.

Our approaches for the parallel construction of the SOG and the proposed model checker have been implemented within prototypes and have been validated through different examples. Our implementations are available in the PMCSOG open source project (https://depot.lipn.univ-paris13.fr/PMC-SOG). The obtained results are encouraging and open several improvement issues.

As perspectives, we think that runtime for model checking can be more reduced by using graphical processing units (GPUs). Indeed, such units provide thousands of processing cores allowing intensive parallelism related to the processing of large

blocks of data like vectors and matrix. Then, the time of computation of an aggregate can be reduced as it is based on operations using vectors (states). Another direction is related to the model checking of large systems. We think that the hybrid approach can be extended to support cloud computing and to benefit from huge provided storage capacity by such technology.

List of Figures

2.1 Example of LTS 12
2.2 Example of Kripke Structure 13
2.3 A Labeled Kripke Structure 14
2.4 MDD (left) and LDD (right) representing the set $\{\langle 0,1\rangle, \langle 0,2\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 3,1\rangle\}$ (For simplicity, we hide paths to 0 for MDD and paths to 0 and 1 for LDD) 16
2.5 An event-based SOG with Obs={a,b} 20
2.6 A state-based SOG, with $AP = \{a,b\}$ 22
2.7 An LKS and its SOG 25

3.1 Shared memory multi-core 31
3.2 Example of BDD and MDD 36
3.3 Runtime of the construction of the SOG 40
3.4 Speedup of the multi-thread algorithm 40

4.1 Distributed memory multi-processor 42
4.2 Runtime of distributed-memory algorithm 47
4.3 Speedup of the distributed-memory algorithm 47

5.1 Architecture of the hybrid approach 50
5.2 Distribution of the aggregates on 12 processes 57
5.3 Comparison between the multi-threaded, distributed and hybrid approaches in term of the obtained runtime. On the X axis are the numbers of cores and on the Y axis are execution times in seconds. . 59
5.4 Comparison between the multi-threaded, distributed and hybrid approaches in term of the achieved speedups. On the X axis are the numbers of CPU cores and on the Y axis are the speedups 60

6.1 Example of canonicalization 64
6.2 Runtime of philo10 example using distributed-memory algorithm . . 68
6.3 Comparison between the multi-threaded, distributed and hybrid approaches in term of the obtained runtime using the canonicalization algorithm. 71

6.4 Comparison between the multi-threaded, distributed and hybrid approaches in term of the achieved speedup using the canonicalization algorithm 72

7.1 Our Model Checker process 76
7.2 UML statechart diagram of the LTL model checker based on POSIX threads 78
7.3 Comparison of the PMCSOG-Sylvan against LTSmin in minimum runtime (above) and maximum speedup (below) for verified formulas (blue) and violated formulas (red) 84
7.4 A sample example of the SOG 85
7.5 A distributed SOG 86

List of Tables

3.1 Comparison between the SOG construction based on BuDDy and Sylvan 38
3.2 Experimental results of the multi-threaded algorithm 39

4.1 Experimental results of the distributed-memory algorithm 46

5.1 Scalability of the Hybrid approach of the construction of the SOG . 56
5.2 The best execution times using multi-threaded, distributed and hybrid approaches 58

6.1 Size of the SOG before and after the Canonicalization 65
6.2 Experimental results of the distributed-memory algorithm with and without canonicalization 66
6.3 Experimental results of the multi-threaded algorithm with and without canonicalization 67
6.4 Experimental results of the hybrid algorithm with and without canonicalization 68
6.5 Size (in bytes) of the exchanged messages between 4 processes . . . 69

7.1 Comparison in terms of minimum runtime between the multi-core PMC-SOG using POSIX threads, the multi-core PMC-SOG using Sylvan and LTSmin 83
7.2 Experimental results of the PMC-SOG and LTSmin 88

Bibliography

[AB86] James Archibald and Jean-Loup Baer. “Cache coherence protocols: Evaluation using a multiprocessor simulation model”. In: *ACM Transactions on Computer Systems (TOCS)* 4.4 (1986), pp. 273–298.

[Ake78] Sheldon B. Akers. “Binary decision diagrams”. In: *IEEE Transactions on computers* 100.6 (1978), pp. 509–516.

[AKH97] Susann C Allmaier, Markus Kowarschik, and Graham Horton. “State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor”. In: *Petri Nets and Performance Models, 1997., Proceedings of the Seventh International Workshop on*. IEEE. 1997, pp. 112–121.

[And90] Thomas E. Anderson. “The performance of spin lock alternatives for shared-money multiprocessors”. In: *IEEE Transactions on Parallel and Distributed Systems* 1.1 (1990), pp. 6–16.

[Bar+10] Jiri Barnat et al. “Divine: Parallel distributed model checker”. In: *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*. IEEE. 2010, pp. 4–7.

[Bar+13] Jiří Barnat et al. “DiVinE 3.0–an explicit-state model checker for multithreaded C & C++ programs”. In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 863–868.

[Bar+18] Jiri Barnat et al. “Parallel Model Checking Algorithms for Linear-Time Temporal Logic”. In: *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 457–507.

[BBR09] Jirı Barnat, Luboš Brim, and Petr Rockai. “DiVinE 2.0: High-performance model checking”. In: *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*. 2009, pp. 31–32.

[Bel+13] Carlo Bellettini et al. “Distributed CTL model checking in the cloud”. In: *arXiv preprint arXiv:1310.6670* (2013).

[BGS06] Roderick Bloem, Harold N Gabow, and Fabio Somenzi. "An algorithm for strongly connected component analysis in n log n symbolic steps". In: *Formal Methods in System Design* 28.1 (2006), pp. 37–56.

[BL99] Robert D Blumofe and Charles E Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[BPW10] Stefan Blom, Jaco van de Pol, and Michael Weber. "LTSmin: Distributed and symbolic reachability". In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 354–359.

[BRR06] Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg. *Petri nets: central models and their properties: advances in petri nets 1986, part I proceedings of an advanced course bad honnef, 8.–19. September 1986*. Vol. 254. Springer, 2006.

[Bry86] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691.

[Bry92] Randal E Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". In: *ACM Computing Surveys (CSUR)* 24.3 (1992), pp. 293–318.

[Bur+94] Jerry R Burch et al. "Symbolic model checking for sequential circuit verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.4 (1994), pp. 401–424.

[BVDP08] Stefan Blom and Jaco Van De Pol. "Symbolic reachability for process algebras with recursive data types". In: *International Colloquium on Theoretical Aspects of Computing*. Springer. 2008, pp. 81–95.

[Cas+94] Stefano Caselli et al. "Experiences on SIMD massively parallel GSPN analysis". In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 1994, pp. 266–283.

[CCM01] Stefano Caselli, Gianni Conte, and Paolo Marenzoni. "A distributed algorithm for GSPN reachability graph generation". In: *Journal of Parallel and Distributed Computing* 61.1 (2001), pp. 79–95.

[CE81] Edmund M Clarke and E Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Workshop on Logic of Programs*. Springer. 1981, pp. 52–71.

[CGN98] Gianfranco Ciardo, Joshua Gluckman, and David Nicol. “Distributed state space generation of discrete-state stochastic models”. In: *INFORMS Journal on Computing* 10.1 (1998), pp. 82–93.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.

[CH91] Armin B Cremers and Thomas N Hibbard. “Axioms for concurrent processes”. In: *New Results and New Trends in Computer Science*. Springer, 1991, pp. 54–68.

[CKZ96] Edmund M Clarke, Manpreet Khaira, and Xudong Zhao. “Word level model checking—avoiding the Pentium FDIV error”. In: *Proceedings of the 33rd annual Design Automation Conference*. ACM. 1996, pp. 645–648.

[Cla+01] Edmund Clarke et al. “Progress on the state explosion problem in model checking”. In: *Informatics*. Springer. 2001, pp. 176–194.

[CLS00] 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*. Springer. 2000, pp. 103–122.

[Dij12] Tom Dijk. “The Parallelization of Binary Decision Diagram operations for model checking”. In: (2012).

[DL+16] Alexandre Duret-Lutz et al. “Spot 2.0—A Framework for LTL and \ω-Automata Manipulation”. In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2016, pp. 122–129.

[Dob96] Hans Dobbertin. “Cryptanalysis of MD5 compress”. In: *rump session of Eurocrypt* 96 (1996), pp. 71–82.

[DP14] Tom van Dijk and Jaco C van de Pol. “Lace: non-blocking split deque for work-stealing”. In: *European Conference on Parallel Processing*. Springer. 2014, pp. 206–217.

[DP15] 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*. Springer. 2015, pp. 677–691.

[DP16] Tom van Dijk and Jaco van de Pol. “Sylvan: multi-core framework for decision diagrams”. In: *International Journal on Software Tools for Technology Transfer* (2016), pp. 1–22.

[DP17] Tom van Dijk and Jaco van de Pol. “Sylvan: multi-core framework for decision diagrams”. In: *International Journal on Software Tools for Technology Transfer* 19.6 (2017), pp. 675–696.

[EKP13] Sami Evangelista, Lars Michael Kristensen, and Laure Petrucci. “Multi-threaded explicit state space exploration with state reconstruction”. In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2013, pp. 208–223.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. “Parallel state space construction for model-checking”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 2001, pp. 217–234.

[Gro+96] William Gropp et al. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel computing* 22.6 (1996), pp. 789–828.

[GV01] Jaco Geldenhuys and Antti Valmari. “Techniques for smaller intermediary BDDs”. In: *International Conference on Concurrency Theory*. Springer. 2001, pp. 233–247.

[GW91] Patrice Godefroid and Pierre Wolper. “A partial approach to model checking”. In: *Logic in Computer Science, 1991. LICS’91., Proceedings of Sixth Annual IEEE Symposium on*. IEEE. 1991, pp. 406–415.

[HB07] Gerard J Holzmann and Dragan Bosnacki. “The design of a multi-core extension of the SPIN model checker”. In: *IEEE Transactions on Software Engineering* 33.10 (2007).

[HDG10] Monika Heiner, Robin Donaldson, and David Gilbert. “Petri nets for systems biology”. In: *Symbolic Systems Biology: Theory and Methods. Jones and Bartlett Publishers, Inc., USA (in Press, 2010)* (2010).

[He09] Y He. *Multicore-enabling a Binary Decision Diagram algorithm*. 2009.

[HIK04] 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*. Springer. 2004, pp. 196–210.

[Hol04] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-Wesley Reading, 2004.

[Hol08] Gerard J Holzmann. "A stack-slicing algorithm for multi-core model checking". In: *Electronic Notes in Theoretical Computer Science* 198.1 (2008), pp. 3–16.

[Hol12] Gerard J Holzmann. "Parallelizing the spin model checker". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2012, pp. 155–171.

[IB02] Cornelia P Inggs and Howard Barringer. "Effective state exploration for model checking on a shared memory architecture". In: *Electronic Notes in Theoretical Computer Science* 68.4 (2002), pp. 605–620.

[Kam98] Timothy Kam. "Multi-valued decision diagrams: Theory and applications". In: *J. Multiple-Valued Logic* 4.1 (1998), pp. 9–62.

[Kan+15] Gijs Kant et al. "LTSmin: high-performance language-independent model checking". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2015, pp. 692–707.

[KP04] 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*. Springer. 2004, pp. 474–483.

[KP08a] Kais Klai and Laure Petrucci. "Modular construction of the symbolic observation graph". In: *ACSD*. IEEE, 2008, pp. 88–97.

[KP08b] 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*. Springer. 2008, pp. 288–306.

[KTD09] Kais Klai, Samir Tata, and Jörg Desel. "Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes". In: *International Conference on Business Process Management*. Springer. 2009, pp. 294–309.

[KTD11] Kais Klai, Samir Tata, and Jörg Desel. "Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes". In: *Data & Knowledge Engineering* 70.5 (2011), pp. 467–482.

[KV92] Roope Kaivola and Antti Valmari. “The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic”. In: *International Conference on Concurrency Theory*. Springer. 1992, pp. 207–221.

[Lam77] Leslie Lamport. “Proving the correctness of multiprocess programs”. In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[LPW10] Alfons Laarman, Jaco van de Pol, and Michael Weber. “Boosting multi-core reachability performance with shared hash tables”. In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc. 2010, pp. 247–256.

[LPW11] Alfons Laarman, Jaco van de Pol, and Michael Weber. “Multi-core LTS min: marrying modularity and scalability”. In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 506–511.

[LS99] Flavio Lerda and Riccardo Sisto. “Distributed-memory model checking with SPIN”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 1999, pp. 22–39.

[Mar96] Andrew Martin. *Machine-Assisted Theorem-Proving for Software Engineering*. Oxford University Computing Laboratory, Programming Research Group, 1996.

[Mat87] Friedemann Mattern. “Algorithms for distributed termination detection”. In: *Distributed computing* 2.3 (1987), pp. 161–175.

[MD98] 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*. IEEE. 1998, pp. 52–57.

[ODP17] Wytse Oortwijn, Tom van Dijk, and Jaco van de Pol. “Distributed binary decision diagrams for symbolic reachability”. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM. 2017, pp. 21–30.

[Oss10] Jörn Ossowski. “JINC: a multi-threaded library for higher-order weighted decision diagram manipulation.” PhD thesis. University of Bonn, 2010.

[Oun+17a] Hiba Ouni et al. “A Parallel Construction of the Symbolic Observation Graph: the Basis for Efficient Model Checking of Concurrent Systems”. In: *SCSS 2017. The 8th International Symposium on Symbolic Computation in Software Science 2017*. Ed. by Mohamed Mosbah and Michael Rusinowitch. Vol. 45. EPiC Series in Computing. EasyChair, 2017, pp. 107–119.

[Oun+17b] Hiba Ouni et al. “Parallel Symbolic Observation Graph”. In: *Ubiquitous Computing and Communications (ISPA/IUCC), 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on*. IEEE. 2017, pp. 770–777.

[Oun+18] Hiba Ouni et al. “Reducing Time and/or Memory Consumption of The SOG construction in a Parallel Context”. In: *Ubiquitous Computing and Communications (ISPA/IUCC), 2018 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2018.

[Oun+19] Hiba Ouni et al. “Towards parallel verification of concurrent systems using the Symbolic Observation Graph”. In: *2019 19th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE. 2019, pp. 23–32.

[QS82] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *International Symposium on programming*. Springer. 1982, pp. 337–351.

[Rod+06] Cássio L Rodrigues et al. “A bag-of-tasks approach for state space exploration using computational grids”. In: *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*. IEEE. 2006, pp. 226–235.

[SD97] Ulrich Stern and David L Dill. “Parallelizing the Murϕ verifier”. In: *International Conference on Computer Aided Verification*. Springer. 1997, pp. 256–267.

[TBD95] ZP Tao, Gregor von Bochmann, and Rachida Dssouli. “Verification and diagnosis of testing equivalence and reduction relation”. In: *Proceedings of International Conference on Network Protocols*. IEEE. 1995, pp. 14–21.

[Val90] Antti Valmari. “A stubborn attack on state explosion”. In: *International Conference on Computer Aided Verification*. Springer. 1990, pp. 156–165.

[VAM96] François Vernadat, Pierre Azéma, and François Michel. “Covering step graph”. In: *International Conference on Application and Theory of Petri Nets*. Springer. 1996, pp. 516–535.

[VDBL13] Freark Van Der Berg and Alfons Laarman. “SpinS: Extending LTSmin with Promela through SpinJa”. In: *Electronic Notes in Theoretical Computer Science* 296 (2013), pp. 95–105.

[VDLVDP13] Tom Van Dijk, Alfons Laarman, and Jaco Van De Pol. “Multi-core BDD operations for symbolic reachability”. In: *Electronic Notes in Theoretical Computer Science* 296 (2013), pp. 127–143.

[VG14] 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.

[Wal94] David W Walker. “The design of a standard message passing interface for distributed memory concurrent computers”. In: *Parallel Computing* 20.4 (1994), pp. 657–673.

[XB00] Aiguo Xie and Peter A Beerel. “Implicit enumeration of strongly connected components and an application to formal verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.10 (2000), pp. 1225–1230.

[ZC10] Yang Zhao and Gianfranco Ciardo. “Symbolic computation of strongly connected components using saturation”. In: (2010).

[Šti13] Vladimír Štill. “State space compression for the DiVinE model checker”. PhD thesis. Bachelor’s thesis. Masaryk University, Faculty of Informatics, 2013. Available at< http://is. muni. cz/th/373979/fi_b, 2013.