Parallel model checking for multiprocessor architecture

Rodrigo Tacla Saad

▶ **To cite this version:**

Rodrigo Tacla Saad. Parallel model checking for multiprocessor architecture. Mathematical Software [cs.MS]. INSA de Toulouse, 2011. English. NNT : . tel-00678352

HAL Id: tel-00678352

https://theses.hal.science/tel-00678352

Submitted on 12 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National des Sciences Appliquées de Toulouse (INSA Toulouse)

Discipline ou spécialité :

Sûreté de logiciel et calcul de haute performance

Présentée et soutenue par :
Rodrigo Tacla Saad

le : mardi 20 décembre 2011

Titre :

Parallel Model Checking for Multiprocessor Architecture

JURY

Dr. François Vernadat, Professeur à l'INSA, Toulouse, France
Dr. Jean-Marie Farines, Professeur à l'Université Federalle de Santa Catarina, Brésil
Dr. Jean-Paul Bodeveix, Professeur à l'Université Paul Sabatier, Toulouse, France
Patrick Farail, Ingénieur Logiciel à AIRBUS, Toulouse, France

---

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
LAAS - CNRS

Directeur(s) de Thèse :
Dr. Bernard Berthomieu, Chargé de recherche au LAAS – CNRS, Toulouse, France
Dr. Silvano Dal Zilio, Chargé de recherche au LAAS – CNRS, Toulouse, France

Rapporteurs :
Dr. Fabrice Kordon, Professor à L'Université P. & M. Curie, Paris, France
Dr. Radu Mateescu, Chargé de recherche à INRIA, Rhône-Alpes, France

Abstract

In this thesis, we propose and study new algorithms and data structures for model checking finite-state, concurrent systems. We focus on techniques that target shared memory, multi-cores architectures, that are a current trend in computer architectures.

In this context, we present new algorithms and data structures for exhaustive parallel model checking that are as efficient as possible, but also "friendly" with respect to the work-sharing policies that are used for the state space generation (e.g. a work-stealing strategy): at no point do we impose a restriction on the way work is shared among the processors. This includes both the construction of the state space as the detection of cycles in parallel, which is is one of the key points of performance for the evaluation of more complex formulas.

Alongside the definition of enumerative, model checking algorithms for many-cores architectures, we also study *probabilistic verification algorithms*. By the term probabilistic, we mean that, during the exploration of a system, any given reachable state has a high probability of being checked by the algorithm. Probabilistic verification trades savings at the level of memory usage for the probability of missing some states. Consequently, it becomes possible to analyze part of the state space of a system when there is not enough memory available to represent the entire state space in an exact manner.

[Keywords:] Parallel Model Checking, Concurrent algorithms and data structures, Formal Methods, Formal Verification and Temporal Logic.

Acknowledgements

It is a pleasure to thank those who helped me to make this thesis possible. I will never forget these last three years, they have changed my perspectives about life in general.

First of all, I would like to thank Dr. François Vernadat and Dr. Bernard Berthomieu for giving me this chance. I had never considered myself to enroll into a Doctorate program before they invited me and for that I will always be grateful to them. Dr. Vernadat has always been more than a friend, his support was paramount to accomplish this thesis. The same can be said for Dr. Berthomieu, whose experience and knowledge allowed me to achieve many of my goals. Thank you again for this chance, for your support, for your advices and for all the exchanges we had during these three years.

I owe my deepest gratitude to my supervisor, Dr. Silvano Dal Zilio, whose encouragement, guidance and support from the very beginning to the end that helped me to overcome all the obstacles for obtaining this thesis. Thank you for the trust and also by giving me the complete freedom to decide the directions of my work. Without his wisdom and experience, this thesis would not have been possible. I do not only consider him as an outstanding professional, but also as an example to be followed. I found in him the right balance between life and work. It has been an honor to work with him.

I extend my thanks to Dr. Jean-Marie Farines, whose encouragement made me considered a graduate career in Computer Science. Since my undergraduation, he has supported me in many ways. Thank you for the advices, this thesis itself is the result from one of your advices.

I would also like to thank the Dr. Radu Mateescu and Dr. Fabrice Kordon for having reviewed this work and for their careful comments. All the

remarks were of great importance to improve this document. I extend my thanks to Dr. Jean-Paul Bodeveix, Dr. Eng. Patrick Farail, members of the examining committee of this thesis.

I would like to show my gratitude to Dr.Alexandre Hamez, Dr.Didier Le Botlan and Dr. Sakkaravarthi Ramanathan for helping me in correcting this document.

I acknowledge the Topcased project for their financial support. They provided me all the tools that were necessary to achieve my goals.

I am indebted to many of my colleagues who supported me during these last three years. I will never forget our coffee breaks where we discussed so different subjects, so many debates about the world, future, humanity, etc... Thank you Dr.Florent Peres, Dr.Pierre-Emmanuel Hladik, Camille Cazeneuve, Jorge Gomez Montalvo, Nouha Abid, Dr.Riadh Ben Halima, Dr. Luiz Douat, Johan Mazel, Nguyen Xuan Hung, Dr.Med Mehdi Jatlaoui, Dr.Franck Chebila, Guillaume Kremer, . . . These moments I spent with you were crucial to my personal and professional development. It was great to share experiences and knowledge with interesting people like you.

I would also like to thank my family for the support they provided me through my entire life. They are for sure my living force, I can not tell how many times I found in them the strength to continue my work. I must acknowledge my father Roberto Elias Saad, my mother Neli Tacla Saad, my brother Fábio Tacla Saad and my sisters Bruna Tacla Saad and Marina Tacla Saad. Without your love and encouragements I would not have finished this thesis.

Lastly, I offer my regards and gratitude to all of those who supported me in any respect during the completion of this thesis.

Contents

List of Figures **vii**

Glossary **xi**

1 Introduction **1**

2 Model Checking — Related Work **9**

2.1 Introduction 10

2.2 What is Model Checking 11

2.2.1 State Graph and Kripke Structure 11

2.2.2 Temporal Logic Formula 12

2.2.3 Model Checking 15

2.3 Parallel Model Checking 17

2.3.1 Parallel Computers 18

2.3.2 Chronology 21

2.3.3 Parallel State Space Construction 22

2.3.4 Parallel LTL Model Checking 29

2.3.5 Parallel CTL Model Checking 35

2.4 Probabilistic Verification 37

2.4.1 Bloom Filters 38

2.4.2 Compact Hash Table 40

2.5 Contributions 41

2.5.1 Parallel State Space Construction 42

2.5.2 Parallel Model Checking 44

2.5.3 Probabilistic Verification 46

3 Parallel State Space Construction **47**

3.1 Introduction . . . 47

3.1.1 Algorithms Overview . . . 48

3.2 General Lock Free Approach . . . 50

3.2.1 Shared and Local Data . . . 51

3.2.2 Different Phases of the Algorithm . . . 52

3.2.3 Experiments . . . 57

3.2.4 Discussion about the experiments . . . 62

3.3 Mixed approach: Localization Table based algorithm . . . 63

3.3.1 Localization Table . . . 64

3.3.2 Algorithm . . . 67

3.3.3 Experiments . . . 68

3.4 Comparison With Other Algorithms and Tools . . . 73

3.5 Conclusions . . . 77

4 Parallel Model Checking With Lazy Cycle Detection — MCLCD **79**

4.1 Introduction . . . 79

4.2 List of Supported Properties — the LRL Logic . . . 82

4.3 Some Graph Theoretical Properties . . . 84

4.4 A Model Checking Algorithm with Lazy Cycle Detection . . . 87

4.4.1 Notations . . . 89

4.4.2 Model Checking Reachability properties — $E(\psi \cup \phi)$ . . . 90

4.4.3 Model Checking Liveness Properties — $A(\psi \cup \phi)$ . . . 90

4.4.4 Model Checking the Leadsto Property — $\psi \leadsto \phi$ . . . 99

4.5 Correctness and Complexity of our Algorithms . . . 101

4.6 Parallel Implementation of our Algorithm . . . 108

4.7 Experimental Results . . . 110

4.7.1 Speedup Comparison Between RG and RPG Algorithms . . . 111

4.7.2 Comparison with a Standard Algorithm . . . 114

4.7.3 Conclusion About the Experiments . . . 116

4.8 Conclusions . . . 118

5 Probabilistic Verification: Bloom Table **127**

5.1 Introduction 128

5.2 Probabilistic Data Structure 130

5.2.1 Bloom filter and Compact Hash Table 130

5.2.2 Bloom Table 131

5.3 Probabilistic Verification 140

5.4 Conclusions 147

6 Conclusions **149**

A Experiments **155**

A.1 Models 155

A.2 Parallel State Space Construction 155

A.3 Probabilistic Verification 161

B Mercury **169**

B.1 Technical Description 169

B.2 Mercury Configurations for Exhaustive Exploration 172

B.3 Mercury Configurations for Probabilistic Exploration 173

B.4 Mercury Configurations for Parallel Model Checking 174

B.5 Installation 175

B.5.1 Usage 175

C Résumé en Français **181**

C.1 Abstract 181

C.2 Introduction 182

C.3 Contribuition 185

C.4 Sommaire : Brève Description de la Thèse 187

C.5 Conclusion 189

Bibliography **195**

List of Figures

2.1 Model Checking development cycle. 11
2.2 Semantics of LTL. 14
2.3 Semantics of CTL. 15
2.4 Shared memory. 19
2.5 Distributed memory. 19
2.6 State space models. 23
2.7 Spin multi-core speedup analysis (Hol08). 28
2.8 Example of the sequential depth-first search post-order of vertices. (BBS01) 30
2.9 Illustration of some operations on a Bloom filter. 39
2.10 Parallel model checking algorithms for shared memory machines. 45

3.1 Parallel memory organization. 49
3.2 Shared and private data overview. 52
3.3 Phases alternation. 53
3.4 Collision resolution with local AVl trees. 57
3.5 Speedup analysis for PH 12 and FMS 7 models. 58
3.6 Occupancy rate for PH 12 with 16 processors. 58
3.7 Collision analysis for FMS 7 and PH 12. 59
3.8 Threshold analysis using 16 processors for PH 12 and FMS 7. 60
3.9 Comparison of Different Implementations. 62
3.10 Algorithm overview. 63
3.11 Insertion in a Localization Table. 67
3.12 Shared and private data overview. 67
3.13 Speedup analysis. 71
3.14 Collisions vs load factor. 72

3.15 Performance vs load factor. 72
3.16 Algorithms selected. 73
3.17 Comparison of Different Implementations. 74
3.18 Average Speedup. 75
3.19 Mean-Standard Deviation. 76

4.1 List of Supported Formulas. 88
4.2 Successful Reverse Graph backward traversal for $A(\psi \cup \phi)$. 94
4.3 Unsuccessful Reverse Graph backward traversal for $A(\psi \cup \phi)$. 94
4.4 Successful Reverse Parental Graph backward traversal for $A(\psi \cup \phi)$. . . 97
4.5 Unsuccessful Reverse Parental Graph backward traversal for $A(\psi \cup \phi)$. . 99
4.6 Leadsto $a \rightsquigarrow b$ where a is ψ and b is ϕ. 99
4.7 Worst-Case Example for the RPG Version (edges in red are in the reverse parental graph). 107
4.8 Formulas and Models in our Benchmark. 120
4.9 PH with Reverse algorithm. 121
4.10 PH with Parental algorithm. 121
4.11 Exploration and cycle detection speedup analysis for PH model. 121
4.12 Peg with Reverse alg. 122
4.13 Peg with Parental alg. 122
4.14 Exploration and cycle detection speedup analysis for Peg model. 122
4.15 TK with Reverse algorithm. 123
4.16 TK with Parental algorithm. 123
4.17 Exploration and cycle detection speedup analysis for TK model. 123
4.18 TK_M with Reverse alg. 124
4.19 TK_M with Parental alg. 124
4.20 Exploration and cycle detection speedup analysis for TK_M model. . . 124
4.21 Simplified graph for Peg-Solitaire (13 tokens). 125
4.22 Simplified graph for TK_M (2 stations). 125
4.23 PH model. 126
4.24 SK model. 126
4.25 TK model. 126
4.26 TK_M model. 126

4.27 PEG model. 126

5.1 Probabilist Algorithms Comparison. 132
5.2 Illustration of the insertion operation on a Bloom Table. 134
5.3 Bloom Table and second storage coupling. 134
5.4 Insertion operation on a Bloom Table with multiple possible insertions. . 136
5.5 Bloom Table Analysis for different number of hash functions k. 138
5.6 Comparison between Bloom Table and Bloom Filter for different number of hash functions k. 139
5.7 Relation k and f. 140
5.8 Relation k and q. 140
5.9 Parallel probabilistic verification algorithm overview. 140
5.10 Bloom filter and Bloom Table Results. 142
5.11 Bloom filter and Bloom Table Results for Sokoban model. 144
5.12 Number of rejected elements for Sokoban model. 145
5.13 Bloom filter and Bloom Table Results for Solitaire model. 145
5.14 Number of rejected elements for Solitaire model. 146
5.15 Probability vs Number of Keys for Peg and Sokoban models. 146
5.16 Bloom filter vs Bloom Table Execution Profile Overview. 147

A.1 Benchmark Examples. 156
A.2 Parallel State Space Construction for the Sokoban and Peg-Solitaire models.157
A.3 Parallel State Space Construction for the Fifteen and FMS models. . . . 158
A.4 Parallel State Space Construction for the Frog and Hanoi models. 159
A.5 Parallel State Space Construction for the Kanban and PH models. . . . 160
A.6 Probability vs Execution Time. 161
A.7 Probability vs Rejected States. 162
A.8 Bloom Filter and Bloom Table Results for Fifteen model. 163
A.9 Number of rejected elements for Fifteen model. 163
A.10 Bloom Filter and Bloom Table Results for FMS model. 164
A.11 Number of rejected elements for FMS model. 164
A.12 Bloom Filter and Bloom Table Results for Frog model. 165
A.13 Number of rejected elements for Frog model. 165
A.14 Bloom Filter and Bloom Table Results for Hanoi model. 166

A.15 Number of rejected elements for Hanoi model. 166
A.16 Bloom Filter and Bloom Table Results for Kanban model. 167
A.17 Number of rejected elements for Kanban model. 167

B.1 Mercury Modules. 170
B.2 Memory Layouts. 171
B.3 Mercury Configurations (we use † to signal our new algorithms). 173
B.4 Algorithms selected for benchmark comparison. 175
B.5 Mercury usage syntax. 176
B.6 Mercury options for different versions. 176
B.7 Mercury options. 177

Glossary

BFS	Breadth-First Search Algorithm, 30
BT	Bloom Table, 46, 131
CTL	Computation Tree Logic, 13
DAG	Directed Acyclic Graphs, 85
DFS	Depth-First Search Algorithm, 30
KS	Kripke Structures, 12
LRL	Leadsto-Reachability-Liveness Logic, 83
LT	Localization Table, 43, 62
LTL	Linear Temporal Logic, 13
LTS	labelled transitions systems, 12
MCLCD	Model Checking With Lazy Cycle Detection, 44, 87
MIMD	Multiple Instructions Multiple Data, 18
Nested-DFS	Nested Depth-First Search Algorithm, 30
NOW	Network of Workstations, 21
PG	Parental Graph, 86, 89
RG	Reverse Graph, 89
SCC	Strongly Connected Components, 30
SIMD	Single Instruction Multiple Data, 18
SISD	Single Instruction Single Data, 18
SPMD	Single Program Multiple Data, 22

Glossary

Chapter 1

Introduction

"Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better."

Donald Knuth

In this thesis, we propose and study new algorithms and data structures for model checking finite-state, concurrent systems. We focus on techniques that target shared memory, multi-cores architectures, that are a current trend in computer architectures.

Model checking is a valuable formal verification method that can be used to avoid the presence of logical errors. In that respect, model checking contributes to improving the safety of embedded systems (and also to improve the level of trust that we can put in them). This is an important goal. Embedded systems are increasingly present in our everyday life and we cannot deny the major impact they have on our societies. Some of these embedded systems—such as those found in the aeronautic or nuclear domains for example—are classified as *critical*, meaning that a failure or a malfunction can result in the injury (or even the death) of the people involved, irreversible damage to equipment, or environmental catastrophes. We can list some outstanding examples of catastrophic failures that have attracted the attention of the public in their time (see (Neu92) for a list of incidents):

- Therac-25 (1985-1987): Between June 1985 and January 1987, a computer-controlled radiation therapy, the Therac-25, severely overdosed six patients due to a software

coding issue (Lev95).

- Ariane 5 (1996): On June 4, 1996, the inaugural launch of the European Ariane 5 rocket ended in a blast! This failure, caused by a malfunction in the control software, was essentially to blame on an internal software exception during execution of a data conversion from 64-bit floating point to 16-bit signed integer value (L^+96).

- NASA Mars Pathfinder (1997): On July 1997, the martian rover started losing information due to several system resets. The system was restarted due to a problem of priority inversion and resulted in delays in relaying data, shortening the duration of the mission (C^+98).

The market demands for more efficient and automated solutions has pushed the complexity of embedded systems to levels never imagined before. For instance, we now develop airplanes that fly longer, with less maintenance time, and using less fuel. The level of efficiency we experiment today is, without doubts, one of the chief achievements of the past decade. However, these achievements come with a price, since systems get more and more expensive to develop (and the probability of successfully completing a new technological project decreases). Although there is no official information about the productivity of embedded software engineers—nor a precise way of computing this metric— in some critical domains like avionics, software engineers are expected to produce no more than one line of code per day on average. Moreover, these figures do not take into account the heavy burden of tests and certification activities that such systems are subject to. The use of model checking can help improve this situation, especially since it is supposed to catch errors early during the design phase of a system, before they become very expensive to fix.

Since the pioneering works of Edmund M. Clarke and Allen Emerson, and of Joseph Sifakis and Jean-Pierre Queille, in the early 1980's, model checking has been successfully used in practice to verify applications such as complex sequential circuit designs (BCL^+94) and communication protocols (JH93). Model checking techniques are attractive because they offer an automatic solution to check whether a model of a system meets its requirements. For instance, it does not require hand constructed proofs like with approaches based on Floyd-Hoare style logics, that can be quite tedious and hard

to scale. Another reason for the growing interest in the use of model checking techniques is that they can be easily integrated into a standard development cycle; they not only help finding errors, but can also provide counter-examples (execution traces) when the system model violates any of its requirements. Later, with the seminal paper (ACD90) from Alur et al., model checking techniques have made their first steps into the analysis of real time systems. Altogether, the model checking approach has emerged as a prominent tool for the design and development of critical, real time systems.

Even though model checking offers a "push button" approach to check finite systems, the size of the state spaces built during verification may grow exponentially large as the complexity of the system increases. Hence, in many cases, model checking may be infeasible in practice. This drawback, known as the *state explosion* problem in the formal methods community, is one of the main challenge faced by model checking. Despite the fact that considerable progress has been made at the theoretical level—for instance with the definition of symbolic model checking and partial orders techniques—there are still classes of systems that cannot benefit from these advanced methods. For example, for models that combine real time constraints, dynamic priorities and data variables. In these cases, we still need to go back to using explicit-state, enumerative model checking techniques.

The main motivation of this PhD thesis is to develop new algorithms and data structures to take advantage of the improvements recently made at the hardware level; namely the advent of affordable, multiprocessor, shared memory servers. Basically, we attack the state explosion problem through the use of brute force! (But not without cunning.) Since the mid 2000's, the major chip-makers have acknowledged a possible end to Moore's law (the Moore's wall), that shaped the evolution of the Personal Computer's architecture based on an increase of the processors frequency. This is a rationale for shifting their attention to multi-core processor architectures, bringing parallel computing technologies to even the simplest of computers: Even netbooks and smartphones have dual core processor nowadays. Furthermore, with the popularization of server-based computing and virtualization technologies (servers hosting multiple virtual machines), we now have access to affordable multiprocessors machines—that is with many multi-core processors—that provides the opportunity to access very large amount of shared (primary storage) memory.

Alongside the definition of enumerative, model checking algorithms for many-cores architectures, we also study *probabilistic verification algorithms*. By the term probabilistic, we mean that, during the exploration of a system, any given reachable state has a high probability of being checked by the algorithm. As a consequence, we accept that some reachable state of the system may not be inspected: While this technique cannot be used to prove the absence of errors, it can be very efficient when we try to find counter-examples. Probabilistic verification trades savings at the level of memory usage for the probability of missing some states. Basically, the idea is to use hash values instead of an accurate representation of a state value. Consequently, it becomes possible to analyze part of the state space of a system when there is not enough memory available to represent the entire state space in an exact manner.

Brief Description of the Thesis Work

The contributions of this thesis can be divided into three main axes: (1) parallel construction of the state space; (2) parallel model checking algorithms; and (3) probabilistic verification methods.

Chronologically, we started our work by studying new algorithms and data structures to construct the state space in parallel. The key points to design an efficient parallel algorithm for shared memory machines are the data structure used to store the set of explored states and the work-load strategy employed to distribute data. We propose two novel approaches based on an optimized data structure: we use independent (distributed) data dictionaries in conjunction with a shared, probabilistic data structure to dynamically distribute the state space.

Our first contribution for parallel state space construction is a speculative algorithm (SZB10) where the states are stored in local data sets, while a shared *Bloom Filter* (Blo70) is used to dynamically distribute the states. Due to the probabilistic character of the Bloom Filter (false positives are possible), we propose a multiphase algorithm to perform exhaustive, deterministic, state space generation. Next, we improve on our previous design and replace the Bloom Filter by a dedicated data structure, the *Localization Table* (TSDZB11). This table is used to dynamically assign newly discovered states and behaves as an associative array that returns the identity of the processor that owns a given state. With this approach, we are able to consolidate a

network of local hash tables into an (abstract) distributed one, removing the need for a multiphased algorithm. Our preliminary results are very promising, we observe performances close to those obtained using an algorithm based on lock-less hash tables (that may be unsafe) and performs well when compared to some classical parallel algorithms.

Another contribution of this thesis consists in the definition of a new data structure for probabilistic, formal verification. After our experiments with parallel state space construction, we studied an enriched Bloom Filter specifically designed for probabilistic verification. We propose a new probabilistic data structure, named Bloom Table, that fulfills a gap we identified between the use of the *hash compact* and Bloom filter data structures. Our Bloom Table not only delivers a small probability of false positive but also improves the time complexity by reducing the number of necessary hash functions. For instance, only two hash keys are needed to deliver a probability of 10^{-5} using only 16 bits per state. (A Bloom filter requires the use of 6 hash keys to achieve similar results.) While Bloom Tables are not inherently a concurrent data structure, we devised our algorithms in order to take advantage of parallel, shared-memory architectures.

Finally, we present two new algorithms for parallel model checking that supports a specific subset of Computation Tree Logic (CTL). In this context, the strategy used to detect cycles is one of the key points of performance for the evaluation of more complex formulas. Our main objective is to propose algorithms that are as efficient as possible, but also "friendly" with respect to the work-sharing policies that are used for the state space generation (e.g. a work-stealing strategy): at no point do we impose a restriction on the way work is shared among the processors. This includes both the construction of the state space as the detection of cycles in parallel. We contribute with a "practical" approach where cycles are detected only at the last stage. We circumvent all the complexities imposed by the detection of cycles in parallel by not doing it explicitly. We present two flavors of our algorithm, one that has linear time complexity and another one that trades a lower memory usage for a bigger time complexity. The difference between these implementations resides in the graph structure stored during the state space exploration. We consider two cases: (1) the case where we have access to the complete state space graph (actually the reverse graph), that is we store all the transitions in memory; and (2) we only have access to one "parent" for each state. While it is common to find algorithms for CTL model checking that do not require the reverse transition relation to be stored in memory (transitions are regenerated when necessary),

this approach is not appropriate when it is not possible (or very expensive) to compute the reverse transition relation of a model. This is, for example, the case when we deal with models combining real-time constraints or data variables.

Before we conclude this section, we would like to emphasize that the contributions of this thesis are not limited to the model checking domain. De facto, the data structures proposed in this work are of interest for any application that performs graph exploration, cycle detection and probabilistic (or lossy) storage in parallel.

Outline

The previous section gives a brief chronological presentation of our work. We review more precisely the contributions of our work in Section 2.5, after we describe the related work. For the remainder of this thesis, we decided to change the presentation order to improve the quality of the manuscript. The section about parallel model checking is placed before probabilistic verification in order to follow a more natural presentation.

We give a brief summary of the contents of the chapters of this document.

Chapter 2 Related Work: In this chapter we present the context of this thesis. We briefly present model checking—in a very general way—and then delves into previous works more relevant for the context of this thesis. We start with related work for parallel and distributed model checking. We try to stress out, in this section, the importance to optimize cycles detection in order to obtain an effective parallel model checking solution. Then, we deal with the most significant works for probabilistic verification, notably the ones based on the probabilistic structures *supertrace*, *multihash* and *hash compact*. We conclude this chapter with a detailed presentation of the contributions made in this thesis.

Chapter 3 Parallel State Space Construction: This chapter describes our approaches for parallel state space construction. We concentrate our efforts on an approach that is scalable, without imposing any restrictions on the way work is distributed among the processing units. In this chapter, we state the main guidelines of our algorithms such as the distribution of memory and work-sharing techniques. This section is followed by the presentation of our (speculative) algorithm, which is a general lock free algorithm for parallel state space construction. We try to stress

the general nature of our approach by giving the results of experiments performed using different data structures (such as AVL trees and hash tables) for the local dictionaries. Then, we give an enhanced version that replaces the *Bloom Filter* with a specialized data structure called a *Localization Table*. This solution improves over our previous version because the small shared memory space not only distribute data but also keeps track of the distribution. Before concluding, we give a comparative study of our enhanced version with other solutions already proposed in the literature.

Chapter 4 Parallel Model Checking With Lazy Cycle Detection: Based on the work we presented at Chapter 3, we define and analyze two new algorithms for parallel model checking that supports a sub-set of CTL formulas. Our main objective is to propose efficient algorithms that do not prevent the use of particular work-sharing policies, that is to say, do not impose any restriction on the way work is shared among the processors during the state space construction and the "property verification" phases. In addition, we also focus our efforts in providing new approaches that requires less memory spaces. To conclude this chapter, we study a set of experiments results obtained with our prototype implementation.

Chapter 5 Probabilistic Verification: The main objective of this thesis is to propose new methods to deal with the state explosion problem. While we base most of our work on an enumerative, explicit-state approach, we realized during the thesis that our data structures could be adapted in order to fit the context of probabilistic verification algorithms. In particular, we identified the existence of a gap between two of the most successful data structures for probabilistic verification, *Bloom Filter* and *hash compact*. Roughly speaking, the probabilistic data structure we present in this chapter delivers a better result than the *hash compact* algorithm when we have less than 40 bits of information per state to represent the state space. On the other hand, our solution improves the execution time when compared to a classical approach based on *Bloom Filters* because it offers a better result without increasing the number of hash functions used. (In fact, our proposition requires only 16 bits per state for an effective state space coverage and is beaten by the *Bloom Filter* only when there are less than 6 hash keys available per state.) We present a theoretical analysis of our data structure together with

an analytical comparison with related solutions. Before concluding, we present empirical results obtained from a large set of experiments in order to demonstrate the effectiveness of our approach.

Chapter 6 Conclusion: We conclude the thesis by defining possible lines for future work.

Chapter 2

Model Checking — Related Work

" An expert is a man who has made all the mistakes which can be made in a very narrow field."

Niels Bohr

In this chapter, we start by presenting model checking briefly, in a very general way, and then we concentrate more deeply on the particular works that are the most relevant for the context of this thesis.

This chapter is organized as follows. Section 2.1 introduces this chapter. We give a general presentation of model checking and temporal logic in Section 2.2. Next, we present in section 2.3 the related work for parallel and distributed model checking. We decided to present both approaches for model checking because—even if unintentionally—the algorithms proposed in the distributed cases influenced the approaches followed for the parallel case. Moreover, we try to stress out in this section the importance of efficient parallel algorithms to detect cycles in order to obtain an effective parallel model checking solution.

In Section 2.4, we study the related work for probabilistic verification, most notably the works based on the use of probabilistic structures such as the *Bloom Filter* and the *Compact Hash Table*.

We conclude this chapter in Section 2.5, where we list the contributions of this thesis.

2.1 Introduction

Model Checking (CE82, QS82) has emerged as a promising automated approach to check whether a system model meets its requirements. Roughly speaking, it has the same impact as performing an exhaustive test, using symbolic evaluations, where every possible scenario is checked for correctness. It may be generically characterized as an inference procedure that decides if a structure M satisfies a given specification (logical formula) ϕ, abbreviated as $M \vDash \phi$.

Model checking techniques are attracting, more and more, the attention of the industries; mainly because they offer a "push button" solution for the verification of finite systems. Unlike with Floyd-Hoare(Cla08) style logic, the model checking approach does not require hand constructed proofs, that can be quite tedious and hard to scale. In addition, model checking uses the expressiveness of temporal logic's to express complex concurrency properties in an elegant and simple fashion. Another positive aspect of model checking is the fact that a counter example is provided when the specification is not satisfied.

Since it was first proposed by Edmund M. Clarke and Allen Emerson in (CE82) and Joseph Sifakis and Jean-Pierre Queille in (QS82), model checking have been successfully used in practice to verify applications such as complex sequential circuit designs and communication protocols. Although model checking offers a "push button" approach to verify finite system, there is still a large gap between possible (or decidable) and feasible. Indeed, there are many cases in which it is not possible to perform the verification of a finite system due to the *state explosion* problem. That is, the number of states that should be inspected can grow exponentially larger in function of the complexity of the system. So large indeed that is goes beyond the available computing resources. The state explosion problem is one of the main challenges faced by model checking researchers.

Despite the fact that considerable progress have been made—such as symbolic model checking or partial-order techniques—there are still classes of systems that cannot benefit from these progress on the algorithmical side. For these systems, a classical—exhaustive state—enumerative approach is still the most appropriate.

In this thesis, the idea is to take benefit of recent advances on the hardware side to improve enumerative model checking techniques. Indeed, we now have access to

computers with larger memory space and to multi-core architectures that makes feasible the verification of larger models, in a reasonable amount of time.

2.2 What is Model Checking

One of the reason for the adoption of model checking techniques is that it can be easily integrated into the development cycle used when developing complex systems. It not only helps in the task of finding errors early (during the conception), but it also provides counter-examples to explain these errors.

Figure 2.1 depicts a simplified overview of the testing cycle using Model Checking techniques. The approach itself can be divided into three phases: (1) the model description is explored until saturation and all possible states are enumerated and stored in a graph structure; (2) the desired property is checked for correctness over the graph structure; (3) if the property is not verified, a counterexample may be provided to reproduce a path leading to the given error. This testing cycle repeats until all properties are verified by the model.

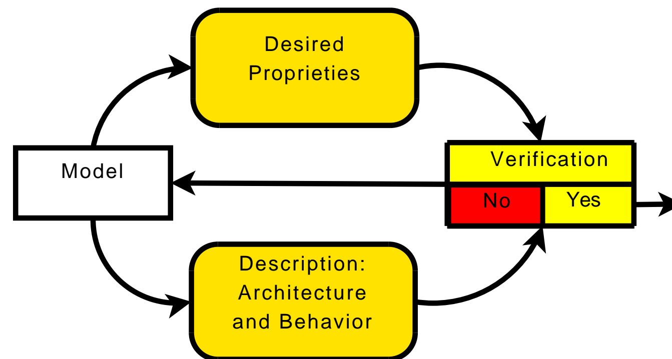


Figure 2.1: Model Checking development cycle.

2.2.1 State Graph and Kripke Structure

The structure commonly used to represent the set of enumerated states are *Kripke structures* (KS) or *labelled transitions systems* (LTS). They differ basically by the information annotated over the states: in a KS, states are annotated with so-called *atomic propositions* while in LTS states and/or transitions are annotated with *actions*. In the reminder of this work, we use only the *kripke structures* representation.

Kripke Structures: A *Kripke structure* (*KS*) is a triple $KS(S, R, s_0)$, where S is a (finite) set of *states*, R is a *transition relation* over S, and $s_0 \in S$ is the initial state. A Kripke Structure is also generally associated to an *interpretation function*, $I : S \to 2^{AP}$, that maps every state $s \in S$ to a subset of atomic properties. The size of KS is defined as $|S| + |R|$ where, $|S|$ denotes the cardinality of the set S.

Some definitions of Kripke Structure consider a set of possible initial states, and not only one single state, but this does not change the expressivity of the model. Also, some definitions require that the transition relation R be *left-total*, meaning that every state in the KS has a successor: for all $s \in R$ there is $s' \in R$ such that $s\ R\ s'$. We should also choose this convention in this thesis, adding a "loop" transition $s\ R\ s'$ if needed. In this case, we define a *deadlock* as a state s whose sole successor is itself and that satisfies the atomic property `dead` (that is $I(s) = \texttt{dead}$).

When we ignore the atomic properties, the Kripke Structure is basically a directed graph, also often called the *state graph* of the system.

The model checking problem may be formalized as follows (Cla99), where the definition of the entailment relation $\models$ depends on the choice of the temporal logic. We discuss temporal logic in the next section.

Model Checking: Given a Kripke structure $KS(S, R, s_0)$ corresponding to a finite-state system, an interpretation function $I : S \to 2^{AP}$, and a temporal logic formula F over the atomic properties in AP, find the set of all states $s \in S$ such that the relation $K, s \models F$ holds. We often simply say that F holds in s.

2.2.2 Temporal Logic Formula

Temporal logic formula (Cla99) is a powerful tool to express the behavior of reactive systems. A temporal logic is an example of modal logic, that is a formal framework able to qualify the truth of a judgment. In a modal logic, the truth of a judgment depends on where, or when, the judgment is held.

A temporal logic expresses properties over the possible (future, current or past) transitions in a reactive system. It has proved to be useful to express the behavior of this class of systems because they define non explicit time operators, such as *eventually* and *never*, which can describe the ordering of events in time without introducing time explicitly.

Temporal logics come in different flavors that differ, basically, by the modalities (operators) used in the logic and the semantics of these modalities.

We define two of the most common temporal logics: Computation Tree Logic (CTL) and Linear Temporal Logic (LTL). They differ in how they handle branching in the underlying Kripke structure. Linear-time logic expresses events along a single computation path. In contrast, branching-time logic takes into account all possible paths from a given state.

The choice for linear or branching-time logics depends on the kind of specifications and systems that need to be analyzed, both have their advantages and their field of application; branching-time logics are better for reactive systems and linear-time logics when only path properties are of interest.

In our work on a model checking algorithm, we propose a new parallel algorithm for a subset of CTL formulas that can also be defined using LTL (see Chapter 4 for more details).

Linear Temporal Logic (LTL)

Linear Temporal Logic (Cla99) is composed of five basic temporal operators used to describe properties of a path through the tree; the X operator means "next time" and requires that the property holds in the second state of the path; the F (G) means "eventually" ("always") and requires that the property holds at some (at every) state on the path; the U means "until" and combines two properties, it holds if there is a state on the path where the second property is true and the first property holds for every preceding state on the path; the R means "release" and it is the logical dual of the U operator. Let p be a state property over the set AP of atomic propositions, LTL formulas are constructed as follows:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid X(\phi) \mid U(\phi,\psi) \mid R(\phi,\psi) \mid F(\phi) \mid G(\phi)$$

Let $M(S, R, I)$ be a Kripke structure over AP; a finite path is a non-empty sequence $\pi = \langle s_0, s_1, ..., s_{n-1}\rangle$ of states $s_0, s_1, ..., s_{n-1} \in S$ such that $(s_i, s_{i+1}) \in R$ for all $0 \leq i < n-1$; n is the length of path π, denoted $|\pi|$; an infinite path is an infinite sequence $\pi = \langle s_0, s_1, ...\rangle$ of states in S such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. We denote π_i for the i-th state in path π and π^i as the tail of the path starting at π_i ($\langle \pi_i, \pi_{i+1}, ...\rangle$). We call

a path in M maximal if it can not be extended. The semantic of LTL is presented in Figure 2.2.

$$
\begin{array}{rcl}
\pi \models p & \text{iff} & p \in I(\pi_0) \\
\pi \models \neg\phi & \text{iff} & \pi \not\models \phi \\
\pi \models \phi_1 \vee \phi_2 & \text{iff} & \pi \models \phi_1 \text{ or } \pi \models \phi_2 \\
\pi \models X(\phi) & \text{iff} & |\pi| > 1 \text{ and } \pi^1 \models \phi \\
\pi \models U(\phi,\psi) & \text{iff} & \text{there is a } k,\ 0 \leq k < |\pi|, \text{ with} \\
 & & \pi^k \models \psi \text{ and for all } i,\ 0 \leq i < k,\ \pi^i \models \phi
\end{array}
$$

Figure 2.2: Semantics of LTL.

The rest of temporal operators are in fact abbreviations, they are reduced from the U operator. The operator F is the abbreviation of U when the first property is always true ($F(\phi) = U(true, \phi)$. The operator G can be obtained from the negation of F operator ($G(\phi) = \neg F(\neg\phi)$) because it is sufficient to prove that a property always holds as long as its complement never happen. Finally, the R operator, which is the dual of U, is reduced to $R(\phi,\psi) = \neg U(\neg\phi, \neg\psi)$.

Computation Tree Logic (CTL)

Computation Tree Logic (CTL)(BPM83, CE82, EC80) differs from LTL because it supports path quantifiers. CTL formulas have the form QL where Q stands for one of the path quantifiers A or E, and L for the linear-time operators, which are the same supported by LTL. These path quantifiers provide universal (A) or existential E) quantification over the paths from a given state. Note that LTL formulas consist of Af formulas where f is a path formula which holds only state subformulas made of atomic properties. CTL formulas are constructed as follows:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid X(\phi) \mid AU(\phi,\psi) \mid EU(\phi,\psi) \mid AF(\phi) \mid EF(\phi) \mid AG(\phi) \mid EG(\phi)$$

Let M be a Kripke structure, a path is an infinite sequence $(s_0, s_1, ...)$ such that $\forall i[(s_i, s_{i+1}) \in R]$ where $s_i \in S$. The notation $M, s_0 \models f$ means that formula f holds at state s_0 in structure M; we simply write $s_0 \models f$ when the structure M is understood. The semantic of CTL is presented in Figure 2.3.

$$
\begin{array}{rcl}
s_0 \models p & \text{iff} & s_0 \in L(p) \\
s_0 \models \neg f_1 & \text{iff} & \neg(s_0 \models f_1) \\
s_0 \models \phi_1 \vee \phi_2 & \text{iff} & s_0 \models \phi_1 \text{ or } s_0 \models \phi_2 \\
s_0 \models EX(\phi) & \text{iff} & s_1 \models \phi \text{ for some state } s_1 \text{ such that } (s_0, s_1) \in R \\
s_0 \models AX(\phi) & \text{iff} & s_1 \models \phi \text{ for all state } s_1 \text{ such that } (s_0, s_1) \in R \\
s_0 \models EU(\phi, \psi) & \text{iff} & \text{for some path } s_0, s_1, ... \\
 & & \exists i[i \geq 0,\ s_i \models \psi \wedge \forall j[0 \leq j < i \Rightarrow s_j \models \phi]] \\
s_0 \models AU(\phi, \psi) & \text{iff} & \text{for all path } s_0, s_1, ... \\
 & & \exists i[i \geq 0,\ s_i \models \psi \wedge \forall j[0 \leq j < i \Rightarrow s_j \models \phi]]
\end{array}
$$

Figure 2.3: Semantics of CTL.

Just like LTL, the F modalities can be expressed using the U operator, $AF(\phi) = AU(true, \phi)$ and $EF(\phi) = EU(true, \phi)$. The G modalities can be defined as the duals of the F modalities, $AG(\phi) = \neg EF(\neg\phi)$ and $EG(\phi) = \neg AG(\neg\phi)$.

It is important to mention that CTL is a subset of a more expressive logic called CTL^* (CES86), indeed CTL differs from CTL^* because each temporal operator X, F, G, U, and R must be immediately preceded by a path quantifier. Moreover, CTL is a particularly simple fragment of the modal μ-calculus(EJS93), CTL modalities can be expressed by means of fixpoint formulas.

2.2.3 Model Checking

Global and Local

The model checking problem can be specified in two ways: local and global. The local model checking problem determines whether a single state satisfies a given property over a given structure (*Kripke Structures* or *labelled transitions systems*). By contrast, the global model checking problem determines if a property is satisfied by all states.

The local model checking is usually preferred for the formal verification of software and hardware systems because the property of interest is often expressed with respect to a specific initial state. In addition, these applications suffer from the well known *state-explosion* problem when the number of states grows exponentially with the number of components (or variables for software). Hence, the use of local model checking helps to address this problem because it might not need to explore the complete structure in order to decide whether the property holds for the initial state.

For other applications which the *state-explosion* problem is not significant and the property of interest is not related to a particular state, the global model checking is more interesting because it gives knowledge about all the states of a structure.

Global model checking: Given a finite model structure, M, and a formula ϕ, determine the set of states in M that satisfies ϕ.

Local model checking: Given a finite model structure, M, a formula ϕ, and a state s in M, determine whether s satisfies ϕ.

Automata-theoretic approach and Semantic Approach

The model checking problem can be solved by different techniques depending on the structure (model) and the logics considered. These techniques differ by the manner they inspect the structure; *semantic* approaches iteratively compute the semantics of the formula over the structure in a inductive manner; *automata-theoretic* approaches reduce the model checking problem into the inclusion problem between automatons.

The semantic approach (Cla99, MSS99) was the first algorithm proposed for model checking (Clarke et al. in (CE82)) in general. In its simplest form, the algorithm determines the set of states S that satisfy a given (CTL) specification f by labeling each state s with the set *label*(s) of subformulas of s which are true in s. The algorithm iterates through a series of stages until the complete formula had been operated; the algorithm presented in (CES86) has time complexity linear in the length of the formula ($|f|$) and the size of the state transition graph ($O(|f| \cdot (|S| + |R|))$). The semantic approach can also be accomplished through the iterative characterization of fixpoints, it allows the evaluation of modal μ-calculus formulas (EJS93). However, the complete evaluation of μ-calculus formulas has an exponential worst-case time due to the alternated nesting of least and greatest fixpoints.

The automata-theoretic approach (Cla99, MSS99) reduces the model checking problem to the automata language acceptance. Let A and B be the system and the specification automatons, roughly speaking, the model checking problem is reduced to check whether the system A satisfies the specification when $L(A) \subseteq L(S)$. Wolper et al., in(Wol86), was the first to propose this approach for linear temporal logic formulas, they successfully bridged the gap between system models and logic formulas by reducing the model checking to the emptiness problem, which is the problem of checking

that none of the sequences accepted by the negation of the specification automaton is accepted by the system automaton. (Let $L(\bar{S})$ be the complement of $L(S)$, the language acceptance can be rewritten as $L(A) \cap L(\bar{B}) = \emptyset$.) Efficient algorithms for LTL model checking are known, one uses the Tarjan's depth first search algorithm (Tar71) for finding strongly connected components for deciding emptiness check in linear time complexity ($O(|S|+|R|)$). There is an alternative algorithm (CVWY92) that performs a nested depth first search whenever an accepting state is found. Both algorithms can be performed on-the-fly, returning errors faster and without exploring the complete state space. However, the latter is more appreciated by its smaller memory footprint when performed on-the-fly.

Probabilistic and Exhaustive

Exhaustive model checking is preferable in comparison with probabilistic because it is the only approach capable of asserting the error free character of a system. Probabilistic verification can only assert that a given error exists, but never the opposite. However, computer resources (i.e. memory) are at price, and sometimes there are not enough resources for an exhaustive exploration of the system state space. In such cases, being able to explore only a portion is still interesting. This approach was first proposed by Holzmann (Hol93) in the context of communication protocols and rapidly spread in academia and industry.

2.3 Parallel Model Checking

The main motivation of this thesis it to perform explicit parallel model checking on multiprocessor machines. In this section, we present all the relevant related work for parallel model checking. Even though we are interested in shared memory architecture, we also decided to present the literature for distributed memory machines because they can be easily implemented into shared memory machines, indeed some of the available solutions we have nowadays were first developed targeting distributed memory machines.

This section is divided as follows. We start with a brief presentation of parallel computers at Section 2.3.1. Next, we give a chronological review of the literature for parallel model checking in Section 2.3.2. Our motivation with this chronological study is to show that model checking solutions have being driven by the hardware market

industry. Section 2.3.3 presents the difficulties and the proposed solutions for the state space construction in parallel, usually the first step for model checking. Finally, solutions for parallel model checking are presented according to the kind of properties they support, Section 2.3.4 presents the related solutions for LTL and Section 2.3.5 for CTL parallel Model Checking.

2.3.1 Parallel Computers

A parallel computer can be characterized as a collection of processing elements that communicate and cooperate to solve large problems. It is expected to reach faster speed and also to be more cost-effective than a high performance single processor.

Parallel computers can be characterized by the way information stream on the system. One of the most popular classification scheme is the Flynn's Taxonomy of computer architecture. (See (El-04) for a complete presentation of parallel computer architecture.) The Flynn's Taxonomy categorizes a parallel computer into four classes based on this notion of information stream — instruction and data. The operations performed by a processor are defined as part of the instruction stream. The data stream is the data exchanged between the processor and the memory.

SISD (Single Instruction Single Data) : a single processor executes a single instruction stream over a single data. Parallelism may be recovered in this model using pipe-lining instructions.

SIMD (Single Instruction Multiple Data) : in this model, all processors execute the same program in lockstep, such that at each "time unit" all active processors are executing the same instruction with different data. Parallelism is exploited by applying simultaneous operations across large sets of data.

MIMD (Multiple Instructions Multiple Data) : this is the less-constrained model. Each processor may execute an instruction or operate on data different from those executed or operated by any other processor during any given time unit. A MIMD architecture can be defined as a computer system consisting of multiple processing units; connected via some interconnection network; with an additional software layer to make processing units interoperate. There are two main types of interconnection networks:

- shared memory: coordination among processes is accomplished through global memory shared among all processes. If all processes access the memory in the same way, it is considered an UMA (Uniform Memory Access) architecture. In the case where each processor has part of the memory attached and the access time depends on the distance to the processor owner, it is classified as NUMA (Non-Uniform Memory Access). Finally, a COMA (Cache-Only Memory Architecture), is a computer architecture such that local memories at each node is only used as cache.

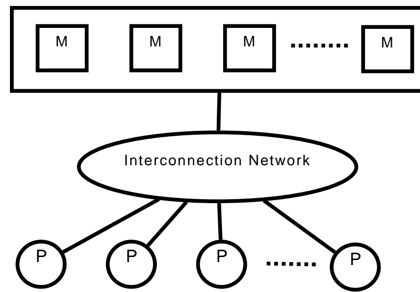


Figure 2.4: Shared memory.

- message passing: in this model, there is no shared memory space among the processors and communication is provided through exchange of messages instead of memory access.

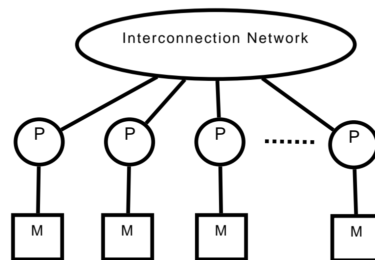


Figure 2.5: Distributed memory.

In the context of this thesis, we deal with MIMD shared memory computer systems.

Measuring the Performance

The measure of performance employed in this work is the *equal duration model* (El-04). We consider this model more appropriate for the parallel programming style we adopted, which is the Single Program Multiple Data (SPMD). We explain the choice for this model in details at Section 2.5.

The equal duration model assumes that a given task can be divided into n equal subtasks and there are n available processors to execute each one of them. Let t_s be the time needed by a single processor to execute the complete task, we can assume that $t_n = \frac{t_s}{n}$ is the time taken for each one of the n processors to execute the subtasks. Since this model is based on the assumption that all processors perform simultaneously, then the time taken for all n processors to execute all n subtasks is $t_N = \frac{t_s}{n}$. We can define the speedup factor as the ratio between the sequential time (t_s) divided by the time taken by all n processors t_N, as follows:

$$
\begin{aligned}
S_p &= \text{speedup factor} \\
S_p &= \frac{t_s}{t_N} = \frac{t_s}{t_s/n} \\
S_p &= n \text{ (linear speedup)}
\end{aligned}
$$

Notice that this model does not take into account all the implicit overheads from the parallel architecture such as communication, synchronizations, etc. A more elaborated model can be achieved if we consider an extra time t_c incurred by all these overheads for the total execution time t_m.

$$
\begin{aligned}
S_p &= \text{speedup factor with overhead} \\
S_p &= \frac{t_s}{t_N} = \frac{t_s}{\frac{t_s}{n} + t_c} = \frac{n}{1 + n \cdot \frac{t_c}{t_s}} \\
\xi &= \frac{1}{1 + n \cdot \frac{t_c}{t_s}} \text{ (efficiency)}
\end{aligned}
$$

The equation shows that the overhead incurred by the parallel architecture affects significantly the speedup factor. For instance, if $t_c \gg t_s$ then the potential speedup factor is $t_s/t_c \ll 1$ and depends on the time lost by the incurred overheads. For the rest of this thesis, we consider that t_c is not significant ($t_c \ll t_s$) and the potential speedup factor depends on the time taken for all n processors to execute the task (t_N). Furthermore, we define another measure of performance called *efficiency* (ξ).

The efficiency measure is the normalization of the speedup factor and is obtained by dividing the speedup factor by n. (The efficiency measure is equal to 1 if the overhead time (t_c) is ignored.)

Finally, we consider two kinds of speedup factors: *relative* and *absolute*.They differ basically by the implementation used to obtain the sequential time t_s. The relative speedup considers the same parallel implementation but using only one processor; The absolute speedup takes into account the most optimized sequential implementation. See (El-04) for a complete presentation of performance analysis for multiprocessor computer architecture.

2.3.2 Chronology

The advent of new technologies, or when they become affordable, enables the development of new solutions. It is not surprising that Model Checking development has been following the hardware improvements for decades. Since its early stage of development at 1980s, the hardware improvements on the domain of the processor's clock frequency have partially guided the community to focus on sequential algorithms, searching for a time and space efficient solution. At that time, parallel machines were suitable only for large laboratories due to their high costs and, by consequence, they were not available to everyone.

Later at 1990s, with the popularization of technologies like the Network of Workstations (NOW), Model Checking made its first steps into parallel computing motivated by the large (distributed) memory spaces available. This technology became popular because it allowed the affordable construction of cluster by using commodity computers. At the beginning, several solutions were proposed to only tackle the analysis of safety properties due to the inherent difficulties in redeploying the efficient sequential algorithms. The first attempts to perform more elaborated properties had already understood that efficiency solutions for parallel Model Checking would require specific algorithms and not the reimplementation of the sequential ones.

In mid 2004, a shift in the hardware industry had brought the discussion of efficient parallel algorithms back, the major chipmakers announced that they were shifting their attention from the processors frequency to the multi-core CPU technology. Although the number of embedded transistors are still following the Moore‘s prediction law (M^+98), it is no longer interesting to continue increasing the processors frequency

due to physical limitations, the energy necessary to cool the processors becomes commercially prohibitive. Hence, larger spaces of shared memory would require more time to be covered because the processors clock-speed will no longer follow the predicted Moore‘s curve. Since then, new algorithms have been specifically proposed for multi-core and multi-processors machines but, even if some of these algorithms scale well, they still do not match the levels of efficiency achieved by the sequential ones.

2.3.3 Parallel State Space Construction

Parallel and Distributed state space construction have been studied in various contexts and different solutions have been proposed. A majority of these solutions adopt a common approach: they can be considered as an “homogeneous” parallelism and they follow a SPMD (Single Program Multiple Data) programming style. The SPMD approach is commonly used to accomplish coarse-grained parallelism and it requires an explicit data and work assignation by the programmer to each processor. Another characteristic about SPMD programs is that the work executed by the threads is typically “homogeneous” in the sense that all threads perform concurrently the same steps.

A common approach to assign work and data is to partition the state space into several chunks, one for each processor available, through a slicing function. This solution is more common on distributed memory environments; they all follow almost the same architecture but differ by the nature of the slicing function, i. e. static or dynamic. In contrast, solutions based on shared memory architectures are more concerned with synchronizations among processors, memory consistency and overheads caused by the extended use of locking systems (mutual exclusion for memory access).

Irregular Problem

The parallel community classifies the parallel state space construction as an irregular problem because of the irregularity of its structure, i.e., the cost to operate this kind of structure is not exactly known or is unknown in advance. As a consequence, the parallel execution of such problems may result in a bad load balancing (GRV95, EL08).

In (EL08), the authors explain that the characteristics of the model under consideration have a key influence on the performance of a parallel algorithm because it may result in extra overhead during the exploration task. Figure 2.6 shows two examples of models that result in different load balancing. Figure a) shows a model where the

parallel exploration will perform like a sequential one, incapable of speedups. Figure b) is the ideal model where every node has more than one successor and, by consequence, enough work will be available to use all processors.

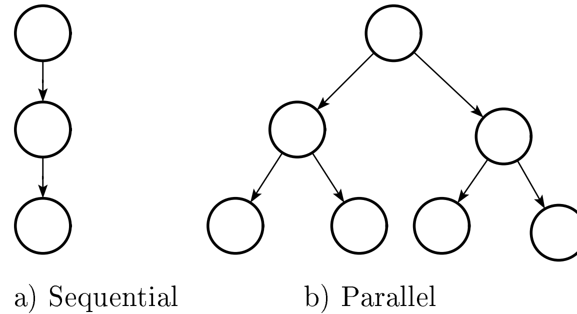


Figure 2.6: State space models.

Sequential State Space Construction

Before looking at the parallel state space construction algorithms, we present the classical sequential algorithm for state space exploration.

The Sequential State Space Construction is the simple and well-known reachability graph algorithm. It starts from the initial state (s_0) by exploring until saturation (*forall* s' *successors of* s) all possible successor states. Every new state found (*if* $s' \notin S$) is stored in the state graph ($S = S \cup s'$). The sequential algorithm is shown in Listing 2.1.

```
1   function VOID reachability(s0 : state, KS : Kripke Structure)
2
3    Set S ← new Set(s0) ;
4    Stack W ← new Stack(s0) ;
5    while (W is not empty) do
6      s ← W.pop();
7      forall s' successor of s in KS do
8         if (s' ∉ S) then
9            // s' is a new state
10           S ← S ∪ {s'} ;
11           W.push(s')
12        endif
13     endfor ;
14   endwhile
```

Listing 2.1: Sequential State Space Construction algorithm.

Distributed Memory

Several of the mechanisms proposed for distributed architectures rely on slicing functions, i.e. $Proc : S \Rightarrow \{0, ..., N-1\}$ where $Proc(s)$ is the owner of state s among N processors. They differ basically by the nature of this function in order to provide both locality and balance. Balance can be measured as spatial or temporal balance: spatial balance means that each processor will receive an equal amount of states; temporal balance means that each processor will be busy most of the time. Locality measures the fact that states which are "related" during the computation should be assigned to nearby processes (typically, the successors of a state should be handled by the same processor). Locality is desired to reduce communication overheads.

We can classify these slicing functions by the manner they are computed. We call user-provided definition when the partition function relies on the user expertise about the system. In (CGN98), Ciardo et al. proposed a partition function based on the stochastic Petri Nets structure in order to achieve both locality and balance, they take into account just a small set of places called control set (P). The idea is that any transition firing that does not involve a place in P corresponds to a state transition between two markings assigned to the same processor, otherwise this transition will correspond to a cross-arc and the new marking will be assigned to another processor. The disadvantage of this approach is that there is no automatic way to suggest this control set, therefore the technique relies on the user intuition to select the set of places that is part of the control set. Similar approaches were presented in (LS99, RBC^{+}06, CCM01).

There is another family of solutions based on dynamic functions (ADK97, LV01, KM05) that tries to deliver homogeneous spatial distribution without the burden of previous knowledge about the system. In general, dynamic load balancing techniques recompute a new partition function whenever the memory utilization of one processor differs more than a fixed percentage. The loading balance phase estimates how much memory each processor has to give/receive to its neighbors. In the end, a new partition function is created and broadcasted to all processors. In contrast, there are some solutions that rely on previous analysis for extracting relevant information to slice the state graph. In (OPE05), the authors try to minimize the cross-arcs by computing a small approximation of the state space. From this prediction, it is possible to extract

the shape of the original system and the relations among the states. A more elaborated mechanism is presented by Nicol et al. (NC97) where they explore heuristic methods to control the distribution of data after an initial random walk.

Finally, there is a group (HKTL07, GMS01, SD97, Pet03, HBB99, BHV00) that does not take into account any structural information from the model and are based only in a mathematical function to partition the graph. These solutions divide the graph through a hash function that takes into account only the spatial balance. As a consequence, it does not handle cross-arcs and may suffers from communication overhead. Even though the standard deviation — measure used to analyze the quality of the physical distribution of states — is smaller than 1% of the mean value, they are highly influenced by the employed static function and the set of chosen examples. Nevertheless, static slicing function is the most used solution, it can be confirmed by the number of available tools (DiVine (BBCR10), Murphi (SD97), CADP (GMS01) and UPPAAL (BHV00)) that employ this technique to distribute the state space.

The anatomy of a basic distributed algorithm can be resumed by the pseudo-code presented below (HBB99):

- all processors start their exploration program;
- processor i, for which $i := Proc(initial)$, starts to explore successor states;
- upon generation of a state $s\prime$ from s:
  - the allocation for $s\prime$ is computed: $j := Proc(s\prime)$
  - if $j = i$ then state $s\prime$ is handled locally;
  - if $j \neq i$ then $s\prime$ and $(s \rightharpoondown^{a\prime})$ are sent to proc. j;
  - all processors process the states received from others, as well as those generated locally;
- the algorithm terminates when each process has no more states to be explored.

Shared Memory

The main advantage shared memory architecture offers over distributed memory is that it provides a shareable memory space for concurrent manipulation, thus obviating the need of passing messages among the processors. As a consequence, there is no 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 assure that a certain processor is accessing the most recent update of the global data. Consequently, synchronization techniques to guarantee mutual exclusive access must be implemented. Nonetheless, to achieve high degree of parallelism, the share of global data that is locked for mutual exclusion must be kept small because synchronization operations are usually time-consuming.

A smaller number of solutions target shared memory machines (HKTL07, Hol08, HB07, IB02, LvdPW10, ADK97, AKH97). They differ mainly by the shared data structure employed and by the work-load strategy chosen to distribute work among the processors. Different data structures impose different locking mechanisms, without mention that the locking grain also plays a significant performance role.

Allmaier et al. (AKH97) was one of the first to address the parallel state space construction on a shared memory machine and the only one that used a data structure different from hash tables. They circumvent the consistency problem of shareable spaces by using locking variables to protect the shared storage structure. They used a Balanced-tree to store the states with a method called splitting-in-advance to reduce the number of data locking, allowing a better concurrent access. The major difficulty with this structure is when an insertion happens into a full node, which forces the node to split into two parts. One of the keys is sent to the parent node, which may also split. This propagation may repeat until it reaches the root vertex. Consequently, a common back to the root propagation is an important point of overhead because it forces the use of several locking variables. The splitting-in-advance method consists in splitting immediately each full node while crossing the Balanced-tree on the way down, regardless of whether an insertion will take place or not. Since non-full nodes serve as barrier, back propagation does not occur because parent nodes can never be full. The result is that each processor holds at most one lock at time. In addition, locking variables are also used to protect the shared stack responsible of load balance, following a *work sharing* scheduling paradigm where new work is distributed to underutilized processors.

Although the implementation of Allmaier et al. reports nearly optimal speedups, the problem of using B-tree's or AVLtrees is the inherent logarithmic time complexity, which can artificially improve the results. Of course, there is an old debate of data

structures and such an analysis is not so simple. However, we believe that hash tables are more appropriated for parallel state space construction, it is not surprising that just a few solutions use trees instead of hash tables. (There is also the work from Bollig et al. (BLW01) for distributed memory machines that uses balanced trees to store the set of explored states.)

The rest of the literature for shared memory machines is based on the use of hash tables to store the states. An example are the DiVinE and Spin multi-core versions, which are the state of the art tools for parallel model checking. DiVinE (BBCR10) for multi-core machines follows their distributed design, they use a static partitioning scheme to distribute the states where each process owns a private hash table.

The parallel version of Spin (HB07, Hol08) makes use of a shared hash table protected by a fine-grained locking technique where only specific parts (that contain a newly generated state) of the hash table are locked. This work has been recently extended with a lock-less shared hash table based on atomic primitives (LvdPW10) (CAS –*Compare & Swap*) in the LTSmin tool. In this work, Laarman et al. proposed a lock-less hash table where locks are emulated with atomic primitives in a separated array. They have for each table slot a *write status* bit that is handled atomically (EMPTY, WRITE and DONE).

Furthermore, we can also mention the work of Inggs et al. (IB02), which use an "unsafe" lockless shared hash table to store the set of explored states . This work implements a hash table without any mutual exclusion locks. The authors emphasize that the duplication caused by the lack of a locking strategy is not relevant compared to the parallel computation power available and that no state will be ignored. All these works report good results, for some models they have speedups close to optimal. Figure 2.7 depicts the speedup analysis for the Spin multi-core presented in (Hol08).

Regarding the work load strategy used by the hash table based solutions, we have basically three main approaches: *static-slicing*, *work stealing* and *stack-slicing*. Like we mentioned before, DiVinE is based on the static-slicing strategy which follows the same guidelines as the distributed solutions presented before. This static partition strategy is not best suited for shared memory machines because most of the time ($\frac{1}{N}$, where N is the number of processors) a processor will have to give away a recently found state. However, this approach allows to achieve satisfactory results with a simple implementation.

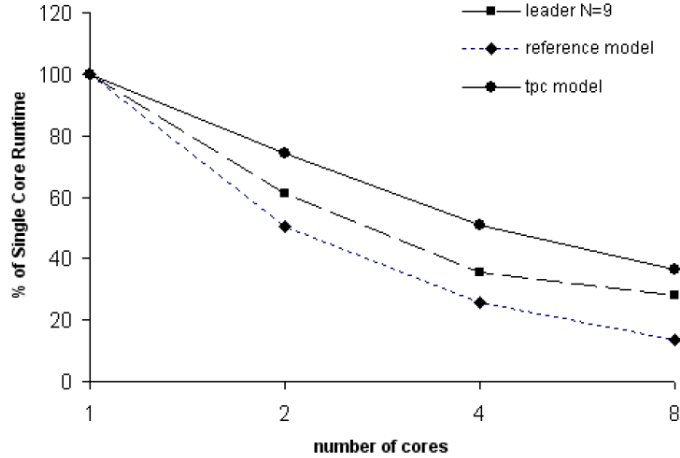


Figure 2.7: Spin multi-core speedup analysis (Hol08).

Following a less restrictive approach than DiVinE, we have Inggs et al. (IB02) that proposed a parallel algorithm for state exploration based on the *work stealing* scheduling paradigm to maintain a dynamic load balance without a blocking phase. The base concept behind this paradigm is that underutilized processors attempt to "steal" work from other processors. This implementation uses a two-queue structure per processor and a hash table to store visited states. The two-queue structure consists in a private and a shared queue that are used to store unexpanded states. Every time a process has no more unexpanded states in its private queue, it has to acquire the mutual exclusion lock to check over its own shared queue for a state. If no state is found, the processor starts searching through all other shared queues until it finds a nonempty queue or finds that all shared queues are empty. Finally, the results reported show that efficiency of the work stealing load balance strategy depends on the division of the state graph (number of successors) and the size of the shared queue. From the experiments presented, optimum results are achieved when the shared queue size of each processor is equal to the branching degree of the graph or one more.

Holzmann et al.(HB07, Hol08) and Laarman et al. (LvdPW10) employed the stack-slicing paradigm to share work among the CPUs. They use shared queues to connect the CPUs into a logical ring structure where each CPU can hand off work only to its right neighbor. The advantage of using a logical ring architecture connecting the CPUs is that each work queue has only one reader and one writer, removing the need for locks

on these structures. The handoff mechanism is controlled by a default depth at which a state transfer to another CPU takes place.

Laarman et al. presented a comparison among Spin, DiVinE and LTSmin in (LvdPW10) using a multiprocessor machine. They observed that LTSmin reports better results than both Spin and DiVinE; the "locking emulation" mechanism proposed by Laarman et al proved effective when compared to Spin, however it should be take into account that Spin is designed specifically for multi-core machines; DiVinE is penalized by its choice of design, it is built over the static partition work-load.

Before concluding this section, it would be interesting to establish a comparison between stack-slicing and work-stealing strategies for dynamic work-load. Actually, as far as we know, there is no work in the literature (model checking or high performance computing community) that does such a comparison. What concerns the model checking domain, we believe that this choice is related to the level of complexity expected and accepted by the developers. Stack-slicing has an advantage in this aspect, unlike work-stealing, it does not require locks to protect the stacks used to share work. However, we believe that work-stealing can spread work faster because a given processor can acquire work from anyone, in contrast with the stack-slicing which spreads work from one processor to another following a logical ring. This feature is interesting for state space construction because we do not know its structure in advance. For instance, consider a state space that alternates its structure between sequential and concurrent path actions.

2.3.4 Parallel LTL Model Checking

In this section we present the related literature for parallel LTL model checking. The parallel algorithms that support Linear Temporal Logic formulas follow the sequential Automata-Theoretic approach (Wol86). The difficulty in this case is to determine whether an accepting state is part of a cycle.

One of the most efficient solution for sequential Model Checking uses the Tarjan's algorithm (Tar71) to find all strongly connected components (SCC); the emptiness problem is reduced to check if an accepting state is part of a SCC; it can be done in linear time in the size of the system but it may incur in extra memory consumption since the states must be stored explicitly for the computation of SCCs.

Another efficient solution and broadly used is (CVWY92) which uses a nested depth first search (Nested-DFS) algorithm to find if an acceptance state is reachable from himself; every accepting state found by the first search triggers a second one to find if it is part of a cycle.

Although both Tarjan's and Nested-DFS algorithms are efficient and simple to implement, they are hard to perform in parallel because they depend on the sequential depth-first search post-order of vertices. It can be illustrated with the example taken from (BBS01) and presented in the Figure 2.8. It is not possible to guarantee that the cycle through the accepting node C will be found by the Nested-DFS algorithm if the parallel DFS exploration does not preserve the DFS order of visited states. For instance, if the nested search starts first from node A, node C will be flagged and the cycle ignored. This problem is inherently sequential and cannot be performed efficiently in parallel (see (Rei85) for a complete discussion).

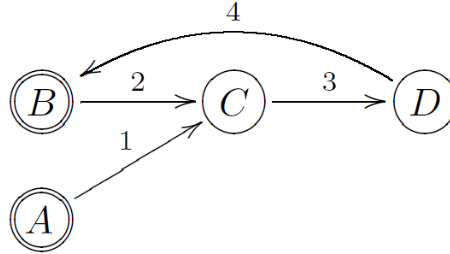


Figure 2.8: Example of the sequential depth-first search post-order of vertices. (BBS01)

The difficulty to find good parallel algorithms for LTL Model Checking can be linked to the fact that "finding a cycle in a graph is an inherently sequential problem" (a claim linked to the fact that the *edge maximal acyclic subgraph* problem is P-complete, see problem A.2.18 in (GHR95)). In this section we present all relevant solutions for parallel LTL Model Checking according to theirs characteristics, we divided them into DFS order and BFS/arbitrary order solutions.

None of the parallel solutions proposed for LTL Model Checking are so efficient like the sequential ones because of the difficulty to assert that an accepting state is part of a cycle. In this section we present all relevant solutions for parallel LTL Model Checking according to their characteristics, we divided them into DFS order and BFS/arbitrary order solutions.

DFS order

The first parallel algorithm for LTL Model Checking was presented by Barnat et al. (BBS01) and was based on a previous work from Lerda et al. (LS99) to deploy the Spin Model Checker on a cluster. Barnat et al. instrumented their exploration engine with a *dependency structure* in order to keep the sequential depth first post-order for the evaluation of accepting cycles: "A nested DFS procedure is allowed to begin from a seed S if and only if all seeds below S have already been tested for cycle detection" (BBS01). This strategy to preserve the sequential post-order comes with the price of undesired synchronizations for the *dependency structure*, moreover, only one nested DFS procedure is allowed at a time. This work targets distributed memory machines (NOW cluster) and makes use of a "generic" static function to distribute work; it was designed to tackle models otherwise untreatable by a single workstation. Hence, execution time (speed up) is not the main concern and only results about the capacity to check bigger models are reported.

The inherent difficulties to keep the sequential DFS order had driven some authors to try a different approach in order to keep the nested search "local", i.e., in a single node (or processor). In (BBC02) and (Laf02), a special static partition function based on the never claim automaton (negation of the Buchi Automaton) is used to "localize" cycles. The main objective is to "... distribute states such that all states of a strongly connect component (or equivalently a cycle) belong to the same equivalence class ..." (Laf02). Hence, all states that belongs to a given cycle will be part of the same *equivalence class*. Assigning each of these classes to only one node is enough to ensure correctness for the nested DFS since the sequential DFS traversal order is still respected within an *equivalence class*. The main drawback of this technique is that the never claim automaton, obtained from LTL formulas, does not hold a significant number of SCC and, by consequence, do not generate a sufficient number of *equivalence class*. (For instance, some formulas may hold only one SCC.)

Next we give some information about three important algorithms that belong to the category of DFS algorithms.

[spin multi-core] Another relevant work that seeks for a simpler approach is (HB07) which uses the notion of *irreversible transitions* to partition the state space

into disjoint subsets. This work was specifically conceived for the multi-core technology and proposes a dual-core extension of the Spin model checker. They use these so called *irreversible transitions* to trigger the nested DFS search; these transitions give two approximately equal and disjoint sets of states when they separate the first from the second search. This design uses only two CPUs, one for each DFS search, and use the *irreversible transitions* to send work from the first to the second CPU. The main advantage of this work is that "... The complexity remains linear in the size of the number of reachable system states, with the same constant factor as applies to the standard nested depth-first search". Although this approach is not scalable for more than two processors, the authors present it as simple and effective solution for dual-core (two CPUs) machines.

[swarm] Another solution for distributed LTL model checking was proposed by Holzmann et al. in (HJG08b, HJG08a). The idea is to have multiple independent instances (workers) of the problem following different exploration heuristics. The main objective is to find errors quickly and not to verify the complete state space. The design is simple, each worker follows a different exploration strategy—i.e., a DFS, BFS or a random order of exploration—until one of them finds an error.

[mc-ndfs] Recently, **swarm** had been extended for multi-core architectures in (LLP+11) and named *multi-core nested DFS* (**mc-ndfs**). They propose a multi-core version with the distinction that the storage state space is shared among all workers in conjunction with some synchronization mechanisms for the nested search. The basic idea is to share information in the backtrap of the nested search. Thus, when a worker backtracks from the nested search, the state involved will be globally ignored through a global coloring scheme, pruning the search spaces for all workers i. Even if in the worst-case each processor might still traverse the whole graph ($O(N \cdot (S + R))$ where N is the number of processors), this work goes one step further to propose an on-the-fly algorithm because the time complexity is still linear in the size of the graph.

Concerning the reported results, as expected, **mc-ndfs** is fast and scales well whenever there is an accepting cycle, otherwise, it suffers a significant loss of performance. The authors justify this loss of performance due to the lack of coloring sharing, "... in the worst case (no accepting cycles) no work is shared between " processors.

BFS/Arbitrary order

The research group that develops DiVinE is one of the most active team and one of the few to propose a different approach for parallel LTL model checking. They introduced a series of four algorithms (BCKP01, BBC03, BCMS04, CP03) not based on the DFS order, these algorithms are indeed based on breath first or arbitrary exploration order. Although they were initially developed for distributed memory machines, the DiVinE team, in (BBR07), reused these algorithms in a search for a good candidate for shared memory machines.

[negc] The first algorithm to follow a different exploration order was presented in (BCKP01) based on the *negative cycles detection* (**negc**). The problem is reduced to finding negative length cycles in the directed graph obtained from the synchronization of the system with a weighted Buchi automaton. In this case, all edges out-coming from accepting states are assigned with -1 and all others with 0, consequently, "negative cycles will simply coincide with accepting cycles and the problem of Buchi automaton emptiness reduces to the negative cycle problem" (BCKP01). They propose a distributed method to solve this problem and compare the results with their previous distributed nested DFS algorithm (BBS01). Despite the fact that it has a theoretical worst-case time complexity in $O(R{\cdot}S)$, the algorithm outperformed (BBS01) (DFS based) because this new algorithm allows for a higher degree of asynchronous parallelism, instead of all the strict synchronizations imposed by the *dependency structure* in (BBS01).

[bledge] Later, Barnat et al., in (BBC03), proposed a distributed memory algorithm named *back-level edge* (**bledge**) based on computing the distance between a vertex, u, and the root following a BFS exploration. This distance is denoted $d(u)$. Indeed, a necessary condition for a path in a graph to be a cycle is that it has two vertices, u and v, following each other such that v is closer to the source than u; an edge (u, v) is called a back-level edge if and only if $d(u) \geq d(v)$. The algorithm computes the BFS distance to detect back-level edges and then check the presence of cycles following a DFS search. The algorithm has a time complexity of $O(S \cdot (S + R))$ and requires some synchronizations to ensure that incorrect distances are never assigned to vertices due to the non-deterministic character of the exploration order in parallel.

[map] Another work that follows the BFS exploration order is the *maximal accepting predecessors* (**map**) (BCMS04). This work is based on the observation that all vertices

on a cycle have exactly the same predecessors: if two vertices belong to the same cycle then they have the same set of predecessors and they belong to this set.

It is not necessary to consider the complete set of vertices in this case, since we are only interested by the accepting states of the automaton. Moreover, it is not necessary to consider all the predecessors of a vertex; the algorithm proposes to use a representative subset of states, called the *maximal accepting predecessors*, obtained from a presupposed linear ordering of vertices.

A drawback is that the set of maximal accepting predecessors may have to be updated dynamically. An advantage is that the algorithm is not bound to a specific traversal order, it does not depend on the sequential DFS postorder. The algorithm works in iterations and has a time complexity of $O = (A^2 \cdot S)$, where A is the number of accepting states.

[owcty] Cerná et al. in (CP03) proposed an algorithm to detect fair cycles that has linear time for weak LTL specifications but has a quadratic worst case complexity. In brief, "the language of the automaton is nonempty if and only if the graphs corresponding to the automaton contains a reachable *fair cycle*, that is a cycle containing at least one state from every accepting set ...". It was inspired from the symbolic algorithm called *One Way Catch Them Young* (**owcty**) (FFK$^+$01), it first computes an approximation of the set of states that contain all fair components and after refines it by successively removing unfair components until it reaches a fixpoint. Although it has a better time complexity in average when compared to the algorithms based on the nested DFS, this work is not on-the-fly and requires several synchronizations in order to refine the set of states. It has a time complexity of $O = (h \cdot (S + R))$ where h is the height of the SCC quotient graph.

The results reported using the distributed version of the **owcty** algorithm could not match the performance of the distributed nested DFS (BBS01) mainly because of the amount of messages and synchronizations required by the distributed implementation. However, it is a good choice for shared memory machines because there is no need to exchange messages and the synchronizations can be achieved just by using barriers and lock regions; a prototype version is presented in (BBR07) and the results reported are very good when compared to other similar algorithms (BCMS04, BBC03, BCKP01). Moreover, (BBR07) addresses the analysis of a shared memory candidate for LTL model checking; all these algorithms were originally developed for distributed machines.

[owcty + map] The state of the art algorithm for parallel LTL model checking was presented by Barnat et al. in (BBR09), they proposed an on-the-fly algorithm by combining the **owcty** and **map** algorithms. The alliance of these two techniques resulted in "a parallel on-the-fly linear algorithm for LTL model checking of weak LTL properties". (Weak LTL properties are those expressible by an automata that has no cycle with both accepting and no-accepting states on its path.) Since **owcty** requires the complete exploration of the state space to work, the accepting cycle detection procedure from **map** is employed but without the re-propagation of accepting predecessors in order to maintain the time complexity linear. On the other hand, if the limited **map** procedure fails to find an accepting cycle, the rest of the algorithm executes the original **owcty**.

Until recently, the results obtained with **owcty + map** indicated that the most appropriate parallel on-the-fly solution would be an heuristic algorithm following a different exploration order than DFS. However, new algorithms such as **mc-ndfs** are showing that under some circumstances, it is possible to achieve a better result following an adapted nested DFS approach. (LLP$^+$11) depicts a comparison between **mc-ndfs** and **owcty + map** for models with and without cycles. For models with accepting cycles, **mc-ndfs** finds counter-examples faster than **map+owcty** due to its depth-first on-the-fly nature, otherwise, in some cases **owcty + map** is faster with a factor of 10 on 16 cores. It is still early to state which one is the best. Meantime, we would like to remind that the worst case scenario for **mc-ndfs** is in fact the case when the formula is valid, hence, this solution is more adapted to find errors fast and not to check if the property is valid.

2.3.5 Parallel CTL Model Checking

In this section, we present the parallel explicit algorithms for CTL model checking. In addition, we also present parallel algorithms for model checking the alternation free μ-calculus, denoted L^1_μ, because of its close relationship to CTL.

CTL model checking can be classified into global and local algorithms. Global algorithms are more suitable to be executed in parallel because cycles can be detected later, indeed, they do not depend on the sequential DFS exploration order because cycles are checked after the complete construction of the state space. While local algorithms decide the problem through a depth-first search, they have the advantage of being on-the-fly.

The first work proposed in this context was for L_μ^1, presented by Bolling et al. in (BLW01), in terms of games (EJS93, Sti99) where the moves correspond to paths in the "game graph" (the Cartesian product of the states s with the formula ϕ). Similar approaches are presented in (BCY02) and, reformulated in terms of solving alternating boolean equation systems (BESs), in (JM05).

Bell et al., in (BH04), propose a distributed version of (CES86). It is based on a labeling procedure and uses the parse tree of the CTL formula to evaluate sub-formulas. The labeling procedure is carried in a distributed manner, a given state is labelled only by its owner following the static slicing strategy used to distribute the states.

In (BLW02), Bollig et al. extend their previous work (BLW01) and propose a parallel local algorithm for L_μ^1 which circumvents the sequential DFS limitation of their initial algorithm by omitting the detection of cycles. This work follows the same idea of "coloring a game graph" but defines a backward color propagation process. Another distributed on-the-fly algorithm was presented in (JM06) where the authors extended their previous work (JM05) on the resolution of BESs. All these works were targeting distributed memory machines and used a static partition to construct

In contrast with the number of solutions proposed for distributed memory machines, just two solutions were conceived for shared memory machines.

[pmc] Inggs et al. (IB06) give the first work specifically developed for shared memory machines and implemented as part of the automata-driven parallel model checker called PMC. It is similar to (BLW01) but it differs in the way winning positions are determined. They use a formalism called Hesitant Alternating Automata (Kup95) to represent the formula specified in CTL*. New games are played locally every time a game position is revisited in order to decide if it is an infinite play (cycle) or part of a different path. For implementing this algorithm, the authors decided not to use locks and therefore to accept a duplication of work.

[(PW08)] Finally, there is (PW08) for shared memory machines based on game graphs. It proposes a parallel algorithm to solve the two-player parity games (Wil01), which is equivalent to model checking for μ-calculus.

2.4 Probabilistic Verification

Probabilistic verification trades memory for the probability of missing some states. The main idea behind this technique is the use of hash values instead of the state value itself, just like a lossy compression strategy where collisions are inevitable. As a result, it becomes possible to analyze part of the state space when there is not enough memory available to represent it in an exact manner. It does not solve the problem of state explosion but it helps to find errors of models previously considered intractable due to memory restrictions. We present below all the significant probabilistic verification works that are related to our proposition.

The first probabilistic technique for model verification was introduced by Holzmann (Hol93) with the *supertrace* algorithm. It was the first work to explore the use of hash values without collision resolution for verification purpose. It brought the possibility of error testing with good coverage ($\approx 98\%$ of the state space) when traditional approaches fail. The effectiveness of this algorithm comes from the data structure used to encode the set of visited states, which is the well known *Bloom Filter* (Blo70) instrumented with two hash functions. This method is memory efficient because states are stored using only two bits, one for each function.

The *Bloom Filter* has been also used in the *multihash* algorithm introduced later by Wolper and Leroy (WL93). The *Multihash* algorithm improves the exploration coverage by increasing the number of hash functions used by the filter. However, the time complexity becomes significant due to the additional generation of hash values and their respective memory accesses over the filter. Some solutions have been proposed to deal with the overhead related to the generation of multiple hash values (DM04, KM06) but, as far as we know, nothing has been done to decrease the number of memory accesses. Moreover, using more hash functions reduces the memory efficiency of the filter because it uses more bits per state.

Another set of celebrated solutions are based on the use of *Compact Hash Tables*, which are capable of increasing the coverage using the same amount of memory but under some circumstances. The idea was first introduced by Wolper and Leroy (WL93) and named *hash compact* algorithm. They argue that storing compressed states (hash values) instead of bits and using a collision resolution scheme for the values stored are enough to deliver a smaller probability of omissions using the same memory space.

Nevertheless, it is not as memory-efficient as the *Bloom Filter* because it requires (recommended) 64 bits per state for an effective coverage. In addition, this algorithm requires a prior knowledge of the state space size in order to dimension the hash table. The memory issue was enhanced later by Stern and Dill (SD95) on the *improved hash compact* algorithm. They argued that Wolper and Leroy analysis could be improved if the probability took into account the calculation of the hash and compressed values independently. Subsequently, Stern and Dill (SD96) proposed a variant of their *improved hash compact* algorithm where the probability of false positives is computed by reasoning about a longest path in the breadth-first search tree of the reachable states. We will skip a comparison with this specific method because we are interested with solutions that are not so complex to parallelize. Indeed, it is costly to respect a given exploration order when the state space construction is performed by a parallel algorithm (see (GHR95) for a list of inherently sequential problems.).

It is important to understand that these solutions are complementary. Dillinger and Manolios (DM04) argue about the categorization of these methods according to the possible estimation of the state space size. The *supertrace* algorithm is the most suitable when the state space size is completely unknown. The *hash compact* holds the best coverage when we know the size rather accurately. Finally, when the size is roughly estimated, a *Bloom Filter* instrumented with more than two hash functions gives a better result. Below, we present these two data structures.

2.4.1 Bloom Filters

A *Bloom filter* (Blo70) is a space-efficient data structure for encoding set membership that is very popular in database and network applications. General theoretical results on Bloom filters can be found in (BMM02), while (DM04) focus more on their use for probabilistic verification. Several variants and extensions have been proposed in the literature. To mention a few, *Counting Bloom filter* (BMM02) uses bytes instead of bits to support the delete operation. *Spectral Bloom Filter* (CM03) is capable of encoding the estimates of frequencies. *Bloomier Filter* (CKRT04) allows the association of an arbitrary function for a subset of elements where the probability of omissions is zero, for all other elements it accepts a small rate of false positive answers. (See (BMM02) for a survey.) In the following, we assume that S is the set of elements (states) we want to store in the Bloom Filter, where $|S| = N$ is the number of elements in S.

Bloom filters support two operations: insertion of an element in the set and test that an element is in the set. A filter B of *dimension* M is implemented as a vector of M bits and is associated with a series of k independent hash functions $(h_i)_{i\in 1..k}$ with image in the interval $0..M-1$. An empty set is represented by a vector with all bits set to 0. Insertion of the element x in B is performed by setting the bits $h_i(x)$ of the vector to 1 for all i in $1..k$. Reciprocally, to query whether an element y is in B, we test that the bits $(h_i(y))_{i\in 1..k}$ are all set to 1 in the vector. If it is not the case, then we are sure that y is not in the set encoded by B. If all the bits are set to 1, we cannot be sure that the element y was inserted in B. Nonetheless, we can compute the probability for y to have been inserted; in the case where y is actually not in the set, we say we have a *false positive*; Moreover the parameters of the Bloom Table can be chosen so as to obtain an arbitrarily low probability for false positives.

In Fig. 2.9 we illustrate insertion and query operations on a Bloom filter with size $M = 16$ and $k = 3$. Starting from an empty set (above), we show the result after the insertion of two elements, x and y. In this example, we have $h_1(x) = 3$, $h_2(x) = 7$ and $h_3(x) = 9$ (actually, the order in which the functions in $(h_i(y))_{i\in 1..k}$ are ran through is not important). Element z is an example of false positive.

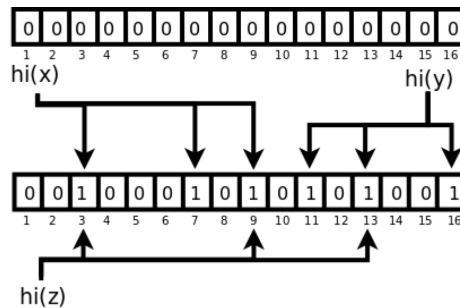


Figure 2.9: Illustration of some operations on a Bloom filter.

The probability for an element to be a false positive is a function of the dimension, M, the number of hash functions, k, and the number of elements inserted in B so far. We can assume that all the elements in the set S have already been inserted, so this last parameter is equal to N. We assume that the k hash functions are independent and perfectly random. As a consequence, for any element x and index $i \in 1..k$, the probability for $h_i(x)$ to be different than some fixed index of B is $(1 - \frac{1}{M})$. Therefore,

after the insertion of N elements (that is $k.N$ writes in the vector), we have the following probability that a given bit is not set (0).

$$p' = (1 - \frac{1}{M})^{k.N} \approx e^{-\frac{k.N}{M}}$$

In the following, we assume that $M \gg 1$ and that the inequality $k.N < M$ is valid. Therefore, we can safely use $e^{-\frac{k.N}{M}}$ as an approximation for $(1-\frac{1}{M})^{k.N}$. The assumption $k.N < M$ is natural. Indeed, it means that there is enough room in the vector B to never write twice at the same position (considering the most favorable case, where each computed hash values are different).

Now, the probability for a given bit to be already set to 1 after N insertions is $1-p'$. Since the probability of false positive is also the probability of finding k bits already set to 1 after N insertions, we have that:

$$P_{Bloom} = (1 - p')^k \approx (1 - e^{-\frac{k.N}{M}})^k \qquad (1).$$

A more detailed discussion on this result in given in (BMM02), where it is also shown that, for fixed values of N and M, there is a value of k that minimizes the probability P_{Bloom}. Hence, there is a way to compute a most efficient value for k if we have a rough estimate of the value of N, that is the size of the state space.

2.4.2 Compact Hash Table

Another celebrated data structure for probabilistic verification is the *hash compact table*. It was introduced by Wolper and Leroy (WL93) as an alternative scheme to obtain a small probability of collisions. They proposed the use of a smaller hash table set with the number of necessary entries. The name *hash compact* comes from the use of hash values instead of the state-description. They argued that this setup instrumented with a collision resolution scheme is sufficient to guarantee very low probability when more than 64 bits are available per state.

From (WL93), considering a hash table of M_h entries where each slot has w bits size and assuming that the overhead required to resolve collisions is negligible, the probability of non omission (p_{no}) for the insertion of N elements is

$$P_{no} \approx e^{-\frac{N^2}{2^w}}.$$

Note that the memory used by the table is of size $M_h \cdot w$. The probability of omissions for the *hash compact table* can be presented as

$$P_{h.compact} = (1 - p_{no}) \approx (1 - e^{-\frac{N^2}{2^w}}) \qquad (2).$$

2.5 Contributions

Before presenting our contributions, it is important to introduce the main guidelines of our work. In this thesis, we perform the formal verification of finite state systems modelled using Petri Nets (Mur89). This means that, in our case, a state is a *marking*, that is a tuple of integers.

Our algorithms have been developed with more complex formalisms in mind, for instance including data, time, etc. We adopted the SPMD programming style and we implemented our algorithms and data structure as part of Mercury, an open-source model checker for multiprocessor machines developed during this thesis (see appendix B).

Section 2.5.1 presents our contributions for parallel state exploration. We propose two approaches: a general speculative algorithm that operates between two phases—exploration and collision resolution—to compute the (exhaustive) state graph of the system; and a mix of distributed and shared hash tables where collisions are solved on-the-fly.

Section 2.5.2 presents our contributions for a practical parallel model checking where "cycles" are detected lazily. The main advantage of our approach is that we devise a new algorithm compatible with different types of state classes abstractions that stores only one transition per state.

Section 2.5.3 presents our contributions on the probabilistic verification of finite systems. We propose a novel probabilistic data structure, called the *Bloom Table*, that fills a gap between the *hash compact* and *Bloom filter*. The Bloom Table is better suited than the Bloom Filter for building concurrent data structures (and algorithms) because it requires less accesses to memory for an equivalent probability of collisions and an equivalent memory consumption.

Finally, we want to stress that the contributions of this thesis are not limited to the domain of formal verification. De facto, the algorithms and data structures proposed

in this work are of interest for any application that performs graph exploration, cycle detection and probabilistic (or lossy) storage in parallel.

2.5.1 Parallel State Space Construction

We presented in section 2.3.3 the related work for distributed and parallel state space construction. The key points to design an efficient parallel algorithm for shared memory machines are the data structure used to store the set of explored states and the workload strategy used to distribute the work (or the states) among the processing units.

In this thesis, we propose two algorithms for parallel state space construction that follow two main principles. The algorithms should be based on a:

- dynamic distribution of data, without prior partitioning; and
- local data spaces (exclusive *write* and shared *read*) for storing the states with only a small, global/shared (*read and write*) data space for synchronizing the computation.

Our first contribution is a speculative algorithm that relies on a novel way to distribute the state space. Our algorithm dynamically assigns states to processors, in the same way we dynamically assign work to processors using a work-stealing strategy. It differs from hash partitions based solutions, i.e. DiVinE multi-core (BBCR10)), where states are assigned statically.

States are stored in local data sets, while a shared Bloom filter is used to dynamically distribute the states. We take advantage of the fast response time and space efficiency of Bloom filter in order to limit undesired synchronizations and increase the locality of memory access. Due to the probabilistic character of the Bloom Filter data structure, we propose a multiphase algorithm to perform exhaustive, deterministic, state space generations. The algorithm iterates between two phases, *exploration* and *collision resolution*, until all possible collisions have been checked and, by consequence, all states have been found.

Our design is original compared to the related solutions (BBCR10, HB07, LvdPW10, IB02), because we do not rely on a unique, big hash table shared among all processors, or a distributed hash table where data is statically assigned. In fact, we build an hybrid algorithm between distributed and shared hash tables where data is dynamically

distributed, according to a probabilistic globally shared structure, and states are stored locally in a distributed fashion (local sets are independent). Our algorithm is quite general and can be adapted to accept different kinds of data structures for local sets. For example, we have already experimented both AVL trees and hash-tables. This work was presented on the 9th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2010) (SZB10).

The second algorithm proposed in this thesis improves on our previous design and replaces the Bloom Filter by a dedicated data structure, the *localization table*. This table is used to dynamically assign newly discovered states and behaves as an associative array that returns the identity of the processor that owns a given state. With this approach, we are able to consolidate a network of local hash tables into an (abstract) distributed one without sacrificing memory affinity—data that are "logically connected" and physically close to each others—and without incurring performance costs associated to the use of locks to ensure data integrity. This work was published in the 10th International Symposium on Parallel and Distributed Computing (ISPDC 2011) (TSDZB11).

Our contributions are twofold. First, for the formal verification community, we define new algorithms for parallel state space construction. Our algorithms are able to exploit parallelism in all possible cases and, unlike algorithms based on slicing functions or heuristic rules, is compatible with dynamic load-balancing techniques. Although we have implemented with the work-stealing approach, nothing prevents our design to work with other strategies such as stack-slicing. Second, for the parallel computing community, we propose new candidates for concurrent hash maps.

Our multiphased algorithm supports different types of data structures (AVLs, hash-tables, etc) for local storage without major changes on the global locking mechanism. Indeed, our speculative approach attenuates the burden for data consistency from the local data structure and shifts it to the iterative character of the algorithm. Moreover, our improved algorithm proposes a new way to use independent (distributed) hash tables as a single shared concurrent hash map; the combination of local hash tables with the localization table provides an interesting implementation for concurrent hash maps that may be useful in other situations.

2.5.2 Parallel Model Checking

We presented in Section 2.3.4 and 2.3.5 an overview of the literature for parallel model checking. We tried to emphasize the need for an effective parallel cycle detection strategy in order to obtain an efficient parallel algorithm. The verification of more elaborated formulas requires the use of efficient algorithms to detect the presence of cycles, such as the use of the Strong Connected Components (SCC for short) abstraction, i.e. (Tar71), or the "nested-DFS" (CVWY92) algorithm. These algorithms are hard to parallelize because they heavily rely on following a particular order when exploring the state graph (for example a DFS order).

We believe that searching only for efficient[1], on-the-fly, parallel solution misses part of the problem. Indeed, to obtain a good complexity in practice, it is also important to be able to benefit from useful work-sharing policies during the complete model checking process; not only during the state space construction.

With the same idea to focus on "pragmatic" optimizations, we also focus on approaches that require less memory space. Our main goal is to propose a solution that is suitable for any state class abstraction without taking into account neither the structure of the model nor its symmetry.

We make the following contributions in the domain of algorithms for parallel model checking.

We define a new algorithm, that we call MCLCD for Model Checking Algorithm with Lazy Cycle Detection. This algorithm is "compatible" with the parallel state space generation techniques described in Chapter 3. By compatible, we mean that we base our approach on the same set of hypotheses; actually, we should say the same absence of restrictions. First, we follow an enumerative, explicit-state approach. We assume that we are in the least favorable case, where *we have no restrictions on the models that can be analyzed*. For instance, we cannot rely on the existence of a symbolic representation for the transition relation, such as with symbolic model checking. Next, *we make no assumptions on the way states are distributed* over the different local dictionaries (we assume the case of a non-uniform, shared-memory architecture). Finally, *we put no restrictions on the way work is shared among processors*, that is to say, the algorithm

[1]Efficient in the sense of "with a good theoretical, worst-case complexity"; such as linear in the size of the state graph for example.

should play nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing.

We propose two versions of MCLCD: a first version based on a reverse traversal of the state graph, called RG, where we need to explicitly store the transitions of the system; and a second version, RPG, where we only need to store a spanning subgraph. The RG version has a linear time and space complexity, in $O(|S| + |R|)$, while the RPG version has a time complexity in $O(|S| \cdot (|R| - |S|))$ and a space complexity in $O(|S|)$.

The main advantage of the RPG version is to provide an algorithm that is efficient in memory and independent of the choice of states classes abstraction. Figure 2.10 lists our algorithms among related solutions for parallel model checking that have already been implemented on shared memory machines.

Algorithm	Time Complexity	Logic Supported
map	$O(\|A\|^2 \cdot \|S\|)$	LTL
owcty	$O(h \cdot \|S\|)$	LTL
negc	$O(\|R\| \cdot \|S\|)$	LTL
bledge	$O(\|S\| \cdot (\|S\| + \|R\|))$	LTL
mc-ndfs	$O(N \cdot (\|S\| + \|R\|))$	LTL
MCLDCD-RG	$O(\|S\| + \|R\|)$	sub-CTL
MCLCD-RPG	$O(\|S\|^2 + \|R\|^2))$	sub-CTL

Figure 2.10: Parallel model checking algorithms for shared memory machines.

Our algorithms follow the classical semantic approach proposed by Clarke et al. in (CES86), with the distinction that we only support a subset of CTL formulas. (We indicate in conclusion of Chapter 4 how this restriction could be lifted). We follow an approach based on labeling states, like with (BH04) and (BLW01, BLW02) in the context of game automata. We choose a semantic approach because we believe that it is more appropriate for a parallel algorithm with dynamic work-load strategies.

Finally, we implemented our algorithms using the work-stealing strategy both for the state space construction phase and the property validation (cycle detection) phase. Related works (HB07, LvdPW10, IB02) use dynamic workload policies for the parallel state space construction only, they do not employ any kind of work-load approach during cycle detection: Holzmann et al. perform the nested search in a exclusive allocated process; Inggs et al. perform (independent) local cycle detection procedures whenever a

node was revisited; and Laarman et al. propose an on-the-fly algorithm where each process performs its own nested search and shares information only to avoid the repetition of nested searches.

2.5.3 Probabilistic Verification

Our main contribution for probabilistic verification is the definition of an enriched *Bloom Filter*, named *Bloom Table* (BT), that is more efficient when we have a rough estimation of the size of the state space. This solution not only delivers a small probability of omission but also improves the execution time by reducing the number of necessary hash functions.

The main difference of our structure, when compared to the classical Bloom Filter, is that we use a vector of "q-bits word" (with $q = 8$ in general) instead of a vector of only bits. Succinctly, a BT with m slots is associated to a sequence of k independent hash functions $(h_i)_{i \in 1..k}$, with values in $1..m$, and another hash function, *key*, that computes values of size $k.q$ bits. To insert a value x inside the BT, the hash-value returned by $key(x)$ is sliced into k sub-words of size q and inserted in the BT at each position $h_i(x)$ for i in $1..k$. Hence, what we obtain is a blend between a Bloom Filter—insertion requires writing in a vector at multiple positions—and a hash-table—insertion may fail because the necessary slots are filled with values different than what is expected.

We position our contribution between the *Compact Hash Table* based solutions and the classical *Bloom Filter*. The probabilistic data structure we propose delivers a better result than the *(improved) hash compact* algorithm using less than 40 bits per state. On the other hand, our solution improves the time complexity when compared to the classical *Bloom Filter* because it offers a better result without increasing the number of hash functions used whenever there are 16 bits available per state.

Chapter 3

Parallel State Space Construction

“ Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.”

Edsger W. Dijkstra

In this chapter we present our novel techniques to perform enumerative parallel state space construction. This chapter is organized as follows. Section 3.1.1 presents the main guidelines of our algorithms such as memory space emplacements and work-sharing techniques. In Section 3.2 we present our general lock free algorithm for parallel state space construction. Section 3.3 presents a devised version based on a mixed approach of distributed and shared hash tables. Before concluding in Section 3.5, we compare in Section 3.4 our approaches with solutions already proposed in the literature.

3.1 Introduction

Verification via model checking is a very demanding activity in terms of computational resources. While there are still gains to be expected from algorithmic methods, it is necessary to take advantage of the advances in computer hardware to tackle bigger models. Obviously, the use of a parallel architecture is helpful to reduce the time needed to check a model because it divides the computation over several processing units instead of one. Another reason, important as well, is the possibility to access a large amount of fast-access memory.

In this chapter we address the problem of generating the state space of finite-state transition systems, often a preliminary step for model checking. In this thesis, we

analyze and propose two novel algorithms for enumerative state space construction targeted at shared memory systems, that are multiprocessor architectures where the memory space can be accessed by all processors. The basic idea behind such algorithms is pretty simple: take a state that has not been explored (a fresh state); compute its successors and check if they have already been found before; iterate. A key point is to use an efficient data structure for storing the set of generated states and for testing membership in this set. With a shared memory architecture, the state space is sliced among all processors and additional efforts are required to ensure data integrity. This is generally obtained through the use of low-level concurrency control mechanisms such as locks and barriers. In the following, we present two algorithms based on novel techniques to construct the state space in parallel without making heavy use of locks.

3.1.1 Algorithms Overview

Our algorithms are based on the same simple design: the global state space is stored in a set of local containers (e.g. hash tables), each controlled by a different processor, while only a small part of the shared-memory is used for coordinating the state space exploration. This is close in spirit to algorithms based on distributed hash tables, with the distinction that we choose to dynamically assign states to processors, that is, we do not rely on an a-priori static partition of the state space.

The main goal of our design is to circumvent the use of locks to protect the local sets by using a lock-less data structure as the shared memory space. Locks are only used for the work-stealing strategy. Moreover, we adopt an “homogeneous” parallelism, which follows the Single Program Multiple Data (SPMD) programming style, such that each processor performs the same steps concurrently. Figure 3.1 briefly presents the memory scheme of our design.

Our first algorithm is a parallel exploration algorithm that constructs the state space in a speculative manner. Strictly speaking, an approximate and relatively small image of the global state space is shared among all processors (or workers). Due to the approximate nature of this image, the algorithm iterates between two phases, *exploration* and *collision resolution*, in order to obtain an exhaustive description of the state space. We call this algorithm “General Lock Free Approach” because its operation mechanism completely isolates the local data structures, enabling the transparent use of different

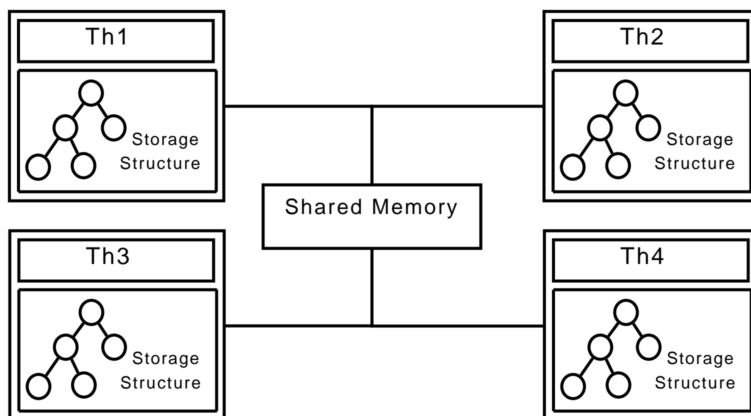


Figure 3.1: Parallel memory organization.

local structures (e.g. AVL trees, hash tables, etc) without the use of locks. This algorithm is presented in detail at Section 3.2.

Inspired by our "General Lock Free Approach", we devised a new algorithm based on an original data structure, called *Localization Table*, that goes one step further on the mixed use of distributed and shared hash tables for shared memory machines. We developed an algorithm that enables the use of distributed hash tables as a single shared concurrent hash map. Unlike our previous algorithm, this new approximate data structure can be used to find the processor that owns a given data item – a state in our case – and not only to assert if the object was already found. This simple addition significantly enhances the performance of our previous algorithm and also simplifies its logic. We give more details about this algorithm in Section 3.3.

Work-Sharing Techniques

Concerning the dynamic work load, our algorithms rely on two different work-sharing techniques to balance the work load between processors. We use these mechanisms alternately during the exploration phase depending on the processor occupancy. First, we use an *active* technique very similar to the work-stealing paradigm of (IB02). This mechanism uses two stacks per processor: a private stack that holds all states that should be worked upon; and a shared stack for states that can be borrowed by idle processors. This shared stack is protected by a lock to take care of concurrent access. The second technique can be described as *passive* and has the benefit to avoid useless

synchronization and contention caused by the active technique. In the passive mode, an idle processor waits for a wake-up signal from another processor willing to give away some work instead of polling other shared stacks. The shift between the passive and active modes is governed by two parameters:

- the *private minimum workload*, which defines the minimal charge of work that should be kept private. The processor will share work only if the charge in its private stack is larger than this value;
- the *share workload*, which defines the ratio of work that should be added in the shared stack if the load in the private stack is larger than the private minimum workload.

Our implementation of the work-stealing paradigm is interesting in its own way since it differs from (IB02) by its use of unbounded shared stacks and the use of a "share workload" parameter.

3.2 General Lock Free Approach

Our first proposition is a speculative algorithm based on two data structures: a lock-free, shared Bloom filter to coordinate the data distribution; and local sets – we use AVL trees in our implementation – to explicitly store the data. We take advantage of the fast response time and space efficiency of Bloom filter in order to limit undesired synchronizations and increase the locality of memory access.

The use of a shared Bloom filter avoids requiring a critical section when writing on local state spaces without sacrificing data integrity. The benefits of this design are better scalability on the number of processors and less contention on memory access. Bloom filters have already been applied for the probabilistic verification of systems; they are compact data structures used to encode sets, but in a way that false positives are possible, while false negatives are not. (See the description of "Bloom Filters" in Section 2.4.1).

We circumvent the probabilistic nature of Bloom filters by proposing an original multiphase algorithm to perform exhaustive, deterministic, state space generation. In the first phase (*exploration*), the algorithm is guided by the Bloom filter until we run out of states to explore. During this phase, states found by a processor are stored locally

in two AVL trees: one for states that, according to the Bloom filter, have already been generated by another processor; the other for fresh states. Since the Bloom filter may, in rare cases, falsely report that a state has already been visited (what is called a false positive), we need to give a special treatment to these *collision* states. This is done in the consecutive phase (*collision resolution*) that takes care of collisions among possible false positives. The algorithm concludes with a *termination detection* phase when there are no more states to explore and no collisions to resolve.

Concerning the operations performed over the shared Bloom filter, we decided for non-protected access to increase the throughput. In the worst case, concurrent accesses lead to duplicated states. For instance, two processors may compete for the same series of slots if they are inserting the same state, what defines a classic situation of date race. In our understanding, we do not consider the duplication of states as a problem because they are compensated by the extra computing power of parallel architecture. Our main concern is to ensure that all states have been explored.

This section is divided as follows. The memory model of our algorithm is given in Section 3.2.1. In Section 3.2.2, we discuss the *exploration*, *collision resolution* and *termination* phases of our algorithm. Finally, we examine experiments performed on a set of typical benchmarks in Section 3.2.3.

In the remainder of the text, we assume that there are N processors and that each processor is given a unique *id*, which is an integer in the interval $0..N-1$.

3.2.1 Shared and Local Data

Our objective is to design a solution adapted to typical shared memory architectures. This means that, in addition to the common difficulties related to shared memory architecture (ensuring data consistency; reducing contention on shared data access; ...), we should also consider the case of Non-Uniform Memory Access architectures (NUMA), where the latency and bandwidth characteristics of memory accesses depend on the processor or memory region being accessed (see Sec. 2.3.1). To improve locality, states generated by a processor are stored in one of two possible local AVL trees, the *state tree* or the *collision tree*. This corresponds to one of the two following cases. Assume that processor i generates a new state s. If a query on the Bloom filter answers that s has not been visited before, the processor may continue generating new states from s. In this case we add s to the state tree of processor i. If the query is positive

– state s may have been visited before – we add s to the collision tree. States in the collision tree will undergo a special treatment to take into account possible false positives.

Each processor also manages two stacks of unexplored states for work-sharing: a *local stack* for storing its private work and a *shared stack* for sharing work with idle processors. Accesses to the shared stack are protected by locks to prevent different processors from requesting the same work. Finally, a shared vector is used to store the current state of processors (either idle or busy) in order to detect termination. Figure 3.2 illustrates the shared and local data structures used in the algorithm.

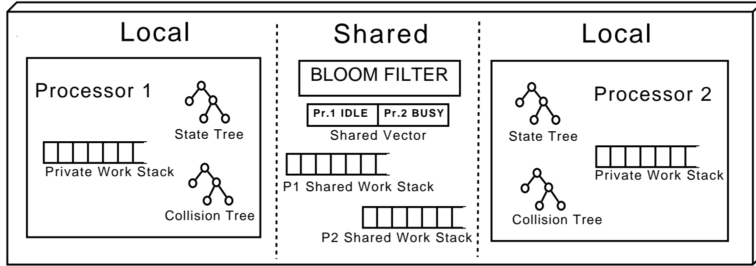


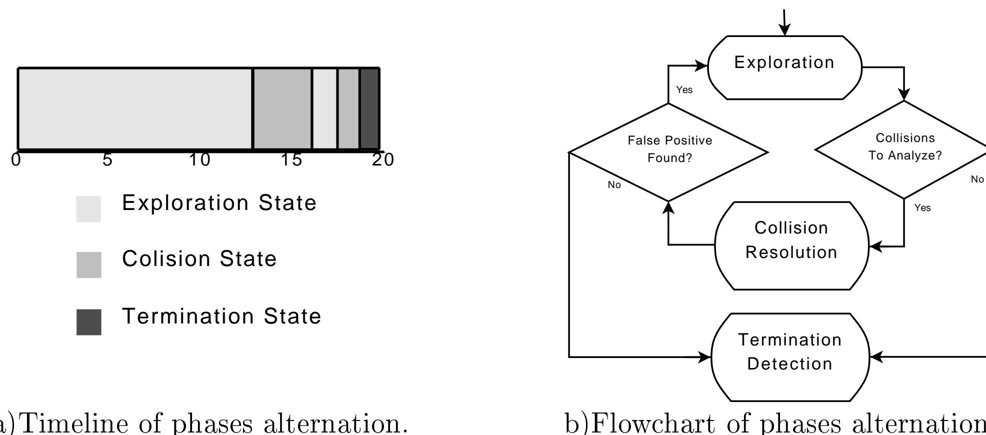
Figure 3.2: Shared and private data overview.

3.2.2 Different Phases of the Algorithm

As mentioned before, our solution makes use of a shared Bloom filter to test whether a state may have already been discovered before. To overcome the problem with false positives, our algorithm iterates between an exploration phase and a collision resolution phase before concluding with a termination detection phase.

The exploration phase takes great advantage of the strong points of a multiprocessor architecture because the shared space is small and all work is done locally. On the opposite, the collision resolution phase puts a lot of stress on the architecture: each processor has to compare the elements in its collision tree with the state tree of all the other processors. As a consequence, the goal is to favor the exploration phase and to reduce the number of iterations. Figure 3.3a) shows the characteristic timeline of phase alternations that we are aiming at. Figure 3.3b) depicts graphically the alternation of phases between exploration and collision resolution. Since iterations are directly

related to the probability of false positive, it is important to dimension the Bloom filter correctly. In our experiments, we typically observe fewer than 3 iterations.



a)Timeline of phases alternation. b)Flowchart of phases alternation.

Figure 3.3: Phases alternation.

In the remaining of this section, we define each phase of our algorithm using pseudo-code. Variable *SS* indicates the current phase of the algorithm. The data structures used in the algorithm are composed of shared and local elements. Shared variables are: (1) the Bloom Filter *BF*, used to test whether a state had already been discovered or not; (2) the bitvector *V*, that stores the state of the processor (0 for idle and 1 for busy); and (3) the shared stacks *Shared_ Stack*[0], ..., *Shared_ Stack*[N-1]. Processor-local variables are the private stack, *private_ stack*, of unexplored states and the two local AVLs: *state_ tree*, to store states discovered by this processor; and *collision_ tree*, to store potential false positives.

Exploration

We give the pseudo-code (Listing 3.1) related to the exploration phase below. The exploration phase proceeds until no new states can be added to the Bloom Filter *BF*. During the exploration, all states appointed by *BF* as already discovered are stored locally in the *collision_ tree*. On the opposite, all newly discovered states are stored locally in the *state_ tree*. Although concurrent accesses to *BF* may seldom result in extra work (state duplication), this is negligible compared to the gain in performance due to the use of a lock-free data structure. Computation switches to the *collision resolution* phase when all processors are idle and there is at least one non-empty local *collision_ tree*. After a

complete iteration, unresolved collisions (false positive) are specially tagged as a means to bypass the *BF* membership test at this phase. We give more details on unresolved collisions in the description of the next phase.

Collision Resolution

The search for collisions (the same state generated in two distinct processors) is done concurrently by each processor through the comparison of its *collision_tree* with the *state_tree* of every other processors. This operation can be implemented efficiently. Indeed, since all these data structures are lexicographically sorted (we use AVL trees for storing states), collisions can be efficiently resolved by comparing trees as ordered lists starting by the leftmost state of each tree. (Figure 3.4 illustrates the synchronization of one collision tree with all state trees.) During this phase, all processors are granted with the read access to the private *collision_tree*s and *state_tree*s.

The advantage of this approach to solve collisions is that if a colliding state s is smaller than a given state of a *state_tree*, no more states of this *state_tree* need to be compared with s. During the *collision resolution*, a state found in the state tree of another processor, say P_i, can be safely deleted from *collision_tree*: it is a "real" collision and it is currently processed by P_i. If the state does not appear in the state tree of another processor then the state is the result of a false positive in the Bloom filter. As a consequence, it will be directly inserted into the private stack of the processor to be expanded during the following exploration phase. We will also mark this state with a special tag to avoid testing it against the Bloom filter a second time. For this reason, if more than one processor find the same false positive, it will result in duplicated states in state space. We prefer to duplicate these states because it would require a complete pass over the false positives found in order to remove the duplicated ones. This operation would demand another synchronization of all N processors. Listing 3.2 presents the pseudo-code for the *collision resolution* phase.

Termination Detection

This phase is responsible for checking if the state space construction should end. Termination detection performs a simple test on the states of the processor and consumes no resources. Assume we arrive in the termination detection phase from the exploration phase, then we can finish the construction if the collision_tree in all processors are

```
1   while SS = Exploration and at least one process is busy do
2      while private_stack is not empty do
3      s ← pop(private_stack);
4         if s is not in BF or s is marked with a special tag then
5            search_and_insert s into state_tree ;
6            let {s1 ,.. ,sj ,... ,sn} ←successors(s) where
7               j ← shared_work_load x n
8               if size(private_stack) > private_work_load then
9                  // Protected action by locks
10                 insert {s1 ,... ,sj} in my shared_stack ;
11                 insert {sj+1 ,... ,sn} in my private_stack
12                 // Share a percentage of new work
13                 if some processor is sleeping then
14                    wake him up
15                 endif
16              else
17                 insert {s1 ,... ,sn} in my private_stack
18              endif
19           endlet
20        else if s is not in state_tree then
21                    search_and_insert s into collision_tree endif
22        endif
23     endwhile
24     //private stack is empty
25     if my shared stack is not empty then
26        transfer work from my shared stack to private_stack
27     else
28        look for a non empty shared_stack to transfer work ;
29        if all shared_stacks empty and at least one processor busy then
30           enter into sleep mode ;
31        endif
32     endif
33  endwhile
34  //Everybody is idle
35  //Protected action by locks
36  SS ← Collision Resolution ;
37  wake up and sync all processors and enter Collision Resolution phase ;
```

Listing 3.1: Exploration phase algorithm pseudo-code.

```
1   leftmost[0..N] ←  leftmost states from state_tree [0..N] ;
2   not_larger[0..N] ← {true,...,true} ;
3   found ← false ;
4   collision ← leftmost state from collision_tree ;
5   while collision is not empty do
6      while not found and leftmost ≠ {∅ , ..., ∅}
7         forall i in 0..N do
8            if not_larger[i] then
9               if collision is smaller than leftmost[i] then
10                 //No more comparisons for this collision
11                 not_larger[i] ← false
12              elsif collision is larger than leftmost[i] then
13                 leftmost[i] :← next ordered element from state_tree[i]
14              else // collision == leftmost[i]
15                 found :← true ;
16                 break forall
17              endif
18           endif
19        endforall
20     endwhile
21     if not(found) then
22        insert collision into private_stack and mark as a special state
23     endif
24     collision ← next ordered element from collision_tree
25  endwhile
26  // No more collision to resolve
27  if private_stack is not empty then
28     // Protected action by locks
29     SS ← Exploration
30  endif
31  if one processor is still busy then
32     enter into sleep mode
33  else
34     wake up and sync all processors ;
35     if SS = Exploration then
36           enter Exploration phase
37     else
38           enter Termination Detection phase
39     endif
40  endif
```

Listing 3.2: Collision resolution phase algorithm pseudo-code.

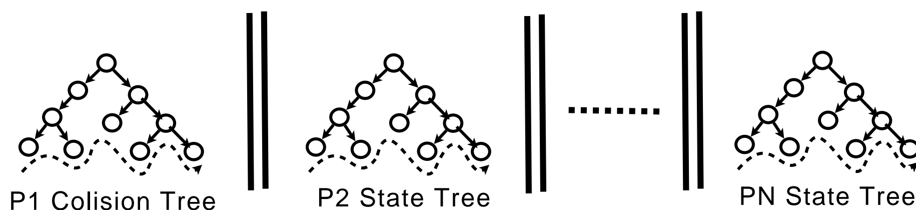


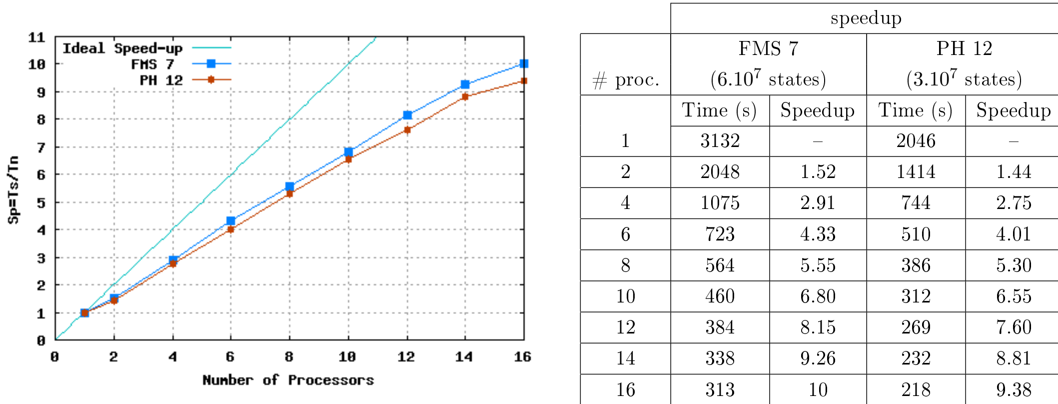
Figure 3.4: Collision resolution with local AVl trees.

empty. In the case we arrive in termination detection phase from the collision resolution phase, then we can finish the construction if the private_stack of all processors are empty.

3.2.3 Experiments

We implemented our algorithm as part of our prototype model checker called MERCURY (Appendix B). In brief, we used the C language with Pthreads (But97) for concurrency and the Hoard Library (BMBW00) for parallel memory allocation. We developed our own library for Bloom filters with support for concurrent insertion. Experimental results presented in this section were obtained on a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208GB of RAM memory, running the Solaris 10 operating system. We worked with a 512MB Bloom filter ($n = 4.10^9$ bits) and 6 chained hash-functions ($k = 6$). These parameters are dimensioned for examples of up to 5.10^8 states, with a small rate ($\approx 2\%$) of false positives.

The finite state systems chosen for this benchmarks are classical examples of Petri Nets taken from (MC99). Together with the perennial Dining Philosophers, we also study the examples of the Flexible Manufacturing System (FMS) and the Kanban System, where the first one is parametrized by the number of subnets and the following two by the weights in their initial marking. We give several results detailing the performance of our implementation. While speedup is the obvious criterion when dealing with parallel algorithms, we also study the memory trade-off of our approach and report on experiments carried out to choose the dimension of the Bloom filter.



# proc.	speedup			
	FMS 7 (6.10^7 states)		PH 12 (3.10^7 states)	
	Time (s)	Speedup	Time (s)	Speedup
1	3132	–	2046	–
2	2048	1.52	1414	1.44
4	1075	2.91	744	2.75
6	723	4.33	510	4.01
8	564	5.55	386	5.30
10	460	6.80	312	6.55
12	384	8.15	269	7.60
14	338	9.26	232	8.81
16	313	10	218	9.38

Figure 3.5: Speedup analysis for PH 12 and FMS 7 models.

Speedup

Figure 3.5 gives the observed relative speedup of our algorithm when generating the state space for 12 philosophers (PH 12) and FMS 7 with a different number of processors. We give the relative speedup (see Sec. 2.3.1), measured as the ratio between the execution time using N processors (t_N) and the time of the same algorithm on one processor (t_s).

Figure 3.6 depicts the system occupancy rate, throughout the duration of the state space computation, for the PH 12 model using all the available processors. The occupancy rate measures the utilization of the machine CPUs. The figure shows high occupancy rate[1] ($\approx 92\%$) for our algorithm, except for a small interval that corresponds to the transition between the exploration and the collision resolution phases. For this experiment, only one pass over the exploration and the collision resolution phases is enough to generate the complete state space. The termination detection phase happens after the collision resolution phase was finished and no false positive was found.

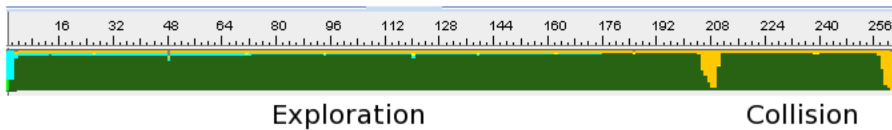


Figure 3.6: Occupancy rate for PH 12 with 16 processors.

[1] The slightly execution time difference is a consequence of the overhead generated by the profiling tool.

Time-Memory Trade-off

Figure 3.7 gives results on the number of *collision nodes* (see Section 3.2.1) used on the FMS 7 and PH 12 examples. We also give the amount of memory required for the collisions tree. The results show an increase of the memory footprint when the number of processors increase. The intuition behind these numbers is quite simple: due to the strong symmetry of the example, if we add more processors, we increase the probability of different processors finding the same state, that is the probability of creating a collision node. As shown with the experiments, the number of collisions generated by our algorithm may be 3 to 4 times greater than the number of states in the worst case (16 processors). What is observed is a general trade-off between memory space and computation time that is often found in parallel algorithms. It should be noted that the use of traditional optimizations, such as partial-order or symmetry reduction techniques, would reduce the number of collisions.

	FMS 7 (6.10^7 states)		PH 12 (3.10^7 states)	
# proc.	# collision tree nodes	Ex. M[a] (GB)	# collision tree nodes	Ex. M[b] (GB)
2	5.10^7	1.0	3.10^7	1.5
4	9.10^7	1.8	6.10^7	3.0
6	11.10^7	2.2	8.10^7	4.0
8	14.10^7	2.8	9.10^7	4.5
10	14.10^7	2.8	10.10^7	5.0
12	16.10^7	3.2	11.10^7	5.5
14	16.10^7	3.2	11.10^7	5.5
16	16.10^7	3.2	12.10^7	6.0

[a] Extra Memory Estimation = $collisions * 20$
[b] Extra Memory Estimation = $collisions * 50$

Figure 3.7: Collision analysis for FMS 7 and PH 12.

A parallel algorithm often trades additional memory space for better execution time. Nevertheless, it is very important to maintain this additional memory usage at an acceptable level. In our case, this means limiting the number of collisions. A straightforward way to deal with this problem is to force the early start of the *collision resolution*

phase as soon as one of the processors reaches a given threshold of collisions states. We can compare this strategy with familiar memory management techniques, such as garbage collection. The choice of the good value for the threshold is a trade-off between execution time and memory usage.

# Th.	FMS 7 (6.10^7 states)				PH 12 (3.10^7 states)			
	T.(s)[a]	Gain	M[b]	Exp.– Col.	T.(s)	Gain	M[c]	Exp. – Col.
1.10^6	762	.43	.3	.37 – .63	284	.76	.8	.58 – .42
2.10^6	488	.68	.6	.56 – .44	237	.91	1.6	.71 – .29
3.10^6	384	.86	.9	.64 – .36	242	.89	2.4	.72 – .28
4.10^6	369	.90	1.2	.64 – .36	226	.95	3.2	.77 – .23
5.10^6	346	.96	1.6	.68 – .32	226	.95	4.0	.79 – .21
6.10^6	370	.90	1.9	.68 – .32	222	.97	4.8	.80 – .20
7.10^6	331	1.00	2.2	.77 – .23	228	.94	5.6	.81 – .19
8.10^6	332	1.00	2.6	.75 – .25	229	.94	6.4	.82 – .18
9.10^6	328	1.01	2.8	.73 – .27	219	.98	7.2	.87 – .13
10.10^6	337	.99	3.2	.76 – .24	219	.98	8	.85 – .15
∞	334	1	3.2	.76 – .24	216	1	6	.87 – .13

[a] Execution Time in seconds
[b] maximal extra memory required $N \times Th \times 20$ bytes
[c] maximal extra memory required $N \times Th \times 50$ bytes

Figure 3.8: Threshold analysis using 16 processors for PH 12 and FMS 7.

We study the impact of the threshold value on the overall performance in Figure 3.8. With this strategy, the maximal extra memory required by our algorithm is given by the formula $N \times Th \times SS$, where N is the number of processors used; Th is the threshold; and SS is the size of the state representation (20 bytes for FMS 7 and 50 bytes for PH 12). Column *Gain* and M depict the relative variation of performance and extra memory required for different values of the threshold for the FMS 7 and PH 12 models using 16 processors. For both models, the experiments show that threshold values above 4.10^6 lead to almost no penalty ($Gain \approx 1$): we observe a drop of performance below 10% ($Gain = 0.90$). In addition, the extra memory required by the collisions for threshold value of 4.10^6 is capped at 1.2 GB instead of the 3.2 GB used in our previous experiment

with the FMS model (see Figure 3.7).

The last column in the table of Figure 3.8, labeled "Exp. — Col.", splits the total execution time into the time spent in the exploration and *collision resolution* phases. The results show an inverse correlation between the ratio of times spent in these two phases and the overall performance: we observe that the speedup decreases when this ratio increases. The intuition behind these numbers is quite simple; with smaller thresholds, there are not enough "newly discovered states" during the exploration phase to compensate for the time spent during the collision phase. As a matter of fact, we observe good time ratio between exploration and collision phases in the experiments without memory recycling (threshold value of ∞). For instance, for both models, a threshold of 10^7 gives almost the same profile than using no memory recycling.

Comparison: AVL vs Hash Tables

We conclude this section of experiments with a comparison between two implementations of our algorithm. We called our approach as "general" because it enables the use of different data structures for local storage. Now, we present a comparison between two different implementations of our algorithm, AVL and Table (see Figure 3.9). AVL stands for the straightforward implementation described in Section 3.2, using AVL Trees as local sets. Table is the same algorithm where AVL trees have been replaced by local hash-tables. Both implementations have their advantages and disadvantages. The AVL implementation proved slower than the other solution. This result is not enough to dismiss the use of AVL. Indeed, while the high algorithmic cost associated to this data structure is a handicap, the choice of AVL has also some benefits. For instance, the use of a sorted data structure in AVL simplifies the *collision resolution* phase, where the state in each local collision tree should be compared against all the other collision trees; this may make the *AVL* solution faster when there are many collisions (hence it could be superior when the number of processors increase). By contrast, Table performs the *exploration* phase faster than *AVL* but requires the pre-allocation of the table. Even though it can be attenuated with resizing strategies, nothing can be done regarding the extra size required for an efficient use; it is known that hash tables performs well when the load factor is around 0.5[1]. Hence, It will impose a significant extra memory use depending on the state size.

[1]Ratio between the number of states in the hash table and its size (number of entries)

	Execution Time (s) with 16 processors	
Model	AVL	Table
Kanban 9 38.10^7 states	2547	1319
FMS 8 24.10^7 states	2003	953
PH 13 14.10^7 states	1306	836

Legend for the algorithm name abbreviations
AVL Local AVL Trees as dictionaries
Table Local Hash Tables as dictionaries

Figure 3.9: Comparison of Different Implementations.

3.2.4 Discussion about the experiments

The experiments conducted with the preliminary implementation of our algorithm show promising speedups on a set of typical benchmarks. While the performance of the algorithm depends on the "geometry" of its input – for instance its concurrency degree – we have consistently obtained good results. For example, we routinely observe efficiency values[1] over 70% while keeping the extra memory needed with our algorithm at a constant level.

Our strategy to force the early start of the *collision resolution* phase proved to be memory efficient, it allowed us to limit the memory used to store the collisions. However, for the model checking domain, this extra memory used by the collision trees may be the difference between a complete or an incomplete run of the algorithm. Consequently, we decided to propose an extension where collisions are solved on-the-fly in order to get rid of both the collision trees and the *collision resolution* phase.

This new algorithm demands flexible data structures, capable of enhancing both the performance and the memory footprint of our algorithm. Next section presents our improved version, where we replaced the shared *Bloom Filter* by a specialized probabilistic data structure that not only dynamically assigns states but also keeps track of the distribution. Hence, whenever a processor finds an old state, it is possible to recover its owner and to resolve on-the-fly if it is a false positive or not.

[1]Efficiency is computed as the ratio between speedup, t_N, and the number N of processors.

3.3 Mixed approach: Localization Table based algorithm

This section presents an improved version of our previous algorithm where collisions are solved on-the-fly. Based on our general (speculative) approach, we devised a new algorithm where data distribution and coordination between processes is made through a special structure called *localization table* (LT), that is a lock-less, thread-safe data structure that approximates the set of states being processed. The localization table is used to dynamically assign newly discovered states and behaves as an associative array that returns the identity of the processor that owns a given state. With this approach, we are able to consolidate a network of local hash tables into an (abstract) distributed one without sacrificing memory affinity – data that are "logically connected" and physically close to each others – and without incurring performance costs associated to the use of locks to ensure data integrity.

The LT is used to test whether a state has already been found and, if so, to keep track of the location – the processor id – where the state is held (see Figure 3.10). The work performed by each processor is pretty simple: generate a state using the model of the system, say s, and check in the LT where it could have potentially been assigned. If s is a newly discovered state, it will be assigned to the processor who generated it. Otherwise, the LT will return the location where the state s is assigned, say $LT(s)$. This approach has the advantage of isolating the local hash tables; each processor has exclusive write access to its local table, whereas concurrent read accesses are unrestricted. Another advantage is that we can easily resize local hash tables, as needed, without blocking the entire exploration. For this new algorithm, we choose to use only hash tables because they perform the state space exploration faster than AVL trees. (See the comparison between AVL trees and Hash tables at Section 3.2.3.)

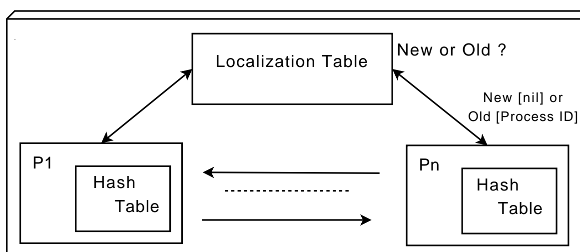


Figure 3.10: Algorithm overview.

One of the advantages of our design is to be thread-safe: operations on a LT are simple and can be implemented using primitive atomic actions. (We define primitive atomic actions as CPU instructions that are guaranteed to be completed without being interrupted by the actions of another thread. This kind of instruction is supported by the majority of x86 processors today.) Another advantage is the small memory footprint of the LT. Furthermore, differently from our general approach, this new algorithm does not duplicate work , i.e., states. To summarize, our goal is to combine the advantages of distributed and shared hash tables for parallel state space construction in a single algorithm. We describe more precisely how the LT is implemented in Section 3.3.1. Section 3.3.2 gives some pseudo-code and further explanations about the algorithm we developed to use our novel LT. In Section 3.3.3, we examine experiments performed on a set of typical benchmarks.

3.3.1 Localization Table

Storing the relation $(s, LT(s))$ – associating each state, s, with the processor identifier that owns it – in a single table would require a very large amount of memory. Actually, it would defeat the need to store the states themselves. Instead, the idea is to use a notion of a *key* associated to a state and to store the association between keys and processors. In our implementation, keys are computed using hash-functions and we will use a scheme based on multiple keys.

A localization table is essentially a "table" that associates a processor id – a value in $1..N$ – to every key in the table. A straightforward implementation is to use an integer vector for the underlying table.

We can implement the table using a vector V of size n and, for computing the key of a state, an independent hash function, h, with image in $1..n$. In this case, we can check if a state s has already been found by looking into the table of processor $V[h(s)]$. While this implementation is simple, its disadvantage is that it is not possible to ensure a fine dynamic distribution of the states if h is not uniform. Indeed, if processor id_1 finds a new state, s, such that $V[h(s)] = id_2$, then we need to transfer s between the two processors. A solution is to increase the size of the vector – but this has a direct impact on the memory consumption – or to use better hashing functions – but this has an impact on the performances.

We propose another implementation of the localization table that improves upon the choice of a vector. Inspired from our previous experience with a *Bloom Filter* (BF), the idea is to use a finite family of hashing functions $h_1, \ldots, h_k$. To test if a state s is in the LT, we search if the key $h_1(s)$ is in LT. If it is not, we know that the state is fresh. If $LT(h_1(s)) = id_1$ then we check if s is in the local table of processor id_1. If it is not the case, we continue searching with the key $h_2(s)$ and so forth.

This is only a rough description of how the LT works. Next, we define more formally the operation of our data structure. In particular, we explain how to deal with states that are not in the processors $LT(h_1(s)), \ldots, LT(h_k(s))$, that we call *collision* states. By convention, our algorithm will route a collision state to the last processor found, that is $LT(h_k(s))$.

A *Localization Table*, L, is defined by two parameters: its size n; and a family of k independent hashing functions $h_1, \ldots, h_k$ with image in $1..m$, where we choose m such that $n \ll m$ (see the discussion about the ratio n/m in Section 3.3.3).

A localization table L of size n is an array of size n containing pairs of the form (p, d), where p is a processor id ($p \in 1..N$) and d is a key ($d \in 1..m$). To look for values inside of L, we use a fixed surjective function map, from $1..m$ to $1..n$. Hence, to check the value associated to the key $h_i(s)$, we look in the array L at index $map(h_i(s))$.

Initially, an empty LT is an array initialized with the value $(0, 0)$. Assume that the processor id attempts to insert a state s into L. The function takes as input a state and a processor id and returns a pair made of: a status, to determine if the element is new (or old); and the identifier of the processor who owns the state. The insertion operation is performed by looking successively at the elements with index $map(h_i(s))$ in L for all $i \in 1..k$. There are four possible cases at each step:

- if $L[map(h_i(s))] = (0, 0)$ then we know for sure that the state is fresh (it has never been added before). We can stop our iteration and write the pair $(id, h_i(s))$ in L. This can be done using an atomic compare and swap operation;

- if $L[map(h_i(s))] = (id', d)$ and $d \neq h_i(s)$ then we cannot decide if the state s has been found and we continue to the next iteration, with the key function h_{i+1};

- if $L[map(h_i(s))] = (id', h_i(s))$ then we answer that s is in the local table of processor id' with high confidence. With this approach, states with the same

```
1  function test_or_insert(s:state, id:1..N) : (status, my_id)
2    (id,d) ← (0,0);
3    for i in 1..k do
4       (id,d) ← LT[map(h_i(s))];
5       if (id,d) = (0,0)   // the slot is empty means that s is fresh
6       then LT[map(h_i(s))]←(my_id,h_i(s));
7            return (new,my_id);
8       elsif d = h_i(s) // the state s may already be in processor p
9       then return (old,id);
10      endif
11   endfor
12   //state s is a collision - assign it to processor d;
13   return (old,id);
```

Listing 3.3: Insertion in the Localization Table

hash value are not handled at the LT level. In our algorithm, these collisions will be spotted when the processor tries to recover the state from the local table of id'. In order to keep the consistency of the LT and to prevent states from being stored more than once, we choose to assign s to the processor id' and this is handled like a collision state;

- if we cannot decide after checking the values of $L[map(h_i(s))]$ for $i \in 1..k$, we also say that s is a collision state and we choose to assign s to the processor id' such that $L[map(h_k(s))] = (id', d)$.

These fours steps are defined in Listing 3.3 by the function test_or_insert(s, my_id) using pseudo-code.

The operation for checking whether a state s is already in the LT is very similar to the insertion function. We test successively if there is an index $i \in 1..k$ such that $L[map(h_i(s))] = (id_i, h_i(s))$, stopping if one of the positions in L is empty. If this is the case, we know that s is not in the LT. If we find no match after k attempts, then we consider that s is a collision state that belongs to id_k.

In Figure 3.11 we illustrate the insertion and test operation for three data items: x, y, and z; performed in this order by the processors P_1, P_2 and P_3. The figure displays a LT of size $n = 4$ with two independent hash functions h_1 and h_2. We assume that $k = 2$ and $m = 31$. The insertion of x requires only one operation since the slot at position $map(h_1(x))$ is initially empty. As a result the slot is associated to processor P_1 for elements with key 17. Element y is inserted at the second attempt, since the slot in

position $map(h_1(y))$ is already filled and that $h_1(y) \neq d_1$. Finally, element z cannot be properly inserted – it is a collision – and it is assigned to processor P_2.

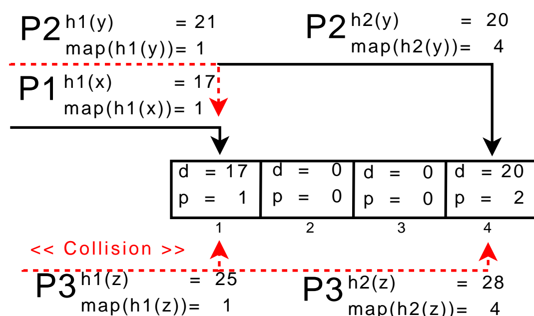


Figure 3.11: Insertion in a Localization Table.

3.3.2 Algorithm

In this section, we give a high-level view of our algorithm described by the pseudo-code of Listing 3.4. Figure 3.12 describes the shared and local data structures used in the algorithm. Each processor manages a “private work” stack of unexplored states and a local hash table to store the states assigned to him. The shared values are: the Localization Table; one bitvector of size N to store the state of the processors (idle or busy), used to detect termination; N stacks – one for each processor – for the work sharing technique described in Section 3.1.1; and finally N collision stacks used to route collisions states to their correct processors.

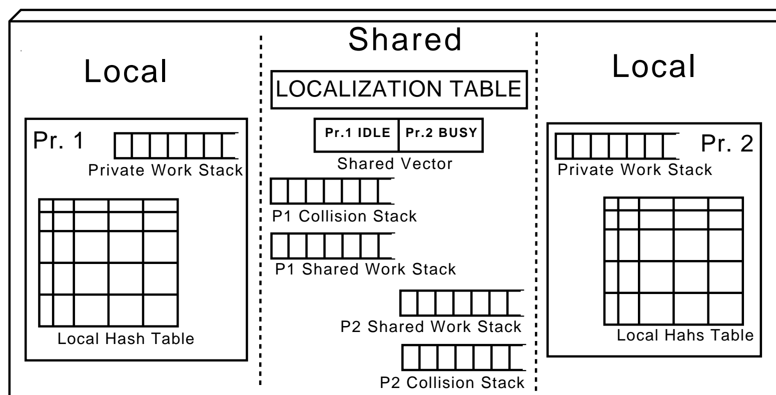


Figure 3.12: Shared and private data overview.

The state space exploration proceeds until no new states can be added to the LT and all stacks are empty. Given a state s, a processor, say my_id, will check the LT to test whether s is new and, otherwise, who is the owner of s. This information is returned by a call to the function test_or_insert(s, my_ id). During the exploration, states that are labeled as new by the LT are stored in the local table of the processor. On the opposite, if the LT returns an owner *id*, then the process performs a look-up operation over the local table of processor *id* to check if the state is really there. If the state is not found, we can tag it as a *collision state* and add it to the collision stack of processor *id*. Collision states are specifically tagged since they bypass the LT membership test and are directly inserted in a local table. When the private work stack is empty, work is transferred from shared work and collision stacks; if they are also empty, the processor may "steal" work from others (as described in Section 3.1.1). The LT is implemented using an atomic "compare and swap" primitive, while locks are only used to protect the access to the other shared data structures – the shared work and collision stacks – which are not resource contention points.

Finally, termination can be easily detected by testing the vector recording the states of processors; the algorithm may safely finish if there is no more state to explore, that is if the stacks of all the processors are empty and if all the processors are idle.

3.3.3 Experiments

We also implemented our algorithm as part of our prototype model checker called Mercury (Appendix B). The implementation followed our previous general approach with the distinction that we replaced the *Bloom filter* by our LT. We developed a library for the *Localization Table* with support for concurrent insertions. The Experimental results presented here are obtained using the same machine, a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208GB of RAM memory, running the Solaris 10 operating system.

For this benchmark, we used models taken from two sources. We have three classical examples: Dining Philosophers (PH); Flexible Manufacturing System (FMS); and Kanban – taken from (MC99) – together with 5 Puzzles models: Peg-Solitaire (Peg); Sokoban; Hanoi; Sam Lloyd's puzzle (Fifteen); and 2D Toads and Frogs puzzle (Frog) – taken from the BEEM database (Pel07). All these examples are based on finite state systems modelled using Petri Nets (Mur89). This means that, in these cases, a state is a

```
1  while (one process still busy)
2    while (Proc[my_id].private_stack not empty)
3       do s ← remove state from Proc[my_id].private_stack;
4          if s not tagged as collision
5          then (status,id) ← LT.test_or_insert(s,my_id); //Concurrent access to LT
6          else (status,id) ← (new, my_id);//Collision state
7          endif
8          if status = old
9          then if s not in Proc[id].local_table
10              then tag s as collision;
11                   add s to Proc[id].collision_stack;
12              endif
13         else
14            add s to Proc[my_id].local_table ;
15                  let {s1 ,.. ,sj ,... ,sn} ←successors(s) where
16                     j ← shared_work_load x n
17                     if size(private_stack) > private_work_load then
18               // Share a percentage of new work
19                        // Protected action by locks
20                        insert {s1 ,... ,sj} in my shared_stack ;
21                        insert {sj+1 ,... ,sn} in my private_stack ;
22                        if some processor is sleeping then
23                           wake him up endif ;
24                        end if
25                     else
26                        insert {s1 ,... ,sn} in my private_stack ;
27                     end if
28                  end let
29         endif
30   endwhile
31   //private stack is empty
32   if my collision_stack is not empty then
33         transfer work from my collision_stack to private_stack
34   endif
35   if my shared_work_stack is not empty then
36         transfer work from my shared_work_stack to private_stack
37   endif
38   if my private_stack is empty then
39         look for a non empty shared_stack to transfer work ;
40         if all shared_stacks empty and at least one processor busy then
41              enter into sleep mode ;
42         endif
43   endif
44   ...
45 endwhile
```

Listing 3.4: Algorithm pseudo-code

marking, that is a tuple of integers. Our algorithm may be adapted to other formalisms, for instance including data, time, etc.

With our computer setup, we are able to tackle examples with approximately 10 billions of states, but we selected models with less than 500 millions of states in order to complete our experiments in reasonable time. (A complete run of our benchmark takes more than a week to finish.)

We study the performance of our implementation on different aspects. While speedup is the obvious criteria, we also study the memory footprint of our approach and the physical distribution of states among processors.

Speedup and Physical Distribution

In Fig. 3.13 we give the observed speedup of our algorithm on a set of examples. We give the absolute speedup (see Sec. 2.3.1), measured as the ratio between the execution time using N processors (t_N) and the time of an optimized, sequential version (t_s). Our implementation delivers some promising speedups. The results also show different behaviors according to the model. For instance, our efficiency may vary between 90% (Hanoi model) and 51% (Kanban model), whereas the system occupancy rate[1] is consistently over 95%. Clearly, the algorithm depends on the "degree of concurrency" of the model – it is not necessary to use lots of processors for a model with few concurrent actions – but this is an inherent limitation with parallel state space construction (EL08), which is an irregular problem.

Concerning the use of memory, we can measure the quality of the distribution of the state space using the *mean standard deviation* (σ) of the number of states among the processors. In our experiments, we observe that the value of σ is quite small and that it stays stable when we change the number of processors (see Fig. 3.19). For instance, we have $\sigma \approx 1.5\%$ for the Hanoi model and $\sigma \approx 7\%$ for Kanban. The difference between values of σ can be explained by the difference in the "degree of concurrency". It may also be affected by the processor's performance, that is, a processor that handles "simpler states"— states whose transition firing involves a small number of operations —may dynamically assign more states than others. Finally, our experiments are also affected by the Non-Uniform Memory Access (NUMA) architecture of our machine, where the

[1] The occupancy rate measures the utilization of the machine CPUs

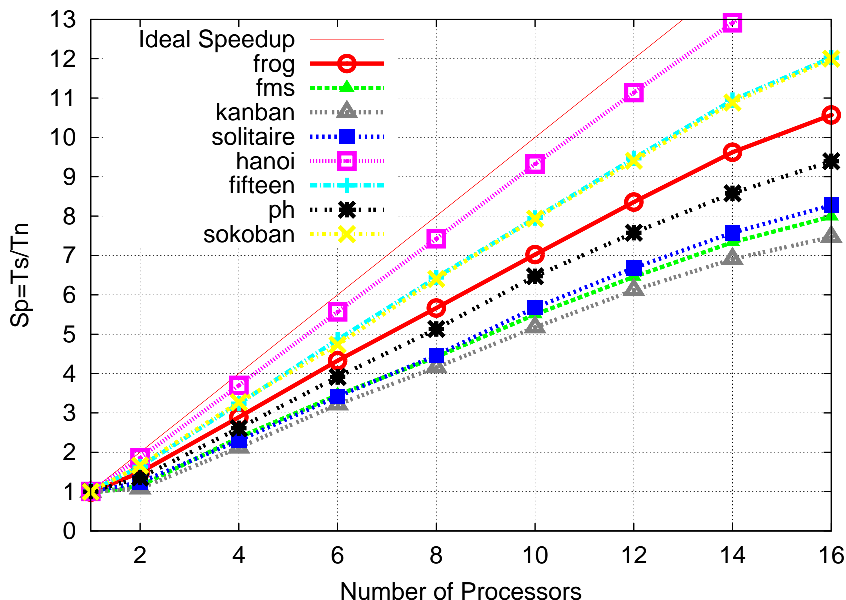


Figure 3.13: Speedup analysis.

latency and bandwidth characteristics of memory actions depend on the processor or memory region being accessed.

Localization Table Configuration and Memory Footprint

The LT data structure is configured using two parameters: its dimension (n) and the number of hash-functions keys (k). The values of these parameters have an impact on the performance. If the dimension is too small, the LT will get quickly saturated and the number of collision states exchanged between processors will increase. It is important to keep the number of collisions as low as possible because they increase the size of the collisions stacks. Moreover, it also affects the execution time due to the overhead necessary to deal with these stacks, without mention that a new state erroneously sent as a collision will have its discovery delayed.

Ideally, a LT of size n is sufficient for a space of n states. However, hash functions are not perfect (uniform), which affects our structure just like with standard hash table. In our experiments, we observe that LT behaves well for load factors (ratio between the number of states in the LT and its dimension) lower than 0.7.

In Fig. 3.14, we display the ratio (in percentage) between the number of collisions

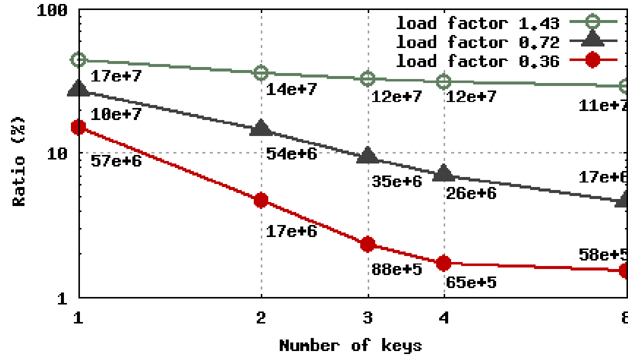


Figure 3.14: Collisions vs load factor.

exchanged and the size of the state space, on the Kanban model ($\approx 4.10^8$), for three different values of the load factor and for different values of k. This Fig. also depicts the absolute number of collisions for each experiment. As expected, smaller load factors give results with smaller number of collisions. For instance, we have approximately 58.10^5 collisions exchanged (ratio of 1.5%) when LT is set with $n = 2^{30} \approx 10^9$ (load factor of 0.36) and $k \leqq 8$.

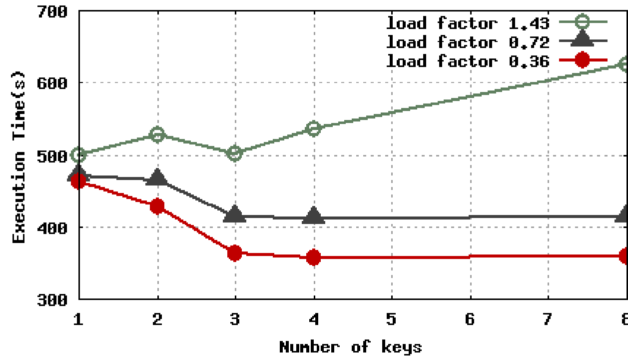


Figure 3.15: Performance vs load factor.

Likewise, in Fig. 3.15 we show the impact of different load factors (choice of the size of the LT) on the execution time of the algorithm for a fixed model. Once again, we observed that our LT gives better results when the load factor is in average smaller than 0.7. Consequently, smaller load factors have a better overall performance because the number of collisions exchanged by the processors is reduced.

For the speedup results given in this Section, we have adjusted the dimension of

the LT to obtain load factors smaller than 0.5 for every models and we have chosen to limit ourselves to at most four hash-functions keys ($k \leq 4$). We decided to fix these settings beforehand in order to not artificially improve our results and also to show the memory efficiency of our solution. To illustrate this point, we may observe that in the experiments of Fig. 3.15, the size of the LT is of 1GB (that is approximately one billion data items) for a load factor of 0.36, which is the only significant memory overhead used by our solution.

3.4 Comparison With Other Algorithms and Tools

Before concluding this chapter, we present a comparison with other solutions proposed in the literature. We developed our own implementation of some classical parallel algorithms based on the use of hash tables. We have implemented these algorithms as part of MERCURY (see Appendix B) . In Fig. 3.16, we briefly describe the different implementations used for this comparison.

Name	Description	Reference
G-AVL	General variant instrumented with local AVL trees as dictionaries	GENERAL
G-Table	General variant instrumented with local Hash Tables as dictionaries	GENERAL
LT	Distributed Table instrumented with our Localization Table	MIXED
Vector	Vector of integers like structure: Localization Table with only one key	MIXED
Static	States are distributed using a static slicing function	(SD97)
Lock-less	Lock-less shared hash table as the shared space	(IB02)
TBB	Unordered hash map as the shared space, from Intel-Threading Building Blocks library	(Rei07)

Figure 3.16: Algorithms selected.

Comparison With Other Algorithms

We present in Figure 3.17 a comparison of our algorithms general (G-AVL and G-TABLE) and mixed (LT and VECTOR) for the same set of experiments we presented in

Figure 3.9 . The results obtained with our algorithm instrumented with LT are at least 4 times better. These results show that our strategy to eliminate the collision resolution phase is not only benefit for the memory footprint but also for the performance as a whole. Indeed, solving collisions on-the-fly is more efficient because the collision resolution phase is resource intensive, both in terms of memory and performance.

	Execution Time (s) with 16 processors			
Model	G-AVL	G-Table	LT	Vector
Kanban 9 38.10^7 states	2547	1319	363	444
FMS 8 24.10^7 states	2003	953	209	227
PH 13 14.10^7 states	1306	836	179	189

Figure 3.17: Comparison of Different Implementations.

Figure 3.18 shows the average (absolute) speedups[1] for the different implementations presented in Figure 3.16. The `Lock-less` implementation has the best performance but it is an unsafe solution, since states may be skipped (BR08). All the other implementations are safe. We include the results for `Lock-less` since it provides a good reference for performance. Our algorithm (**LT**) performs better than all the other implementations for all models. As we mentioned earlier, the difference in performance between `Vector` and **LT** is mainly due to the non-uniformity of hash functions. This difference is significant especially for Sokoban and Kanban models (see the load factor analysis for the Kanban model at Fig 3.15). Concerning `Static`, an explanation for the better performances is that we exchange less states between processors: in some experiments with `Static`, we can observe that up to 96% of the states have not been found by their owner. The gain in performance compared to `TBB` (based on a lock-less, non-lossy hash table found in the Intel-TBB (Rei07)) may be explained by the adequacy of our data structure to our targeted application (state space generation). Indeed, in this application, we have many more reads than writes (state spaces have more transitions than states). The LT has several benefits in this case: (1) it delivers a low complexity mechanism to grant exclusive write access for the local hash-table; (2) the structure is

[1] The average speedup is the mean of all speedups achieved by a given implementation for all models.

cache-friendly since data are stored in-place (avoiding pointers); and (3) the use of local hash tables improves memory affinity, which is important for NUMA machines.

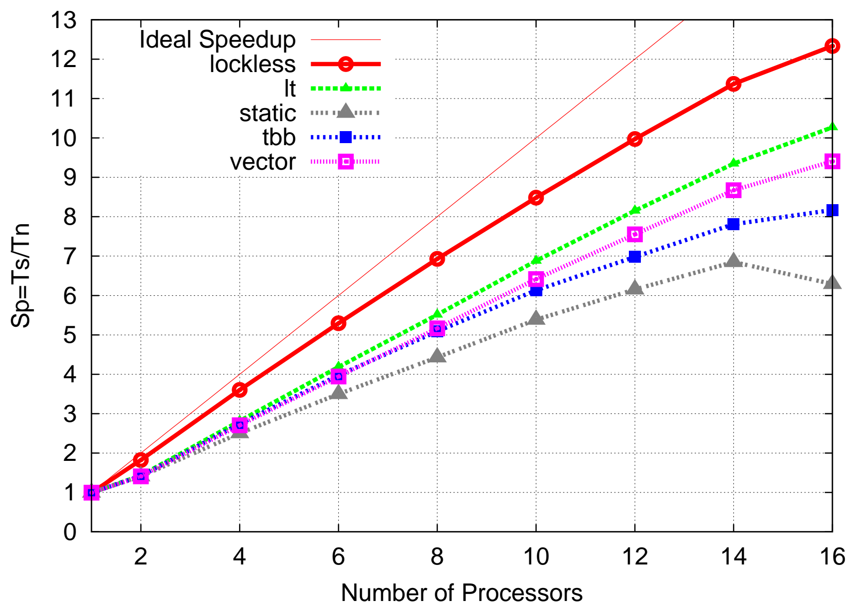


Figure 3.18: Average Speedup.

Concerning the memory distribution, we display the average mean-standard deviation for all implementations in Fig. 3.19. The results show that the best distribution, by far, is from the `Static` version. We can observe that all other implementations have similar distributions. (The anomalous values for $N = 16$ can be explained by the fact that, in this case, we use all the processors of our computer.) In the context of this work, we use no heuristics to ensure an uniform partition of states, so the quality of the distribution depends on the model "degree of concurrency" and the performance of processors. The rest of this benchmark is given at Appendix A.

Comparison With Other Tools

We have also compared our implementation with "state of the art" verification tools that provide a parallel implementation. We looked both at the Spin and DiVinE tools. We give some performance results but it is difficult to make a fair comparison. For one thing, it has proved difficult to port available implementations on the configuration used for our experiment. For instance, our benchmark with DiVinE and Spin are based on

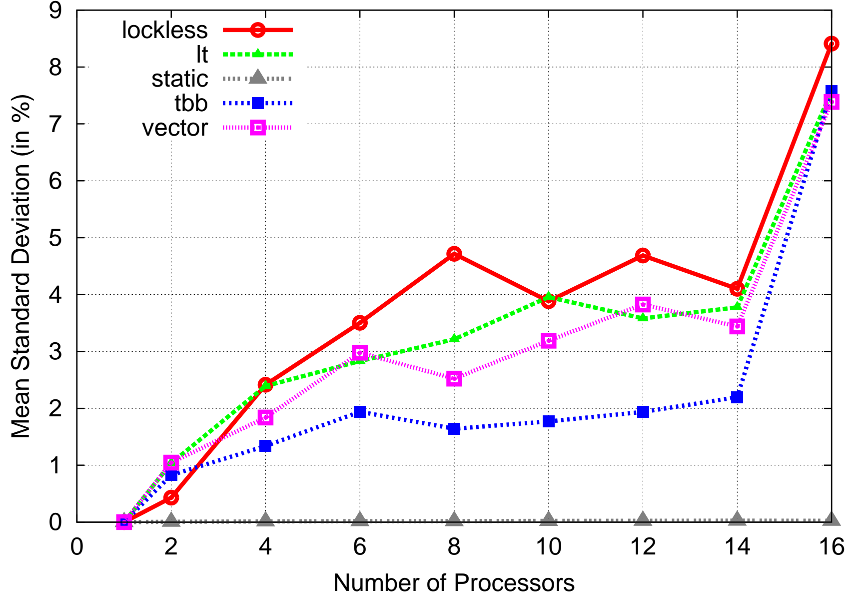


Figure 3.19: Mean-Standard Deviation.

Linux instead of Solaris, which means that we take advantage of more efficient libraries. On the other hand, a major discrepancy lies in the fact that we compare an algorithm with a tool. For instance, we do not make use of any "general optimizations" techniques, such as local caches, data-alignment optimizations, etc. Also, while Spin works with compiled models, we currently use interpreted models. DiVinE accepts both models but we use their interpreted variant for this comparison.

After these words of caution on the significance of the comparison, we give some results obtained on the Sokoban model. Using the parallel version of Spin on our benchmarks, we observe a maximum speedup of 3.6 using 8 cores ($73s$). Nonetheless, the sequential performance of Spin ($264s$) is about 3 times better than our prototype implementation of **LT**. In our experiments, **LT** is marginally faster than Parallel Spin when both are running on 12 cores and is faster using the 16 available cores. The computation for Spin is of $81.2s$ for 12 cores and 82.3 for 16 cores, while we generate the state space in $80s$ with 12 cores and $64s$ with 16 cores using **LT**.

Concerning DiVinE – whose sequential performance is about 40% better than our prototype implementation – **LT** matches the performance of DiVinE when both are using 10 cores and outperforms it of about 20% using 16 cores. More precisely, the

running time for DiVinE is of $96.5s$ with 10 cores – while **LT** time is of $96s$ – and $84.9s$ with 16 cores. It is possible to connect this result with the comparison given in Fig. 3.18. Indeed, DiVinE is based on a static slicing function to distribute the states – as in the `Static` implementation of Fig. 3.16 – and the difference of performance between **LT** and `Static` is of about 30% on 10 cores and of almost 70% for 16 cores.

These preliminary results against two of the most popular parallel model-checker are very encouraging since we have only a prototype implementation of **LT**.

3.5 Conclusions

In this chapter we presented two novel algorithms for parallel state space construction target at shared memory multiprocessor machines. We built our work over an innovative memory design; we used distributed sets structures to store the states locally and a small shared memory space to manage the dynamic distributions of states. Our approach takes into account spatial balance by dynamically assigning states to processors and managing states locally. Based on this unique design, we proposed two variants: a general (speculative) algorithm—where collisions are solved using inter-process synchronization—and a mixed algorithm, where collisions are solved on-the-fly. Our experimental results show that the mixed version is the best choice in practice.

In the context of our experiments, we worked more specifically with system described by Petri Nets. Nonetheless, our algorithms are quite general and could be applied to different formalisms for describing finite transition systems (or finite abstractions of infinite-state models): we only require a simple way to represent states and a function to generate successors. While we provide an implementation that works with an explicit representation of states, our work can be applied alongside with traditional optimizations for reducing the state space size, such as partial-order and state compression techniques. We consider a black-box approach which is orthogonal to the representation details of the state space.

A first implementation of our mixed algorithm (LT based) shows promising results as we observed speedups consistently better than with other parallel algorithms. For instance, our experimental results show that efficiency varies between 90% and 50%, depending on the “degree of concurrency” of the model. In addition, our memory footprint is almost negligible when compared to the total memory used for storing the state

space. For example, in the worst case (the Kanban model, with 25GB) we consume less than 4% of the memory for the LT and the different stacks used by our algorithm. That is approximately 1GB of memory.

The same benchmark also shows that our implementation fares well when compared with related tools. Indeed, our experiments show that our solution performed very well against an industrial strength lock-less hash table, the concurrent hash map implementation provided in the Intel-TBB. This may be explained by the fact that we provide a concurrent data structure for encoding sets that is optimized for the case where deletions are very rare and the same item may be inserted several times, whereas the Intel-TBB provides a general implementation. This is very encouraging since we obtained these results with minimal optimizations (i.e. without resorting to global caches, data-alignment optimizations, etc.), so there is still room for improvements. Altogether, our solution fulfilled our goal of having both the best temporal and spatial balance as possible.

For future works, we are experimenting a new version of our algorithm where LT is used to define a kind of lazy locking policy strategy. The idea behind is to grant write access to all processors for any of the local hash table as long as they lock the table completely. We extended the local hash tables with lightweight locks which are controlled by a shared LT. So, every time a processor finds a state, and if it is a new state then it gets directly the lock for its own table and the right to insert it locally. Otherwise, LT returns the possible owner (hash table) for ownership verification. If after checking, it finds out that the state is indeed a new one, the processor who found the given state tries to lock that table in order to insert the state. If the processor fails to get the lock, it gives away and send it as an LT collision. Our preliminary experiments show that we can obtain similar (or sometimes slightly better) results but using a smaller LT. To give actual figures, in this case we can use a LT of $1MB$ in contrast with the $1GB$ used in some of our experiments. We still have many open questions about this new version and therefore it is early to say it is a better/usable solution.

Chapter 4

Parallel Model Checking With Lazy Cycle Detection — MCLCD

"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."
Edsger W. Dijkstra

In this chapter we present two new algorithms to perform model checking in parallel. This chapter is organized as follows. Section 4.1 introduces the main guidelines of our algorithms. In Section 4.2 we define the set of logical formulas supported in our approach. After the definition of basic results on graph theory, we describe our parallel algorithms using pseudo-code in Section 4.4. We study the fundamental properties—complexity, termination, soundness, ...—in Section 4.5. Before concluding, we give a set of experimental results performed with our prototype implementations in Section 4.7.

4.1 Introduction

We describe and analyze a new parallel model checking approach that is compatible with the parallel state space generation techniques described in Chapter 3. By compatible, we mean that we base our approach on the same set of hypotheses; actually, we should say the same absence of restrictions. First, we follow an enumerative, explicit-state approach. We assume that we are in the least favorable case, where *we have no restrictions on the models that can be analyzed.* For instance, we cannot rely on the

existence of a symbolic representation for the transition relation, such as with symbolic model checking. Next, *we make no assumptions on the way states are distributed* over the different local dictionaries (we assume the case of a non-uniform, shared-memory architecture). Finally, *we put no restrictions on the way work is shared among processors*, that is to say, the algorithm should play nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing (see the discussion in Sect. 2.5.2).

We decided to define our own parallel algorithm for model checking instead of trying to parallelize existing, state-of-the-art, sequential algorithms. We can give a simple, theoretical justification to support our choice. In the sequential case, many efficient model checking algorithms rely on the computation of Strongly Connected Components (SCC) or, at least, rely on following a specific order when exploring the state space graph—generally a Depth-First Search (DFS) or breadth-first search order. This is the case, for example, in most of the *automata-theoretic* approaches for model checking LTL[1]. These algorithms rely on efficient methods to detect the presence of cycles in a graph, such as Tarjan's algorithm (Tar71) or "nested-DFS" (CVWY92). While this class of sequential algorithms are very efficient—their complexity is linear on the size of the state graph—they do not lend themselves to parallelization. Indeed, it is known since the 1980's that "depth-first search is inherently sequential" (Rei85), more precisely, that it is related to problems that are P-space complete. This gives strong evidence that trying to parallelize this class of automata-theoretic, sequential, algorithms is not the right way to go, at least if we expect a significant speedup.

We can give a second justification, that is more related to the implementation choices made in our work. Indeed, algorithms that rely on exploring the state space graph in a specific order go against our requirement that the state space exploration should be friendly to traditional work-sharing techniques.

Based on these observations, we decided to follow an alternative approach that we call *semantic model checking* . This is the approach initially proposed by Clarke and Emerson (CE82, Cla99) for CTL model checking. In its simplest form, a semantic algorithm works by labeling each state of the system with the "sub-formulas" of the initial specification that are true for this given state. Labels are computed iteratively

[1]We use the term automata-theoretic to denote algorithms in which model checking is reduced to composing the system with an automaton that accepts traces violating the specification; and then using graph algorithms to search for a counterexample trace.

until we reach a fix-point, that is until we cannot add new labels. Similar algorithms are also used for evaluating modal μ-calculus formulas (EJS93) (if we leave out the extra-complexity involved with modal fixpoint alternation).

To obtain an efficient parallel model checking algorithm, we decided to limit ourselves to a restricted subset of temporal logic formulas. We made the choice of a subset of formulas, expressible both in CTL and LTL, that is equivalent to the requirement specification language supported by Uppaal (BDL04). This subset includes formulas for expressing basic invariant and reachability properties, but also more complex properties, such as $\psi \leadsto \phi$, meaning that every state where ψ holds eventually "leads to" a state where ϕ holds.

Our approach is quite simple. The algorithm is based on two separate steps: (1) a forward, constrained exploration of the state graph—where we start labeling each state with local information—followed by (2) a backward traversal—where we propagate information towards the root of the state graph—to check if the resulting graph has an infinite path. (We can avoid the case of dead states—states without successors—by adding a trivial loop to each such state.) It should be observed that, since we work with finite state systems, any infinite path includes at least one cycle. This remark explains why, in the remainder of this chapter, we will often focus our attention on the problem of identifying a cycle in a graph.

We propose two versions of our algorithm that differ by the way we store the state graph in memory. In the first version, we assume that, for every reachable state, we have a constant time access to the list of all its "parents". Basically, it means that we store the *reverse graph* (RG) structure of the state space[1]. In the second version, we assume that we have access to only one of the parents, meaning that we may have to recompute some transitions dynamically. We say that the second version is based on a *reverse parental graph* (RPG).

The advantage of this second version is to save memory space. The gain in memory space can be very important; something we have experienced in our experiments. Indeed, if we use the symbol S to denote the number of states (vertices) in the graph, the size of the data structure for our first algorithm is of the order of $O(S^2)$, in the worst case, while it is of the order of $O(S)$ for the second version.

[1]In the context of this work, we prefer to use the inverse of the transition relation because we propagate labels from a state to its parents

4.2 List of Supported Properties — the LRL Logic

Model checking is a problem with two variables: it depends both on (1) the formal language used to express the models, and (2) the theoretical framework used to express the specification of the system. In this thesis, we follow a classical approach in which properties are expressed using temporal logic formulas.

Several approaches for model checking exist in the literature, supporting a different set of temporal logics. Linear Temporal Logic (LTL) and Computation tree logic (CTL) are the most popular options, mainly because there exist efficient algorithms; algorithms with a linear time complexity in the size of the state graph.

While sequential algorithms for model checking are a well-studied research subject, we believe that it is still early to claim that the problem is settled in the parallel case. Indeed, to the best of our knowledge, there is no good parallel algorithm that: covers a "complete logic" (such as the whole of LTL or CTL); has a linear, worst-case complexity which does not duplicate work; and provides a good speedup independent if the property holds or not.

In the context of our work, we do not try to define a parallel algorithm with all these good properties and favor speedup. We decided to trade off the expressiveness of the specification language for a better parallel complexity. Furthermore, in the case of the second version of our algorithm, we are willing to abandon our requirement over the linear, worst-case complexity of the algorithm. We show with our experimental results that this may prove a good choice in practice.

We limit ourselves to a restricted subset of temporal logic formulas that corresponds to the requirement specification language supported by Uppaal (BDL04). This specification language includes formulas for expressing basic reachability, safety and liveness formulas and can be expressed as a subset of CTL formulas, with the distinction that we follow a *local*, instead of global, model checking semantic; that is to say, we are interested by properties that are valid for the initial state, and not from any other given state.

We give the list of supported properties below, with a simple explanation for each property. We use the symbols $\phi, \psi, \ldots$ to denote *predicates*, that is statements that may be true or false depending only on the state on which they are evaluated[1].

[1] Since we define a general model checking algorithm, we do not specify exactly what is the language

- $E\Diamond(\phi)$: this property expresses the *possibility* for ϕ to hold. The property is valid if there is a path, starting from the initial state, that reaches a state where ϕ holds. This corresponds to the formula $EF\,\phi$ in CTL (note that we are only interested by the validity of this formula for the initial state).

- $E\Box(\phi)$: this property expresses that ϕ holds *potentially always*. The property is valid if there is an infinite path, starting from the initial state, where ϕ holds for every state s in the path. This corresponds to the formula $EG\,\phi$ in CTL.

- $A\Diamond(\phi)$: this property expresses that ϕ always *eventually* holds. The property is valid if, for every path starting from the initial state, there is a state where ϕ holds. This corresponds to the formula $AF\,\phi$ in CTL. The property $A\Diamond(\phi)$ is true if $E\Box(\neg\phi)$ is false.

- $A\Box(\phi)$: this property expresses that ϕ is an *invariant*; it always holds. The property is valid if, for every reachable state, ϕ holds. This corresponds to the formula $AG\,\phi$ in CTL. The property $A\Box(\phi)$ is true if $E\Diamond(\neg\phi)$ is false.

- $\psi \leadsto \phi$: the *leadsto* property expresses that from every state where ψ holds, eventually ϕ will hold. This corresponds to the formula $AG\,(\psi \Rightarrow AF\,\phi)$ in CTL. This is the only property that corresponds to a CTL formula with nested modalities.

- $E\,(\psi \cup \phi)$: the *reachability* property is valid if there is a path, starting from the initial state, such that ψ holds until we reach a state where ϕ holds. This is an extension of the possibility property since $E\Diamond(\phi)$ is equivalent to $E\,(\text{True} \cup \phi)$. This property is not part of Uppaal's specification language and encompasses the formulas $E\Diamond(\phi)$ and $A\Box(\phi)$.

- $A\,(\psi \cup \phi)$: the *liveness* property is valid if for every path, starting from the initial state, the predicate ψ holds until we reach a state where ϕ holds. This is an extension of the eventuality property since $A\Diamond(\phi)$ is equivalent to $A\,(\text{True} \cup \phi)$. This property is not part of Uppaal's specification language and encompasses the formulas $A\Diamond(\phi)$ and $E\Box(\phi)$.

of predicates. This parameter may be changed depending on the underlying languages used to define the models.

We call our specification language LRL, for Leadsto-Reachability-Liveness. As we said in the description of LRL, every property corresponds to an equivalent CTL formula. LRL is strictly less expressive than CTL. In particular, the language does not allow the nesting of formulas (modalities are always applied to predicates). Nonetheless, we believe that our specification language is expressive enough for many practical cases; the choice of this language by the designers of Uppaal gives at least some support to the claim that this subset provides a good balance between expressiveness and efficiency.

Formal Semantics of LRL. Next, we define the semantic of our specification language with respect to a given *Kripke structure*, KS, using an entailment relation. The definition is very similar to the traditional presentation of the semantic of CTL formulas. We use a Kripke structure $KS = (S, R, s_0)$ to represent the behavior of the system. We use S to denote the set of states, $s_0 \in S$ to denote the initial state, and R to denote the transition relation between states in E. We use the symbol τ to define an infinite sequence of states (a trace) $s_0 \cdot s_1 \cdot \ldots$ such that $s_i R s_{i+1}$ for all i. Finally, we use the notation $(S, R, s_0) \vDash F$ to denote that formula F holds for the Kripke structure (S, R, s_0) and $s \vDash \phi$ to denote that the predicate ϕ holds for s. Considering the different equivalences between properties that we already gave, we simply need to define the semantics for the formulas $E\,(\psi \cup \phi)$, $A\,(\psi \cup \phi)$, and $\psi \rightsquigarrow \phi$.

$$
\begin{array}{rcl}
(S, R, s_0) \vDash E(\psi \cup \phi) & \text{iff} & \text{for some path } \tau = s_0 \cdot s_1 \cdot \ldots \\
& & \exists i[i \geq 0 \wedge s_i \vDash \phi \wedge \forall j[0 \leq j < i \Rightarrow s_j \vDash \psi]] \\[1ex]
(S, R, s_0) \vDash A(\psi \cup \phi) & \text{iff} & \text{for all path } \tau = s_0 \cdot s_1 \cdot \ldots \\
& & \exists i[i \geq 0 \wedge s_i \vDash \phi \wedge \forall j[0 \leq j < i \Rightarrow s_j \vDash \psi]] \\[1ex]
(S, R, s_0) \vDash \psi \rightsquigarrow \phi & \text{iff} & \text{for all path } \tau = s_0 \cdot s_1 \cdot \ldots \\
& & \forall i[(i \geq 0 \wedge s_i \vDash \psi) \Rightarrow \exists j[i < j \wedge s_j \vDash \phi]]
\end{array}
$$

Before defining our model checking algorithm, we give some simple results from graph theory that will be useful to prove the soundness of our approach.

4.3 Some Graph Theoretical Properties

Our model checking algorithm is based on an iterative exploration of the state space graph. The goal is to prove that a given invariant is valid for every infinite path in the state space. Since we work with finite state systems, any infinite path includes at least

one cycle. Hence, in the remainder of this chapter, we will often focus our attention on the problem of identifying a cycle in a Kripke structure. To this end, we need to define some properties of Directed Acyclic Graphs (DAG).

To give an example; to check the property $A\Diamond(\phi)$ on a Kripke structure KS, it is enough to generate the subset KS_ϕ of KS obtained by "stopping" our exploration whenever we find a state s where ϕ holds. Then, the property is true if KS_ϕ is a DAG; otherwise there would be a cycle of states in KS where ϕ never holds.

Definition 4.3.1. A finite Directed Graph $G(V, E)$ is an ordered pair (V, E) comprising a finite set V of vertices and a finite set E of edges, (v_i, v_j), such that v_i and v_j are in V for all edges. A finite Directed Acyclic Graph (DAG) is a finite directed graph $G(V, E)$ with no *cycles*, that is there is no way to find a sequence of edges $v_0 \cdot \ldots \cdot v_{n+1}$ such that $(v_i, v_{i+1}) \in E$ for all index i in $0..n$ and $v_0 = v_{n+1}$.

We prove that, in a finite DAG, there is always at least one vertex that has no children (what we call a *leaf*) and one vertex without parents (what we call a *root*). In the following, we say that a leaf has *out-degree zero* and that a root has *in-degree zero*.

Lemma 4.3.1. *In a finite DAG $G(V, E)$ there exists at least one vertex in V with in-degree zero and at least one vertex in V with out-degree zero.*

Proof. The proof is by contradiction on the definition of a maximal path in G. We say that $\tau = v_1 \cdot \ldots \cdot v_n$ is a proper path in G if all the vertices in τ are different. We say that τ is a maximal path if it is a proper path and if there are no proper path of size $n + 1$. If the in-degree of v_1 is not equal to zero, then there exists a vertex $w \in V$ such that $(w, v_1) \in E$. If w is a vertex in τ then we have found a cycle, which contradicts the fact that G is a DAG. Otherwise, $w \cdot \tau$ is a proper path of G of size $n + 1$, which contradicts the fact that τ is maximal. Following a similar reasoning, we can show that the out-degree of v_n is necessarily zero. $\square$

We give another property related to leaves in a DAG. Our algorithm mostly relies on the following observation: a finite graph is acyclic if, whenever we recursively remove all the leaves, we eventually end up with an empty graph. By recursively removing the leaves, we mean removing a leaf from the graph—together with all its incoming edges—and starting over with the remaining graph. The procedure stops when no more nodes can be removed.

Actually, we use a slightly stronger property and rely on the fact that it is enough to stop removing leaves when all the vertices have in-degree zero (the graph has only root nodes). This property is expressed by Theorem 4.3.2.

Theorem 4.3.2. *A finite directed graph $G(V, E)$ is a DAG if and only if, by recursively removing the leaves, we finally end up with a graph that only has root nodes.*

Proof. The property is trivial if G is the empty graph.

Otherwise, assume G_0 is a finite DAG with $n + 1$ vertices. By lemma 4.3.1, we know that there is at least one vertex l_0 in G_0 that is a leaf. If we remove this vertex and its incoming edges from G_0, the resulting graph G_1 is a finite DAG with n vertices (removing a vertex cannot introduce a cycle). We can repeat this operation to obtain a sequence $(G_i)_{i \leq n}$ of finite DAG with less and less vertices. Therefore, the graph G_n has only one vertex and this vertex is a root. Actually, there is an index $k \leq n$, that is the smallest value such that all the vertex in G_k have in-degree zero.

To prove the other direction, we assume that we have a finite sequence of graphs $G_i(V_i, E_i)$, with $i \in 0..n$, such that $G_0 = G$, all the vertices in G_n are roots, and we obtain the graph G_{i+1} by removing one leave from G_i. Let l_i denotes the vertex removed from G_i. The rest of the proof is by contradiction. Assume we have a cycle in G, that is to say, there is a path $\tau = v_0 \cdot \ldots \cdot v_{m+1}$ such that $(v_i, v_{i+1}) \in E$ for all index i in $0..m$ and $v_0 = v_{m+1}$. Let v_j be the first vertex in τ to be erased from the graph; that is, there is an index $k \in 0..n$ such that $v_j = l_k$ and all the other states in τ are in G_{k+1}. On one hand, the vertex v_j is well-defined. Indeed, if no vertex from τ is ever erased, then the path τ is also a path in G_n, which contradicts the fact that all the vertices in G_n have in-degree zero. On the other hand, since v_j is a leaf in G_k, this contradicts the fact that there is an edge from v_j to one of the vertex in τ. In consequence, the graph G as no cycles. □

To conclude this section, we study properties of Parental Graphs, that is a spanning subgraph such that all the nodes, except the root(s), have an in-degree of one.

Definition 4.3.2. We say that a directed graph, $PG(V_p, E_p)$, is a parental graph of $G(V, E)$ if: (1) PG is a subgraph of G that has the same vertex set (that is $V_p = V$ and $E_p \subseteq E$) and (2) for every vertex $v \in V$, if v is not the root in G then v has an in-degree of one in PG.

To obtain a parental graph PG, from a directed graph G, it is enough to keep only one edge coming in for every vertex in G and delete the others. Also, if G is acyclic,

then all its parental graphs are acyclic. (Actually, in this case, the parental graph is a spanning tree of the reverse graph.)

The following theorem states an important connection between a graph and its parental graphs: if PG is a parental graph of (the finite directed graph) G, then the set of leaves of PG subsumes the leaves of G. Indeed, a leaf of G is necessarily a leaf of PG, but the opposite may be false. Thus, we can also conclude that G has necessarily some cycles if we fail to find an out-degree zero vertex in PG that is also in G.

In the following sections, we will use the leaves of a parental graph PG as a set of candidates—an approximation—for finding the leaves of G, saving us from testing all the nodes in the graph.

Theorem 4.3.3. *Let G be a finite directed graph and PG be a parental graph of G. If the graph G is acyclic then PG has at least one leaf that is also a leaf in G.*

Proof. Let G be a finite directed graph and PG be a parental graph of G. By Lemma 4.3.1, since G is acyclic, there is at least one leaf in G; Moreover, since PG is a subgraph of G, a vertex of out-degree zero in G must also have out-degree zero in PG (a parental graph has less edges). Therefore the leaf in G is also one of the leaf of PG. □

A corollary of this property is that the relation between G and PG is "stable" when we remove the same leaf from both graphs. This property ensures that it is sound to use a parental graph when we recursively remove all the leaves from a graph.

Corollary. *Assume v is a zero out-degree (leaf) node in G; when we remove the vertex v from PG, we obtain a parental graph of the graph obtained after removing v from G.*

In this chapter, we will essentially work with *reverse graphs* and *reverse parental graphs.*

Definition 4.3.3. The reverse graph of a directed graph $G(V, E)$ is the graph $G^{-1}(V, E^{-1})$ such that the edge (u, v) is in G if and only if the edge (v, u) is in G^{-1}. A reverse parental graph of G is a parental graph of G^{-1}.

4.4 A Model Checking Algorithm with Lazy Cycle Detection

Our parallel algorithms for model checking, named MCLCD (for Model Checking With Lazy Circle Detection), is based on two separate steps: (1) a forward exploration of the

state graph (in collaboration with the state space construction), where we label each state with some "local" information; followed by (2) a backward traversal—and label propagation phase—to check if the resulting graph is a DAG.

In the second step, we do not explicitly look for cycles (like in a "nested-DFS" approach for example). We rather follow a "lazy approach" in order to avoid all the inherent complexities related to the parallel detection of cycles. The second step can be easily implemented in parallel, each processing unit updating the labels of its own states.

A first optimization is to constraint the state space exploration in order to generate only the portion of the state graph that is important to prove or disprove the specification. This approach is quite similar to techniques for on-the-fly model checking because, for some class of formulas, we can sometimes disprove the specification before generating the complete state space. For example, in the case of reachability formulas, such as the invariant formula $A\Box(\phi)$, we will of course stop exploring as soon as we find a state satisfying the predicate $\neg\phi$.

The backward traversal is performed only for safety and liveness formulas; it is not necessary for reachability formulas. We define these different classes of formulas in Figure 4.1 and list, for each formula, whether they involve a backward step.

Formula	Interpretation	Forward	Backward	Classification
$E\ (\psi \cup \phi)$	$E\ (\psi \cup \phi)$	x		Reachability
$A\ (\psi \cup \phi)$	$A\ (\psi \cup \phi)$	x	x	Liveness
$E\Diamond(\phi)$	$E\ (\text{True} \cup \phi)$	x		Reachability
$A\Diamond(\phi)$	$A\ (\text{True} \cup \phi)$	x	x	Liveness
$E\Box(\phi)$	$\neg A\Diamond(\neg\phi)$	x	x	Safety
$A\Box(\phi)$	$\neg E\Diamond(\neg\phi)$	x		Safety
$\psi \leadsto \phi$	$A\Box(\neg\psi \vee A\Diamond\phi)$	x	x	Liveness
$A\Box A\Diamond(\phi)$	$true \leadsto \phi$	x	x	Liveness

Figure 4.1: List of Supported Formulas.

This algorithm is not very original. We already said that this is, basically, the semantic approach initially proposed by Clarke and Emerson (CE82, Cla99) for CTL model checking. The same remark applies to the computation of the "fixed point" in parallel. Our main contribution, from the algorithmic viewpoint, is the definition of

a version of this algorithm based on the reverse parental graph. To the best of our knowledge, this approach is totally new. Indeed, most model checking algorithms for CTL avoid to store the transition relation explicitly. But these approaches always rely on some assumptions about the models, for instance that it is possible to compute the "reverse" transition relation efficiently. We do not make this assumption in our case (this assumption is not valid, for example, with models that mix real-time constraints and data variables). We still define a version of our algorithm based on the reverse transition graph because it is useful to prove the soundness of our method and for studying the theoretical complexity.

From the interpretation of formulas listed in figure 4.1, we see that it is enough to provide a model checking procedure for only three formulas: (reachability) $E\,(\psi \cup \phi)$, (liveness) $A\,(\psi \cup \phi)$, and (leadsto) $\psi \leadsto \phi$. We describe our model checking procedure for each of these three cases.

4.4.1 Notations

We assume that we perform model checking on a Kripke system $KS(S, R, s_0)$. We will use, interchangeably, the notation KS for the Kripke structure (S, R, s_0) and for the directed graph (S, R), also called the state graph.

The expression $|S|$ is used to denote the cardinality of S (and therefore the number of reachable states), while $|R|$ is the number of transitions. Inside asymptotic notations (big O notations) we will simple use the symbols S and R when we really mean $|S|$ and $|R|$.

We assume that every state $s \in S$ is labeled with a value, denoted suc(s), that records the out-degree of s in KS. The value of suc(s) is set during the forward exploration phase. Initially, suc(s) is the cardinality of the set of successors of s in KS, that is suc(s) $= |\{s' \mid s\ R\ s'\}|$. We decrement this label during the backward traversal of the state graph; when the value of suc(s) reaches zero, we say that s is *cleared* from the state graph. In our pseudo-code, we use the expression suc(s).dec() to decrement the value of the label suc for the state s in KS, and the expression suc(s).set(i) to set the label of s to some integer value i.

When we deal with the reverse parental graph version of our algorithm, we assume that we implicitly work with one particular parental graph of KS, denoted PKS. In

this case, we assume that every state $s \in S$ is also labeled with a value, denoted sons(s), that record the out-degree of s in PKS. We also label each state $s \in S$ with a state, denoted father(s), that is the predecessor of s in PKS. (The label father(s) makes sense only if s is not s_0, the initial state of KS.)

Initially, the value of sons(s) is equal to zero. The value of this label will be incremented during the forward exploration of KS, when we build PKS (that is, we select the transitions from KS that will be stored in PKS). This operation is denoted sons(s).inc() in our pseudo-code. We will decrement the value of sons(s) during the backward traversal phase.

4.4.2 Model Checking Reachability properties — $E\,(\psi \cup \phi)$

To check the formula $E\,(\psi \cup \phi)$, we basically search for states satisfying the predicate ϕ in the state graph. More precisely, we stop exploring a path whenever we find a state such that (1) ϕ holds or (2) $\neg\psi \wedge \neg\phi$ holds. In the first case, we can stop the exploration and return that the property is true. Otherwise, we stop the exploration on this path because the property does not hold. The exploration continues over the set of unexplored paths until (1) or (2) holds. The property is false if (1) never holds.

We give the pseudo-code for checking the formula $E\,(\psi \cup \phi)$ in Listing 4.1. The inputs are the atomic properties ψ and ϕ and the initial state (or vertex) s_0. The algorithm uses a stack, W, to store the "working states" (that corresponds to paths that still need to be explored) and a set, S, to store the states that have already been visited. The algorithm returns true as soon as the property ϕ is found, otherwise, it returns false.

The function is the same for the two versions of our algorithm; based on the reverse graph or the reverse parental graph data structure.

4.4.3 Model Checking Liveness Properties — $A\,(\psi \cup \phi)$

To check the formula $A\,(\psi \cup \phi)$, we basically search for states satisfying the predicate ϕ in the state graph. Like for reachability properties, we stop exploring a path whenever we find a state such that (1) ϕ holds or (2) $\neg\psi \wedge \neg\phi$ holds.

If we find an occurrence of case (2), we now at once that the property is false. In the other case, we start a second phase, after the forward exploration is over, in order to detect cycles. We call this second phase the *clearing phase*, because it consists in

```
1   function BOOL check_e(ψ : pred, φ : pred, s0 : state)
2     Set S ← new Set(∅) ;
3     Stack W ← new Stack(s0) ;
4     while (W is not empty) do
5       s ← W.pop() ;
6       if (s ⊨ φ) then
7         return TRUE
8       elsif (s ⊨ ψ) then
9         forall s' successor of s in KS do
10          if (s' ∉ S) then
11            // s' is a new state
12            S ← S ∪ {s'} ;
13            W.push(s')
14          endif
15        endfor
16      endif
17    endwhile ;
18    return FALSE
```

Listing 4.1: Algorithm for the formula $E\,(\psi \cup \phi)$

recursively removing the leaves node from the graph. This process ends either when we finally reach the initial state (which means the property is true), or when no states with zero out-degree can be found (in which case we know that there is a cycle). The validity of this process is a direct corollary of Theorem 4.3.2.

To give an example of how this algorithm works, we can consider the case of the formula $A\Diamond(\phi)$ (that is $A\,(\mathrm{True}\cup\phi)$). In the first step, we stop exploring a path whenever we find a state where ϕ holds. The forward exploration ends when all possible paths have been explored and none of them violates the constraints, i.e., ϕ eventually holds for all paths. Then we start removing the nodes with out-degree zero from the graph. The first "leaves" in the graph are necessarily states where the predicate ϕ holds. Next, we also remove states that have had all their successors removed. We can give a simple explanation of the validity of this method. Indeed, at each step we remove states belonging to the set X defined by the following recursive equation: a state s is in X if either (1) $s \models \phi$, or (2) all the successors of s are in X. Basically, we compute the semantics of the modal μ-calculus formula $\mu X.(\phi \vee [\mathrm{True}]X)$, that is equivalent to the LRL formula $A\Diamond(\phi)$.

We give the pseudo-code for checking the formula $A\,(\psi \cup \phi)$ in Listing 4.2. Like in the previous case, the inputs are the atomic properties ψ and ϕ and the initial state s_0.

```
1  function BOOL check_a(ψ : pred, φ : pred, s0 : state)
2      Stack A ← new Stack(∅) ;
3      // Start with the forward exploration
4      if forward_check_a(ψ , φ, s0, A) then
5          // If all forward constraints are respected, start the backward phase
6          return backward_check_a(s0, A)
7      else
8          // We found a problem during the forward exploration
9          return FALSE
10     endif
```

Listing 4.2: Algorithm for the formula $A(\psi \cup \phi)$

The algorithm uses a stack, A, to collect the states where ϕ holds during the forward exploration phase. The algorithm also uses two auxiliary functions, forward_check_a and backward_check_a, that are defined later. The implementation of these two functions depend on the version of the algorithm that we use. We start by studying the case where we use the *Reverse Graph* data structure and then the *Reverse Parental Graph*.

Algorithm for the reverse graph version — RG

We give the pseudo-code for the function forward_check_a in Listing 4.3. The last parameter of this function, A, is a stack that is used to collect the "leave nodes" of the state graph; the states where ϕ holds. These states will be the starting points in our backward traversal of the graph.

The function forward_check_a basically performs the same operations than the function check_e of Section 4.4.2, with some minor differences. More precisely, the function does not return when we find a state where ϕ holds. Instead, we continue our exploration on other paths of the state graph. On the opposite, we stop the exploration and return false whenever we find a state where both ϕ and ψ do not hold or a dead state where ψ holds.

During the forward exploration phase, we label each state s with a value that is the number of successors of s in the initial state graph (the Kripke structure). During the backward traversal phase of the algorithm, we will decrement this value each time we remove a successor of s. Intuitively, a state can be removed as soon as it is tagged with zero. We never actually remove a state from the graph. Instead, when a processor

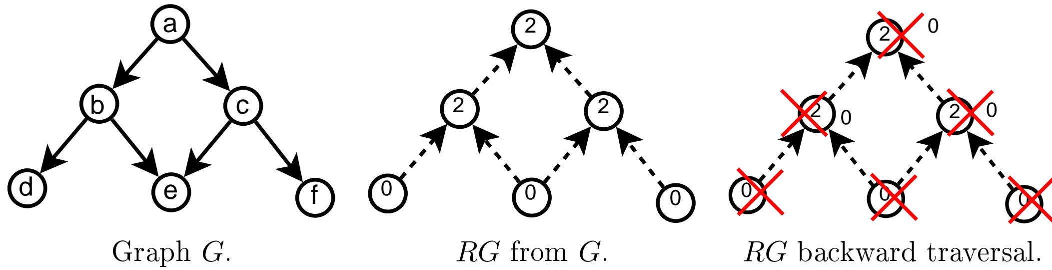
```
1  function BOOL forward_check_a(ψ : pred, φ : pred, s0 : state, A : Stack)
2      Set S ← new Set(∅) ;
3      Stack W ← new Stack(s0) ;
4      while (W is not empty) do
5          s ← W.pop();
6          if (s ⊨ φ) then
7              // we clear state s from KS
8              suc(s).set(0) ;
9              A.push(s)
10         elsif (s ⊨ ψ) then
11             // we tag s with its number of successors
12             suc(s).set(number of successors of s in KS) ;
13             // check if s is not a dead state
14             if (suc(s) = 0)
15                 return FALSE
16             endif
17             // and continue the exploration
18             forall s' successor of s in KS do
19                 if (s' ∉ S) then
20                     // s' is a new state
21                     S ← S ∪ {s'} ;
22                     W.push(s')
23                 endif
24             endfor ;
25         else return FALSE
26         endif
27     endwhile ;
28     return TRUE
```

Listing 4.3: Forward exploration for the formula $A(\psi \cup \phi)$ with Reverse Graph

changes the label of a state s to 0[1], we also decrement the labels of all the parents of s in the graph. Hence the choice of storing the reverse of the transition function in the data structure.

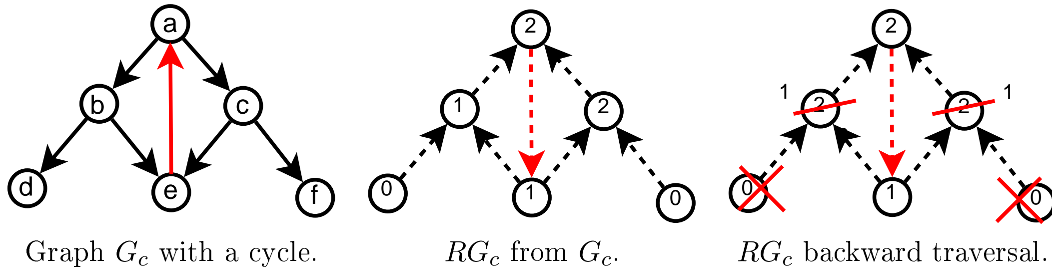
As an example, we show the result of a backward exploration for two Kripke structures, G and G_c in Figure 4.2 and 4.3. The graph G_c is obtained by adding an edge to G (pictured in red) that results in the presence of a cycle. In each figure, we show: (1) the graph; (2) the resulting reverse graph with the initial labeling of each node; and (3) the result of the clearing phase. In the last case, a red cross means that the state is cleared, while a simple mark is used for states that have been updated.

Figure 4.2 gives an example of a successful backward traversal, while figure 4.3 gives an example of an unsuccessful clearing phase.



Graph G. RG from G. RG backward traversal.

Figure 4.2: Successful Reverse Graph backward traversal for $A\,(\psi \cup \phi)$.



Graph G_c with a cycle. RG_c from G_c. RG_c backward traversal.

Figure 4.3: Unsuccessful Reverse Graph backward traversal for $A\,(\psi \cup \phi)$.

Listing 4.4 gives the pseudo-code for the function backward_check_a, that implements the clearing phase. We start by clearing all the states in A which are, by construction, states s such that suc(s) is zero. When a state is cleared, we decrement the label of all its parents ($suc(s').dec()$) and check which ones can be cleared ($suc(s') == 0$).

[1] We assume that the decrementing operations are done in parallel.

```
1  function BOOL backward_check_a(s0 : state, A : Stack)
2     while (A is not empty) do
3        s ← A.pop() ;
4        // the property is true if we reach the initial state
5        if (s = s0) then
6           return TRUE
7        endif
8        // otherwise we check if the predecessors of s can be cleared
9        forall s' parent of s in KS do
10          suc(s').dec() ;
11          if (suc(s') = 0) then
12             A.push(s')
13          endif
14       endfor
15    endwhile ;
16    return FALSE
```

Listing 4.4: Backward exploration for the formula $A(\psi \cup \phi)$ with Reverse Graph

The algorithm stops if the initial state, s_0, can be cleared or if there are no more state to update.

Algorithm for the reverse parental graph version — RPG

We give the pseudo-code for the forward exploration function, forward_check_a, in the case of the parental graph version (see Listing 4.5).

The difference, in this case, is that we only have access to the reverse parental graph data structure. As a consequence, we can only access one of the parents of a state in constant time (what we call the father of the state). In our implementation, we choose as father for a state s', the first state, say s, that leads to s' in the exploration. But the algorithm could work with any other choice.

For the clearing phase, we rely on the parental graph structure to "propagate" the cleared states toward the root of the state graph. We give the pseudo-code for the backward traversal phase in Listing 4.6. The algorithm iterates between two behaviors, *clearing* and *collecting*. The *clearing* behavior is similar to the pseudo-code for the RG algorithm (see Listing 4.4), with the difference that we decrement only the father of a state and not all the predecessors. When there are no more labels to decrement—and if the initial root is not yet cleared—the algorithm starts looking for states that can be cleared. For this, we test all the states s such that $sons(s) == 0$; that is, such that all

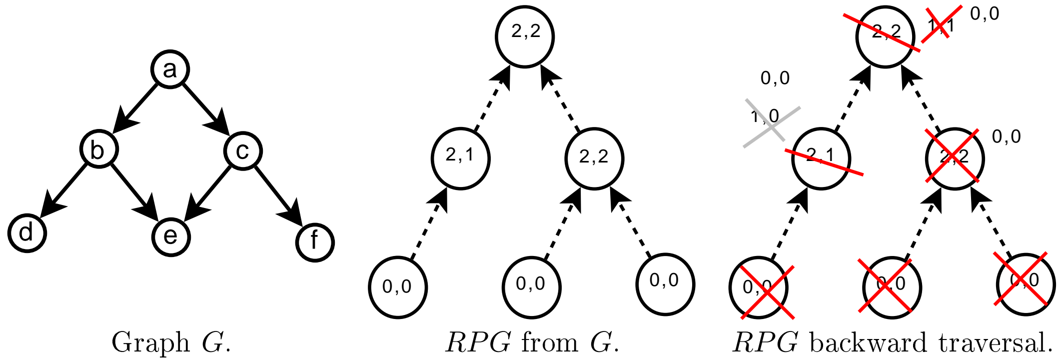
```
1  function BOOL forward_check_a(ψ : pred, φ : pred, s0 : state, A : Stack)
2     Set S ← new Set(∅) ;
3     Stack W ← new Stack(s0) ;
4     while (W is not empty) do
5        s ← W.pop() ;
6        if (s ⊨ φ) then
7           suc(s).set(0) ;
8           A.push(s)
9        elsif (s ⊨ ψ) then
10          suc(s).set(number of successors of s in KS) ;
11          // check if s is not a dead state
12          if (suc(s) = 0)
13             return FALSE
14          endif
15          forall s' successor of s in KS do
16             if (s' ∉ S)
17                // then s is the father of s' PKS
18                S ← S ∪ {s'} ;
19                father(s').set(s) ;
20                sons(s).inc();
21               W.push(s')
22             endif
23          endfor ;
24       else return FALSE
25       endif
26    endwhile ;
27    return TRUE
```

Listing 4.5: Forward exploration for the formula $A(\psi \cup \phi)$ with Reverse Parental Graph

the sons of s have been cleared. In this case, to check if s can be cleared, we have to recompute all its successors in KS (because this information is not stored in the RPG) and to check whether they have been cleared also (if their suc label is zero).

The advantage of this strategy is that we do not have to consider all the states in the graph, just a subset of it. Indeed, we know from Theorem 4.3.3 that this subset is enough to test the presence of a cycle. At the opposite, the drawback of this approach is that we may try to clear the same vertex several times, which may be time consuming.

We show the result of the backward propagation phase for two different cases in figures 4.4 and 4.5. In each diagram, the vertices are decorated with the pair of labels $(\text{suc}, \text{sons})$.



Graph G. RPG from G. RPG backward traversal.

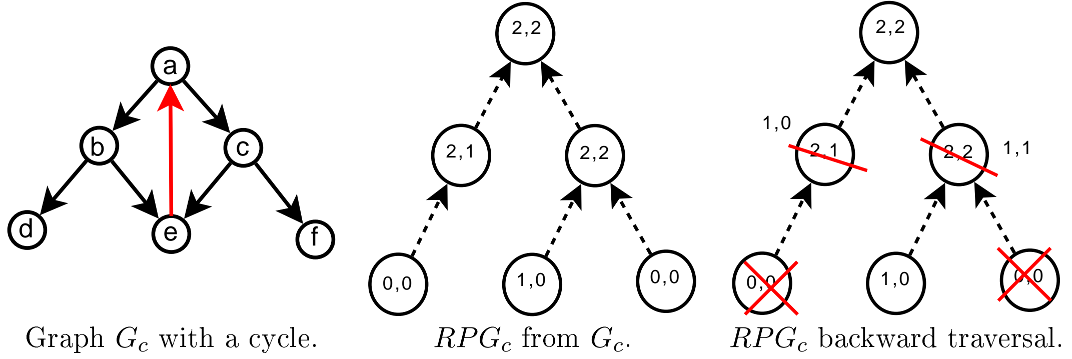
Figure 4.4: Successful Reverse Parental Graph backward traversal for $A\,(\psi \cup \phi)$.

Figure 4.4 gives an example of a successful backward traversal. Initially, the stack A contains the states d, e, f. After they are removed, we have $\text{sons}(b) = 0$, $\text{suc}(b) = 1$ and $\succ(c) = \text{sons}(c) = 0$. Then state c can be cleared. At this point, we have only one zero in-degree node, b, in the reverse parental graph. We need to recompute one transition to retrieve the fact that e is a successor of b. Since e was already removed, we can finally clear b and then reach the initial state a.

We give an example of unsuccessful backward traversal in Figure 4.5. Unlike the previous case, when the backward propagation ends, we have two zero in-degree node in the parental graph, namely b and e. Since $\text{suc}(b) = \text{suc}(e) = 1$, we cannot clear any of these states. Therefore the process ends and, by Theorem 4.3.3, we know that there must be a cycle in G_c.

```
1  function BOOL backward_check_a(ψ : pred, φ : pred, s0 : state, A : Stack)
2      over ← FALSE
3      while (not over)
4          while (A is not empty) do
5              //Clearing
6              s ← A.pop() ;
7              // the property is true if we reach the initial state
8              if (s = s0) then
9                  return TRUE
10             endif ;
11             // otherwise we check if the father of s can be cleared
12             s' ← father(s) ;
13             sons(s').dec() ;
14             suc(s').dec() ;
15             if (suc(s') = 0) then
16                 A.push(s')
17             endif
18         endwhile
19         //Collecting
20         // if we have no more states to clear in A we try to find
21         // candidates among the states with no children in PKS
22         forall s such that sons(s) = 0 and suc(s) ≠ 0 in KS do
23             if test(s) then
24                 suc(s).set(0) ;
25                 A.push(s)
26             endif
27         endforall
28         if (A is empty) then
29             //No good candidate was found, end backward search
30             over ← TRUE
31         endif
32     endwhile ;
33     return FALSE
34
35 function BOOL test(s : state)
36     forall s' successor of s in KS do
37         if suc(s') ≠ 0 then
38             // at least one successor is not cleared
39             return FALSE
40         endif
41     endfor
42     return TRUE
```

Listing 4.6: Backward exploration for $A(\psi \cup \phi)$ with Reverse Parent Graph



Graph G_c with a cycle. RPG_c from G_c. RPG_c backward traversal.

Figure 4.5: Unsuccessful Reverse Parental Graph backward traversal for $A\,(\psi \cup \phi)$.

4.4.4 Model Checking the Leadsto Property — $\psi \leadsto \phi$

To check the formula $\psi \leadsto \phi$, we need to prove that there is no cycle that can be reached from a state where ψ holds, without first reaching a state where ϕ holds. Indeed, otherwise, we can find an infinite path where ϕ never holds after an occurrence of ψ. Figure 4.6 gives an example of graph for which the formula is valid.

This observation underlines the link between checking the formula $\psi \leadsto \phi$—locally, for the initial state—and checking the validity of $A\Diamond(\phi)$—globally, at every state where ψ holds. As a consequence, we can use an approach similar to the one used for liveness properties in the previous section. The main difference is that, instead of clearing the initial state, we have to clear all the states where ψ hold.

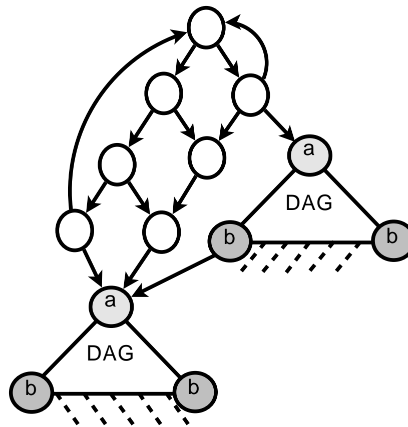


Figure 4.6: Leadsto $a \leadsto b$ where a is ψ and b is ϕ.

We give the pseudo-code for checking the formula $\psi \leadsto \phi$ in Listing 4.7. Compared

```
1  function BOOL check_leadsto(ψ : pred, ϕ : pred, s0 : state)
2     Stack A ← new Stack(∅) ;
3     Stack P ← new Stack(∅) ;
4     forward_check_leadsto(ψ , ϕ, s0, A, P) ;
5     return backward_check_leadsto(A, P)
```

Listing 4.7: Algorithm for the formula $\psi \leadsto \phi$

```
1  function void forward_check_leadsto(ψ : pred, ϕ : pred,
2                                      s0 : state, A : Stack, P : Stack)
3     Set S ← new Set(∅) ;
4     Stack W ← new Stack(s0) ;
5     while (W is not empty) do
6        s ← W.pop();
7        if (s ⊨ ϕ) then
8           suc(s).set(0) ;
9           A.push(s)
10       else
11          suc(s).set(number of successors of s in KS) ;
12          if (s ⊨ ψ) then
13             P.push(s)
14          endif ;
15          forall s' successor of s in KS do
16             if (s' ∉ S) then
17                S ← S ∪ {s'} ;
18                W.push(s')
19             endif
20          endfor ;
21       endif
22    endwhile ;
23    return void
```

Listing 4.8: Forward exploration for the formula $\psi \leadsto \phi$ with Reverse Graph

to the algorithm for liveness, we use an additional parameter, P, that is a stack where we collect all the states such that ψ holds. Another difference is that we need to explore the state graph totally (we say that leadsto is a global property).

Algorithm for the reverse graph version — RG

We give the pseudo-code for the function forward_check_leadsto in Listing 4.8. The function adds to the stack A (resp. P) all the state where ϕ (resp. ψ) holds. During the forward exploration, we also set the states in A as cleared (we set the label suc to zero) because they will be the starting points for our backward traversal.

```
1  function BOOL backward_check_leadsto(A : Stack, P : Stack)
2     while (A is not empty) do
3        s ← A.pop() ;
4        forall s' parent of s in KS do
5           suc(s').dec() ;
6           if (suc(s') = 0) then
7              A.push(s')
8           endif
9        endfor
10    endwhile ;
11    while (P is not empty) do
12       s ← P.pop() ;
13       if (suc(s) ≠ 0) then
14          return FALSE
15       endif
16    endwhile ;
17    return TRUE
```

Listing 4.9: Backward exploration for $\psi \leadsto \phi$ with Reverse Graph

We give the pseudo-code for the backward traversal in Listing 4.9. Again, the code is similar to the backward traversal for the liveness case. The main difference is in the termination condition: the function returns true if all the states in P have been labeled as cleared.

Algorithm for the reverse parental graph version — RPG

The algorithm for forward and backward analysis with the Reverse Parental Graph are similar to the previous cases.

Backward traversal (see figure 4.11) starts from the *accepted* vertices and terminates when there is no more vertex to clear (set its out-degree to zero). It returns true if all the *seeds* vertices are cleared. Its implementation is similar to the pseudo-code presented at figure 4.6, the algorithm iterates between two phases: *clearing* and *collecting*.

4.5 Correctness and Complexity of our Algorithms

In this section, we give a correctness proof for the MCLCD algorithm. We also study the complexity of our algorithm in the sequential case. We more specifically study the case

```
1  function void forward_check_leadsto(ψ : pred, φ : pred,
2                                       s0 : state, A : Stack, P : Stack)
3    Set S ← new Set(∅) ;
4    Stack W ← new Stack(s0) ;
5    while (W is not empty) do
6      s ← W.pop();
7      if (s ⊨ φ) then
8        A.push(s) ;
9        suc(s).set(0)
10     else
11       suc(s).set(number of successors of s in KS) ;
12       if (s ⊨ ψ) then
13         P.push(s)
14       endif ;
15       forall s' successor of s in KS do
16         if (s' ∉ S) then
17           S ← S ∪ {s'} ;
18           father(s').set(s) ;
19           sons(s).inc() ;
20           W.push(s')
21         endif
22       endfor
23     endif
24   endwhile ;
25   return void
```

Listing 4.10: Forward exploration for $\psi \leadsto \phi$ with Reverse Parent Graph

```
1   function BOOL backward_check_leadsto(ψ : pred, φ : pred,
2                                        s0 : state, A : Stack, P : Stack)
3     over ← FALSE
4     while (not over)
5       while (A is not empty) do
6         //Clearing
7         s ← A.pop() ;
8         // we check if the father of s can be cleared
9         s' ← father(s) ;
10        sons(s').dec() ;
11        suc(s').dec() ;
12        if (suc(s') = 0) then
13          A.push(s')
14        endif
15      endwhile
16      //Collecting
17      // if we have no more states to clear in A we try to find
18      // candidates among the states with no children in PKS
19      forall s such that sons(s) = 0 and suc(s) ≠ 0 in KS do
20        if test(s) then
21          suc(s).set(0) ;
22          A.push(s)
23        endif
24      endforall
25      if (A is empty) then
26        //No good candidate was found, end backward search
27        over ← TRUE
28      endif
29    endwhile ;
30    // the property is true if all the state in P are cleared
31    while (P is not empty) do
32      s ← A.pop() ;
33      if (suc(s) ≠ 0) then
34        return FALSE
35      endif ;
36    endwhile ;
37    return TRUE
38
39  function BOOL test(s : state)
40    forall s' successor of s in KS do
41      if suc(s') ≠ 0 then
42        // at least one successor is not cleared
43        return FALSE
44      endif
45    endfor
46    return TRUE
```

Listing 4.11: Backward exploration for $\psi \rightsquigarrow \phi$ with Reverse Parent Graph

of the liveness formula $A\,(\psi \cup \phi)$. The results obtained for this case can be generalized to our whole logic.

The correctness of our algorithm is based on the fact that the computation will stop (and return the boolean value FALSE) if there is at least one cycle in KS, that is to say, we cannot perform a complete backward traversal and reach the initial state.

Theorem 4.5.1 (Termination)**.** *The MCLCD algorithm, for model checking the logic LRL on a finite Kripke Structure, terminates for all inputs.*

Proof. We only consider the case for the formula $A\,(\psi \cup \phi)$, that is the function check_a. The other cases are similar. We prove the termination of check_a by proving the termination of the two functions that it calls: forward_check_a and backward_check_a.

The function forward_check_a (see for instance Listing 4.3 for the RG case) generates a subset of the state graph by pruning all the transitions after a state where ϕ holds. At each iteration of the function, we consider a different state taken from a stack—denoted S in the pseudo-code—that contains a subset of the reachable states. Since the state graph is finite, the function will always terminate.

For the function backward_check_a, termination follows from the fact that, at each iteration, we decrement the value of at least one of the labels suc(s). □

Theorem 4.5.2 (Completeness)**.** *If the system KS satisfies the formula F then the MCLCD algorithm returns the boolean value TRUE when run with the value (KS, F).*

Proof. We only consider the case for the formula $A\,(\psi \cup \phi)$. The proof is essentially the same for the RG and RPG cases. (The two algorithms have a similar behavior; only their complexity differ.) We assume that $KS \vDash F$ and study the result of the expression check_a(ψ, ϕ, s_0).

By the definition given in Section 4.2, we have $KS \vDash A\,(\psi \cup \phi)$ if and only if (INV): for all maximal path $\tau = s_0 \cdot s_1 \cdot \ldots$ in KS there is an index i such that $s_i \vDash \phi \wedge \forall j[0 \leq j < i \Rightarrow s_j \vDash \psi]$.

First, we prove that the call to forward_check_a(ψ, ϕ, s_0, A) returns TRUE. The proof is by contradiction. Let us assume that the forward exploration returns FALSE (we know that the forward exploration terminates). Hence, there must be a path $\tau = (s_i)_{i \in 1..n}$ such that $s_i \vDash \psi \wedge \neg\phi$ for all $i < n$ and $s_n \vDash \neg\psi \wedge \neg\phi$. Since we can always extend this path into a maximal path of KS, this contradicts (INV). Therefore, the forward exploration must succeeds.

In the remainder of this proof, we assume that KS_f denotes the subset of the state graph generated during the forward exploration and that A_{init} is the set of states s

such that suc(s) = 0. It is exactly the states with out-degree zero in KS_f and the set of states in KS_f where ϕ holds. By construction, since at least one state must satisfy ϕ, the set A_{init} is not empty. Also, the parameter A is set to A_{init} when we start the backward exploration phase.

Now, we consider the backward exploration for the reverse graph case. Let A_i denotes the set of states that are cleared in KS_f after the ith iteration of the while loop in the call to backward_check_a. The sequence of sets $(A_i)_{i \in 1..n}$ can be defined by the following induction: (1) $A_0 = A_{init}$, and (2) for all $i > 0$, we have $A_{i+1} = A_i \cup \{s \| \forall s'.s \; R \; s' \Rightarrow s \in A_i\}$. Since it is an increasing sequence, bounded by the finite set S, it finally reaches a limit, $A_f \subseteq S$. If we use the terminology defined in Sect. 4.3, $S \setminus A_f$ are exactly the states that are left in KS_f when we recursively remove all the cleared states.

To prove that the backward exploration also succeeds, it is enough to show that $KS \vDash A(\psi \cup \phi)$ entails $s_0 \in A_f$. From (INV), we have that KS_f must be a DAG. Otherwise we could find a maximal path in which ϕ never holds. Therefore, by Theorem 4.3.2, we have $s_0 \in A_f$, as needed. □

Theorem 4.5.3 (Soundness)**.** *If the MCLCD algorithm returns TRUE for a given pair (KS, F) of a Kripke Structure and a LRL formula then $KS \vDash F$.*

Proof. Once again, we only give the proof in the case where F is the formula $A(\psi \cup \phi)$. The proof is essentially the same for the RG and RPG cases. (The two algorithms have a similar behavior; only their complexity differ.)

Let us assume that the call to check_a(ψ, ϕ, s_0) returns TRUE. Then the forward and backward exploration must both succeeds.

We assume that KS_f is the subset of the state graph computed during the forward exploration. By construction, the predicates ψ holds for all the states in KS_f except for its leaves, where ϕ holds.

Next, we show that if the backward exploration succeeds then KS_f is a DAG. The proof is the same than with Theorem 4.5.2 and makes use of the other part of the equivalence given by Theorem 4.3.2

Since KS_f is a DAG, all the maximal paths in KS_f must eventually reach a state where ϕ holds. Therefore, since each maximal path in KS is necessarily an extension of a maximal path in KS_f, the same property is true with KS, which means that $KS \vDash A(\psi \cup \phi)$. □

Next, we study the worst-case complexity of our algorithms. We obtain different results for the RG and RPG versions of the MCLCD algorithm. (Recall that, inside

asymptotic notations, we use the symbols S and R when we really means $|S|$ and $|R|$.)

Theorem 4.5.4 (Complexity)**.** *The worst-case time complexity of the algorithm is in the order of $O(S + R)$ for the RG version and in the order of $O(S \cdot (R - S))$ for the RPG version.*

Proof. In the worst-case, we need to perform both a forward and a backward exploration.

The complexity of the forward exploration is trivially bounded by the size of the state space: in the worst-case, we need to explore all the states and test all the transitions. Hence the complexity of this first phase is (linear) in $O(S + R)$.

The complexity of the backward exploration depends on the underlying data structures used to encode the Kripke structure. For each version of our algorithm, the worst case is when the property is true.

If we rely on a reverse graph structure—and if we assume that the property is true—then we need to clear all the states in the graph and, for every transition (edge) in the reverse graph, we need to update the label of its head. Therefore, the complexity of the backward exploration is also $O(S+R)$ in the worst case. That is, the overall complexity is in $O(2 \cdot (S + R))$ for the RG version of our algorithm. This result is consistent with the complexity of the CTL model checking algorithm defined in (CES86), that has a time complexity of $O(|\phi| \cdot (S + R))$, where $|\phi|$ is a measure of the complexity of the specification. Indeed, in our case, the most complex formula (the leadsto property) has complexity 2.

The analysis is quite similar for the version based on the reverse parental graph. The main difference is that we may have to recompute some transitions in KS several time. At least, we need to recompute all the transitions whose "inverse" is not stored in the reverse parental graph.

Since a Kripke structure is a weakly connected graph, we have that $|R| > |S|$ and that $|R| - |S| + 1$ is a bound to the number of transitions not stored in the RPG. At each iteration, we need to test the successors of the states, s, such that $\text{sons}(s) = 0$. This is in order to find the zero out-degree node in the Kripke structure, that is, the states s such that $\text{suc}(s) = 0$. In the worst case, we may have to re-compute all the transitions that are not stored in RPG.

We clear at least one state and remove at least one transition at each iteration of the backward check function. Therefore, there is at most $|S| - 1$ iterations and, at the ith iteration, there is at most $|S| - i$ states that are not cleared and $|R| - |S| - i + 1$ transitions that may need to be re-computed. Thus, the complexity of our algorithm, for the ith iteration, is bounded by $(|S| - i) + (|R| - |S| - i + 1)$. As a consequence,

the complexity of the backward traversal can be bounded by the following expression, which is in $O(S \cdot (R - S))$.

$$\begin{aligned}\sum_{i \in 1..|S|-1} (|S| - i) + (|R| - |S| - i + 1) &= 1/2.|S|.(|S| - 1) + (|R| - 3/2.|S| + 1).(|S| - 1) \\ &= (|R| - |S| + 1).(|S| - 1)\end{aligned}$$

□

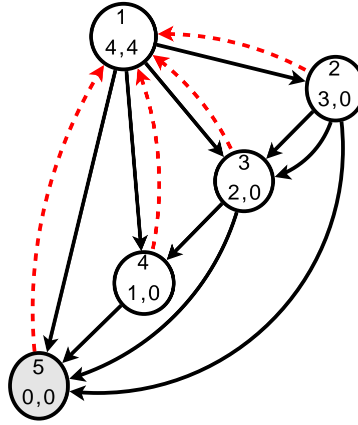


Figure 4.7: Worst-Case Example for the RPG Version (edges in red are in the reverse parental graph).

Since the number of transitions is bounded by $|S|^2$, we obtain a complexity in the order of $O(S^2)$ for the RG version and of $O(S^3)$ for the RPG version. We can show that this is an asymptotic tight bound for the complexity of the backward exploration.

In the case of the RPG version, we give an example of state graphs such that the complexity is asymptotically equivalent to S^3 (up to a constant). In Fig. 4.7, we give an example of a state graph, with 5 states, that fall in the "worst complexity" case. Following the same structure than in this example, we can build a family of graphs K_N with N states and $1/2.N.(N-1)$ transitions (for any $N > 2$); K_N is a directed graph with N vertices that has the maximum possible number of edges for a DAG.

The backward exploration on K_N will require $N - 1$ iterations. Actually, when we remove the only leave in the graph K_N we obtain the graph K_{N-1}. A more precise analysis of the behavior of the backward exploration of the graph K_N gives a complexity

that is a factor of $N^3 + O(N)$; if we denote $C(N)$ the complexity for the graph K_N then we have the relation $C(N+1) = 1/2.N.(N-1) + N + C(N)$.

In conclusion, we have shown that the RG version of the algorithm has a time complexity that is a factor of $|S|$ better than the RPG version. At the same time, the space complexity is better for the RPG version than for the RG version by the same factor: the space complexity is in $O(S)$ for the RPG version and in $O(S^2)$ (or $O(S+R)$) for the RG version. Note that this is a complexity result for the worst-case, obtained using a family of graphs with unbounded degree. (The maximal degree of graph K_N is $N-1$.) In particular, we can expect a smaller difference in time complexity if we consider that the out-degree of the state space is bounded.

4.6 Parallel Implementation of our Algorithm

While we consider a Random Access Machine (RAM) model for the complexity results given in the previous section, we have not specifically fixed the abstract computational model that is used to interpret the semantics of our pseudo-code. Most particularly, we can easily adapt the same code to a Parallel RAM model, following the Single Program Multiple Data (SPMD) programming style that we adopted for our algorithms in Chapter 3.

In a SPMD context, all processing units will execute the same functions, as defined in Sect. 4.4. Following this approach, the forward exploration phase and the cycle detection (backward traversal) phase can both be easily parallelized. Then, for the model checking function themselves—for instance the function check_a—we only need to synchronize the termination of the forward exploration with the start of the backward label propagation. At each point, a processing unit can terminate the model checking process if he can prove (or disprove) the validity of the formula before the end of the exploration phase.

We consider a shared memory architecture where all processing units will share the state space (using the mixed approach that we introduced in Section 3.3) and where the working stacks are partially distributed (such as the stacks W, A and P used in our pseudo-code). For most of our pseudo-code, it is enough to rely on atomic "compare and swap" primitives to protect from parallel data races and other synchronization issues; typically, compare-and-swap primitives will be used when we need to test the value of a

label or when we need to update the label of a state (for instance with expressions like sons(s).dec()). Together with the compare-and-swap primitive, we use our combination of distributed, local hash tables with a concurrent localization table to store and manage the state space.

For the RG version of the algorithm, we can ensure the consistency of our algorithm by protecting all the operations that manipulate a state label. (We made sure, in our pseudo-code, that every operation only affects one state at a time.)

The parallel version of RPG is a bit more complicated. This problem is related to the behavior *collection*, that needs to check all the successors of a given state to see if they are cleared. First, this operation is not atomic and it is not practical to put it inside a critical section (it would require a mutex for every state). If two processors collect the same state, then the father of this state will be decremented two times (later) at the *clearing* procedure. Second, the *collection* operation must be performed after all processors have finished the *clearing* operation, otherwise, Theorem 4.3.3 can not be applied to our algorithm. For instance, if the processors are allowed to perform asynchronously both *clearing* and *collection* operations, then a state may be forgotten to be collected because one of its successors has not been cleared yet.

We solve the parallel issues for RPG through the synchronization of all processors before both *clearing* and *collection* operations. The synchronization ensures that no states will be forgotten to be collected. Then, we take advantage of our distributed local hash tables to avoid the concurrent access problem. Each processor is restricted to perform the *collection* operation over the states stored in its own table.

To conclude with the parallel version of our algorithm, we use a work-stealing strategy (see Sect. 3.1.1) to balance the work-load between the different phases of our algorithm. During the exploration phase, we use the same strategy than in our algorithm for parallel state space construction, where each processor holds two stacks for unexplored states (one private stack and one shared stack). For the backward traversal, we use the same idea of two stacks for the accepted vertices (the stack called A in our pseudo-code); whenever a thread has no more vertex to clear, it tries to “steal” non-cleared vertices from other processors.

4.7 Experimental Results

We have implemented several versions of our model checking algorithm as part of our prototype model checker MERCURY (Appendix B). They are built on top of our previous algorithm for parallel state space exploration (see Sect. 3.3). We basically follow the same guidelines than in our implementation of state space generation: we use the C language with Pthreads (But97) for concurrency and the Hoard Library (BMBW00) for parallel memory allocation. We use our *Localization Table* to store the set of explored states. Experimental results presented in this section were obtained on a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208 GB of RAM memory, running the Solaris 10 operating system.

For this benchmark, we selected some of the models already used in Section 3.3.3. Figure 4.8 lists the formulas and models selected for our experiments. We choose the Dining Philosophers (PH), the Token Ring (TK), the Peg-Solitaire (Peg) and the Sokoban (SK) models in order to have different types of state graphs for the verification.

The PH and TK benchmarks are classical models that are very well-suited for verification methods based on partial order techniques due to their high degree of interleaving and symmetries. The puzzle models (Peg and SK) have an opposite behavior and have graphs structures that "are almost" DAG.

For each model, we list the formulas that have been checked and give an informal description of the specification. In each case, we try to have a mix of valid and invalid properties. We experimented with all the formulas: reachability ($E\Diamond\phi$), safety ($A\Box\phi$ and $E\Box\phi$), liveness ($A\Diamond\phi$) and leadsto ($\psi \leadsto \phi$). Actually, our tool uses an equivalent ASCII syntax: `E<>`, `A<>`, `E[]`, `A[]`, and `==>` for the modalities in LRL and `- phi` for the negation on predicates. We also use a special predicate, `dead`, to denote the states without successors.

For the model TK, we experimented with two different versions: one that is simply called TK, which is the classical Token Ring example, where starvation is possible—a process can be perpetually denied access to the service ("token")—and a second version, that we call TK_M, which is a modified model that avoids the resource starvation problem. Similarly to our previous analysis, all these examples are based on finite state systems modeled using Petri Nets (Mur89).

The rest of the section is divided in two main parts. First, we perform a speedup analysis of our model checking program. We also try to analyze how our new approach for detecting cycles participate to the total speed-up of the method. Finally, we present a broader comparison of the two solutions proposed in this chapter with a third variant where we do not store the transitions, that is to say, only the states are stored and the reverse transition relation is are re-computed dynamically during the backward phase.

4.7.1 Speedup Comparison Between RG and RPG Algorithms

We analyze the speedup of our parallel model checking algorithm for the benchmark described in Figure 4.8. We give the relative speedup and the execution time for the reverse and reverse parental graph versions our algorithm. In addition, we also give the separate speedup obtained in each phase of the algorithm—during the exploration (forward) and cycle detection (backward) phases—in order to better analyze the advantages of our approach.

Experimental Results for the Dining Philosophers Model — PH

We give the speedup analysis for the Dining Philosophers (PH) in Figures 4.9 and 4.10: we display the speedup and the execution time for different configurations, from 1 to 16 processors, and for both versions of our algorithm. Each diagram has one line chart for each kind of formulas. The results are fairly good, with an average efficiency of 65% for the reverse version and 56% for the parental version.

We can explain the abnormal behavior for the reachability formula $E <>$ by the fact that the algorithm is very fast in this case—it takes less than one second to finish in average—and therefore the results are prone to experimental fluctuations. (We can use Fig. 4.23 to compare the execution time for the different formulas.)

Although we could expect the parental algorithm to have a very good speedup—it has more opportunities to benefit from the parallelism—the relative speedup for the RPG version is not as good as the one obtained by the RG version. This surprising result is partially linked to the fact that the sequential execution time (execution time on one processor) is significantly lower for the parental version than for the reverse version. On of the reasons explaining this behavior is that the forward exploration phase of the RPG algorithm is much faster (there is less information to write in memory).

We try to pinpoint more precisely what is the impact of our two different versions on the performance in Figure 4.11. We give several bar charts where we decompose the speedup into three separate values: the speedup for the forward exploration phase, the speedup for the backward exploration phase (cycle detection), and finally the overall speedup. For a better view of the impact of each phases, we display the "cumulative" execution time in Fig. 4.23, obtained when checking the PH model with 16 processors.

For the PH model, we can observe that, in both variants, the speedup for the exploration phase is: (1) much more important than the one obtained in the cycle detection phase; and (2) almost equal or slightly better than the total speedup. We remind the reader that there is no cycle detection phase when we model-check the formulas `E<>` or `A[]`.

Concerning the RG algorithm (the top graph from Fig. 4.11), we observe that the speedup for the cycle detection phase is very poor for the PH model. This can be explained by the fact that this phase is very short (we can see in Fig. 4.23 that the time spent in this phase is negligible compared to the time spent in the forward exploration) and do not create enough work for several processors; which means that our work stealing strategy has no effect. The results are more interesting for the Peg model, that we study later.

Concerning the RPG algorithm (the lower graph in Fig. 4.11), we observe better speedups for the cycle detection phase, with values slightly superior to 4 when using 16 processors. We also observe that, for more than 10 processors, the execution time of the RPG and the RG versions are almost identical (see Fig. 4.9 and 4.10). Like with the RG version, the speedup of the algorithm for the PH model mostly comes from the forward exploration phase. In particular, we observe that the efficiency of the exploration phase is the key factor for the performance of the algorithm.

Experimental Results for the Peg Solitaire Model — PEG

We performed the same kind of analysis with a model corresponding to the puzzle game *Peg-Solitaire* (see Fig. 4.12 and 4.13). This model offers a good benchmark for our method because the state graph of the system is acyclic. In this benchmark, we only consider one specification, that is an instance of liveness property (`A<> dead`).

Like in the previous case, we give the "cumulative" execution time in Fig. 4.27, obtained when checking the PEG model with 16 processors.

Our results with the PEG model contrast with the ones obtained with the Dining Philosophers. We observe very good speedups for both versions of our algorithm (with an efficiency of 75%). Also, it is interesting to remark that, even though the relative speedups for the RG and RPG versions are similar, there is a significant difference between their execution time.

Figure 4.14 gives the speedup achieved by the exploration and cycle detections phases independently. We can see that the cycle detection phase has a bigger impact and better speedups. For instance, we have a speedup of approximately 11 for 16 processors for both versions. We can explain this behavior by the fact that, for the formula that we check on the PEG model, we need to completely explore the state graph. It means that the occupancy rate of the processors is better in this example and therefore we take benefit from our work stealing strategy. This is an evidence that our approach is optimized for the worst case scenario when model checking a system, that is for the case when the property is true.

Experimental Results for the Token Ring Models — TK and TK_M

Finally, we give the results for a model corresponding to the Token Ring protocol. We consider two versions of the protocol. TK stands for the classical "implementation", where starvation is possible. TK_M is a modified version, without starvation, which means that "whenever a process requires a resource, it will eventually be granted the right to use it".

In our benchmark, the most interesting specification is related to starvation, that is an example of the leadsto property: `wait_1 ==> cs_1`. We also consider two examples of safety property: `A[] - (cs_1 + ... + cs_22 > 1)` and `E[] - (cs_1 + ... + cs_22 = 0)`. For these two examples, the first formula do not require a backward exploration phase.

Our results are given in Fig. 4.15, 4.16 and 4.17 for the TK model and Fig. 4.18, 4.19 and 4.20 for the TK_M model.

This example is interesting because we can compare the performance of our algorithms using: state spaces that are very similar; using the same formula; but with a specification that is true in one case and false in the other.

Figures 4.15 and 4.16 show the speedup and execution time for the three formulas given in Fig. 4.8 for the TK model. We observe similar results than with the PH model:

the two versions have similar parallel execution time and the total speedup is mainly dominated by the (forward) exploration phase. Like with the PH model, we can explain these results by the fact that the cycle detection phase is very short and stop before completely exploring the state space.

Figures 4.18 and 4.19 show the results of a similar experiment for the TK_M model but with a small number of stations, 20 instead of 22. We decided to reduce the number of stations due to the long runtime necessary for the execution with small number of processors, i.e. the sequential time for the parental algorithm is around 12000 seconds for this experiment.. Like we mentioned before, the formula `wait_1 ==> cs_1` is valid for TK_M , that is to say, we need to explore the complete state graph in the backward exploration phase. This explain why, in this case, the results are more similar to the PEG case than to the PH case.

There are some differences though. The execution time of the RPG version is significantly slower and does not scales well for TK_M. We can explain this loss of performance by the number of iterations in the backward exploration. For the PEG model, our algorithm requires 29 iterations and almost one billion transitions are re-computed. for the TK_M model, while the algorithm requires 98 iterations to re-compute $2 \cdot 10^9$ transitions. The difference is quite important, especially since the TK models has four times less states than PEG. (TK_M has $5 \cdot 10^7$ states and $4 \cdot 10^8$ transitions; PEG has $2 \cdot 10^8$ states and $22 \cdot 10^8$ transitions.)

This difference in behavior can be explained by the influence of the state space's "shape" on the algorithm. To give a good idea of what is intended by the notion of "shape" in this context, we display in see Fig. 4.21 and 4.22 the state graphs for simplified versions of PEG (only 13 pegs) and TK_M (only 2 stations). Comparing the state graphs for TK_M and PEG, we see that there are less opportunities with TK_M to clear a large number of states in the same iteration.

4.7.2 Comparison with a Standard Algorithm

In this section, we compare our approach with a "standard" algorithm for model checking CTL. We assume that we know how to efficiently compute the predecessors of a state in the state graph, that is, that we can directly compute the reverse transition relation. This is true for the models used in our benchmarks, because we know how to easily compute the inverse of a transition in a Petri Net.

In this case, we can simply use the same code than for the RG version of the MCLCD algorithm, but compute the predecessor relation instead of relying on the reverse graph. Since we do not need to store the transition relation, we call this new version of our algorithm NO_GRAPH. The NO_GRAPH version of the algorithm is interesting for several reasons:

- Obviously, it is even more memory efficient than the RPG version. This means that we have the same benefits than the RPG version for the forward exploration, that is, a good speedup due to the fact that we write less information on memory. We also have the same benefits than the RG version for the cycle detection phase, that is, we will never have to re-compute the transitions;
- We can reuse the same data structures and synchronization patterns than in our implementations of the RG and RPG versions. We also use exactly the same models, expressed in the same modelling language. This means that we can really compare algorithms and not only implementations
- Finally, NO_GRAPH is a “standard” algorithm, that is representative of the current state of the art for semantic-based model checking algorithms.

Next, we give experimental results comparing our implementation of the three versions of the MCLCD algorithm (RG, RPG and NO_GRAPH) using 16 processors on our test machine. Figures 4.23 to 4.27 give a series of bar charts where we put in evidence the time required for each phase of the algorithm (exploration and cycle detection).

For the Dining Philosophers (PH) and the Sokoban (SK) models, we observe that: (1) NO_GRAPH has the best execution time; (2) the time spent in the forward phase is the same for NO_GRAPH and RPG; and (3) the RPG algorithm matches the RG algorithm because its gain in performance during the forward exploration exceeds is loss of performance during the cycle detection.

The second observation is not surprising since the forward phase is almost the same for the RPG and NO_GRAPH versions. (RPG should be a little bit slower than NO_GRAPH because we need to store one additional pointer in each state for the father, but this is not noticeable.)

We perform a similar comparison with our two models for the Token Ring protocol (TK and TK_M). As in the previous case, the performances of the versions are essentially the same for the `A[]` and `E[]` formulas. We can even observe that, for the TK

model and the `E[]` formula, RPG beats NO_GRAPH. The same is true in the case of the PEG model (see Fig. 4.27).

The result is far more different for the leadsto formula (`==>`). We remind the reader that, in this case, the formula is false for TK and true for TK_M. We observe that the execution time with RPG are around 7 times slower than with NO_GRAPH.

4.7.3 Conclusion About the Experiments

We have observed two main categories of behaviors in the analysis of our experimental results. We have examples of *complete backward traversal* and examples of *negligible backward traversal.*

Negligible backward traversal We put in this category the examples where the time spent in the backward exploration phase is negligible compared to the overall execution time. This is the case, for instance, if the specification is false and the cycle detection phase terminates early. In this category of experiments, there is no significant differences between RG and RPG. This is mainly because the gain in performance during the forward exploration phase outweighs the extra work performed during the cycle detection phase.

Complete backward traversal. We put in this category the examples where the cycle detection phase needs to run through all the state space. We observed a significant difference in performance between the RG and RPG versions in this case. The extra work performed by the RPG version becomes the dominant factor, up to a point where it accounts for nearly all the execution time. We also observed that, in this case, the "shape" of the state graph has a strong impact on the performance of the algorithm; in particular, the RPG algorithm does not scale well when the backward traversal phase requires a lot of iterations between the clearing and collecting steps.

We can draw some conclusions from these benchmarks. The RG version of our algorithm appears to be the best choice when we expect to spend a non-negligible part of the execution time in the cycle detection phase. When the backward phase is short, the NO_GRAPH and RPG versions are a good choice. This is in particular true with reachability properties (because we only perform the forward phase), but also

if we expect the property to be false (because we expect the cycle detection phase to terminate early).

The RPG is still interesting with very large state spaces, when we do not have enough memory to store the complete transition relation. We can observe, when we compare the RPG and NO_GRAPH versions, that the time lost re-computing the same transitions several time may not always be too much of a drawback. We see with the PEG model that RPG is substantially better than NO_GRAPH, while the opposite is true for the TK_M model.

A real advantage of the RPG version is to impose no restrictions on the models that are checked. Several model checking algorithms rely on the fact that the transition relation needs not be stored. Very often, this optimization is based on the fact that it is possible to compute the reverse transition relation[1]. But this is not always practical, or even possible. This is the case, for example, when model checking Timed Petri Nets (MF76) using State Class Graphs, a common abstraction for representing a Kripke Structure with real-time constraints; Given a state class, it is only possible to compute a superset of the predecessors. More generally, computing the reverse transition relation is not possible for systems that manipulate data variables. In this case, RPG is the best solution.

To conclude, both RG and RPG can be useful; RPG being the good choice if we are limited by the memory space or we expect the specification to be true. Although RPG may requires a lot more computations, it can be applied on models that are not tractable with the reverse graph version. For instance, we performed an experiment with the European Peg-Solitaire game (37 pegs) with our setup (208 GB of RAM). The state space of this model has 3.10^9 states and 3.10^{10} transitions. Assuming that each transition would use 8 bytes of memory to store the reverse relation between two given vertices, we would need at least 240 GB of memory only to store the edges of this graph. On the opposite, we only need 15 GB to store the states and we can check this example with RPG (with the specification `A<> dead`) in 19,662 s, divided in 3,817 s for the exploration phase (less than 20% of the computation time) and 15,845 s for the cycle detection phase.

[1] There are also solutions based on recursive traversals of the state graph, but they essentially store the transition relation in the stack, rather than the heap.

4.8 Conclusions

In this chapter, we have described some ongoing work concerning parallel model checking algorithms for finite state systems. This is, chronologically, the last work that was performed during this PhD thesis. For this reason, it is also the chapter that has the most opportunities for future work.

We have based our approach on three main principles:

- *Thou shalt not restrict the modelling language*: we only need to be able to compute the successors of a state;
- *Thou shalt not restrict the way states are distributed*: because we should be able to reuse the same algorithm with different state space construction methods; and, finally
- *thou shalt no put restrict the way work is shared among processors*: because the algorithm should play nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing.

We define two versions of a new algorithm, called MCLCD, that supports specification expressed in a subset of CTL. Our algorithms are based on a standard, semantic model checking algorithm for CTL that specifically targets parallel, shared memory machines. We defined two versions of the same algorithm: a Reverse Graph (RG) version, that explicitly stores the transition relation in memory; and a Reverse Parental Graph (RPG) version, that only requires a "spanning subtree" of the transition relation.

We show that the RG version has a linear time and space complexity ($O(S + R)$), while RPG that has time complexity in $O(S \cdot\cdot (S - R))$ and space complexity in $O(S)$. In these expressions, S stands for the number of reachable states in the system and R for the number of transitions. If we interpret these complexity results using only the number of states—we have R in $O(S^2)$—it is clear that RPG trades computation time (in $O(S^3)$ for RPG against $O(S^2)$ for RG) for memory space (in $O(S)$ for RPG against $O(S^2)$ for RG).

De facto, we use the reverse parental graph structure as a mean to fight the state explosion problem. In this respect, this approach has a similar impact than algorithmic techniques like *sleep sets* (used with partial orders methods), but with the difference

that we do not take into account the structure of the model. Moreover, our approach is effective regardless or the formalism used to model the system.

Our prototype implementation shows promising results for both the RG and RPG versions of the algorithm. The choice of a "labeling algorithm" based on the out-degree number has proved to be a good match for shared memory machines and a work stealing strategy; for instance, we consistently obtained speedups close to linear with an average efficiency of 75%. Our experimental results also showed that the RPG version is able to outperform the RG version for some categories of models.

For future works, we are studying an improved version of our algorithms that supports the complete set of CTL formulas. Actually, we already have what is needed to model-check the whole of CTL. Indeed, we can follow the approach proposed by Clarke for CTL model checking (CES86) and reduce the problem of checking a "nested" formula Φ to the problem of checking $|\Phi|$ *basic formulas*. In this context, $|\Phi|$ is an integer value measuring the complexity of the formula Φ—or the number of "sub-formulas" in Φ—and basic formulas correspond to the formulas $E(\phi \cup \psi)$ and $E\Box(\phi)$[1]. A naive implementation of this approach would be to manage $|\Phi|$ copies of our labels (sons and suc) in parallel, but this could have an adverse effect on the memory consumption. At the moment, we are still considering several strategies for model checking CTL formulas using a bounded number of labels.

[1] we do not consider the modality *next* in this discussion but it is not difficult to add it in our framework.

Model	Formula	Description	Results
Sokoban(SK) $7 \cdot 10^7$ states $18 \cdot 10^7$ trans.	`E<> win`	The Game has a wining move	*true*
	`E[] - win`	There is an infinite match.	*true*
Philosophers (PH) $14 \cdot 10^7$ states $17 \cdot 10^8$ trans.	`E<> gr1 /\ gl1`	A philosopher can eventually eat.	*true*
	`A<> (gr1 /\ gl1)`	A philosopher will eventually eat.	*false*
	`E[] - (gr1 /\ gl1)`	A philosopher may never eat.	*true*
	`A[] - ((gr1 /\ gl1) /\(gr2 /\ gl2))`	Two philosophers cannot both eat at the same time (mutual exclusion).	*true*
	`(wl1 /\ wr1) ==> (gl1 /\ gr1)`	Whenever a philosopher wants to eat, then eventually it will eat (starvation)	*false*
Solitaire (33 pegs) (PEG) $18 \cdot 10^7$ states $15 \cdot 10^8$ trans.	`E<>(peg_1 +... + peg_33 = 1)`	There is a sequence of moves leading to a winning position (only one peg left).	*true*
	`A<> dead`	Every sequence of moves eventually end in a dead position (no more pegs to remove).	*true*
Token Ring (22 stations) (TK/TK_M) $23 \cdot 10^7$ states $22 \cdot 10^8$ trans.	`wait_1 ==> cs_1`	No process should wait indefinitely to enter its critical section.	(TK) *false* (TK_M) *true*
	`A[] -(cs_1 + ... + cs_22 > 1)`	We cannot have more than one process in critical section at any time.	*true*
	`E[] -(cs_1 + ... + cs_22 = 0)`	We can find a scenario where no process enters its critical section.	*true*

Figure 4.8: Formulas and Models in our Benchmark.

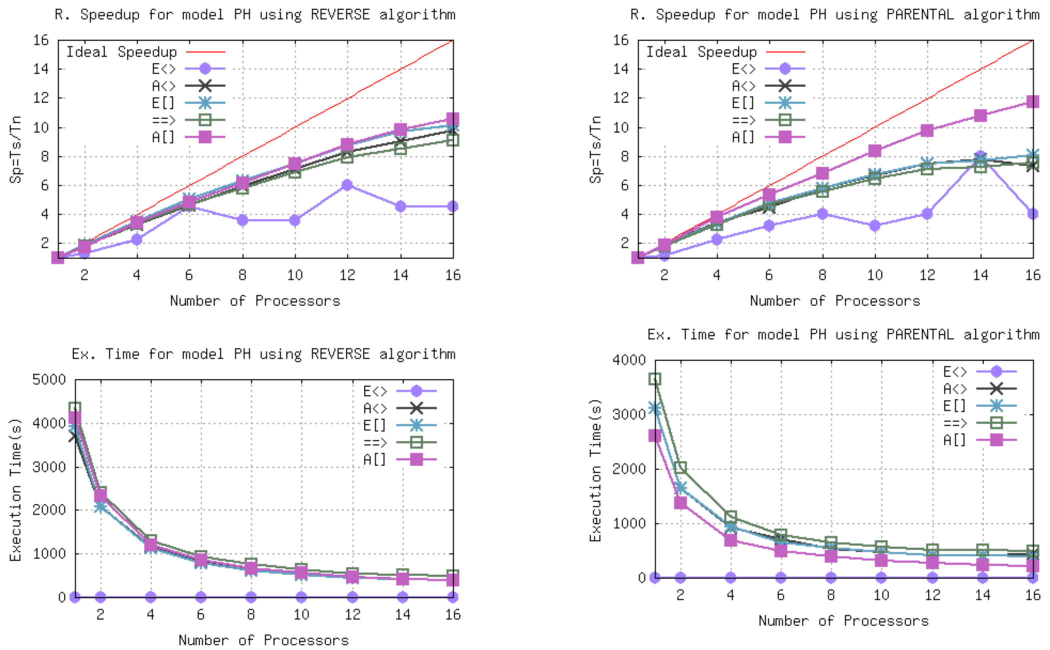


Figure 4.9: PH with Reverse algorithm.

Figure 4.10: PH with Parental algorithm.

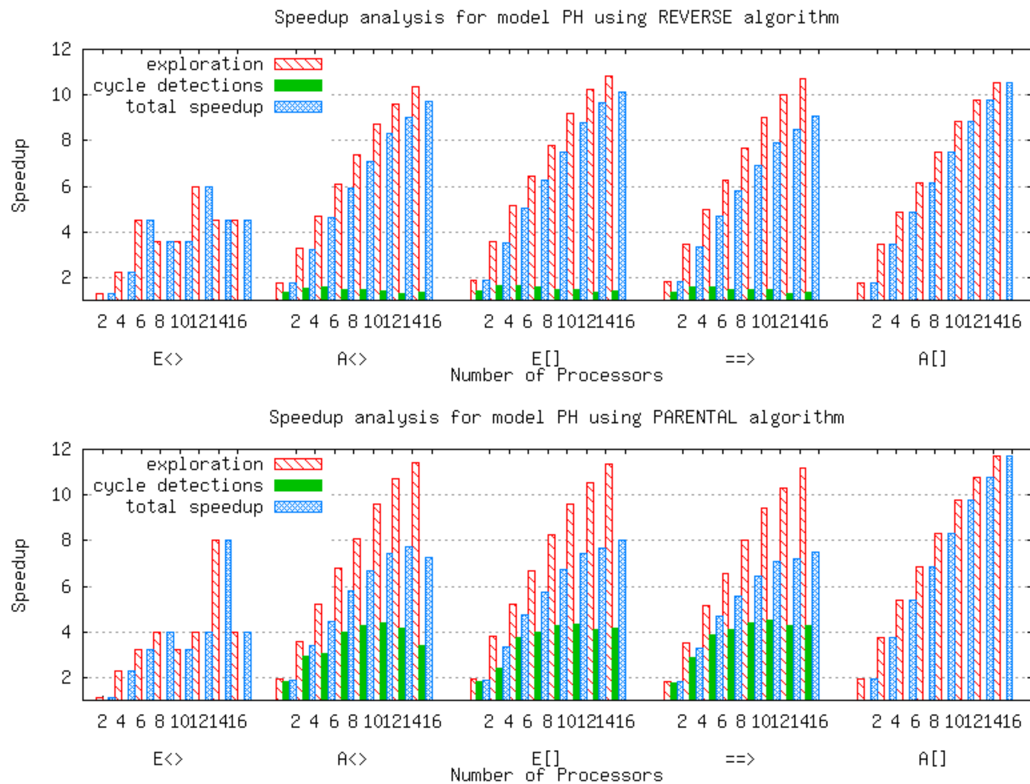
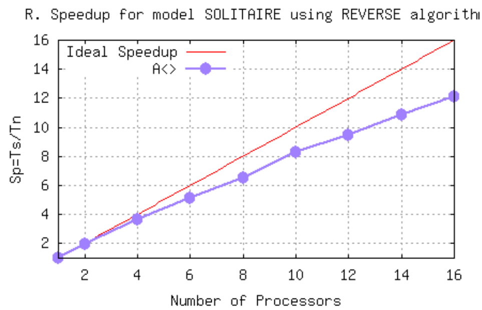


Figure 4.11: Exploration and cycle detection speedup analysis for PH model.



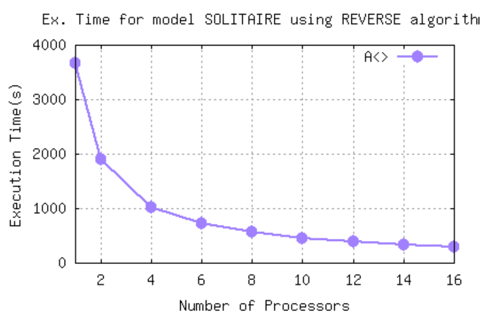
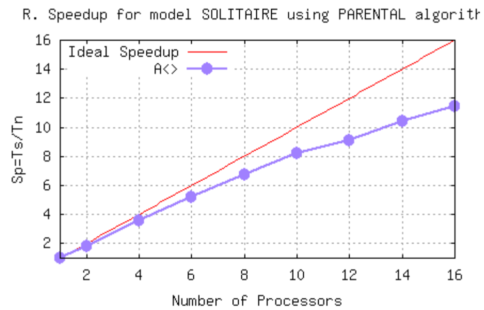


Figure 4.12: Peg with Reverse alg.



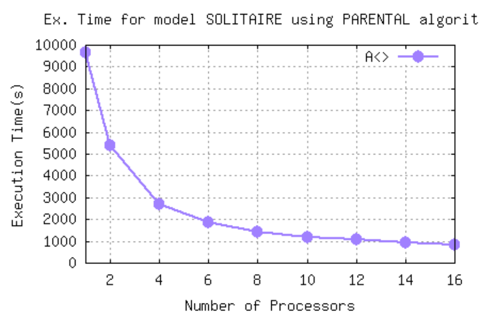
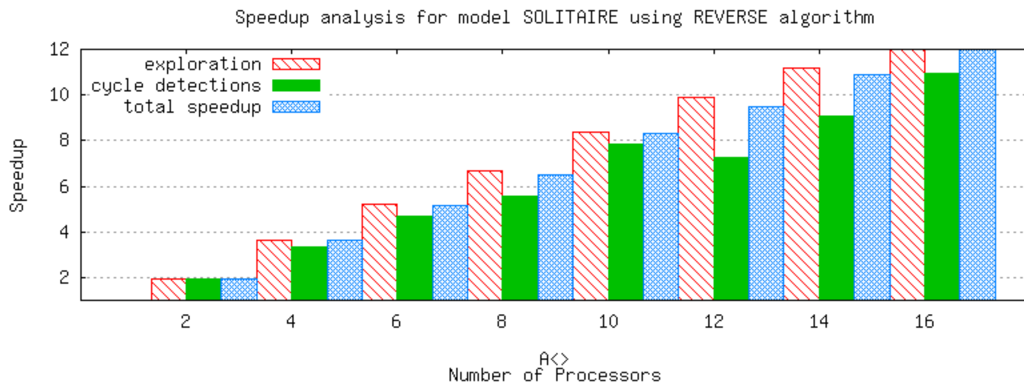


Figure 4.13: Peg with Parental alg.



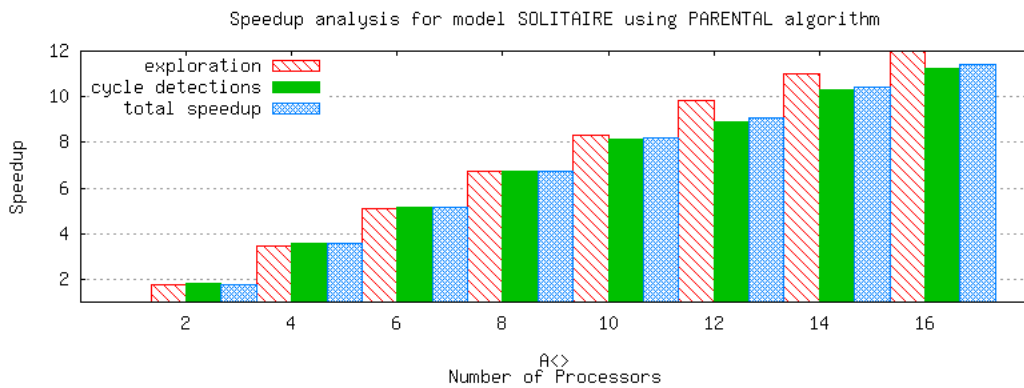


Figure 4.14: Exploration and cycle detection speedup analysis for Peg model.

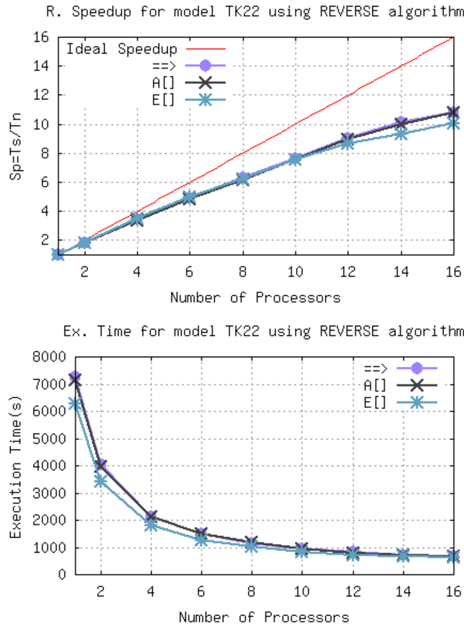


Figure 4.15: TK with Reverse algorithm.

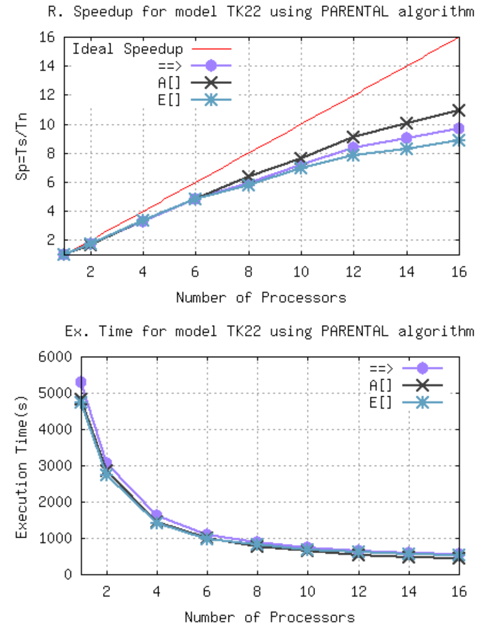


Figure 4.16: TK with Parental algorithm.

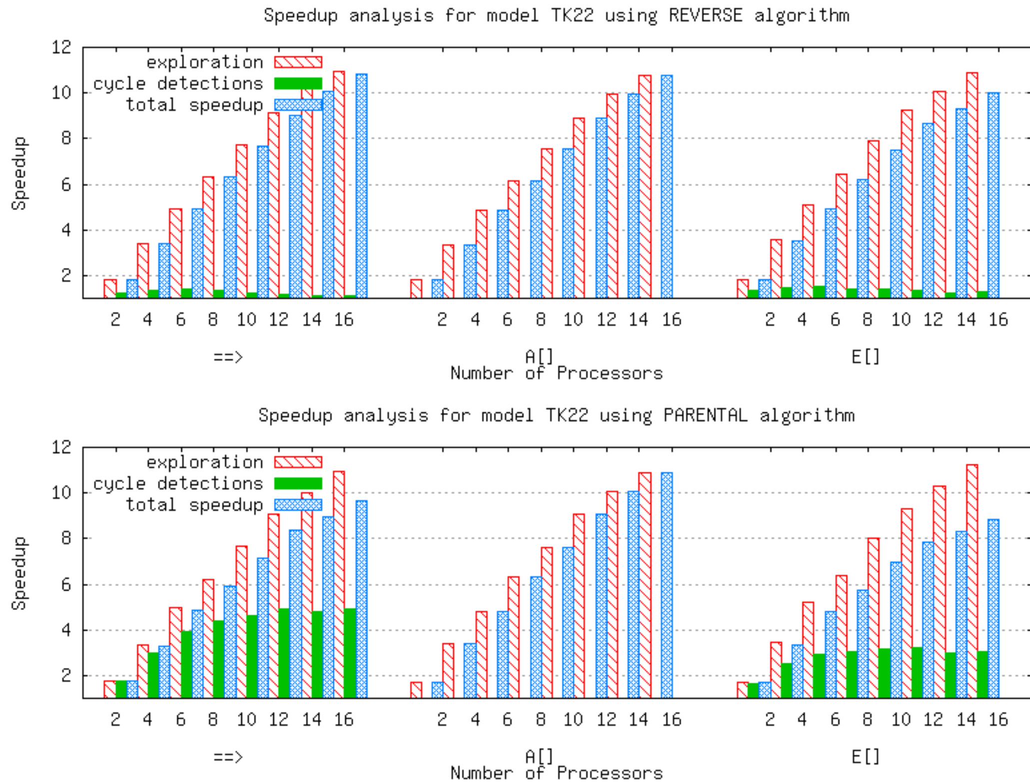


Figure 4.17: Exploration and cycle detection speedup analysis for TK model.

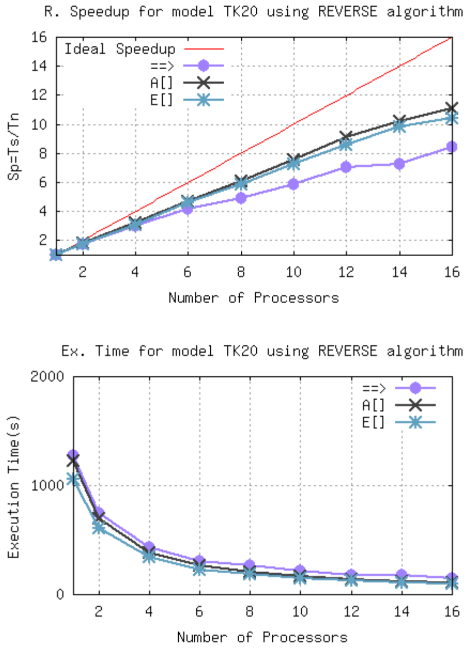


Figure 4.18: TK_M with Reverse alg. .

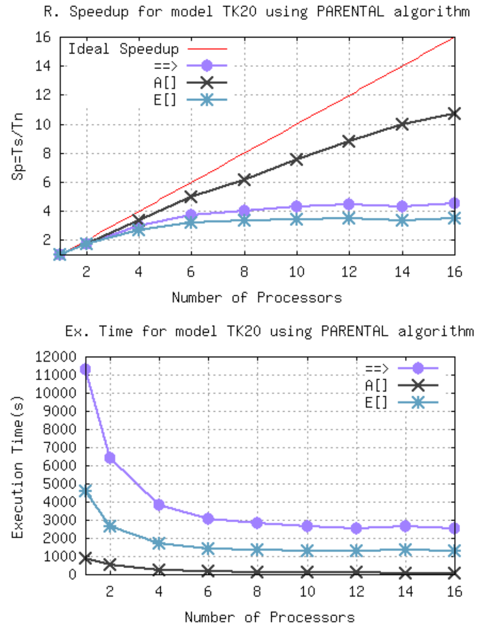


Figure 4.19: TK_M with Parental alg. .

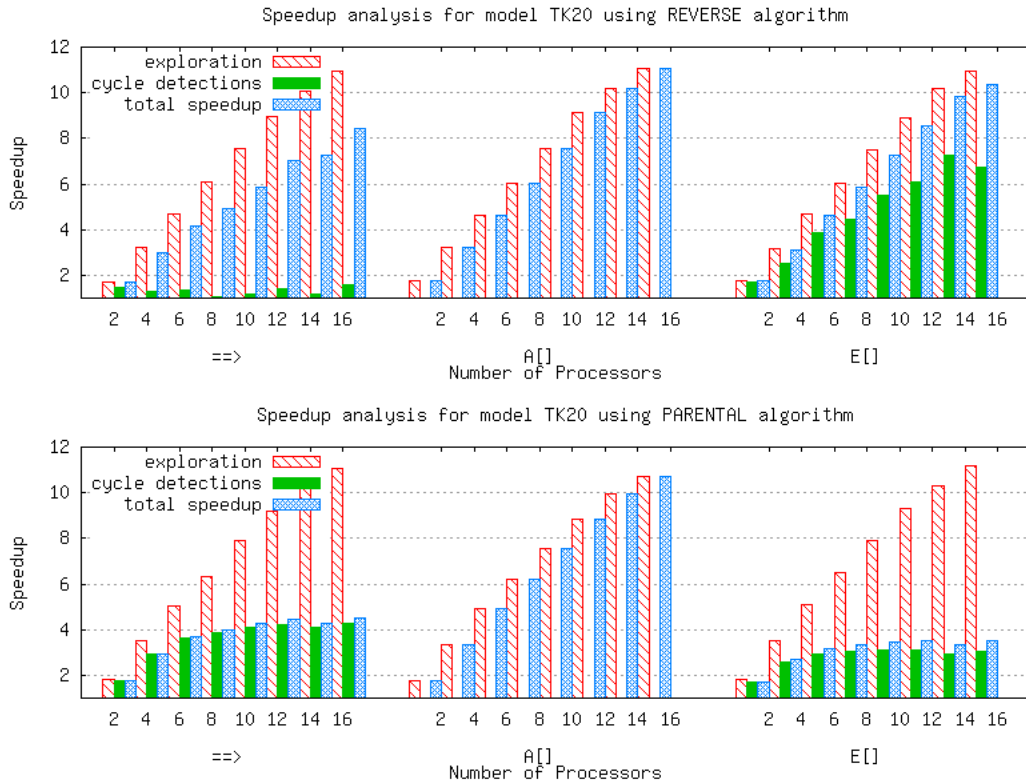


Figure 4.20: Exploration and cycle detection speedup analysis for TK_M model.

Figure 4.21: Simplified graph for Peg-Solitaire (13 tokens).

Figure 4.22: Simplified graph for TK_M (2 stations).

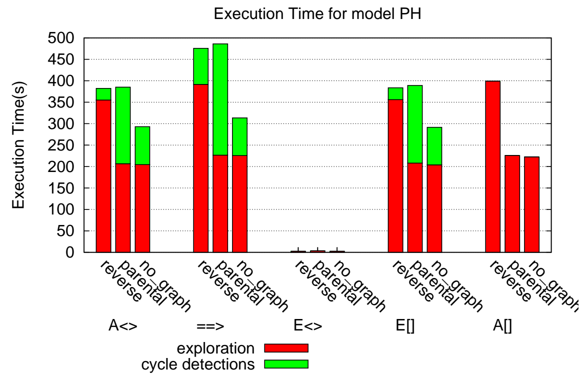


Figure 4.23: PH model.

Figure 4.24: SK model.

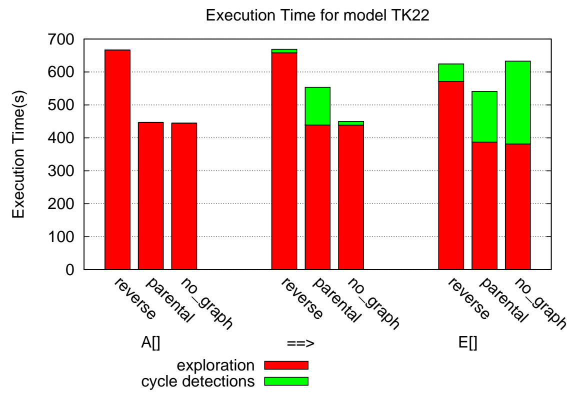


Figure 4.25: TK model.

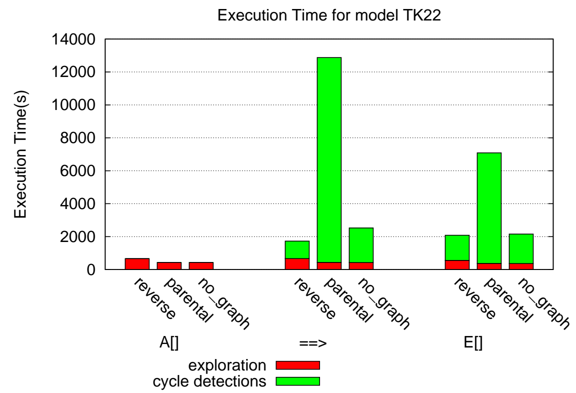


Figure 4.26: TK_M model.

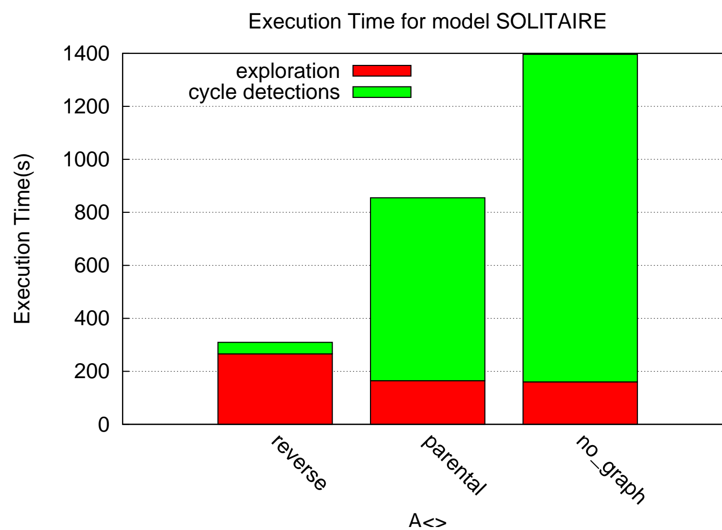


Figure 4.27: PEG model.

Chapter 5

Probabilistic Verification: Bloom Table

“ Beware of bugs in the above code; I have only proved it correct, not tried it.”

Donald Knuth

In this chapter, we describe a new data structure for probabilistic state space generation. In the context of this work, we use the term *probabilistic* to mean that, during the exploration of a system, any given reachable state has a high probability of being checked by the algorithm. As a consequence, we accept that some reachable state of the system may not be inspected; but this loss of accuracy is generally counterbalanced by the fact that probabilistic verification algorithms require less computing space. It does not solve the problem of state explosion but it helps to find errors of models previously considered intractable due to memory restrictions.

This chapter is organized as follows. Section 5.1 introduces the domain of probabilist verification of formal models. We define our Bloom Table data structure in Section 5.2. In Section 5.3, we use our novel data structure to perform the probabilistic state space generation, we give several experimental results that are compared to the theoretical analysis introduced in Section 5.2. Finally, in Section 5.4, we conclude this chapter with a discussion about its advantages and disadvantages.

5.1 Introduction

This idea of a probabilistic verification algorithm was first proposed by Holzmann (Hol93) in the context of the verification of communication protocols. Basically, the idea is to store the "hash" of a state (i.e. one or multiple hash keys) instead of the whole state itself. There are different ways to store these hash keys, each combination of these choices leading to a different algorithm. Two main categories of choices emerge: storing hash keys as positions in a vector (as it is the case with Bloom filter and Holzmann's supertrace algorithm); and storing hash keys directly as values (as it is the case in the hash compact algorithm). We have a straightforward gain in memory if the amount of information needed for encoding the hash keys, say h bits, is less than the size of representing one state. On the other hand, we may believe that a newly generated state has already been found if we have a *collision*; that is two states having similar hash keys. This is the only source of errors. Therefore, we have a small probability of missing a state if the size of the state space is small compared to 2^h (and provided we use good hash-functions).

Probabilistic verification has been extensively studied and several solutions has been proposed. Some of the most successful tools in this domain rely on Bloom filters, a celebrated data structure for implementing set membership. This structure delivers a good trade-off between state space coverability and memory used. It is a compact structure for set membership testing where the probability of omission—false positive results—depends on the number of hash functions used and the ratio of the number of inserted elements with its dimension. However, high coverability (low probability of omissions) comes at a price, since it may require the use of a large number of hash functions. Consequently, it will increase the execution time because it requires the generation of a larger number of hash values and, by the same way, a larger number of memory access to the filter. Another problem with the use of a Bloom Filter is that the precision of the filter—and therefore the coverability ratio of the algorithm—may be inadequately poor if its parameters is not well tuned with respect to the number of entries.

We propose an optimized data structure based on the notion of Bloom Filter (see the description of "Bloom filters" in Section 2.4.1). We present and analyze a new probabilistic verification scheme based on an enriched Bloom Filter, named Bloom Table (BT

for short). When compared to simpler hash-based techniques, this new data structure is able to increase the probability of generating the whole state space of a system—what we call the *coverability* ratio, or the accuracy of the algorithm—provided when we have a rough estimate of the size of this state space. If we fix the maximal amount of memory that can be used during the state space generation, our algorithm based on Bloom Tables will have a better accuracy, without a significant impact on performances, and in particular without involving the use of a large number of intermediate hash function. Our results show that only two hash functions are enough for a probability of omissions close to 10^{-5}, where a "classical" approach based on Bloom Filters needs around 5–6 hash functions to achieve a similar result.

The main difference of our structure, when compared to the classical Bloom Filter, is that we use a vector of "words" instead of a vector of bits. (We redefine the term word as the vector slot size in bits.) Succinctly, a BT with M slots is associated to a sequence of k independent hash functions $(h_i)_{i \in 1..k}$, with values in $1..M$, and another hash function, *key*, that computes values of size $k.q$ bits. To insert a value x inside the BT, the hash-value returned by $key(x)$ is sliced into k sub-words of size q and inserted in the BT at each position $h_i(x)$ for i in $1..k$. Hence, what we obtain is a blend between a Bloom Filter—insertion requires writing in a vector at multiple positions—and a hash-table—insertion may fail because the necessary slots are filled with values different than what is expected. A significant difference with hash table, and the main drawback of our structure, is that we cannot easily adapt the hash table's collision resolution strategies (such as separate chaining or open addressing) in the case of a failed insertion.

In a Bloom Table, test membership is successful if the concatenation of the k sub-words found at position $h_i(x)$, for i in $1..k$, matches $key(x)$. Otherwise the test fails. Consequently, the BT divides the chances of omissions by a factor of $2^{k.q}$ when compared to a Bloom filter (that is the probability for $key(x)$ to be equal to a random sequence of $k.q$ bits). Moreover, setting up the data structure is simple. Knowing the amount of memory available (T), the size of the "words" in bits (q) and the number of hash keys (k), the maximal number of elements that can be inserted in the BT is $N = \dfrac{T}{k.q}$. In addition, we can adjust the number of bits used per element if there is an estimation of the number of states to be encoded—since increasing the value of q leads to a better accuracy.

5.2 Probabilistic Data Structure

Before defining the Bloom Table data structure (section 5.2.2), we compare both Bloom filter and Compact hash table data structures in order to show the advantages of our work.

5.2.1 Bloom filter and Compact Hash Table

Before defining our Bloom Table, we compare the advantages and disadvantages of the Bloom filter and Compact Hash Table for implementing (probabilistic) test membership algorithms. (See the description of "Bloom filters" and "Compact hash table " data structures in Section 2.4.1 and Section 2.4.2, respectively.)

We define the notion of "hash signature" of a data structure as the ratio between the size of the hash computed and the number of bits effectively used; it helps us to define a measure of memory efficiency because it shows how many bits of information are implicitly stored per bit used. To compute the hash signature of a structure, we take the size of the hash function used divided by the number of bits necessary to insert a new element . The hash signature is a measure of the conciseness (memory space efficiency) of the algorithm. Nonetheless, we cannot only rely on this measure to compare data structures. Indeed, it does not take into account the implicit sharing mechanism that occurs in structures like the Bloom Filter, where the representation of different states can have one or more bits in common.

We assume that S is the set of elements that should be inserted in these data structures, where N is the cardinality of S. We consider: (1) an approach based on a Bloom filter of dimension $M_b = 2^m$—a vector of M_b bits—with $M_b \gg N$, and (2) a version of the hash compact algorithm based on a hash table of size $M_h = 2^h$, storing hash keys encoded using w bits. To compare data structures that have a similar memory footprint, we suppose that $M_b \approx M_h.w$.

The supertrace or multihash algorithms offer a good trade-off between coverage and memory used because they have a "hash signature" of size $HS_{Bloom\ Filter} = m$ per bit used. However, it renders difficult to decide if a given element is a false positive when all bits are set because there is no information attached to these bits. By contrast, hash compact does not promote any kind of memory sharing and delivers a smaller "hash signature" per bit used. It explicitly assigns a "word" of size w bits for a given state

in order to reduce the number of false positives. The hash signature in this case is $HS_{Hash\ Compact} = \frac{h+w}{w}$, which is the hash computed (h bits) to map the state into the hash table plus the state hash value (w bits) divided by the w bits used. Comparing the hash signature of both structures, $HS_{Bloom\ Filter} > HS_{Hash\ Compact}$, the Bloom filter is more memory efficient than the Compact hash table because it carries more information per bit used, meantime, Compact hash table has a smaller number of false positive when a minimum amount of bits is available per state. So, increasing the size of the value stored may reduces the number of false positives but increases the memory requirements. Our interpretation of this comparison is that we can obtain a new data structure (Bloom Table) with an intermediate hash signature between Bloom filter and Compact hash table, being more memory efficient than Compact hash table and having a smaller rate of false positives than Bloom filter.

Fig.5.1 compares **BT** with supertrace (BF), hash compact (HC) and improved hash compact (HCI) algorithms using the same amount of memory (2^{31} = 268MB) for different number of entries. (See the presentation of BF, HC and HCI at Section 2.4) This figure helps us to position our contribution; the hash compact algorithm is effective when the states are stored using at least 64 bits values ($\frac{3.4\ 10^7}{2^{31}}$ bits); the improved hash compact delivers a better result than hash compact but it needs at least 40 bits per state ($\frac{5.3\ 10^7}{2^{31}}$ bits) – conversely, the result is not significant for any number of entries above 5.3 10^7 (see improved hash with 32 bits); the classical Bloom filter with 2 hash functions is the most memory-efficient (2 bits per state); the **BT** data structure, configured with 2 hash functions with a “word” of 8 bits size, gives the lower probability of omissions using only ($k \cdot q = 2 \cdot 8$) 16 bits per state.

5.2.2 Bloom Table

Our *Bloom Table* is a natural blend of the Bloom filter and the Compact hash table data structures. The goal is to keep a good accuracy and a low space complexity in the cases where the number of elements that need to be inserted is approximately known. When compared to these two data structures, we conceived BT as an extension of the Bloom filter that has a better hash signature than Compact hash table (hash compact) and a smaller number of false positives than supertrace and multihash while using the same amount of memory.

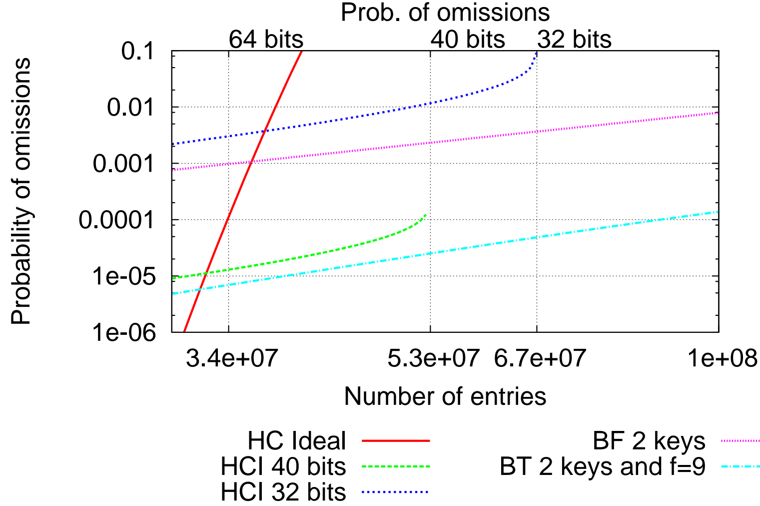


Figure 5.1: Probabilist Algorithms Comparison.

A *Bloom Table* B is a vector of "words", instead of bits. We use q to denote the size (in bits) of each word and M for the size of the vector, also called the dimension of B. As with Bloom Filters, a BT is also associated to a series of k independent hash functions $((h_i)_{i \in 1..k})$ with image in $0..M-1$. The values of q, M and k are parameters of the Bloom Table. Finally, a BT is also associated to a hash function *key*, with image in $0..(2^{k.q}-1)$. The role of *key* is to produce values that can be easily transformed into k different words of size q. The *key* function will be used to attach small pieces of information when inserting a new element; to help decrease the rate of false positives.

An empty Bloom Table is represented by a vector with all positions set to $\mathbf{0}_q$, that is the q-bit word all made of 0's. The operation of inserting a new element x in the Bloom Table B is simple. We compute the value of $key(x)$ and "slice" the result into k words of size q, say $w_1, \ldots, w_k$[1]. Next, we insert the word w_i at position $h_i(x)$ in the vector B for each index i in $1..k$. We only insert a word at a position where the vector was empty—its value is equal to $\mathbf{0}_q$—so as to never overwrite previous information. Another successful case is if the value found in the vector—the *resident* value—is already equal to w_i. We say that we have a partial collision. The insertion procedure is deemed *successful* if all k pieces are inserted. Otherwise, we say that the insertion is *incomplete*.

[1]We assume that each of these words is different from $\mathbf{0}_q$.

The membership test follows a similar logic. To test if an element x is in B, we collect the words at position $h_1(x), ..., h_k(x)$ and compare their concatenation to the value of $key(x)$. The test is successful if the two values are equal.

We consider different cases of insertions and tests. This distinction is important since we will obtain different probabilities of omissions depending on the number of "words" that were successfully inserted. There are four possible status for insertion:

- Complete: when all slots are set, at least one word is inserted, and all the resident values match;
- (Possible) False Positive: when all slots are already set and all the resident values match. This case is similar to the omission mechanism that we described with *Bloom Filter*.
- Incomplete: when at least one slot could not be set and the resident value do not match;
- Not Possible: when all slots are already set and none of the resident values match;

In Figure 5.2 we depict the insertion operation for three elements (x, y and z in this order) in a Bloom Table with ($k = 2$) two independent hash function h_1 and h_2. We also test the membership of an element (f). For the other parameters of the BT, we have a dimension $M = 16$ and a size of words $q = 3$ bits. For convenience, we use an octal notation for the words inserted in B so, for instance, 6 stands for the word 110 and 23 is the concatenation of the two q-bits words 010 and 011. The figure shows a *successful* insertion for an element x such that $key(x) = 15$, $h_1(x) = 3$ and $h_2(x) = 9$. It also shows two unsuccessful tests: the insertion of y is *incomplete* because there is at least one slot that could not be inserted ($w(y)_1 = 2$ and the value of B at position $h_1(y) = 9$ is 5); the insertion of z is *not possible* because none of the slots can be inserted. Finally, the element f is cataloged as a *false positive* because all slots are already set and all the values match.

In our implementation of Bloom Table—used for the probabilistic algorithm defined in Section 5.3—we consider that elements defined as *incomplete* or *not possible* are not part of the set encoded by the BT. Instead, we add a secondary data structure to the BT to store these "failed insertions". These elements are stored separately not to degrade the probability of BT due to their inability to insert the complete sliced *key*.

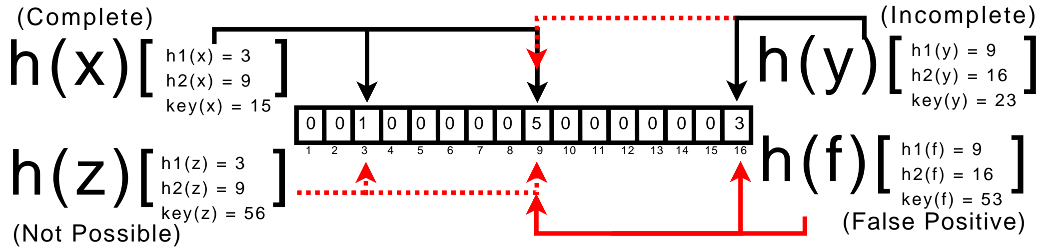


Figure 5.2: Illustration of the insertion operation on a Bloom Table.

This second data structure—or *overflow* table—may be probabilistic or "exact". We could also choose a cache table instead. In the context of this work, we choose an exact one (Hash Table) in order to better analyze the results of our algorithm. Even though these elements are stored outside of the Bloom Table, it is not problematic. For every failed membership test (when all slots are set and at least one slot does not match), we forward the membership query to the overflow table. Figure 5.3 gives a simple flowchart representing the operations of a BT in conjunction with a second storage device for incomplete insertion and partial match tests.

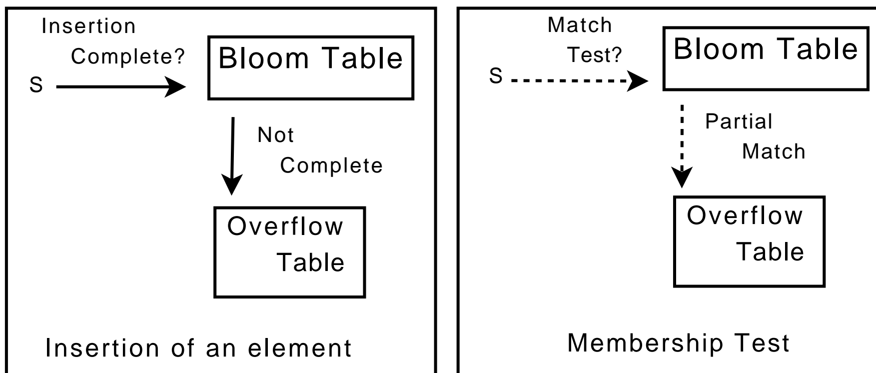


Figure 5.3: Bloom Table and second storage coupling.

Our motivation for not considering incomplete elements as part of the BT is to decrease the probability of omissions. For instance, assume we accept elements with only one (sub-word) failed insertion out of k, then the probability of omissions is multiplied by a factor of approximately $k.2^q$. Additionally, the size of the overflow table can be kept to a small fraction of the size of the BT when the accuracy of the BT is good. In our experiments, for instance, we show that with a right choice of parameters, the

size of this overflow table is less than 1% of the size of the BT. In the next section, we present a simple extension to Bloom Table that can help reduce the size of the overflow table.

Bloom Table with Multiple Possible Insertions

We want to insert as many as possible elements in the BT in order to avoid filling the overflow table. To this end, we extend the operations of the Bloom Table using a mechanism similar to the *open addressing* strategy used in some hash tables implementations. The idea is to allow multiple attempts for finding empty slots in the vector when inserting a new element. At each attempt, we try to insert the word at a new position. We use f to denote the maximum number of attempts that are allowed.

Our strategy is simple, we associate to each one of the k words a set of f independent hash functions, $(h_i^j)_{i \in 1..k, j \in 1..f}$. The "basic" Bloom Table described in the previous section corresponds to the case where $f = 0$.

In a BT with f possible insertions, B, the operation of inserting a new element x is as follows. We compute the tuple of words $(w_1, \ldots, w_k)$ using the *key* function. For insertion of the word w_i, we try each slot in B at the positions given by $h_i^j(x)$ for all index j in $1..f$ in this order. We write w_i in the first slot that is empty (whose value is $\mathbf{0}_q$) or whose resident value is w_i. The insertion will fail if none of these f position is eligible.

With this extension, each "word" has f chances to find an empty slot or a match. This strategy helps to increase the number of elements inserted completely, that is, elements whose *key* can be completely reconstructed from the BT. Figure 5.4 shows the complete insertion of two elements, y and z, in a BT with $f = 2$. These two elements were classified as *incomplete* or *not possible* in Figure 5.2 because some (or all) of the "words" in $key(y) = \{w_1 = 2, w_2 = 3\}$ and $key(z) = \{w_1 = 5, w_2 = 6\}$ not match. From Figure 5.4, the insertion of the element y is performed at the second attempt ($h_{1 \cdot 2}(y) = 11$) of the first piece $w_1 = 2$ of its key ($key(y)$) because it had failed to insert it at the slot $h_{1 \cdot 1}(y) = 9$. The insertion of the element z is analogous, it is successful after one additional attempt for each piece w_1 and w_2. Later, at Section 5.3, our experimental results show that the portion of rejected elements (stored outside) is smaller than 1% of the total of inserted elements.

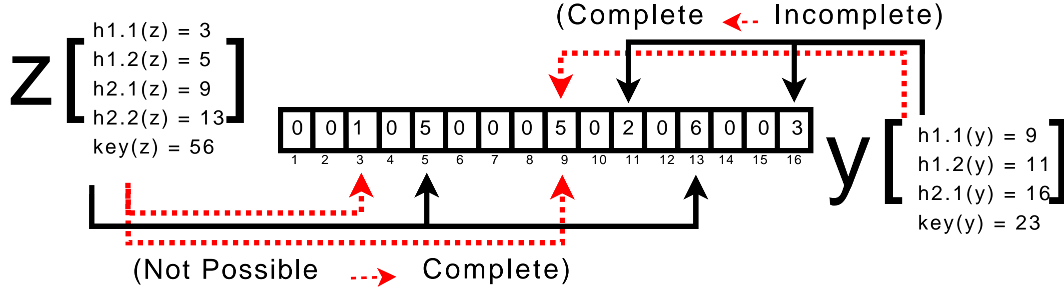


Figure 5.4: Insertion operation on a Bloom Table with multiple possible insertions.

5.2.2.1 Probability of omissions

In this section, we study the probability of a *false positive* in a Bloom Table. We consider the case of inserting a random element, say x, in a Bloom Table where N elements have already been inserted. The goal is to provide an analytical expression, P_{BT}, that approximate the probability for the insertion to report that x is a false positive. (Note that we are not able to identify if x is an omission or an element that was already inserted.) The probability P_{BT} will be a function of $N = 2^n$, the dimension of the table $M = 2^m$, the number of hash functions k, the number of possible attempts f, and the word size q. Later, at Section 5.3, we show the coherence of our analysis by comparing with actual results obtained from our experiments.

First, we consider the case of the "basic" Bloom Table ($f = 0$). An element x is a false positive in the Bloom Table B if the following two conditions are simultaneously true: (1) for each index i in $1..k$, the BT has a resident value at position $h_i(x)$ (that is different from $\mathbf{0}_q$); and (2) the value of $B[h_i(x)]$ is equal to w_i (using the same notations than in Section 5.2.2). Therefore, P_{BT} is the product of two probabilities: (1) the probability that the k slots related to x have already been written, which is the probability of omission for a Bloom Filter (P_{Bloom}) that we computed in Section 2.4.1; and (2) the probability for k random words of q bits length to be equal to some given tuple $w_1, \ldots, w_k$, that is $({}^1/_{2^q-1})^k$. (This last probability is not simply $2^{-q.k}$, since we need to rule out the case of a random word being equal to $\mathbf{0}_q$.) Like in the case of Bloom filters, we can safely assume that $M \gg 1$ (the Bloom Table is sufficiently large)

and that the inequality $k.N < M$ is valid. In this case, we have

$$\begin{aligned} P_{BT} &= P_{Bloom}.(^1/_{2^q-1})^k \\ &\approx (1-e^{-\frac{k.N}{M}})^k.(^1/_{2^q-1})^k \\ &\approx \left(\frac{(1-e^{-\frac{k.N}{M}})}{2^q-1}\right)^k \end{aligned}$$

The formula giving the probability of false positive for a Bloom Table reveals the dual behavior of the data structure. In the case $q = 1$, we find that the BT has the same behavior than a Bloom Filter. On the other hand, if $q > 1$ and all k slots are set, we find that the behavior of the Bloom Table is close to a Compact hash table, with an omission rate of approximately $2^{-q.k}$.

Now, we consider our technique of multiple insertions to avoid as much as possible elements from being stored outside. (Remind that we use a kind of *open addressing* strategy but limited to f attempts per slot.) So, every time a slot is already taken (not empty) and the residual value does not match, BT tries another slot up to f chances. Before continuing, it is reasonable to assume that these f chances are in fact mutually exclusive events, as long as no new slot is set. From this assumption, we define an upper bound probability of omissions for the non-ideal BT. With this in mind, we define two auxiliary variables: β and α. β is the probability to find a non empty slot after the insertion of N elements ($\beta = 1 - e^{\frac{-k \cdot N}{M}}$) and $1 - \alpha$ is the probability that the residual value does not match ($1 - \frac{1}{2^q} = 1 - \alpha$). Considering that an attempt to insert a slot fails if and only if it finds a non empty slot whose value does not match, the probability of a given chance i ($P(f_i)$) to fail is

$$\begin{aligned} \alpha &= \tfrac{1}{2^q-1} \\ \beta &= 1 - e^{\frac{-k \cdot N}{M}}. \\ P(f_i) &= \beta.(1-\alpha) \\ P(f_1 \cup f_2 \cup \cdots \cup f_n) &= P(f_1) + P(f_2) + \cdots + P(f_n) \end{aligned}$$

Hence, the upper bound probability of omissions P_{BT} consists in $f-1$ failed attempts ($\beta.(1-\alpha)$) to find a match before the fth one, which is a match ($\beta.1$). Taking into account that we had a failed match ($1-\alpha$) before we try the f chances, we have:

$$P_{BT} \approx P_{Bloom}.\left(\frac{(1-\alpha)+(f-1).(\beta.(1-\alpha))+\beta}{\alpha}\right)^k$$

Without loss of generality, we assume $1-\alpha \approx 1$. Consequently, we get the following probability for BT

$$
\begin{aligned}
1-\alpha &\approx 1 \\
P_{BT} &< P_{Bloom}.\left(\frac{1+(f-1).(\beta)+\beta}{\alpha}\right)^k \\
P_{BT} &< P_{Bloom}.\left(\frac{1+f.(1-e^{-\frac{k.N}{M}})}{2^q-1}\right)^k
\end{aligned}
$$

which is the rate of omissions for BT using k sets of f hash functions each one.

Now, we depict a series of figures to present a graphical analysis of the probability P_{BT}, and also, to put in evidence how it is affected by the parameters k, q and f. For all of these figures, we used a theoretical BT of size $M = 2^{28}$, words of $q = 8$ bits size and $N = 10^7$ elements. This setup is not a coincidence, we decided for these values because they are similar to one of the experiments we present later at Section 5.3. These figures help us to establish a comparison between our theoretical and empirical values.

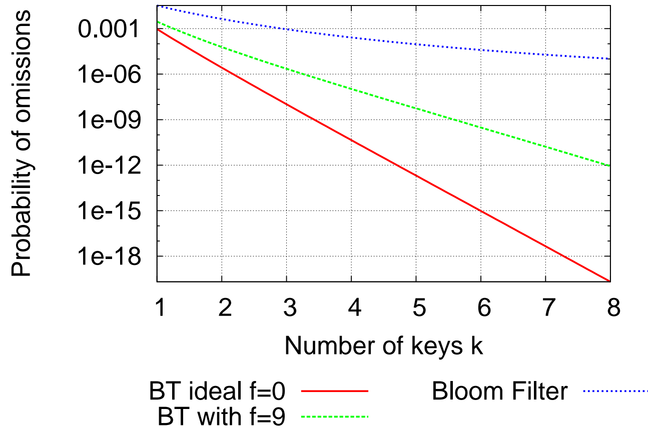


Figure 5.5: Bloom Table Analysis for different number of hash functions k.

Figure 5.5 compares the probability of omissions for the ideal ($f = 0$) and the non ideal BT varying the number of hash functions k (or sets). For the non-ideal BT, we use up to 9 extra chances ($f = 9$). We also depict the probability for the *Bloom Filter* using the same amount of memory — a theoretical *Bloom Filter* of size $M = 2^{31}$. Note that increasing the number of keys (k) improves the probability of omissions, but on the other hand, the difference between the ideal and the non-ideal BT becomes more significant due to our strategy of f chances. Indeed, a good choice for f and k is a trade

off between a low probability of omissions in conjunction with a low rate of rejected elements (elements forward to the overflow table).

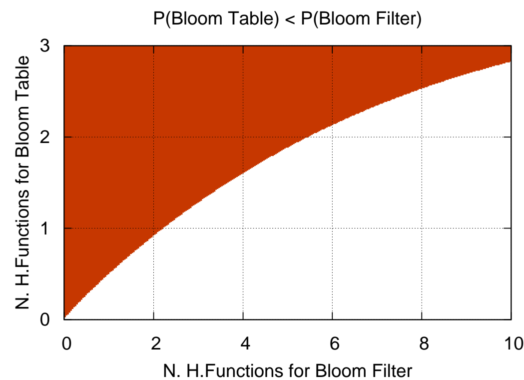


Figure 5.6: Comparison between Bloom Table and Bloom Filter for different number of hash functions k.

Figure 5.6 presents a theoretical comparison between a classical *Bloom Filter* and a non ideal BT with $f = 9$ chances; the highlighted area illustrates the region where **BT** delivers a smaller probability of omissions using fewer hash functions. Later, on Section 5.3, our results confirm this theoretical comparison and show that only two hash functions are enough for a probability of omissions closer to 10^{-5}; a classical filter needs around 5—6 hash functions to achieve the same result.

Figure 5.7 and Figure 5.8 clarify the relationship between k, f and q. These figures highlight some curves which have the same probability but with different parameters. Figure5.7 show that the probability of omission degrades with f. Indeed, the number of f chances must be carefully chosen not to significantly worsen the results. On the other hand, higher values for the parameter f avoids the undesirable problem of rejected elements. At Section 5.3, we try three different values for f (4, 9 and 14) in order to show how this parameter can influence the behavior of the structure in general.

Figure 5.8 presents the relation between the number of keys (k) and the size of the word in bits (q). Increasing the size of the *word* decreases the probability of omissions because it reduces the chances of collisions, on the other hand, it increases the number of bits used per element. In our experiments, we choose to use words of one byte size ($q = 8$) for matters of simplicity.

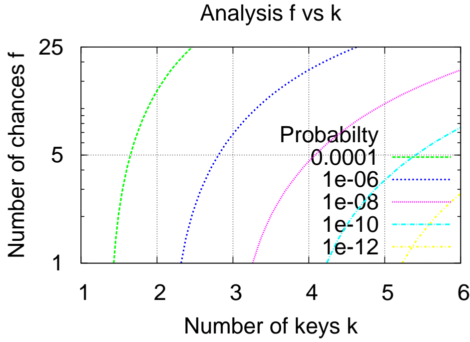


Figure 5.7: Relation k and f.

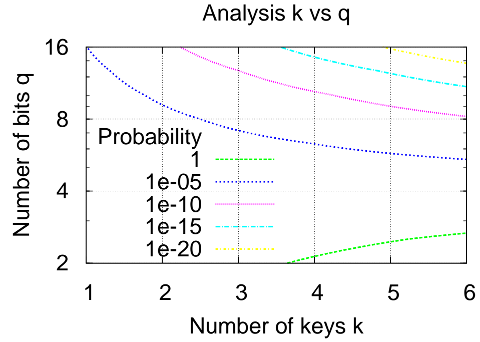


Figure 5.8: Relation k and q.

5.3 Probabilistic Verification

In this section, we go back to our initial motivation, that is the formal verification of systems. We present an application of Bloom Tables as a data structure in a probabilistic verification algorithm. The algorithm is based on the algorithm for exhaustive state space exploration we presented at Section 2.3.3. It starts from the initial state ($S :=$ $Initial$;) by exploring until saturation (*for each* $a \in$ *Enabled(s) do*) all possible successor states ($s_{new} := NewState(s, a)$). Every new state found (*if* $s_{new} \notin S$) is stored in the state graph ($S := S \cup \{n_{new}\}$) with their input arcs ($s \rightarrow^{a} s_{new}$). So, the key point for probabilistic verification is the data structure used to encode the approximate set of generated states.

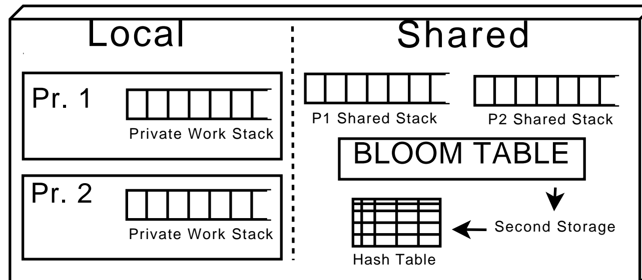


Figure 5.9: Parallel probabilistic verification algorithm overview.

Following the guidelines of this thesis, we implemented a parallel version of this algorithm in order to take benefits from the performance gain offered by multiprocessor

architectures. The parallel implementation is obtained by using a thread-safe concurrent version of our Bloom Table, to store the visited states, and the "work-stealing" strategy to distribute work among the processors. Our parallel implementation is based on our work for parallel exhaustive state space generation (see chapter 3). Like we mentioned before, in this work we decided for a shared hash table as the overflow table in order to best analyze the results. Figure 5.9 depicts an abstract view of the algorithm design.

Experimental Results

We implemented our algorithm as part of our prototype model checker called MERCURY (Appendix B). In brief, we used the C language with Pthreads (But97) for concurrency and the Hoard Library (BMBW00) for parallel memory allocation. We developed our own library for the Bloom Table with support for concurrent insertions and we used Bob Jenkins's hash function (Jen97) to generate hash keys from states. The experimental results presented here are obtained using a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208GB of RAM memory, running the Solaris 10 operating system.

For our experiments, we used the same set of models we experimented with our parallel state space construction algorithm (Section 3.3.3). We have three classical examples: Dining Philosophers (PH); Flexible Manufacturing System (FMS); and Kanban — taken from (MC99) — together with 5 Puzzles models: Peg-Solitaire (Peg); Sokoban; Hanoi; Sam Lloyd's puzzle (Fifteen); and 2D Toads and Frogs puzzle (Frog) — taken from the BEEM database (Pel07). All these examples are based on finite state systems modeled using Petri Nets (Mur89).

For each one of the models selected for our benchmark, we performed a comparison experiment with a *Bloom Filter* using the same amount of memory used by the *Bloom Table* where $M_{BF} = q \cdot M_{BT}$ slots. For the Bloom filter experiments, we tried three different setups for the number of hash functions k; we performed the supertrace ($k = 2$) for performance comparison; we also experimented $k = 5$ and $k = 8$ for state space coverage comparison. For the Bloom Table, we used slots of 8 bits size ($q = 8$) and different setups for m, k and f; we performed testes using $k = \{2, 3\}$ and $f = \{4, 9, 14\}$ to observe the relation between state space coverage and rejected elements from BT, which are the elements stored outside of BT.

The complete benchmark is given at Appendix A. We carried out several experiments to analyze the rate of omissions ($p_{omissions} = 1 - \frac{States\ Found}{States\ Total}$), the execution time and the the rate of rejected elements ($\frac{States\ Rejected}{States\ Total}$). In brief, Bloom Table delivers smaller probabilities of false positive in a shorter execution time when compared to the Bloom filter. Our strategy of multiples insertion f significantly reduces the number of elements inserted at the overflow table (rejected elements). Below, we give a detailed presentation for two experiments among them (Sokoban and Peg-Solitaire).

Legend	
k	number of keys
m	Filter size in bits ($\|M\| = 2^m$)
Om. States	Rate of omitted states Om $= (1 - \frac{States\ Found}{States\ Total})$
T	Time in seconds
f	number of chances
Rej.	Rate of rejected states Rej $= (\frac{States\ Rejected}{States\ Total})$
ld	$\frac{k.n}{m}$ (Load Factor)

Model	Bloom Filter				
States (n)	k	m	ld	Om.	T(s)
Sokoban 7.10^7	2	31	0.06	3.10^{-3}	82
	5	31	0.16	3.10^{-5}	146
	8	31	0.26	2.10^{-6}	183
Solitaire 15.10^7	2	32	0.07	2.10^{-3}	567
	5	32	0.17	6.10^{-5}	803
	8	32	0.28	7.10^{-6}	1067

a)Bloom filter Results.

Model	Bloom Table						
States (n)	k	m	f	ld	Om.	Rej.	T(s)
Sokoban 7.10^7	2	28	4	0.52	6.10^{-6}	0.015	96
	2	28	9		8.10^{-6}	4.10^{-4}	93
	2	28	14		7.10^{-6}	1.10^{-5}	88
	3	28	4	0.78	2.10^{-7}	0.13	122
	3	28	9		4.10^{-7}	0.03	124
	3	28	14		1.10^{-6}	8.10^{-3}	128
Solitaire 15.10^7	2	29	4	0.56	1.10^{-5}	0.05	632
	2	29	9		1.10^{-5}	4.10^{-3}	604
	2	29	14		1.10^{-5}	5.10^{-4}	631
	3	29	4	0.83	5.10^{-7}	0.27	732
	3	29	9		3.10^{-7}	0.17	795
	3	29	14		8.10^{-7}	0.13	899

b)Bloom Table Results.

Figure 5.10: Bloom filter and Bloom Table Results.

Figure 5.10.a) depicts the experiments for Sokoban and Solitaire models using the Bloom filter data structure. For each model, we tried different setups (k and M) with the purpose to achieve high coverage ($p_{omissions} \approx 10^{-5}$). The coverage is improved increasing the number of independent hash keys (k) used by the filter. Although it allows to omit less states without increasing the use of memory, it impacts significantly on the time spent by the algorithm to explore "all possible states". For instance, the experiments running with $k = 8$ for both models took practically twice of the time when

using $k = 2$. Moreover, we observe that high coverability (10^{-5}) is achieved with at least 5 keys.

Figure 5.10.b) presents the results for the same experiments but using our *Bloom Table*. We performed different experiments for each model in order to show the impact of k and f over $p_{omissions}(Om.\ States)$ and the memory used ($M \cdot q$). The number of keys k has a factual impact over the probability of omissions. From $k = 2$ to $k = 3$, we can notice the slight improvement of $p_{omissions}$ from 10^{-6} to 10^{-7}, respectively. However, the number of elements rejected by BT also increases, the rate of elements stored outside goes from 1.5% to 13% of the complete state space when $f = 4$; these are the elements forward to the overflow table and which are not considered as part of BT. This increase on the number of rejected elements is due to the load factor ld, which goes from 0.52 to 0.78. Indeed, only 4 chances are not enough to find an empty slot.

The main weakness of our approach is the number of rejected elements. They have a negative impact over the memory used because it requires additional memory space to store these states. From 5.10, we performed experiments using different values for f ($4, 9$ and 14) in order to keep this weakness at an acceptable level. The increase in the number of chances from 4 to $9, 14$ has a positive impact on the number of elements encoded by BT. For instance, the number of reject elements for the Sokoban model using $f = 9$, $k = 2$ and $M = 2^{28}$ is almost negligible ($Rej = 0.04\%$). In addition, the number of omitted states ($p_{omissions}$) remained virtually unchanged; there is a small depreciation, but the order remains the same (10^{-6}).

Figures 5.12 and 5.11 illustrates the results we presented at Figure 5.10 for the Sokoban model. Figure 5.11 depicts the relation between the probability of omission with the execution time. The lines from this figure highlight experiments that use the same configuration, i.e., the same dimension (m,$M = 2^m$) and, in the case of BT, the same number of chances f. The number of hash keys k used for each experiment is given close to the concerning point. We tried two setups for the BT size m; for the experiments performed using $m_{BT} = 2^{28}$, we have a load factor of $ld = 0.52$ when $k = 2$ and of 0.78 when $k = 3$; for the experiments performed using $m_{BT} = 2^{29}$, we have a load factor of 0.26 when $k = 2$ and of $ld = 0.39$ when $k = 3$. For the Bloom filter, we also experimented two setups, $m_{BF} = 31$ and $m_{BF} = 32$. For the remind of this section, $M_{BF} = 2^{m_{BF}}$ stands for the Bloom filter size and $M_{BT} = 2^{m_{BT}}$ for the BT size.

The results reported at Figure 5.11 are consistent with the theoretical analysis we presented in Figure 5.6, where we showed that the Bloom filter would need at least 5—6 hash keys to achieve a similar probability. For instance, note that all Bloom filter results with $m_{BF} = 31$ and $k \leqslant 5$ are above the BT experiments that used the same amount of memory ($m_{BT} = 28$).

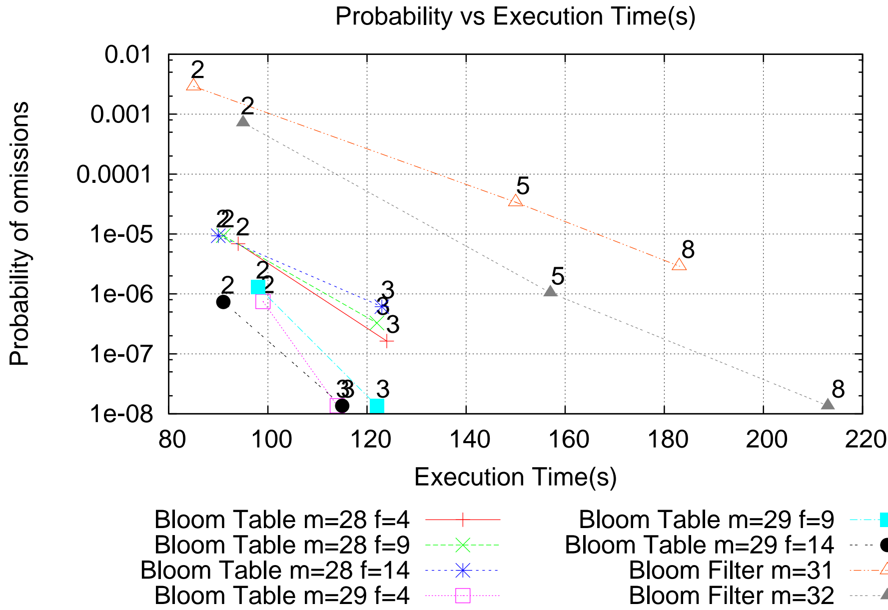


Figure 5.11: Bloom filter and Bloom Table Results for Sokoban model.

Concerning the number of rejected elements, Figure 5.12 gives an overview for the experiment with the Sokoban model. The rate of rejected elements decreases significantly with the increase of f. Regarding the difference between the experiments using 2 or 3 keys (k), it is a consequence of the load factor, i.e., the ratio between the number of elements inserted and the number of available places.

Figures 5.13 and 5.14 depicts the same analysis but for the Solitaire model. We tried again two setups for the BT size m; for the experiments performed using $m_{BT} = 2^{29}$, we have a load factor of $ld = 0.67$ when $k = 2$ and of ≈ 1 when $k = 3$; for the experiments performed using $m_{BT} = 2^{30}$, we have a load factor of 0.33 when $k = 2$ and of $ld = 0.50$ when $k = 3$. The number of rejected elements are smaller than 10% whenever $ld < 0.7$. For the same load factor, these number of rejected elements is smaller than 1% when $k = 2$ and $f = 9$ or $f = 14$. Furthermore, excepted when $ld \approx 1$, the experiments

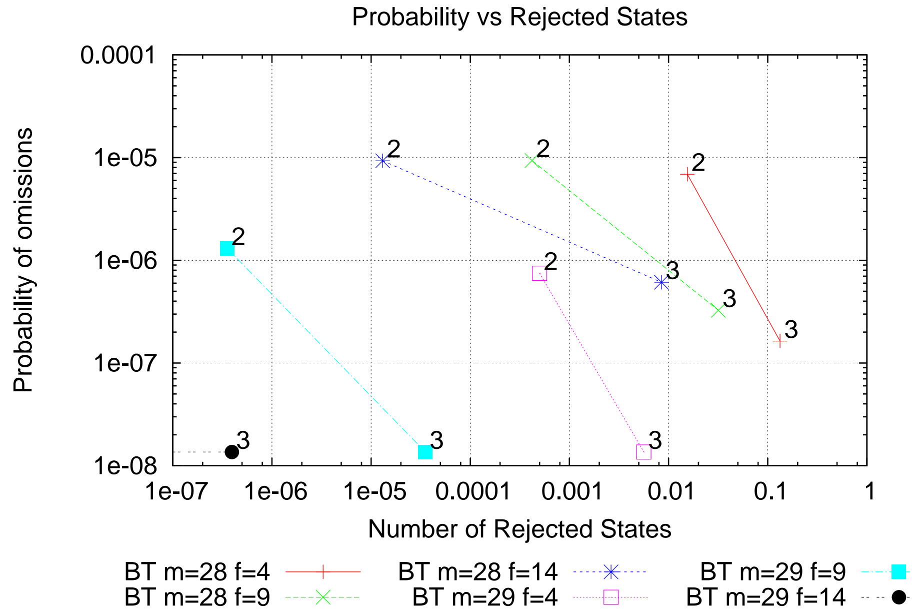


Figure 5.12: Number of rejected elements for Sokoban model.

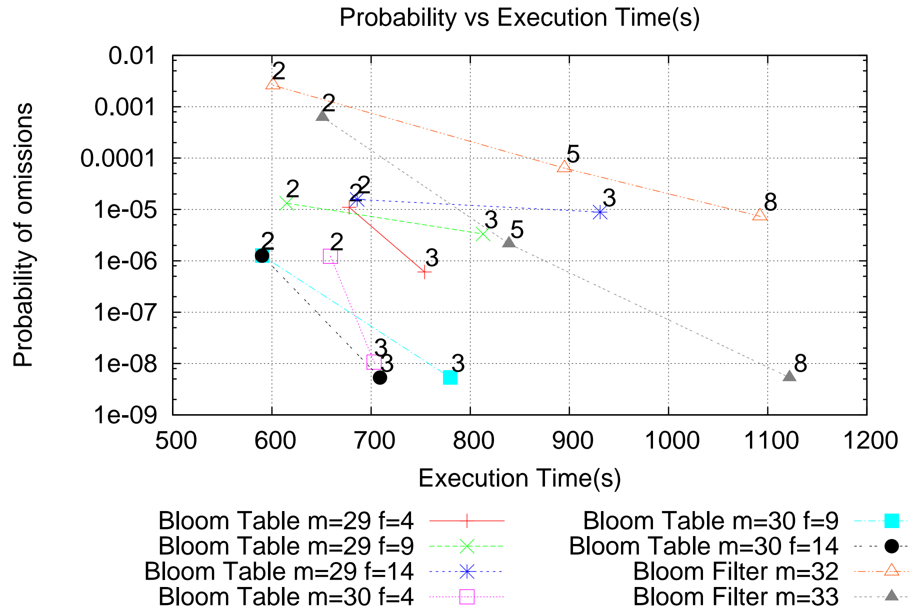


Figure 5.13: Bloom filter and Bloom Table Results for Solitaire model.

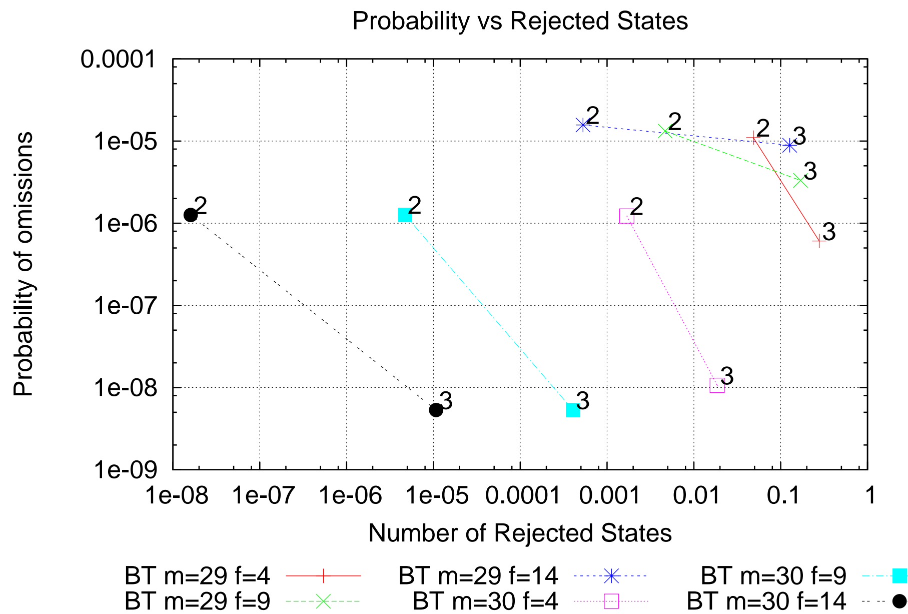


Figure 5.14: Number of rejected elements for Solitaire model.

with $f = 9$ and $f = 14$ reduced the number of rejected elements without degrade the probability of omissions. As might be expected, the strategy to increase the number of chances f does not significantly improve the number of elements encoded at BT for the experiments with high load factors ($ld \approx 1$).

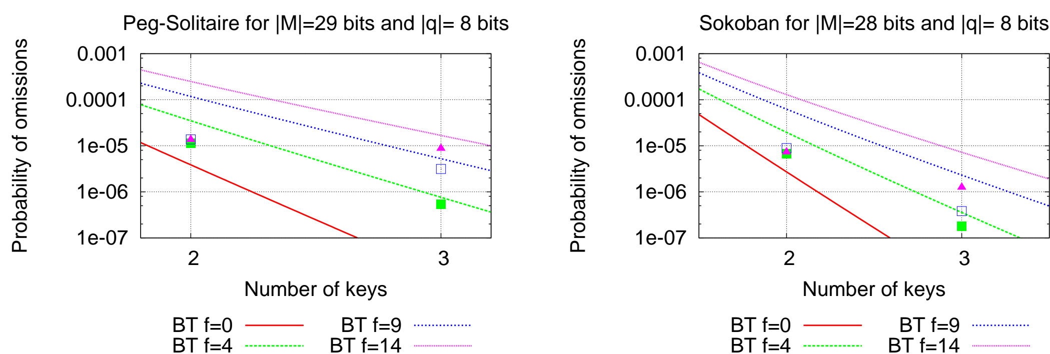


Figure 5.15: Probability vs Number of Keys for Peg and Sokoban models.

Figure 5.15 presents graphically the experiments performed for the Solitaire and Sokoban models using the *Bloom Table*. It compares the obtained experimental results

with the theoretical probability (P_{BT}) presented in section 5.2.2.1. From these graphs, the values obtained for the experiments using $k = 3$ and $f = \{4, 9, 14\}$ are close to the theoretical ones, by contrast, there is almost no difference for the experiments using $k = 2$. Indeed, our theoretical analysis defines an upper bound when the *load factor* between the number of used and available slots is heavily loaded. For such a case, the nth element inserted will certainly need to try all f chances until it finds an appropriate slot. As an illustration, the Solitaire model with $k = 2$ uses a total of 30.10^7 slots (if all states are inserted) among $|M_{BT}| = 2^{29}$ slots, which means a *load factor* of 0.69; for $k = 3$ we have a *load factor* close to 1. In addition, heavy *load factor* also affects the rate of rejected elements.

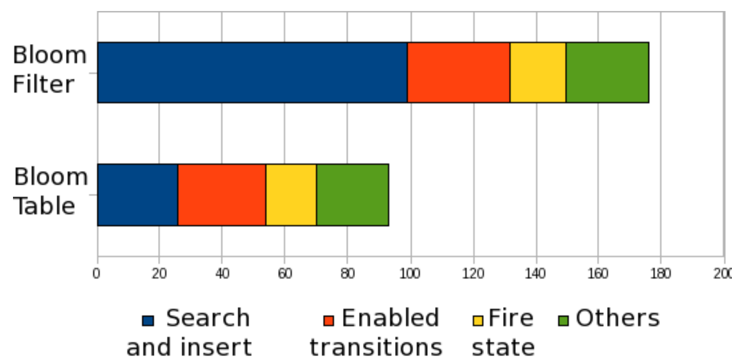


Figure 5.16: Bloom filter vs Bloom Table Execution Profile Overview.

Finally, we present a brief comparison of the execution profile of both data structures. Figure 5.16 depicts the execution profile (in seconds) for both Bloom filter ($k = 8$ and $m_{BF} = 31$) and Bloom Table ($k = 2$, $m_{BT} = 28$ and $f = 14$) for the Sokoban model. We present the time spent divided into four main groups: search and insert operations over the data structure, selection of enabled transitions, operation to fire a transition and others. BT spends fourth of the time needed by the Bloom filter to execute the operations of searching and inserting.

5.4 Conclusions

In this chapter, we presented and analyzed a novel probabilistic data structure for set encoding inspired from the Bloom filter data structure. The main difference of our structure is that we use a vector of small pieces ("words") of information instead of

bits. This small modification allowed us to enhance the probability of the Bloom Table without making use of a large number of keys.

The empirical results we presented in this chapter demonstrated that our data structure performs well and high coverage is achieved without make use of several hash functions, indeed only two are enough. We also show that our strategy of f "chances" can successfully decrease the number of rejected elements allowing us to encode more elements at BT. For instance, less than 1% of elements are stored at the overflow table when BT is set with nine additional chances ($f = 9$). In addition, when compared to the Bloom filter, our data structure achieved similar or better results in a shorter execution time, up to 50% better.

Concerning the use of an overflow table to store the elements rejected by BT, like we mentioned, we decided for an hash table (elements are stored in full size) in order to better analyze our results. However, nothing prevents us from replacing the overflow table with a probabilistic structure or a cache table or even a disk based database (since less than 1% of the membership tests are forward to the overflow table).

For future works, we want to investigate the probabilistic verification of more complex formulas, such as safety and liveness formulas. The probabilistic verification tool that we implemented in this PHD thesis supports only reachability formulas. We believe that it is interesting to use probabilistic tools for finding errors and we encourage its use in premature stages of the development cycle of critical system. A possible suite of tests could rely on probabilistic verification in the early stages and exhaustive in later stage. The probabilistic verifications tests performed early would not require deep levels of abstraction to reduce the number of states due to their efficiency in terms of memory. Abstractions would be necessary for the exhaustive tests performed later in the development cycle.

Chapter 6

Conclusions

" We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Turing

This thesis describes our efforts to perform model checking of finite systems on multiprocessors and multi-core machines. We proposed new data structures and novel algorithms that are applied to three main application domains: parallel state space construction, parallel model checking and probabilistic verification.

Obviously, we want to benefit from the increased computing power brought by multi-core machines. Moreover, the transition to concurrent model checking algorithms seems unavoidable. There is no doubt that all model checking tools will run on multi-core computers; for the simple reason that every new computer—of reasonable power—is equipped with multi-core processors[1].

Apart from the gain in performance, we are also very much interested in the huge amount of fast-access memory provided by current multiprocessor servers; like the one used in our experiments. Recall that all the experimental results presented in this manuscript have been performed on a server with 208 GB of RAM—we affectionately called the machine Brutus—quite an increase compared to the 8–32 M available to the first model checking tools in the 1980s.

In Chapter 3 of this thesis, we describe new algorithms to perform exhaustive, explicit, state space construction in parallel. Our work is build on a unique design,

[1]Actually, we already have users asking for a concurrent version of the model checking toolbox tina (BB04).

that is based on a dynamic distribution of states and the use of lock-less concurrent data structures. Chapters 4 and 5 describe new approaches to perform parallel model checking and probabilistic verification that are built on top of this parallel state space generation infrastructure.

We see exhaustive model checking and probabilistic verification as two complementary approaches. On one hand, probabilistic methods should be considered whenever we try to find counter-examples to a specification. On the other hand, our exhaustive approach is more appropriate in the last phases of the conception of (the model of) a system, or when we need to check complex specifications. For this reason, we optimized our model checking algorithm, called MCLCD, for the case where the specification is true.

We propose memory efficient algorithms for the two approaches. Indeed, we believe that memory is the most important resource for (explicit) model checking. This is the reason why we put a lot of efforts on reducing the memory usage; even if it means trading memory space for computation time. For example, we define a version of our MCLCD algorithm that do not require to store the transition relation of a system. We believe that the solution we propose in this context is original.

Likewise, we define a new data structure for probabilistic verification, called BT, that requires only 2 bytes per state in average, while still offering a very good coverage. (This data structure has also been designed in order to be "friendly" for concurrent programming). For example, with BT, we have a 10^{-5} probability of missing a state in a system with one billion states and using 2.5 GB of memory. It is not possible to reach the same accuracy using the "hash-compact" approach and the same amount of memory. Similarly, a "Bloom Filter" approach using two hash-keys will have an accuracy in the order of $10^{-2} - -10^{-3}$ with the same setup. More than 5 hash keys are necessary to obtain an accuracy of 10^{-5} but, in this case, the computation is slower than with BT.

Next, we summarize the contributions of this thesis.

[Parallel State Space Construction]

In Chapter 3, we define two new algorithms for parallel state space construction. We believe that our contributions are of interests in several domains. In the formal verification domain, we define new algorithms for parallel state space construction. In the

parallel computing domain, we propose new lock-less data structures for concurrent hash maps.

Our approach is based on the use of distributed, local dictionaries (for storing states) and a probabilistic data structure (e.g. Bloom Filters) for dynamically assigning the states to a process.

We propose two algorithms: a general (speculative) algorithm—where collisions are solved using inter-process synchronization—and a mixed algorithm, where collisions are solved on-the-fly. Our experimental results show that the mixed version is the best choice in practice.

The main innovation is the definition of a data structure, named Localization Table (LT), that is used to coordinate a network of local hash table in order to obtain an efficient concurrent hash map.

We observe very good speedups in our experimental results. The speedup for the mixed algorithm is consistently better than with other parallel algorithms. We also show that our implementation performs well when compared with related tools; we show that, in our algorithm, it is better to use the localization table than the concurrent hash map provided by Intel-TBB (an industrial strength lock-less hash table).

We believe that our Localization Table can be of great interest outside of the domain of parallel model checking. For this reason, we are planning to provide a functional API of our distributed hash table—completely self contained—that could be used in other situations and that will require only minimal configuration.

[Parallel Model Checking]

In Chapter 4 we define an algorithm for model checking CTL formulas. We choose a semantic-based algorithm—adapted for concurrent, shared memory architectures—because we believe that it is more appropriate for a parallel algorithm with dynamic work-load strategies.

We have two versions of this algorithm: a first version based on a reverse traversal of the state graph, called RG, where we need to explicitly store the transitions of the system; and a second version, RPG, where we only need to store a spanning subgraph. The RG version has a linear time and space complexity, in $O(S + R)$ (where S is the number of states and R the number of transitions), while the RPG version has a time complexity in $O(S \cdot (R - S))$ and a space complexity in $O(S)$. The main advantage of

the RPG version is to provide an algorithm that is efficient in memory and independent of the choice of states classes abstraction.

Our prototype implementation shows promising results for both the RG and RPG versions of the algorithm. The choice of a "labeling algorithm" has proved to be a good match for shared memory machines and a work stealing strategy; for instance, we consistently obtained speedups close to linear with an average efficiency of 75%. Our results also show that the RPG algorithm is able to match the performance of RG for some type of models.

We believe that the two versions of the algorithm are complementary. RG is the best choice whenever there is enough memory to store the complete state graph, otherwise, RPG should be used despite its higher time complexity.

For future works, we are studying an improved version of our algorithms that supports the complete set of CTL formulas.

[Probabilistic Verification]

The last technical chapter of this thesis, chapter 5, is dedicated to a new approach for probabilistic verification. We define a new data structure inspired from our work on the localization table and compare it with the *Bloom Filter* and *Compact Hash Table* data structures. The main difference of our structure, when compared to the classical *Bloom Filter*, is that we use a vector of "w-bits words" instead of a vector of bits.

When compared to simpler hash-based techniques, our Bloom Table is able to increase the probability of generating the whole state space of a system provided when we have a rough estimate of the size of this state space.

Our experimental results show that our data structure performs well and results in a good coverage of the state space without requiring too many operations on hash functions. (In our experiments, 2 hash-functions are generally enough.) Additionally, when compared to the use of a *Bloom Filter*, our data structure achieve similar or better results with a shorter execution time, sometimes by a factor of 2.

For future works, we want to investigate the probabilistic verification of more complex formulas, such as safety and liveness formulas. The idea is to be able to use probabilistic verification methods, at the same time than exhaustive model checking

algorithm, without any restriction on the specification. Probabilistic verification techniques will be used early in the development cycle, while model checking will be use to “certify” the absence of errors at the model-level

Appendix A

Experiments

This appendix presents all the experiments we performed for probabilistic verification and parallel state space construction. Section A.1 describes the set of models used in our benchmarks. Section A.2 presents all the experiments we performed for parallel state space construction (see Chapter 3). In Section A.3, we give all the experiments we performed for probabilistic verification (see Chapter 5).

A.1 Models

The finite state systems chosen for our benchmarks were taken from two sources. We experiment three classical examples of Petri Nets from (MC99) together with 5 Puzzles models from the BEEM database (Pel07). Figure A.1 presents all selected models highlighting their respective version. The first column illustrates the abbreviations used fallowed by a brief description. The version chosen for each model was motivated by the number of states, we selected models with less than 5.10^8 states. The last columns gives the source from where each model was extracted.

A.2 Parallel State Space Construction

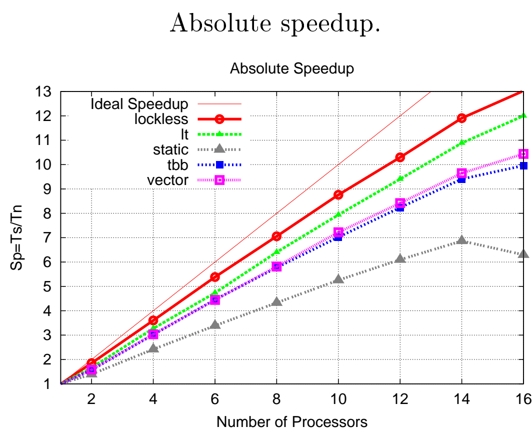
In Chapter 3, we presented two algorithms for parallel state space construction. In Section 3.4, we presented a comparison of our mixed variant with other algorithms proposed on the literature. We reported the average results we obtained on our set of experiments. Below, we depict in details each experiment giving the absolute speedup, the physical distribution of states and the execution time between 6 to 16 processors.

Model States	Description	Parameter	Source
PH 14.10^7	Dining Philosophers	*13 subnets*	(MC99)
FMS 24.10^7	Flexible Manufacturing System	*initial marking* *weight 8*	(MC99)
Kanban 38.10^7	Kanban System	*initial marking* *weight 9*	(MC99)
PEG 18.10^7	Peg-Solitaire Game	*version=2,* *crossways=1*	(Pel07)
Sokoban 7.10^7	Computer Maze Game	*version=2*	(Pel07)
Hanoi 38.10^7	Tower of Hanoi puzzle	n=*17*	(Pel07)
Fifteen 23.10^7	Sam Lloyd's fifteen puzzle	*cols=4,* *rows=3*	(Pel07)
Frog 53.10^7	2D Toads and Frogs puzzle	n=*6,* m=*5*	(Pel07)

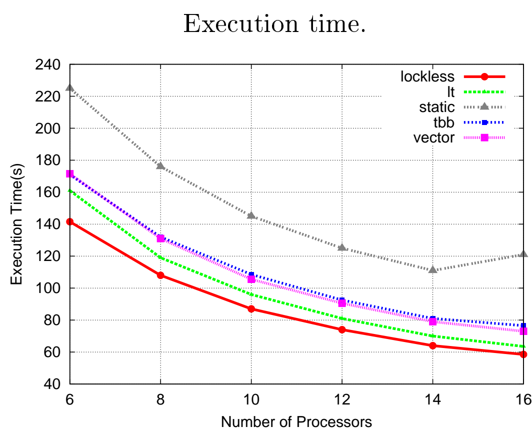
Figure A.1: Benchmark Examples.

- Sokoban

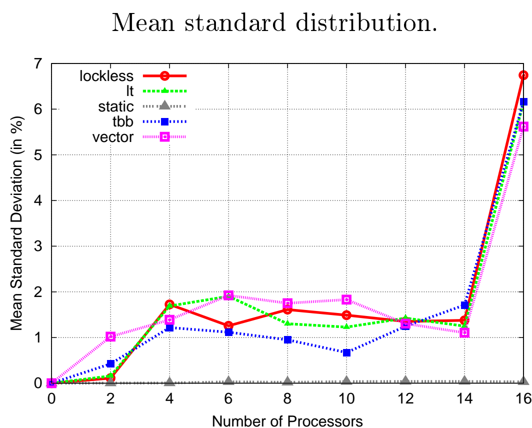
Absolute speedup.



Execution time.

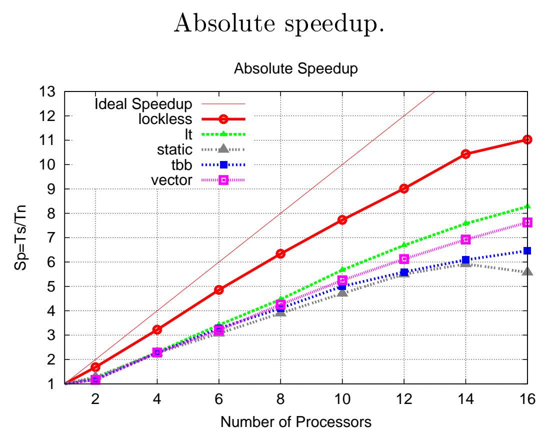


Mean standard distribution.

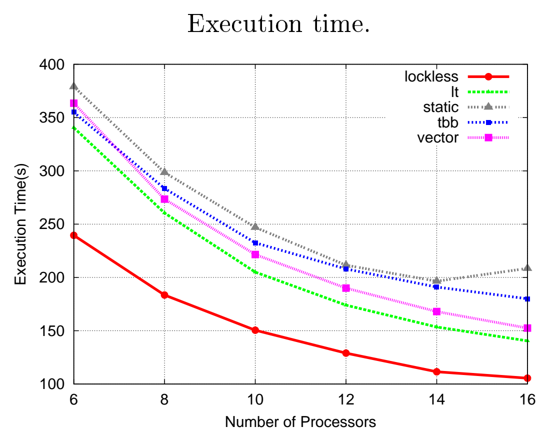


- Peg-Solitaire

Absolute speedup.



Execution time.



Mean standard distribution.

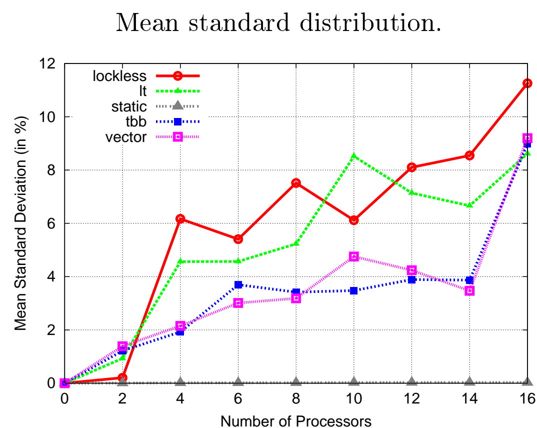
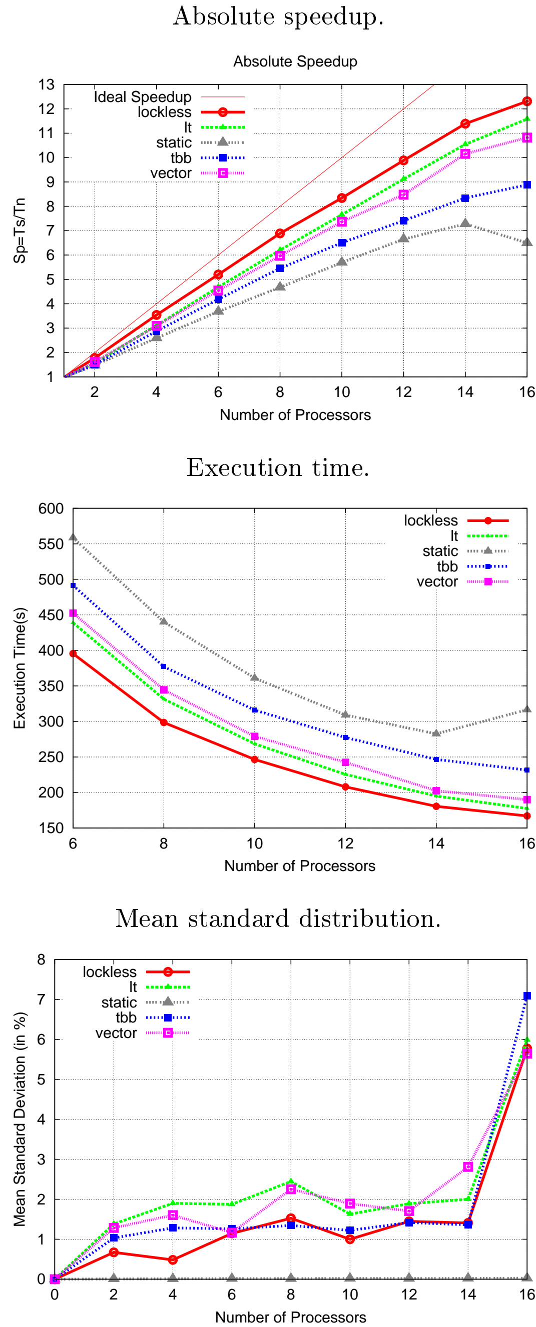


Figure A.2: Parallel State Space Construction for the Sokoban and Peg-Solitaire models.

Fifteen

FMS



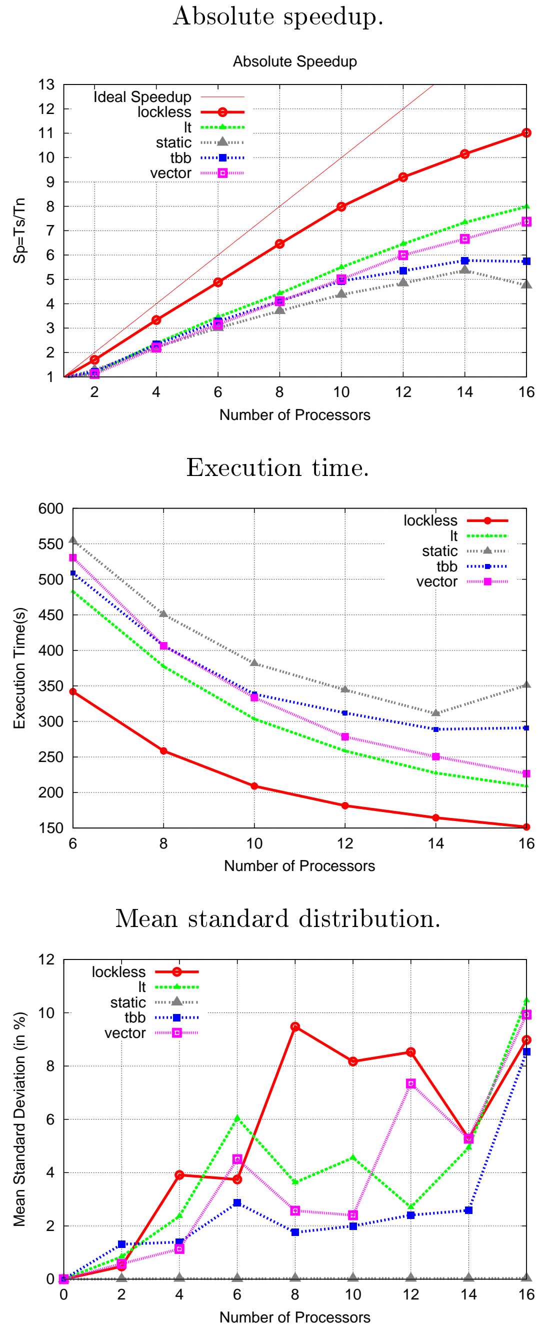
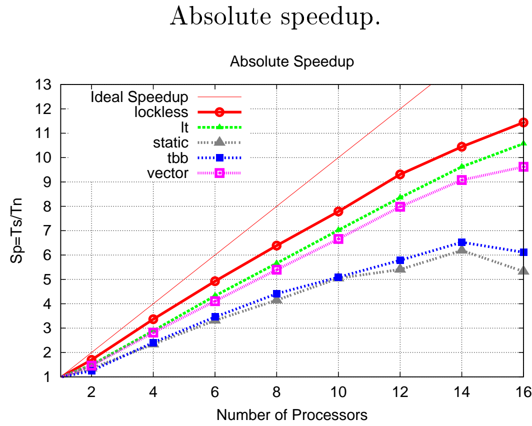


Figure A.3: Parallel State Space Construction for the Fifteen and FMS models.

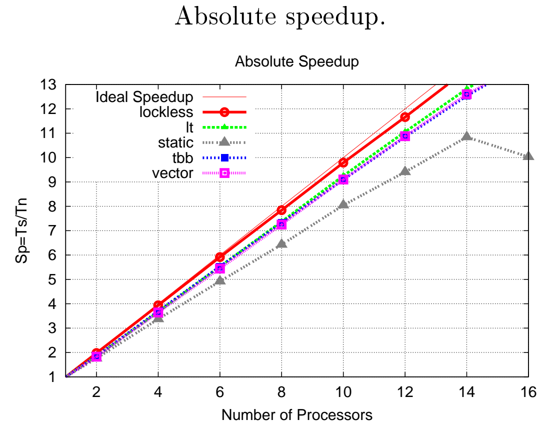
Frog

Absolute speedup.

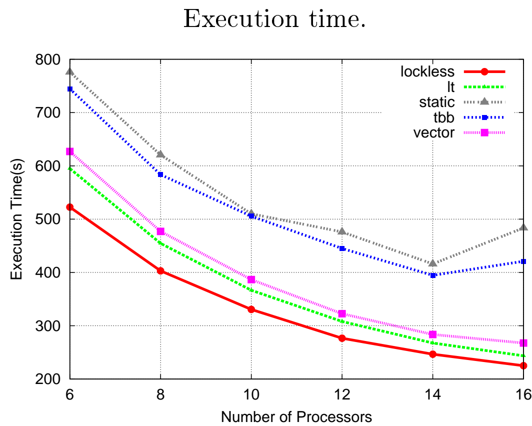


Hanoi

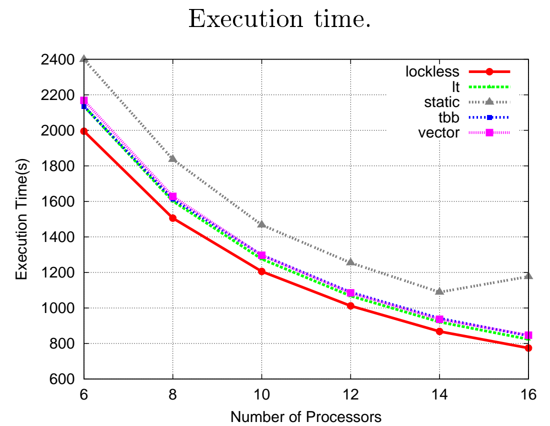
Absolute speedup.



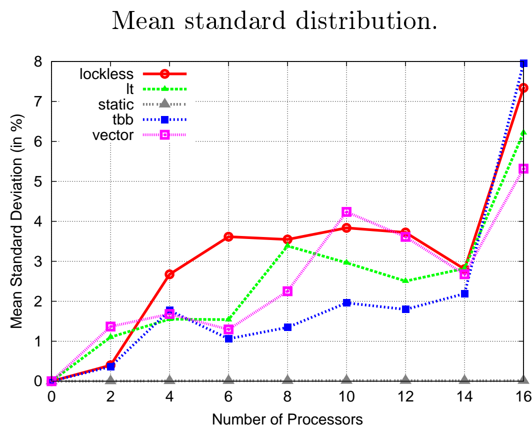
Execution time.



Execution time.



Mean standard distribution.



Mean standard distribution.

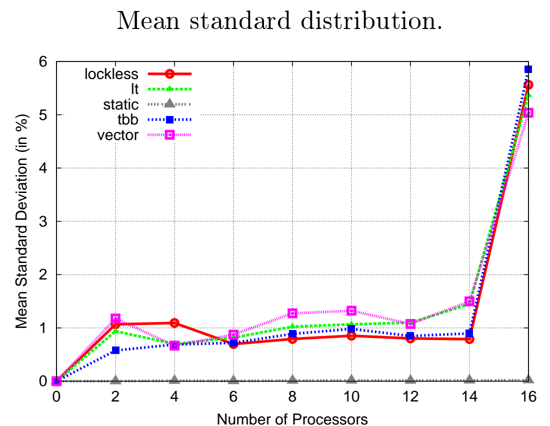
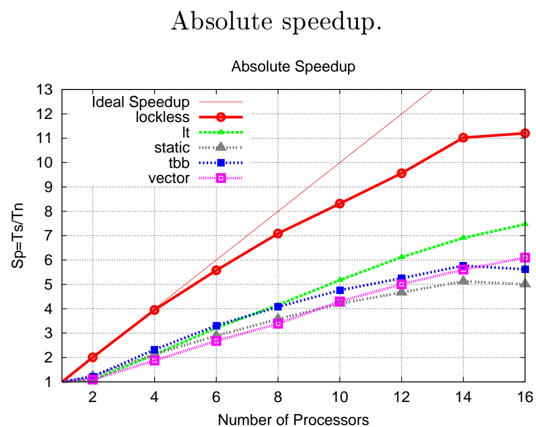


Figure A.4: Parallel State Space Construction for the Frog and Hanoi models.

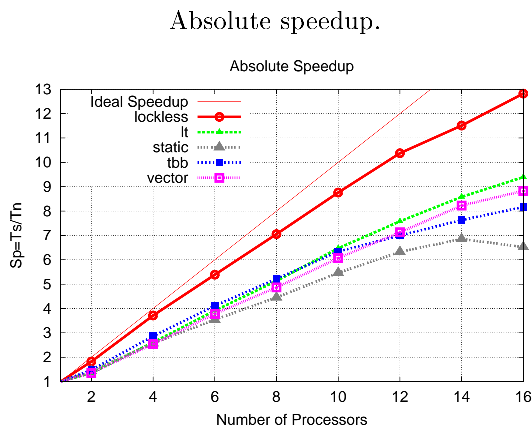
Kanban

PH

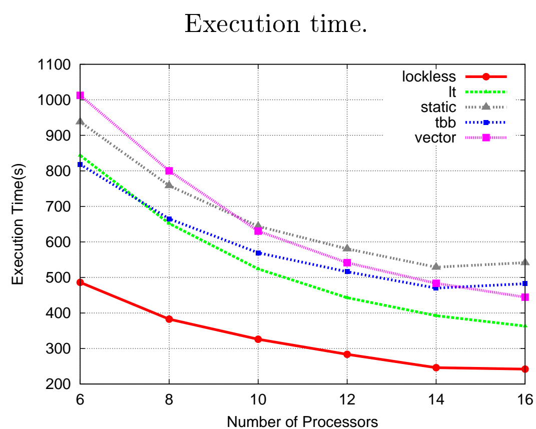
Absolute speedup.



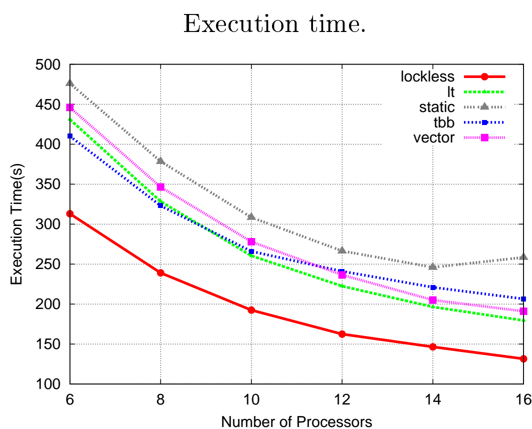
Absolute speedup.



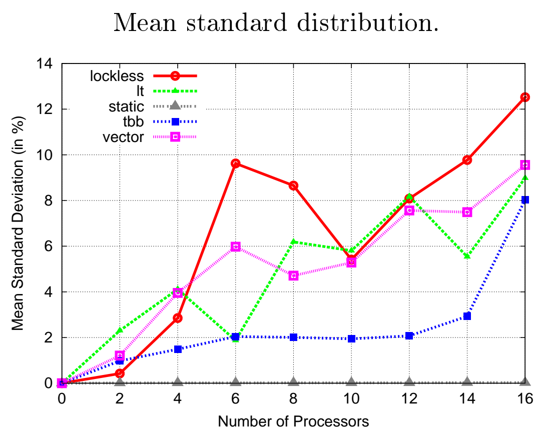
Execution time.



Execution time.



Mean standard distribution.



Mean standard distribution.

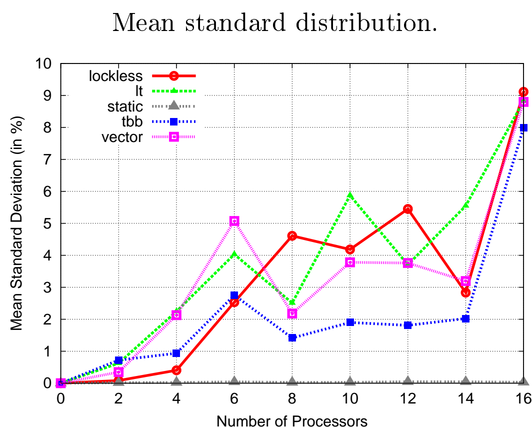


Figure A.5: Parallel State Space Construction for the Kanban and PH models.

A.3 Probabilistic Verification

In Chapter 5 we introduced and analyzed a new probabilistic data structure called Bloom Table. In this section, we report the set of experiments we performed in the context of probabilistic verification. For each model, we performed a comparison experiment with a *Bloom Filter* using the same amount of memory used by the *Bloom Table* where $M_{BF} = q \cdot M_{BT}$ slots. For the Bloom filter experiments, we tried three different setups for the number of hash functions k; we performed the supertrace ($k = 2$) for performance comparison; we also experimented $k = 5$ and $k = 8$ for state space coverage comparison. For the Bloom Table, we tried several experiments in order to analyze the impact of different load factors (m) over the parameters k and f. We used slots of 8 bits size ($q = 8$) and different setups for m, k and f; we performed testes using $k = \{2, 3\}$ and $f = \{4, 9, 14\}$.

For these experiments, the load factor is obtained by the equation

$$ld = \frac{k.n}{m}$$

where k is the number of keys, n the number of states inserted and m the number of available slots.

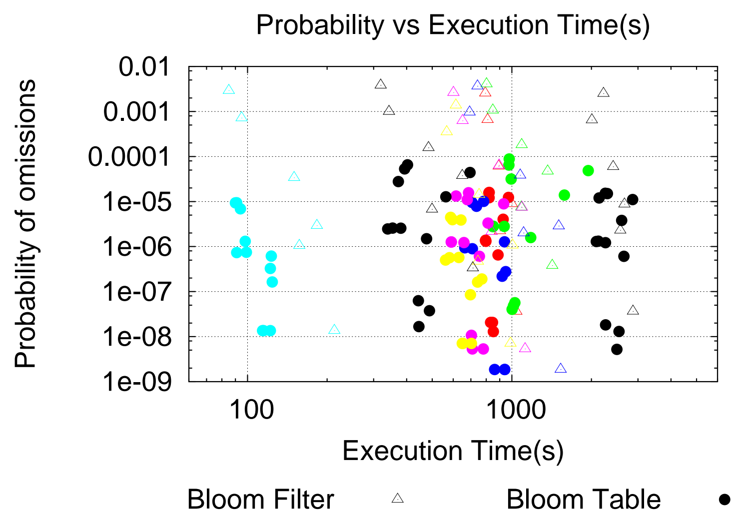


Figure A.6: Probability vs Execution Time.

Figure A.6 compiles together all the experiments we performed (each different color represents an example) and compares the results achieved using the Bloom filter (triangles) and the *Bloom Table* (dot circles) data structures. This figure shows the rate of omissions ($p_{omissions} = 1 - \frac{States\ Found}{States\ Total}$) and the execution time for each experiment. We observe the predominance of Bloom filter results (triangles) in the upper right, in contrast with the *Bloom Table* results (dot circles) on the bottom left. This figure shows briefly that, for the our benchmark, Bloom Table delivers smaller probability of false positive in a shorter execution time. Later, we give a more detailed presentation for each experiment.

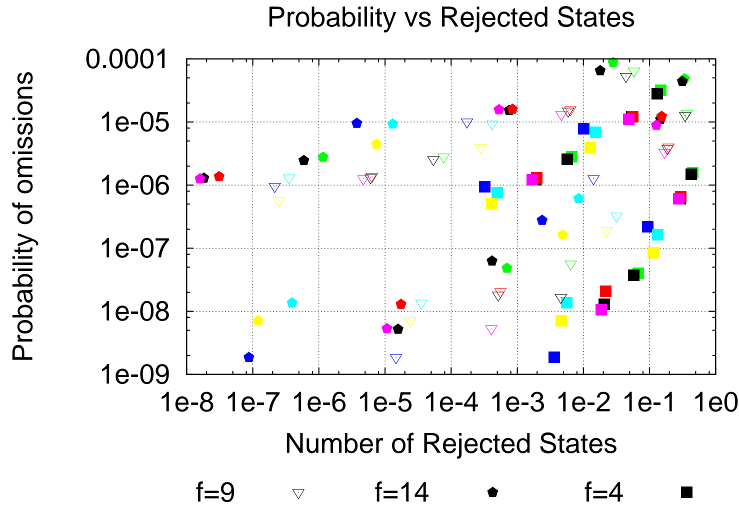


Figure A.7: Probability vs Rejected States.

Figure A.7 depicts the relation among the parameter $f = \{4, 9, 14\}$ (number of chances), the rate of rejected elements ($\frac{States\ Rejected}{States\ Total}$) and the probability to omit a single state for the experiments we performed. The experiments with $f = 9$ and $f = 14$ are more predominant on the left side. So, the parameter f allows BT to encode more elements and, by consequence, less elements are inserted into the overflow table. The experiments placed at the upper right corner show a worsening of their relative probabilities with the increase of the number of chances because of the load factor between BT and the number of entries, more explanations are given below (Figure 5.10).

- Fifteen: Figures A.8 and A.9 depicts the probabilistic experiments for the Fifteen model. We tried two setups for the BT size m; for the experiments performed using $m = 2^{29}$, we have a load factor of $ld = 0.85$ when $k = 2$ and of ≈ 1 when $k = 3$; for the experiments performed using $m = 2^{30}$, we have a load factor of 0.42 when $k = 2$ and of $ld = 0.64$ when $k = 3$. The number of rejected elements are smaller than 10% whenever $ld < 0.9$. For the same load factor, the number of rejected elements are close to 1% when $k = 2$ and $f = 9$ or $f = 14$.

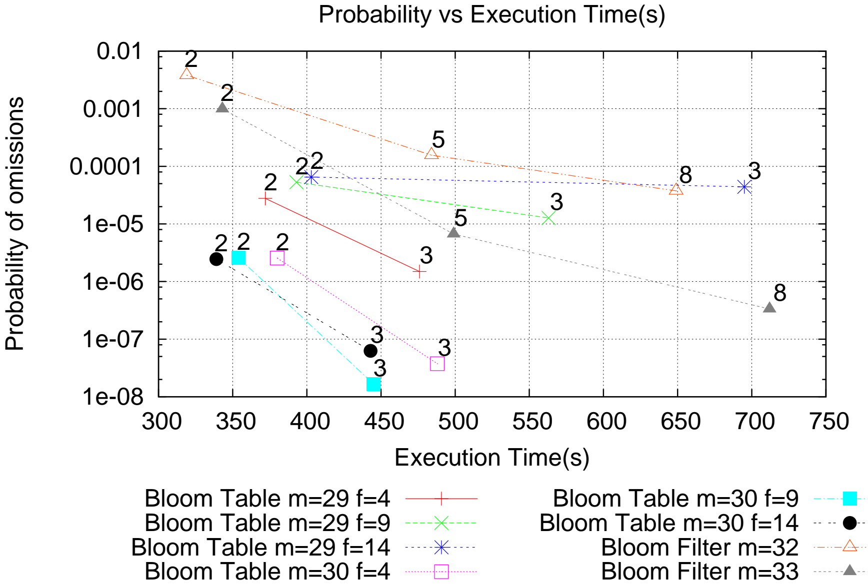


Figure A.8: Bloom Filter and Bloom Table Results for Fifteen model.

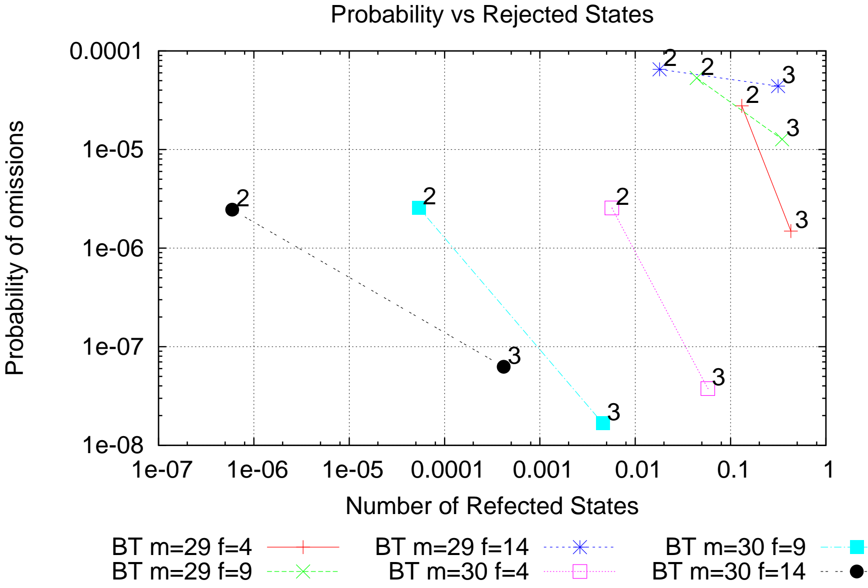


Figure A.9: Number of rejected elements for Fifteen model.

- FMS: Figures A.10 and A.11 depicts the probabilistic experiments for the FMS model. We tried two setups for the BT size m; for the experiments performed using $m = 2^{29}$, we have a load factor of $ld = 0.89$ when $k = 2$ and of ≈ 1 when $k = 3$; for the experiments performed using $m = 2^{30}$, we have a load factor of 0.44 when $k = 2$ and of $ld = 0.67$ when $k = 3$. The number of rejected elements are smaller than 15% whenever $ld < 0.9$. For the same load factor, the number of rejected elements are close to 5% when $k = 2$ and $f = 9$ or $f = 14$.

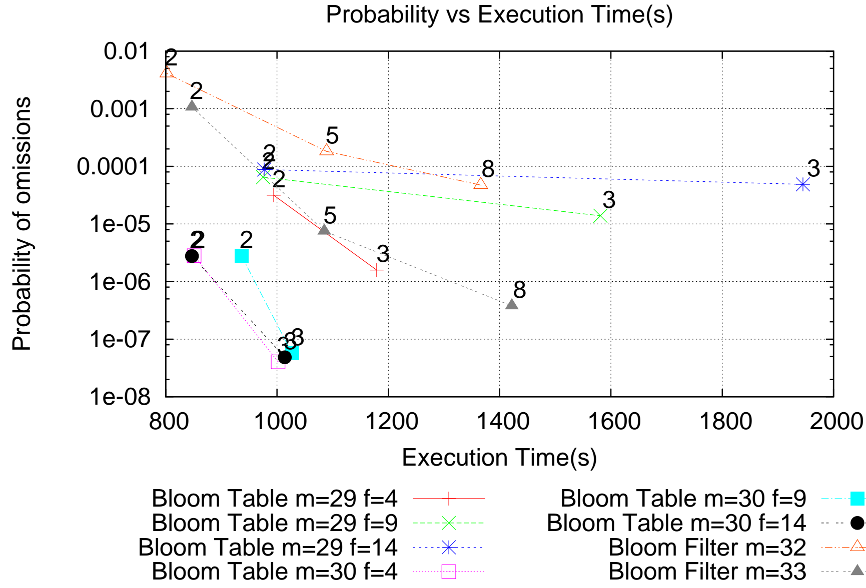


Figure A.10: Bloom Filter and Bloom Table Results for FMS model.

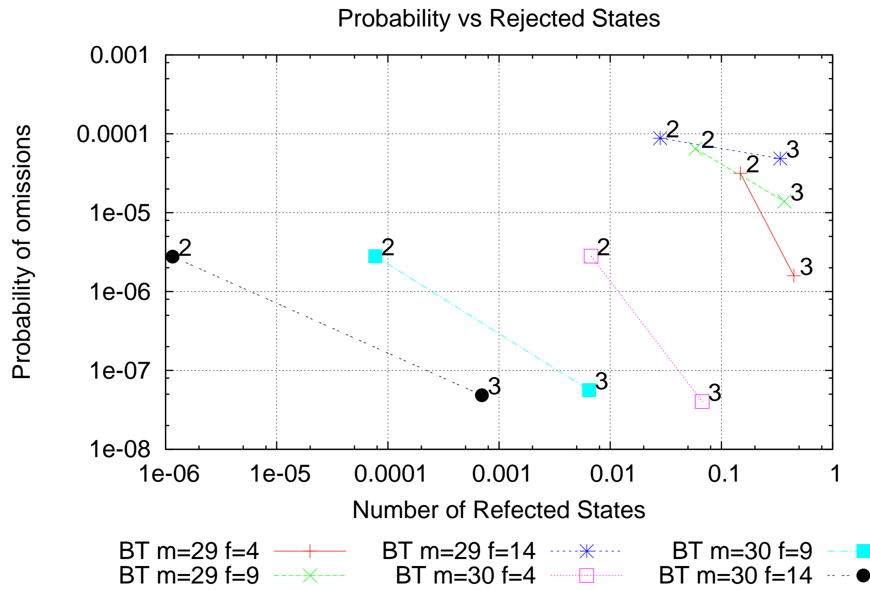


Figure A.11: Number of rejected elements for FMS model.

- Frog: Figures A.12 and A.13 depicts the probabilistic experiments for the Fifteen model. We tried two setups for the BT size m; for the experiments performed using $m = 2^{31}$, we have a load factor of $ld = 0.49$ when $k = 2$ and of $ld = 0.74$ when $k = 3$; for the experiments performed using $m = 2^{32}$, we have a load factor of 0.24 when $k = 2$ and of $ld = 0.37$ when $k = 3$. Regarding the experiments, the number of rejected elements are smaller than 1% for all the experiments where $f = 9$ or $f = 14$, independent of load factor.

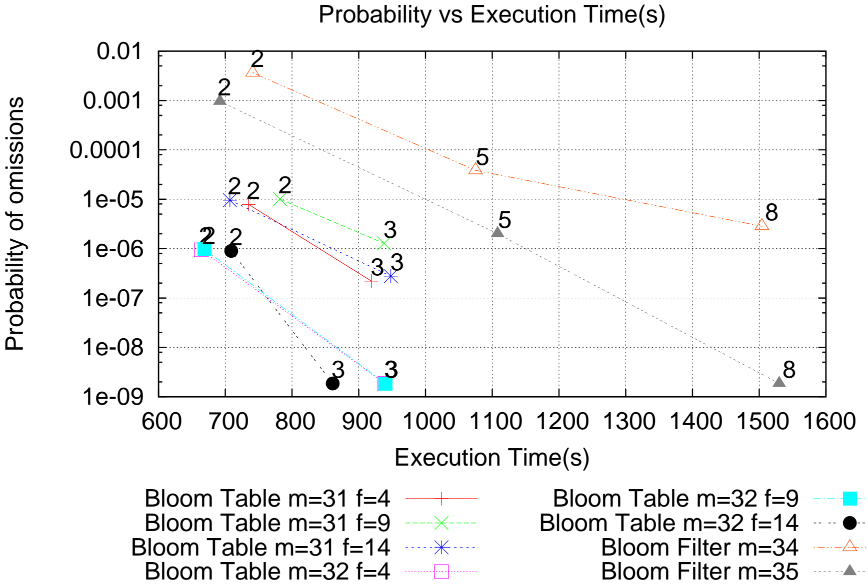


Figure A.12: Bloom Filter and Bloom Table Results for Frog model.

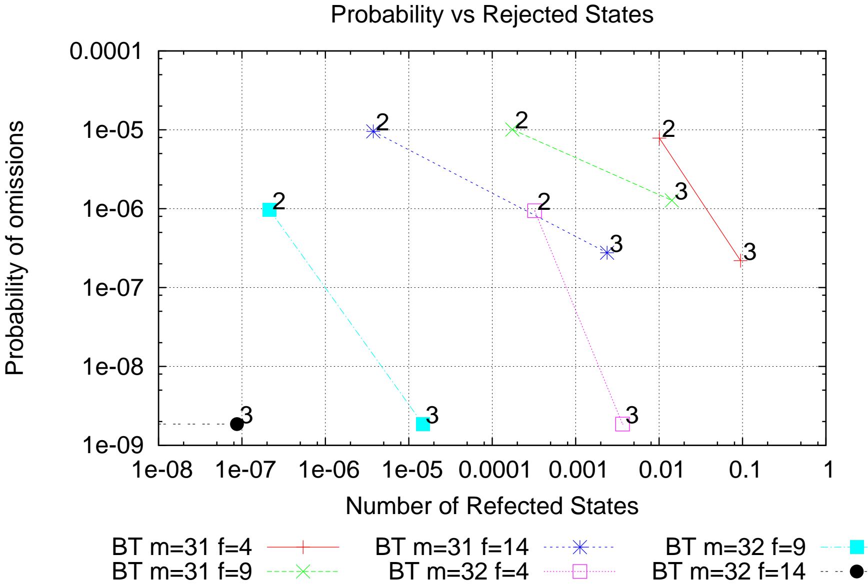


Figure A.13: Number of rejected elements for Frog model.

- Hanoi: Figures A.14 and A.15 depicts the probabilistic experiments for the Hanoi model. We tried two setups for the BT size m; for the experiments performed using $m = 2^{30}$, we have a load factor of $ld = 0.7$ when $k = 2$ and of $ld \approx 1$ when $k = 3$; for the experiments performed using $m = 2^{31}$, we have a load factor of 0.35 when $k = 2$ and of $ld = 0.53$ when $k = 3$. For the experiments where $ld < 7$, the number of rejected elements are smaller than 1% for all whenever $f = 9$ or $f = 14$.

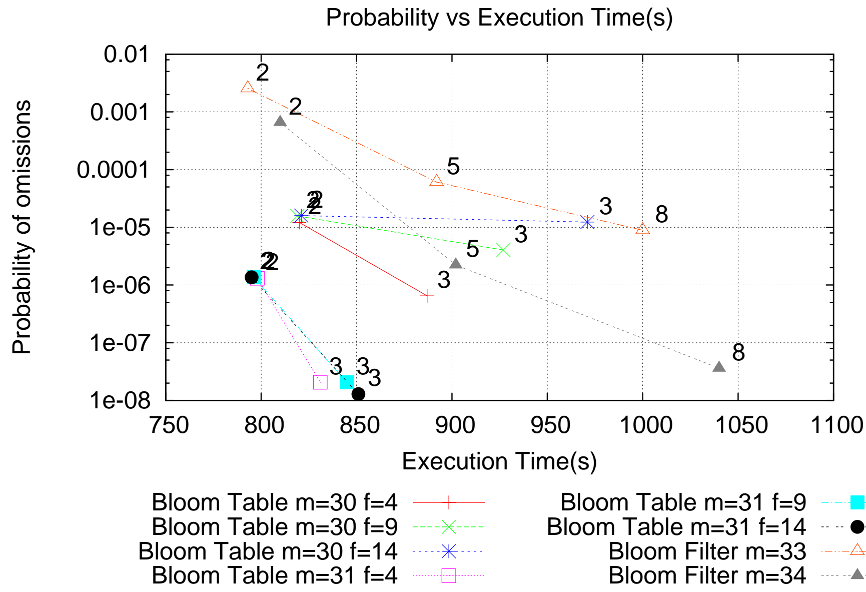


Figure A.14: Bloom Filter and Bloom Table Results for Hanoi model.

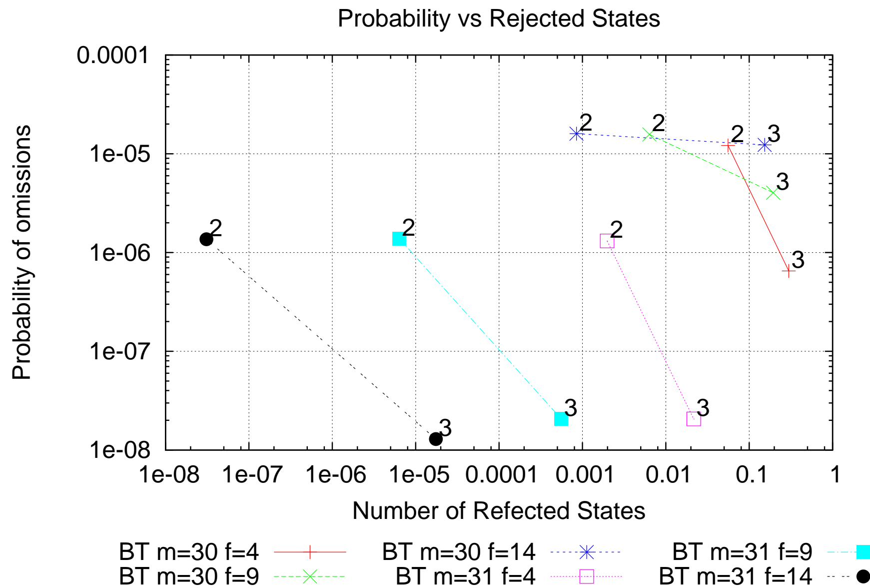


Figure A.15: Number of rejected elements for Hanoi model.

- Kanban: Figures A.16 and A.17 depicts the probabilistic experiments for the Kanban model. We tried two setups for the BT size m; for the experiments performed using $m = 2^{30}$, we have a load factor of $ld = 0.7$ when $k = 2$ and of $ld \approx 1$ when $k = 3$; for the experiments performed using $m = 2^{31}$, we have a load factor of 0.35 when $k = 2$ and of $ld = 0.53$ when $k = 3$. For the experiments where $ld < 7$, the number of rejected elements are smaller than 1% for all whenever $f = 9$ or $f = 14$.

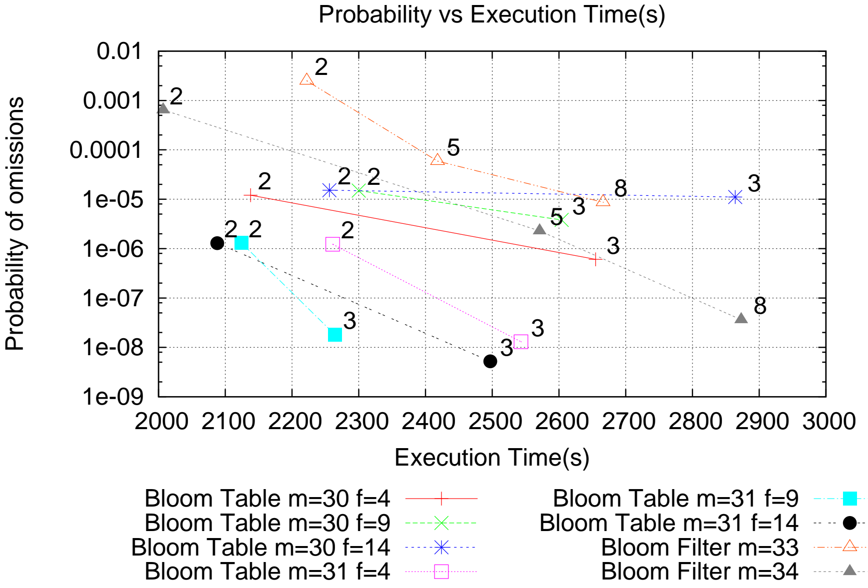


Figure A.16: Bloom Filter and Bloom Table Results for Kanban model.

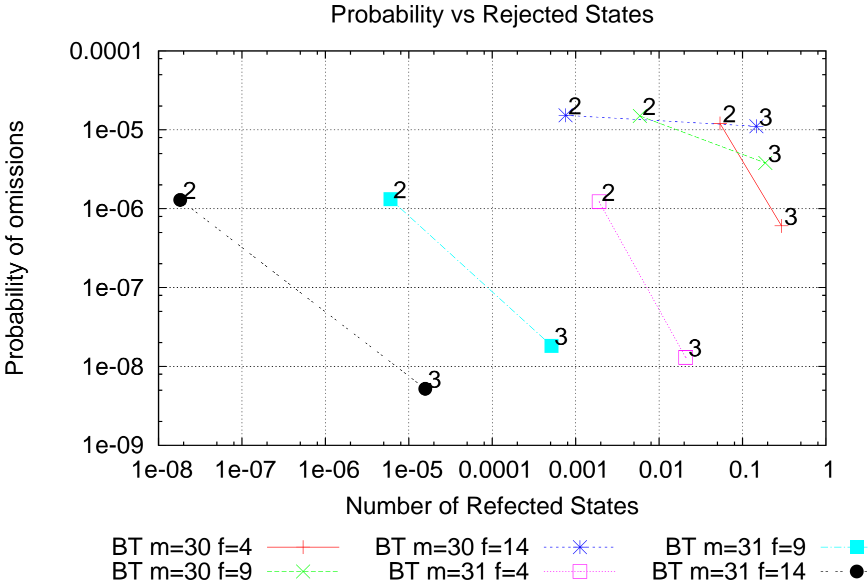


Figure A.17: Number of rejected elements for Kanban model.

Appendix B

Mercury

In this section, we describe our prototype software, called MERCURY, that has been developed in order to experiment different approaches for Parallel State Space construction and Parallel Model Checking. All the algorithms implemented in MERCURY follow a SPMD approach, such that each processor executes the same program. MERCURY has been developed to be highly modular and extensible. The software is composed of separate, interchangeable modules that accept different memory layouts (shared/local data) and synchronization mechanisms. Moreover, the software view of states is abstract and can be easily extended to take into account data structures and time classes. Altogether, we experimented 11 versions, two for probabilistic, six for exhaustive state space construction and three for parallel model checking (subset of CTL formulas).

B.1 Technical Description

MERCURY is implemented using the C language with Pthreads (But97) for concurrency and the Hoard Library (BMBW00) for parallel memory allocation. Except for the Concurrent Hash Table from the Intel-TBB framework (Rei07) (that is used to perform benchmarks), we implemented our own libraries for the data dictionaries. It is instrumented to support Bob Jenkins's hash function (Jen97). For all our experiments, we used binaries generated using the Oracle Sun compiler (cc); we currently support Oracle Solaris operating system.

MERCURY is designed to accommodate different configurations. As we mentioned before, the performance of a parallel algorithms is strongly linked to the concurrent

data structures that are used (data dictionaries in our case). In the context of this project, we considered a modular architecture in order to experiment with different dictionaries and layouts. We have five different classes of modules (we give an overview of the relations between modules in Figure B.1):

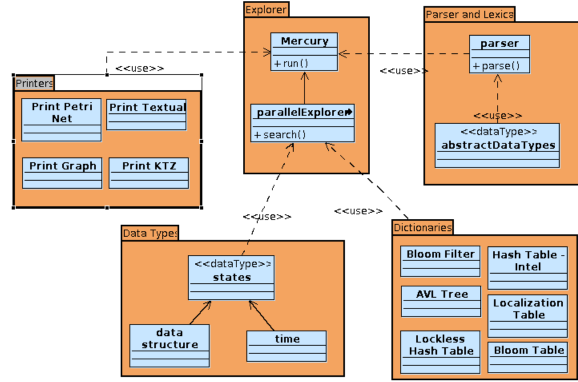


Figure B.1: Mercury Modules.

1. A parser and lexical module to interpret the model. We currently accept Petri Net models expressed in the .net format (the same format that is used in Tina).

2. A graph exploration engine module to define the strategy for exploring the graphs (e.g. DFS or BFS). The module is also in charge of implementing the work-sharing strategies, see Chapter 3.

3. A data type module to define the representation of states. We have experimented with different representation for markings. This module will also be used when we extend our models with data and time.

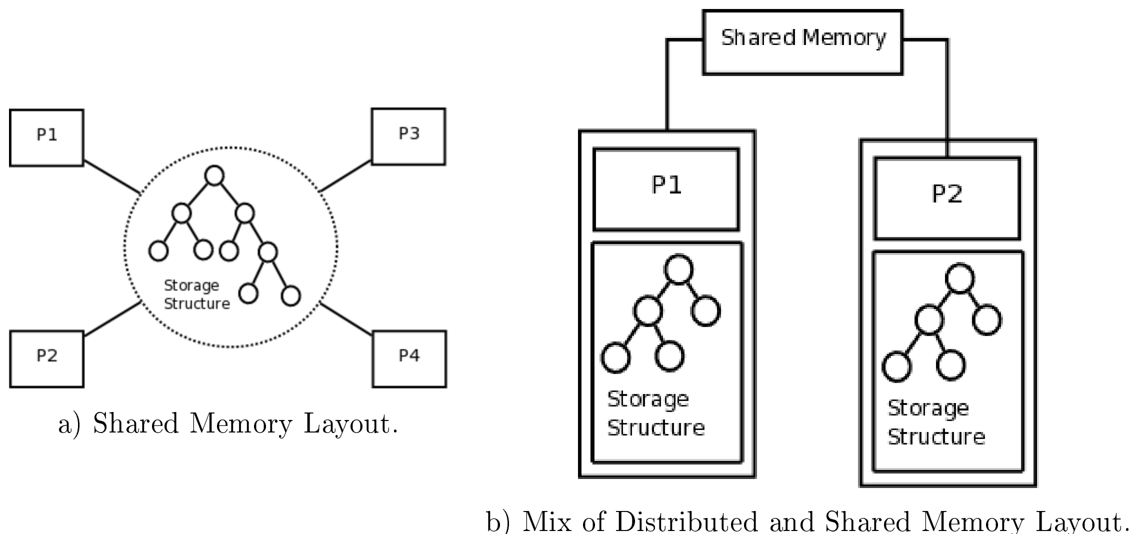
4. The dictionaries modules, that define the different concurrent data structures used in our algorithms.

5. A model checking module that enables the exploration engine module to verify more elaborated formulas, see Chapter 4.

6. A printer module to output the state graph.

Each module is completely isolated from the others and they are all composed together through a common interface. A given *configuration* is obtained by assembling

together a set of modules. The configurations distinguish one from another by selecting a different dictionary from the dictionaries module. We schematize the two memory layouts used by MERCURY in Figure B.2. The first one, Figure B.2-a), is the layout commonly used in shared memory algorithms, such that there exist only one data dictionary, completely shared among all processors. The second one, Figure B.2-b), is an experimental layout inspired from the distributed algorithms. It consists of a small shared space supplemented by local dictionaries to store the states. This layout is optimal for Non-Uniform Memory Architectures (NUMA), where the latency and bandwidth characteristics of memory accesses depend on the processor or memory region being accessed, since it lessen inter-processor communications.



a) Shared Memory Layout.

b) Mix of Distributed and Shared Memory Layout.

Figure B.2: Memory Layouts.

We briefly describe different configurations that have been tested with MERCURY. In this context, a configuration stands for a choice among all the possible modules offered by MERCURY and, most importantly, a choice of data structure (in the dictionaries module). We have five configurations that correspond to exhaustive (deterministic) algorithms; three corresponding to already existing algorithms (used for benchmarking) and two corresponding to algorithms that we defined (SZB10, TSDZB11). Two other configurations correspond to "probabilistic algorithms'. We summarize all the implemented versions in a table (see Figure B.3) that lists the dictionaries and layouts used in each case.

B.2 Mercury Configurations for Exhaustive Exploration

We define five configurations of Mercury that correspond to exhaustive algorithms: **Static**, **Lockless** and **TBB** are classical implementations and had been included as part of our experiments to evaluate their temporal and space balance. From the observations of these classical solutions, we proposed two novel implementations, **General** and **Vector/Dist**, that corresponds to the algorithms described in Chapters 3 (see (SZB10, TSDZB11)).

Static Partition (**Static**): The static hash partition is an implementation of the algorithm defined in (SD97) and described in Section 2.3.3. Several tools use a similar algorithm due to its simplicity. It was implemented using local Hash Tables and use a hash function for statically distributing the states (the slicing function). This configuration is *safe*, meaning that all the state space is generated.

Lockless Hash Table (**Lockless**): This version is a simple implementation using an "unsafe" Hash Table as the global shared dictionary, that is a global hash-table whose access is not protected by a lock. Due to the absence of locks, it is not possible to ensure the data integrity and, as consequence, data may be lost: this configuration is *unsafe*. We experimented this version in order to establish a performance limit, since this algorithm does not suffer from any synchronization penalties.

Concurrent Hash Table (Safe) (**TBB**): This configuration is similar to the Lockless Hash Table version but use a commercial implementation of a safe, concurrent, lock-less hash table (the Intel Threading Building Blocks ((Rei07)) as the shared dictionary. This configuration is *safe*.

General Lock-Free (**General**): this configuration corresponds to the multiphase algorithm described in (SZB10) (see Section 3.2). This configuration is *safe*.

Localization Table Guided (**Vector/Dist**): this configuration corresponds to the "asynchronous" algorithm described in (TSDZB11) (see Section 3.3). This configuration is *safe*. This implementation has two variants (**Vector/Dist**) that differ by the configuration parameters of the *localization table*.

Name	Description	Exh.	Prob.	Safe	Layout
General†	A shared Bloom filter complemented by local dictionaries	x		x	b)
Dist†	Distributed Table instrumented with our Localization Table	x		x	b)
Vector†	Localization Table replaced by a simple Vector of integers	x		x	b)
Static	States are distributed according to a static Hash function	x		x	a)
Lockless	Lockless shared hash table as the shared space	x			a)
TBB	Unordered hash map as the shared space, which is part of the Intel-threading blocs library	x		x	a)
Bloom	Bloom Filter for the shared dictionary		x		a)
BTable†	Bloom Table for the shared dictionary		x		a)

Figure B.3: MERCURY Configurations (we use † to signal our new algorithms).

B.3 MERCURY Configurations for Probabilistic Exploration

We conducted several experiments on parallel, probabilistic state space construction using MERCURY. We define two configurations based on the use of a “probabilistic” data structure in order to save memory space. Obviously, the drawback of this approach is that it is not possible to have full confidence on the outcome of model checking, since the actual state space may not be completely explored. Nonetheless, a “probabilistic verification result” may still be helpful to find errors in a model and some model checking tools (most notably the Spin tool) provide this facility.

Bloom Filter (**Bloom**): this configuration follows the shared memory layout (see Figure B.2-a) and use a shared Bloom Filter for the dictionary. All states are compressed in hash values and symbolically stored at the Filter. For “well dimensioned” Bloom filters (and for acceptable performances), we typically explore 98% of the state space.

Bloom Table (**BTable**): inspired from Bloom Filter, we have defined a novel prob-

abilistic structure with a better coverage ratio. This new structure is actually an enriched Bloom Filter instrumented to prevent the insertion of false positives values. In this implementation, potential false positives are explicitly stored in a separated dictionary. (See Chapter 5 for a complete presentation of the **BTable** data structure.)

B.4 Mercury Configurations for Parallel Model Checking

We extended Mercury engine with the Model Checking with Lazy Cycle Detection technique (MCLDC) presented at Chapter 4 for parallel model checking. Our preliminary implementation with Mercury supports the sub-set of CTL formulas depicted at Section 4.2. Internally, Mercury stores the state space in one of the three accepted graph structures: Reverse, Parental and No_graph. They differ basically by the number of edges they store per vertex.

- Reverse: The Reverse graph RG from a given graph $G(S, E)$ consists of a triplet $RG(S, R)$ where R is binary reverse relation of the edges from E.
- Parental: The Parental graph ($RPG(S, P)$) is an abstraction of Reverse where only the parental relations are stored in R, all other edges are discarded.
- No_graph: The No_Graph is an implementation where none of the relation between vertices are stored, only the set of generated states are stored. This idea is available whenever is possible to establish the reverse relation function between vertices.

Mercury uses RG and RPG graph structures only when the backward traversal is necessary, and each one of them has its advantages. The use of a graph structure like RG allows MCLDC to have linear time complexity even running in parallel ($O(|S| + |E|)$). By contrast, RPG helps MCLDC to save memory by reducing the number of edges (the number of edges is equal to the number of vertices), on the other hand the time complexity is no longer linear ($O(|S|^2 + |E|^2)$). (See Chapter 4 for a complete presentation of the MCLDC technique.)

B.5 Installation

As we mentioned, MERCURY is a prototyping tool that supports 8 different configurations (6 for exhaustive and 2 for probabilistic model checking). For each one of them – and for efficiency reason – we generate a different binary executable using the tool builder *dmake*. A normal compilation is achieved by calling *dmake* with one of the parameters listed in Figure B.4. This table shows the respective parameter for each implementation.

Name	Conf
Bloom	dmake -f Makefile CONF=Parallel_Bloom clean
BTable	dmake -f Makefile CONF=Parallel_Probabilistic clean
General with Hash Table	dmake -f Makefile CONF=Parallel_local_Hash_Table clean
General with AVL Tree	dmake -f Makefile CONF=Parallel_local_Tree clean
Static	dmake -f Makefile CONF=Parallel_Static clean
Vector – Dist	dmake -f Makefile CONF=Parallel_local_Dist_Hash_Table clean
Lockless	dmake -f Makefile CONF=Parallel_Hash_Table_Lockless clean
TBB	dmake -f Makefile CONF=Parallel_Hash_Table_TBB clean
Model Checking	dmake -f Makefile CONF=Parallel_MC clean

Figure B.4: Algorithms selected for benchmark comparison.

B.5.1 Usage

MERCURY binaries are command-line programs (see Figure B.5). A normal execution is performed by giving the optional flags followed by a valid input file. As input, it accepts only textual Petri Net models written using the .net format (the same that is used by Tina). By default it outputs a textual description of the system.

The exploration of a system will normally generates a textual output, except if the option -*aut* is given. In this case, MERCURY will generate a transition system in textual mode. A second group of options can be used to define some initial parameters of the algorithm. They are useful when the user can provide an approximate value for the size of the state space. A summary of all supported options are presented in Figure B.7.

```
usage: mercury [-h | -help] [-p] [-R] [-th n]
               [-bls n] [-blk n] [-hts n] [-smode 0-1]
               [-f 'formula'] [-graph 0-2]
               [-v | -q | -aut ]
               [infile] [outfile]
```

Figure B.5: Mercury usage syntax.

Mercury supports different versions for the parallel state space construction (see Figure B.6). For each version, Mercury can be invoked with specific options to boost the execution. These options may be used to specify initial parameters like the hash table size, the bloom filter size, the number of hash function keys insertions, etc. In practice, the defaults value chosen for these parameters have been carefully selected so has to obtain good performances in our test machine (default parameters may have to be changed depending on the multiprocessor computers that is used). We can list, for each configuration, the set of flags that are supported.

Name	Supported Flags
Bloom	-h,-help,-p,-R,-th,-bls,-blk,-NET,-v,-q,-aut
BTable	-h,-help,-p,-R,-th,-bls,-blk,-NET,-v,-q,-aut
General	-h,-help,-p,-R,-smode,-th,-hts,-bls,-blk,-NET,-v,-q,-aut
Static	-h,-help,-p,-R,-th,-hts,-NET,-v,-q,-aut
Vector – Dist	-h,-help,-p,-R,-th, -hts, -bls,-blk,-NET,-v,-q,-aut
Lockless	-h,-help,-p,-R,-th,-hts,-NET,-v,-q,-aut
TBB	-h,-help,-p,-R,-th,-hts,-NET,-v,-q,-aut
Model Checking	-h,-help,-p,-R,-th, -hts, -bls,-blk, -f 'formula', -graph (0|1|2), -NET,-v,-q,-aut

Figure B.6: Mercury options for different versions.

```
FLAGS            WHAT                                   DEFAULT
-h | -help       this mode
-p               just parses and check input
parallel configurations for Localization Table:          0
-smode 0         ASYNCHRONOUS
-smode 1         SYNCHRONOUS
-smode 2         MIXTE
-smode 3         STATIC
-th n            number of threads                       2
bloom configurations:
-bls n           Bloom Filter Size in bits               32
-blk n           number of Bloom Keys                    8
bloom probabilistic:
-blevels n       number of chances                       3
-bdc     n       size difference(in bits) in cascade     1
-baprox  n       approximate number of keys
-bifp            ignore false positive states
Memory configurations:
-hts n           Hash Table Address Size in bits         25
-sc n (0|1|2|3) State Compression                        0
0:No_Compression  1:Huffman  2:RLE  3:LZ7
Work Load Sharing Options:
-adp             adaptative work load enabled
CTL model Checking:
-f 'formula'     enable ctl mchecking for the given formula
-graph (0|1|2)   Type of graph
     0:Reverse  1:Parental  1:No Graph
input net format flags:
-NET             textual net input
-tts             textual net with data input
output format and options flags:
-v | -q          textual output (full | digest)        -v
files:
infile           input file (stdin if -)               stdin
outfile          output file (stdout if - or absent)   stdout
```

Figure B.7: MERCURY options.

Model checking sur Architecture Multiprocesseur

Résumé en Français

Annexe C

Résumé en Français

C.1 Abstract

Nous proposons de nouveaux algorithmes et de nouvelles structures de données pour la vérification formelle de systèmes réactifs finis sur architectures parallèles. Ces travaux se basent sur les techniques de vérification par model checking. Notre approche cible des architectures multi-processeurs et multi-cœurs, avec mémoire partagée, qui correspondent aux générations de serveurs les plus performants disponibles actuellement.

Dans ce contexte, notre objectif principal est de proposer des approches qui soient à la fois efficaces au niveau des performances, mais aussi compatibles avec les politiques de partage dynamique du travail utilisées par les algorithmes de génération d'espaces d'états en parallèle ; ainsi, nous ne plaçons pas de contraintes sur la manière dont le travail ou les données sont partagés entre les processeurs.

Parallèlement à la définition de nouveaux algorithmes de model checking pour machines multi-cœurs, nous nous intéressons également aux *algorithmes de vérification probabiliste*. Par probabiliste, nous entendons des algorithmes de model checking qui ont une forte probabilité de visiter tous les états durant la vérification d'un système. La vérification probabiliste permet des gains importants au niveau de la mémoire utilisée, en échange d'une faible probabilité de ne pas être exhaustif ; il s'agit donc d'une stratégie permettant de répondre au problème de l'explosion combinatoire.

[Mots-clés :] Model Checking en Parallèle, Algorithme et Structure de Données concurrents, Méthode Formelle, Vérification Formelle et Logiques Temporelles.

C.2 Introduction

Dans cette thèse, nous proposons et étudions de nouveaux algorithmes et structures de données pour la vérification formelle de systèmes finis, plus spécifiquement les techniques de model checking. Nous nous concentrons sur des techniques qui ciblent les machines multi-processeurs et multi-cœurs à mémoire partagée, qui sont une tendance actuelle.

Model Checking est une méthode de vérification formelle utilisée pour assurer l'absence d'erreurs de logique. À cet égard, le model checking contribue à l'amélioration de la sécurité des systèmes embarqués (et aussi pour améliorer le niveau de confiance que nous pouvons mettre en eux). Ceci est une réussite importante. Les systèmes embarqués sont de plus en plus présents dans notre vie quotidienne et nous ne pouvons nier l'impact majeur qu'ils ont sur nos sociétés. Certains de ces systèmes embarqués — comme ceux trouvés dans les domaines de l'aéronautique ou du nucléaire — sont classés comme critiques, ce qui signifie qu'une panne ou un dysfonctionnement peut entraîner un dommage corporel (ou même le décès) des personnes impliquées, des dommages irréversibles à l'équipement, ou des catastrophes environnementales. Nous pouvons énumérer quelques exemples remarquables de défaillances catastrophiques qui ont attiré l'attention du public en leur temps (voir (Neu92) pour une liste des incidents) :

- Therac-25 (1985-1987) : Entre Juin 1985 et Janvier 1987, une machine de radiothérapie contrôlé par ordinateur, le Therac-25, overdose sévèrement six patients en raison d'un problème de codage de son logiciel de contrôle (Lev95).
- Ariane-5 (1996) : En juin 1996, le lancement inaugural de la fusée européenne Ariane-5 s'est terminé dans un échec total ! Cet échec, provoqué par un dysfonctionnement dans le logiciel de contrôle de guidage, a été essentiellement causé par une exception interne pendant la conversion de données flottante en 64 bits à 16 bits (L^+96).
- La NASA Mars Pathfinder (1997) : En Juillet 1997, le rover Mars Pathfinder a commencé à perdre des informations en raison de plusieurs redémarrages du système. Le système a été redémarré en raison d'un problème d'inversion des priorités, ce qui a raccourcis la durée de la mission (C^+98).

Les demandes du marché pour des solutions plus efficaces et automatisées ont poussé la complexité des systèmes embarqués à des niveaux jamais imaginé auparavant. Par

exemple, nous construisons des avions qui consomment moins de carburant, volent plus longtemps et dont la durée de maintenance est moindre qu'auparavant. Le niveau d'efficacité que nous expérimentons aujourd'hui est, sans doute, une des réalisations principal de la dernière décennie. Toutefois, ces réalisations ont un prix, puisque les systèmes coûtent de plus en plus cher à développer (et la probabilité de réussir un nouveau projet technologique diminue). Bien qu'il n'existe aucune information officielle sur la productivité des ingénieurs logiciels embarqué - ni d'une façon précise de calcul de ce métrique - dans certains domaines critiques comme l'avionique, les ingénieurs logiciels ne produisent pas plus d'une ligne de code par jour en moyenne. Par ailleurs, ces chiffres ne prennent pas en compte le lourd fardeau des tests et des activités de certification auxquels ces systèmes sont soumis. L'utilisation de model checking peut aider à améliorer cette situation car cela aide à détecter les erreurs pendant la phase de conception d'un système, avant qu'elles ne deviennent très coûteuses à réparer.

Depuis les travaux pionniers de Edmund M. Clarke et Allen Emmerson, et de Joseph Sifakis et Jean-Pierre Queille, au début des années 1980, le model checking a été utilisé avec succès dans la vérification de certaines applications telles que la conception des circuits intégrés (BCL+94) et des protocoles de communication (JH93). Les techniques de model checking sont attrayantes car elles offrent une solution automatique pour vérifier si un système (modèle valide) répond à ses exigences. Par exemple, il ne nécessite pas des preuves construite à la main, comme c'est le cas avec les approches fondées sur des logiques de style Floyd-Hoare qui peut être assez fastidieux et difficile de passer à l'échelle. Une des autre raison de l'intérêt croissant pour l'utilisation des techniques de model checking est qu'elles peuvent facilement être intégrées dans un cycle de développement standard ; elles peuvent non seulement aider à trouver des erreurs, mais peuvent également fournir des contre-exemples (les traces d'exécution) lorsque le modèle du système viole certaines de ses exigences. Au final, l'approche model checking s'est révélé être un outil important dans la conception et le développement de systèmes critiques.

Même si les techniques de model checking offrent une approche "push button" pour la vérification des systèmes finis, la taille de l'espace des états construits durant la vérification peut croître de façon exponentielle par rapport à l'augmentation de la complexité du système. Ainsi, dans de nombreux cas, cette technique peut se révéler inutilisable en pratique. Cet inconvénient, connu comme le problème d'explosion des états, est l'un

des principaux défis dans le domaine du model checking. Malgré le fait que des progrès considérables aient été réalisés sur le plan théorique —par exemple avec la définition de méthodes symboliques et des techniques d'ordres partielles— des classes de systèmes ne peuvent pas bénéficier de ces méthodes avancées. Par exemple, pour des modèles qui combinent contraintes de temps réel, des priorités dynamiques et de données externes. Dans ces cas, nous avons encore besoin de revenir à l'utilisation des techniques classiques de model checking énumératives où les états sont stockés de manière explicite.

La motivation principale de cette thèse est de développer de nouveaux algorithmes et structures des données afin de profiter des améliorations récemment apportées au niveau du matériel (hardware) ; à savoir l'avènement d'un coût abordable des serveurs de mémoire partagée (machines multi-processeurs). Fondamentalement, nous attaquons le problème d'explosion des états à travers l'utilisation de la force brutale ! Depuis le milieu des années 2000, les principaux fabricants de puces ont reconnu une possibles fin à la loi de Moore en ce qui concerne l'augmentation de la fréquence d'horloge du processeur ("the Moore's wall"). Par conséquence, l'industrie de hardware ont déplacé leur attention vers des architectures de processeur multi-cœurs, apportant la technologie de la computation parallèle même au plus simple des ordinateurs : même les netbooks et les les smartphones ont un processeur dual core de nos jours. De plus, avec la vulgarisation de l'informatique basée sur le serveur et les technologies de virtualisation (serveurs hébergeant de multiples machines virtuelles), nous avons maintenant accès pour un prix abordable à des machines multiprocesseurs —avec de nombreux processeurs multi-core— ce qui offre la possibilité d'accéder à de grands espaces de mémoire partagée de façon plus efficace.

Parallèlement à la définition de nouveaux algorithmes pour les machines multi-core, nous étudions aussi des algorithmes de vérification probabiliste. Par le terme de probabiliste, nous voulons dire que, pendant l'exploration d'un système, n'importe quel état atteignable a une forte probabilité d'être visité par l'algorithme. En conséquence nous acceptons que certains états joignables du système ne puissent pas être inspectés : cette technique ne peut pas être utilisée pour prouver l'absence d'erreurs, elle peut être très efficace quand nous essayons de trouver des contre-exemples. La vérification probabiliste échange l'économie au niveau de la mémoire utilisée pour la probabilité de manquer certains états. Fondamentalement, l'idée est d'utiliser des valeurs de hachage au lieu d'une représentation exacte d'état. Par conséquent, il devient possible d'analyser

une partie de l'espace d'état d'un système où il n'y a pas assez de mémoire disponible pour représenter l'espace d'état entier de manière exacte (énumérative).

C.3 Contribuition

Les contributions de cette thèse peuvent être divisés en trois axes principaux : (1) la construction de l'espace d'état en parallèle, (2) des algorithmes de model checking en parallèle, et (3) des méthodes de vérification probabilistes.

Chronologiquement, nous avons commencé notre travail en étudiant de nouveaux algorithmes et structures de données pour construire l'espace d'état en parallèle. Les points clés pour concevoir un algorithme efficace en parallèle pour les machines à mémoire partagée sont la structure de données utilisée pour stocker l'ensemble des états explorés et la stratégie de partage de travail employée pour distribuer les états. Nous proposerons deux nouvelles approches basées sur une structure des données optimisée : nous utilisons des structure des données indépendants (distribués) en conjonction avec une structure partagée probabiliste pour distribuer dynamiquement l'espace d'état.

Notre première contribution pour la construction de l'espace d'état parallèle est un algorithme spéculatif (SZB10) où les états sont stockés dans des ensembles de données locales, tandis qu'un Bloom Filter (Blo70) partagé est utilisé pour distribuer dynamiquement les états. En raison du caractère probabiliste du Bloom Filter (faux positifs sont possibles), nous proposons un algorithme qui opère en phase, afin d'effectuer une génération exhaustive et déterministe de l'espace d'état. Ensuite nous améliorons notre conception précédente et nous remplaçons le filtre de Bloom par une structure de données dédiée, appelée par nous de Localization Table (TSDZB11). Ce tableau est utilisé pour attribuer dynamiquement les états récemment découverts et se comporte comme un tableau associatif qui retourne l'identité du processeur qui possède un état donné. Avec cette approche, nous sommes en mesure de consolider un réseau de tables des hachages locales dans une distribué, supprimant la nécessité d'un algorithme qui opère en phase. Nos résultats préliminaires sont très prometteurs, on observe des performances proches de ceux obtenus en utilisant un algorithme basé sur une table de hachage non protégée (risques probables d'incohérences) et se comporte bien par rapport à certains algorithmes parallèles classiques.

Notre deuxième contribution est la présentation de deux nouveaux algorithmes de model checking en parallèle, qui prennent en compte un sous-ensemble spécifique de la logique Computation Tree Logic (CTL). Dans ce contexte, l'un des points clés de performance pour l'évaluation de formules plus complexes est la stratégie utilisée pour détecter des cycles dans le comportement du système. (Par exemple, un comportement qui se répète indéfiniment). Notre objectif principal est de proposer des algorithmes qui ne sont pas seulement efficaces mais aussi "amical" à l'égard des politiques de partage du travail qui sont utilisées pendant la génération de l'espace d'états en parallèle (par exemple la stratégie "work-stealing") : à aucun moment nous imposons une restriction sur la façon que le travail est partagé entre les processeurs. Cela inclut la construction de l'espace d'états comme la détection de cycles en parallèle. Nous contribuons à une approche pratique où les cycles sont détectés uniquement à la dernière étape. Nous contournons toutes les complexités imposées par la détection de cycles en parallèle en ne le faisant pas explicitement. Nous présentons deux approches de notre algorithme, celui avec une complexité temporelle linéaire et un autre qui échange une plus faible utilisation de la mémoire pour une complexité temporelle plus grande. La différence entre ces implémentations réside dans la structure utilisée pour stocker l'espace d'états traversés. Nous considérons deux cas : (1) le cas où nous avons accès au graphe de l'espace d'état complet (en fait le graphe inverse), nous stockons toutes les transitions dans la mémoire, et (2) nous avons seulement accès à un parent pour chaque état. Alors il est commun de trouver des algorithmes pour le model checking de formules CTL qui ne nécessitent pas de stocker en mémoire la relation de transition inverse (les transitions sont régénérées lorsque nécessaire), cette approche est pas inapproprié quand il n'est pas possible (ou très cher) de calculer la relation de transition inverse d'un modèle. C'est par exemple le cas quand nous traitons avec des modèles combinant les contraintes temps réel, la priorité entre les transitions ou lorsque le système est étendu avec des données externes.

Une autre contribution de cette thèse c'est la définition d'une nouvelle structure de données pour la vérification formelle probabiliste. Après nos expériences avec la construction de l'espace d'état en parallèle, nous avons étudié une version enrichie de la structure Bloom Filter spécialement conçue pour la vérification probabiliste. Nous proposons une nouvelle structure probabiliste de données, nommée Bloom Table, qui

remplit une lacune que nous avons identifiée entre l'utilisation de la structure des données Hash Compact et le Bloom Filter. Notre Bloom Table ne fournit pas seulement une faible probabilité de faux positifs, mais améliore également la complexité du temps en réduisant le nombre de fonctions de hachage nécessaires. Par exemple, seulement deux clés de hachage sont nécessaires pour livrer une probabilité de 10^{-5} en utilisant uniquement 16 bits par l'état. (Un Bloom Filter nécessite l'utilisation de 6 clés de hachage pour atteindre des résultats similaires.) Alors que les Bloom Tables ne sont pas intrinsèquement une structure de données conçue pour l'utilisation concurrente, nous avons imaginé nos algorithmes afin de profiter d'une architecture parallèle à mémoire partagée.

Avant de conclure cette section, nous tenons à souligner que les contributions de cette thèse ne sont pas limitées au domaine model checking. De facto, les structures des données proposées dans cet ouvrage sont intéressantes pour toute application qui effectue l'exploration de graphe, la détection de cycle ou le stockage probabiliste (ou avec pertes) en parallèle.

C.4 Sommaire : Brève Description de la Thèse

La section précédente donne une brève présentation de notre travail. Ci-dessous, nous donnons un bref résumé du contenu des chapitres présents dans ce document.

Chapitre 2 Related Work : Dans ce chapitre, nous présentons le contexte de cette thèse. Nous présentons brièvement le model checking—d' une manière très générale—puis nous plongeons dans la littérature pertinente au contexte de cette thèse. Nous commençons par les travaux de model checking distribué et parallèle. Nous essayons de mettre en évidence l'importance d'optimiser la procédure de détection des cycles afin d'obtenir une solution parallèle efficace. Ensuite, nous présentons les solutions les plus significatives pour la vérification probabiliste, notamment celles basées sur les structures probabilistes supertrace, multihash et hash compact. Nous concluons ce chapitre par une présentation détaillée de la contribution de cette thèse.

Chapitre 3 Parallel State Space Construction : Ce chapitre décrit notre approche pour la construction de l'espace d'états en parallèle. Nous concentrons nos efforts sur une approche qui est évolutive, sans imposer aucune restriction sur la façon

dont le travail est réparti entre les processeurs. Dans ce chapitre, nous indiquons les principales directives de nos algorithmes tels que la distribution de la mémoire et les techniques de partage du travail. Cette section est suivie de la présentation de notre algorithme (spéculative), qui est un algorithme général non bloquant pour la construction d'espace d'états en parallèle. Nous essayons de souligner le caractère général de notre approche en donnant les résultats de nos expériences, effectuées en utilisant différentes structures de données (comme les arbres AVL et les tables de hachage) pour les structures de données locales. Ensuite nous donnons une version améliorée qui remplace le Bloom Filter par une structure spécialisée appelée Localisation Table. Cette solution améliore notre version précédente parce que non seulement elle fait la distribution dynamique des données, mais également assure le suivi de la distribution. Avant de conclure, nous montrons une étude comparative de notre version améliorée avec d'autres solutions déjà proposées dans la littérature.

Chapitre 4 Parallel Model Checking With Lazy Cycle Detection : Basé sur les travaux que nous avons présentés au chapitre 3, nous définissons et analysons deux nouveaux algorithmes pour les techniques de model checking en parallèle qui prennent en charge un sous-ensemble de formules CTL. Nos principaux objectifs sont de proposer des algorithmes efficaces, qui n'empêchent pas l'utilisation de certaines stratégies de partage dynamique du travail, c'est-à-dire qu'ils n'imposent pas de restriction sur la façon dont le travail est partagé entre les processeurs lors de la construction de l'espace d'état ni pendant la phase de vérification de la formule CTL fourni. De plus, nous nous intéressons également au développement de nouvelles approches qui nécessitent moins d'espaces de mémoire. Avant conclure ce chapitre, nous étudions un ensemble de résultats obtenus avec nos approches.

Chapitre 5 Probabilistic Verification : L'objectif principal de cette thèse est de proposer de nouvelles méthodes pour traiter le problème d'explosion de l'état. Alors que la plus grande partie de notre travail se base sur une approche énumérative, nous avons réalisé au cours de cette thèse que nos structures de données pourraient être adaptées au contexte des algorithmes de vérification probabiliste. En particulier, nous avons identifié l'existence d'un écart entre les deux structures de données les plus rependues pour la vérification probabiliste, Bloom Filter et Hash Compact. Grosso modo, la structure de données probabiliste que nous

présentons dans ce chapitre offre un meilleur résultat que Hash Compact lorsque nous avons moins de 40bits d'information par état pour représenter l'espace d'état. D'autre part, notre solution améliore le temps d'exécution par rapport à une approche classique basée sur les Bloom Filters car elle offre un meilleur résultat sans augmenter le nombre de clés de hachage utilisées. (En effet, notre proposition nécessite que 16bits par état pour atteindre une couverture efficace de l'espace d'état ; un Bloom Filter nécessite au moins 6 clés de hachage par état pour obtenir un résultat similaire.) Nous présentons une analyse théorique de notre structure de données avec une comparaison analytique des solutions similaires. Avant de conclure, nous présentons les résultats obtenus à partir d'un ensemble d'expériences afin de montrer l'efficacité de notre approche.

Chapitre 6 Conclusion : Nous concluons cette thèse en définissant les lignes possibles des travaux futurs.

C.5 Conclusion

Dans cette thèse, nous avons présenté nos efforts pour réaliser le model checking de systèmes finis sur les machines multi-processeurs et multi-core. Nous avons proposé de nouvelles structures de données et de nouveaux algorithmes qui sont appliqués à trois domaines d'application principaux : la construction de l'espace d'état en parallèle, les techniques de model checking et la vérification probabiliste.

Évidemment nous voulons profiter de la puissance de calcul accrue apportée par les machines multi-cores. Par ailleurs, la transition vers des approches parallèle pour la vérification formelle nous semble inévitable. Il ne fait aucun doute que tous les outils de model checking se dérouleront sur les ordinateurs multi-core ; pour la simple raison que chaque nouvel ordinateur est équipé d'un processeur multi-core.

Outre le gain en performance, nous sommes également très intéressés par l'énorme quantité de mémoire d'accès rapide fournie par les serveurs multiprocesseurs actuels ; comme celui utilisé dans nos expériences. Rappelons que tous les résultats expérimentaux présentés dans ce manuscrit ont été effectués sur un serveur avec 208 Go de RAM —affectueusement appelée Brutus— laquelle représente une augmentation significative par rapport au $8 - 32$M à la disposition aux premiers outils de model checking dans les années 1980.

Dans le chapitre 3 de cette thèse, nous avons décrit de nouveaux algorithmes pour effectuer la construction exhaustive de l'espace d'état en parallèle. Notre travail est construit sur une conception unique, qui est basée sur une distribution dynamique des états et de l'utilisation de structures de données non bloquant. Les chapitres 4 et 5 décrivent nos nouvelles approches pour effectuer les techniques de model checking exhaustive et probabiliste en parallèle, lesquelles sont construites sur le dessus de notre algorithme pour la construction de l'espace d'états en parallèle.

Nous comprenons que les approches exhaustive et probabiliste sont complémentaires. D'une part, les méthodes probabilistes devraient être considérées quand nous essayons de trouver des contre-exemples à une spécification donnée. D'autre part, notre approche exhaustive est plus appropriée dans les dernières phases de la conception d'un système, ou quand nous avons besoin de vérifier les spécifications complexes. Pour cette raison, nous avons optimisé notre algorithme de model checking, appelé MCLCD, pour le cas où la spécification est vraie.

Nous proposons des algorithmes efficaces pour la mémoire ces deux approches. En effet, nous croyons que la mémoire est la ressource la plus importante pour les techniques de model checking. Il s'agit de la raison pour laquelle nous avons concentré nos efforts sur la réduction de l'utilisation de la mémoire, même si cela signifie la négociation d'un temps de calcul plus élevé en retour d'une utilisation d'un espace mémoire plus réduit. Par exemple, nous définissons une version de notre algorithme MCLCD qui ne nécessite pas de stocker la relation complète de transition d'un système donnée.

Nous définissons aussi une nouvelle structure de données pour la vérification probabiliste, appelé Bloom Table, qui nécessite que 2 octets par état en moyenne, tout en offrant une très bonne couverture. (Cette structure de données a également été conçue pour être conviviale pour la programmation concurrente). Par exemple, avec Bloom Table que utilise seulement 2.5GB de mémoire, nous avons une probabilité de 10^{-5} de manquer un état dans un système avec un milliard états. Il n'est pas possible d'atteindre la même précision en utilisant la structure de données hash-compact avec la même quantité de mémoire. De même, un Bloom Filter avec deux clefs de hachage aura une précision de l'ordre de 10^{-2}-10^{-3} avec la même configuration. Plus de 5-6 clés de hachage sont nécessaires pour le Bloom Filter afin d'obtenir une précision de 10^{-5} mais, dans ce cas, le calcul est plus lent que avec BT.

Ci-dessous, nous résumons les conclusions de cette thèse.

[Construction de l'espace État en parallèle]

Au chapitre 3, Nous définissons deux nouveaux algorithmes pour la construction de l'espace d'état parallèle. Nous croyons que nos contributions sont intéressant dans plusieurs domaines. Dans le domaine de la vérification formelle, nous définissons de nouveaux algorithmes pour la construction de l'espace d'état en parallèle. Dans le domaine du calcul parallèle, nous proposons de nouvelles structures de données concurrentes non bloquant.

Notre approche est basée sur l'utilisation d'un ensemble des structures de données locaux (pour le stockage des états) et d'une structure de données probabiliste (par exemple Bloom Filter) pour gérer la distribution dynamique des états entre les processeurs. Nous proposons deux algorithmes : un général (spéculative) —où les collisions sont résolues à l'aide de synchronisations intra-processus— et un algorithme mixte plus spécialisé où les collisions sont résolues à la volée. Nos résultats expérimentaux montrent que la version mixte est le meilleur choix dans la pratique.

La principale innovation est la définition d'une structure de données, nommé Localization Table, qui est utilisée pour coordonner un réseau de tables de hachage locales afin d'obtenir une structure de données concurrent et efficace.

Nous observons des bonnes accélérations dans nos résultats expérimentaux. L'accélération de l'algorithme mixte est toujours meilleur que d'autres algorithmes parallèles. Nous avons également montré que notre mise en oeuvre se comporte bien par rapport aux outils similaires ; nous montrons aussi que, dans notre algorithme, il est préférable d'utiliser notre Localization Table que la structure concurrent fournie par Intel TBB.

Nous croyons que notre Localization Table peut être d'un grand intérêt en dehors du domaine de la vérification formelle. Pour cette raison, nous prévoyons de fournir une API fonctionnelle de notre structure de données concurrent complètement autonome, qui pourrait être utilisé dans d'autres situations et qui ne nécessitent que une configuration minimale.

[Model Checking parallèle]

Dans le chapitre 4, nous définissons un algorithme pour le model checking de formules CTL. Nous choisissons un algorithme fondé sur la sémantique et adapté aux

architectures de mémoire partagée, parce que nous croyons qu'il est plus approprié pour une application parallèle doté d'une politique dynamique de distribution de travail.

Nous avons développé deux versions de cet algorithme : une première version basée sur le graphe reverse de l'espace d'états, appelé RG, où nous avons besoin de stocker explicitement toutes les transitions du système et une deuxième version, RPG, où nous avons seulement besoin de stocker un sous-graphe couvrant.

La version RG a une complexité en temps linéaire et en espace, en $O(S + R)$ (où S est le nombre d'états et R le nombre de transitions), tandis que la version RPG a une complexité en temps de $O(S \cdot (R - S))$ et une complexité spatiale en $O(S)$. Le principal avantage de la version RPG est de fournir un algorithme qui est efficace dans la mémoire et indépendante du choix des classes d'abstraction de l'espace d'états.

Notre mise en oeuvre montre des résultats prometteurs pour les versions RG et RPG de notre algorithme. Le choix d'un algorithme d'étiquetage s'est révélé être un bon match pour les machines à mémoire partagée et pour la politique de contrôle de charge dynamique work-stealing ; par exemple, nous avons obtenu des accélérations près du linéaire avec une efficacité moyenne de 75%. Nos résultats montrent aussi que l'algorithme RPG est en mesure d'égaler la performance de RG pour un certain type de modèles.

Nous croyons que les deux versions de l'algorithme sont complémentaires. RG est le meilleur choix quand il y a assez de mémoire pour stocker complètement le graphe reverse, sinon, RPG devrait être utilisé malgré sa complexité élevée en temps. Pour nos travaux futurs, nous étudions une version améliorée de nos algorithmes qui prend en charge tout l'ensemble des formules CTL.

[Vérification probabiliste]

Le dernier chapitre technique de cette thèse, le chapitre 5, est dédié à une nouvelle approche pour la vérification probabiliste. Nous définissons une nouvelle structure de données inspirée de nos travaux avec la Localization Table et nous la comparons avec les structures probabilistes Bloom Filter et Hash Compact. La différence principale de notre structure, par rapport au Bloom Filter, est que nous utilisons un vecteur de "w-bits mots" au lieu d'un vecteur de bits.

Lorsque notre Bloom Table est comparé à de simples techniques de hachage, Bloom Table est en mesure d'améliorer la probabilité de générer l'espace d'état complet d'un système fourni quand nous avons une estimation de la taille de cet espace d'états.

Nos résultats expérimentaux montrent que notre structure de données fonctionne bien et il est possible d'obtenir de bonne couverture de l'espace d'état sans exiger de nombreuses opérations de fonctions hachage. (Dans nos expériences, deux clés sont généralement suffisantes.) Par rapport à l'utilisation d'un Bloom Filter, notre structure de données peut atteindre des résultats semblables ou de meilleurs avec un temps d'exécution plus court, parfois par un facteur de 2.

Pour les travaux futurs, nous voulons étudier la vérification probabiliste de formules plus complexes, comme la sûreté et les formules de vivacité. L'idée est d'être capable d'utiliser des méthodes de vérification probabiliste en même temps que les algorithmes exhaustifs de model checking, sans aucune restriction sur la spécification. Les techniques de vérification probabilistes seront utilisées au début du cycle de développement, tandis que le model checking exhaustive sera utilisé pour "certifier" l'absence d'erreurs au niveau du modèle

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414 –425, June 1990. 3

[ADK97] S. Allmaier, S. Dalibor, and D. Kreische. Parallel graph generation algorithms for shared and distributed memory machines. In *Proceedings of the Parallel Computing Conference PARCO*, volume 1253 of *LNCS*, pages 207–218, Bonn, Germaney, 1997. Springer. 24, 26

[AKH97] S. C Allmaier, M. Kowarschik, and G. 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*, pages 112 –121, June 1997. 26

[BB04] F. Vernadat B. Berthomieu, P.-O. Ribet. The tool TINA – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14), 2004. 149

[BBC02] J. Barnat, L. Brim, and I. Cerna. Property driven distribution of nested DFS. In *Proc. Workshop on Verification and Computational Logic*, DSSE Technical Report, page 1–10, 2002. 31

[BBC03] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 106 – 115, October 2003. 33, 34

[BBCR10] J. Barnat, L. Brim, M. Ceska, and P. Rockai. DiVinE: parallel distributed model checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, page 4–7. IEEE, 2010. 25, 27, 42

[BBR07] J. Barnat, L. Brim, and P. Rockai. Scalable multi-core LTL Model-Checking. In *Model Checking Software*, volume 4595 of *LNCS*, page 187–203. Springer, 2007. 33, 34

[BBR09] J. Barnat, L. Brim, and P. Rockai. A Time-Optimal On-the-Fly parallel algorithm for model checking of weak LTL properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, page 407–425. Springer, 2009. 35

[BBS01] Jiri Barnat, Lubos Brim, and Jitka Stribrna. Distributed LTL model-checking in SPIN. In Matthew Dwyer, editor, *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 200–216. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45139-0_13. vii, 30, 31, 33, 34

[BCKP01] Lubos Brim, Ivana Cerna, Pavel Krcal, and Radek Pelanek. Distributed LTL model checking based on negative cycle detection. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 96–107. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45294-X_9. 33, 34

[BCL+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(4):401 –424, April 1994. 2, 183

[BCMS04] L. Brim, I. Cerna, P. Moravec, and J. Simsa. Accepting predecessors are better than back edges in distributed LTL model-checking. In *Formal Methods in Computer-Aided Design*, page 352–366, 2004. 33, 34

[BCY02] Lubos Brim, Jitka Crhová, and Karen Yorav. Using assumptions to distribute CTL model checking. *Electronic Notes in Theoretical Computer*

Science, 68(4):559 – 574, 2002. PDMC 2002, Parallel and Distributed Model Checking (Satellite Workshop of CONCUR 2002). 36

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, LNCS, page 200–236. Springer–Verlag, September 2004. 81, 82

[BH04] Alexander Bell and Boudewijn R. Haverkort. Sequential and distributed model checking of petri nets. *International Journal on Software Tools for Technology Transfer*, 7(1):43–60, April 2004. 36, 45

[BHV00] G. Behrmann, T. Hune, and F. Vaandrager. Distributing timed model checking—how the search order matters. In *Computer aided verification*, page 216–231, 2000. 25

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. 4, 37, 38, 185

[BLW01] Benedikt Bollig, Martin Leucker, and Michael Weber. Parallel model checking for the alternation free μ-Calculus. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 543–558. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45319-9_37. 27, 36, 45

[BLW02] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the Alternation-Free μ-Calculus. In Dragan Bosnacki and Stefan Leue, editors, *Model Checking Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 501–522. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46017-9_11. 36, 45

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000. 57, 110, 141, 169

[BMM02] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, page 636–646, 2002. 38, 40

[BPM83] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, 1983. 10.1007/BF01257083. 14

[BR08] Jiri Barnat and Petr Rockai. Shared hash tables in parallel model checking. *Electronic Notes in Theoretical Computer Science*, 198(1):79 – 91, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007). 74

[But97] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997. 57, 110, 141, 169

[C+98] J. Crow et al. NASA formal methods specification and analysis guidebook for the verification of software and computer systems, volume II: a practitioner's companion. NASA office of safety and mission assurance, 1998, vol. 2. *NASA Oce of Safety and Mission Assurance*, 1998. 2, 182

[CCM01] S. Caselli, G. Conte, and P. Marenzoni. A distributed algorithm for GSPN reachability graph generation. *Journal of Parallel and Distributed Computing*, 61(1):79 – 95, 2001. 24

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131, pages 52–71. Springer-Verlag, Berlin/Heidelberg, 1982. 10, 14, 16, 80, 88

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986. 15, 16, 36, 45, 106, 119

[CGN98] Gianfranco Ciardo, Joshua Gluckman, and David Nicol. Distributed state space generation of Discrete-State stochastic models. *INFORMS JOURNAL ON COMPUTING*, 10(1):82–93, 1998. 24

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, page 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. 38

[Cla99] E Clarke. *Model checking.* MIT Press, Cambridge Mass., 1999. 12, 13, 16, 80, 88

[Cla08] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. 10

[CM03] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, page 241–252, New York, NY, USA, 2003. ACM. 38

[CP03] Ivana Cerna and Radek Pelanek. Distributed explicit fair cycle detection (Set based approach). In Thomas Ball and Sriram Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 623–623. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-44829-2_4. 33, 34

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, 1992. 10.1007/BF00121128. 17, 30, 44, 80

[DM04] Peter Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In Alan Hu and Andrew Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 367–381. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30494-4_26. 37, 38

[EC80] E. Emerson and Edmund Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of

Lecture Notes in Computer Science, pages 169–181. Springer Berlin / Heidelberg, 1980. 10.1007/3-540-10003-2_69. 14

[EJS93] E. Emerson, C. Jutla, and A. Sistla. On model-checking for fragments of μ-calculus. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-56922-7_32. 15, 16, 36, 81

[El-04] Hesham El-Rewini. *Advanced computer architecture and parallel processing.* Wiley, New York, NY, 2004. 18, 20, 21

[EL08] Jonathan Ezekiel and Gerald Luttgen. Measuring and evaluating parallel State-Space exploration algorithms. *Electronic Notes in Theoretical Computer Science*, 198(1):47 – 61, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007). 22, 70

[FFK+01] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Vardi, and Zijiang Yang. Is there a best symbolic Cycle-Detection algorithm? In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 420–434. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45319-9_29. 34

[GHR95] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, USA, 1995. 30, 38

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In Matthew Dwyer, editor, *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45139-0_14. 25

[GRV95] T. Gautier, J. Roch, and G. Villard. Regular versus irregular problems and algorithms. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms*

for Irregularly Structured Problems, volume 980 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60321-2_1. 22

[HB07] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33:659–674, 2007. 26, 27, 28, 31, 42, 45

[HBB99] B. Haverkort, A. Bell, and H. Bohnenkamp. On the efficient sequential and distributed generation of very large markov chains from stochastic petri nets. In *Petri Nets and Performance Models, 1999. Proceedings. The 8th International Workshop on*, page 12–21, 1999. 25

[HJG08a] G. J Holzmann, R. Joshi, and A. Groce. Swarm verification. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, page 1–6, Washington, DC, USA, 2008. IEEE Computer Society. 32

[HJG08b] Gerard Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 134–143. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85114-1_11. 32

[HKTL07] Alexandre Hamez, Fabrice Kordon, Yann Thierry-Mieg, and Fabrice Legond-Aubry. dmcG: a distributed symbolic model checker based on GreatSPN. In Jetty Kleijn and Alex Yakovlev, editors, *Petri Nets and Other Models of Concurrency – ICATPN 2007*, volume 4546 of *Lecture Notes in Computer Science*, pages 495–504. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73094-1_29. 25, 26

[Hol93] G. J Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25(9):981 – 1017, 1993. Protocol Specifications, Testing and Verification. 17, 37, 128

[Hol08] Gerard J. Holzmann. A Stack-Slicing algorithm for Multi-Core model checking. *Electronic Notes in Theoretical Computer Science*, 198(1):3 –

16, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007). vii, 26, 27, 28

[IB02] Cornelia P. Inggs and Howard Barringer. Effective state exploration for model checking on a shared memory architecture. *Electronic Notes in Theoretical Computer Science*, 68(4):605 – 620, 2002. PDMC 2002, Parallel and Distributed Model Checking (Satellite Workshop of CONCUR 2002). 26, 27, 28, 42, 45, 49, 50, 73

[IB06] Cornelia P. Inggs and Howard Barringer. CTL* model checking on a shared-memory architecture. *Formal Methods in System Design*, 29(2):135–155, July 2006. 36

[Jen97] B. Jenkins. Algorithm alley: Hash functions. *Dobb's Journal*, September 1997. 141, 169

[JH93] Gerard J and Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25(9):981 – 1017, 1993. Protocol Specifications, Testing and Verification. 2, 183

[JM05] Christophe Joubert and Radu Mateescu. Distributed local resolution of boolean equation systems. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, page 264–271, Washington, DC, USA, 2005. IEEE Computer Society. 36

[JM06] Christophe Joubert and Radu Mateescu. Distributed On-the-Fly model checking and test case generation. In Antti Valmari, editor, *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 126–145. Springer Berlin / Heidelberg, 2006. 10.1007/11691617_8. 36

[KM05] Rahul Kumar and Eric G. Mercer. Load balancing parallel explicit state model checking. *Electronic Notes in Theoretical Computer Science*, 128(3):19 – 34, 2005. Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004). 24

[KM06] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In Yossi Azar and Thomas Erlebach, editors,

Algorithms – ESA 2006, volume 4168 of *Lecture Notes in Computer Science*, pages 456–467. Springer Berlin / Heidelberg, 2006. 10.1007/11841036_42. 37

[Kup95] O. Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, The Technion, 1995. 36

[L+96] J.L. Lions et al. Ariane 5 flight 501 failure. *Report by the Inquiry Board, Paris*, 19:1, 1996. 2, 182

[Laf02] A. L Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical 00176, Universität Freiburg, Institute of Computer Science, 2002. 31

[Lev95] N.G. Leveson. *Safeware: system safety and computers*, volume 12. Addison-Wesley Boston, MA, New York, NY, USA, 1995. 2, 182

[LLP+11] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. Wijs. Multi-Core nested Depth-First search. In T. Bultan and P-A. Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Tapei, Taiwan*, volume online pre-publication of *Lecture Notes in Computer Science*, London, July 2011. Springer Verlag. 32, 35

[LS99] Flavio Lerda and Riccardo Sisto. Distributed-Memory model checking with SPIN. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48234-2_3. 24, 31

[LV01] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In Matthew Dwyer, editor, *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45139-0_6. 24

[LvdPW10] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 247 –255, October 2010. 26, 27, 28, 29, 42, 45

[M+98] G. E Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. 21

[MC99] Andrew Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In Susanna Donatelli and Jetty Kleijn, editors, *Application and Theory of Petri Nets 1999*, volume 1639 of *Lecture Notes in Computer Science*, pages 691–691. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48745-X_2. 57, 68, 141, 155, 156

[MF76] P. Merlin and D. Farber. Recoverability of communication Protocols–Implications of a theoretical study. *Communications, IEEE Transactions on*, 24(9):1036 – 1043, September 1976. 117

[MSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-Checking. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 848–848. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48294-6_22. 16

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, April 1989. 41, 68, 110, 141

[NC97] David M. Nicol and Gianfranco Ciardo. Automated parallelization of discrete State-Space generation. *Journal of Parallel and Distributed Computing*, 47(2):153 – 167, 1997. 25

[Neu92] Peter G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. *SIGSOFT Softw. Eng. Notes*, 17(1):23–32, January 1992. 1, 182

[OPE05] Simona Orzan, Jaco van de Pol, and Miguel Valero Espada. A state space distribution policy based on abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 128(3):35 – 45, 2005. Proceedings of the 3rd

International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004). 24

[Pel07] Radek Pelanek. BEEM: benchmarks for explicit model checkers. In *Proceedings of the 14th international SPIN conference on Model checking software*, page 263–267, Berlin, Heidelberg, 2007. Springer-Verlag. 68, 141, 155, 156

[Pet03] D. Petcu. Parallel explicit state reachability analysis and state space construction. In *Proceedings of Second International Symposium on Parallel and Distributed Computing, ISPDC*, page 13–14, 2003. 25

[PW08] Jaco van de Pol and Michael Weber. A Multi-Core solver for parity games. *Electronic Notes in Theoretical Computer Science*, 220(2):19 – 34, 2008. Proceedings of the 7th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2008). 36

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, page 337–351, London, UK, 1982. Springer-Verlag. 10

[RBC+06] C. L Rodrigues, P. E.S Barbosa, J. M Cabral, J. C.A de Figueiredo, and D. D.S Guerrero. 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*, pages 226 –235, September 2006. 24

[Rei85] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985. 30, 80

[Rei07] J. Reinders. *Intel threading building blocks.* O'Reilly, 2007. 73, 74, 169, 172

[SD95] Ulrich Stern and David Dill. Improved probabilistic verification by hash compaction. In Paolo Camurati and Hans Eveking, editors, *Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 206–224. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60385-9_13. 38

[SD96] Ulrich Stern and David L. Dill. A new scheme for Memory-Efficient probabilistic verification. In *in IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, page 333–348, 1996. 38

[SD97] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. In *In Computer Aided Verification. 9th International Conference*, page 256–267. Springer-Verlag, 1997. 25, 73, 172

[Sti99] C. Stirling. Bisimulation, modal logic and model checking games. *Logic Journal of IGPL*, 7(1):103–124, 1999. 36

[SZB10] Rodrigo T. Saad, Silvano Dal Zilio, and Bernard Berthomieu. A general Lock-Free algorithm for parallel state space construction. *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, 0:8–16, 2010. 4, 43, 171, 172, 185

[Tar71] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114 –121, October 1971. 17, 29, 44, 80

[TSDZB11] Rodrigo T. Saad, Silvano Dal Zilio, and Bernard Berthomieu. Mixed Shared-Distributed hash tables approaches for parallel state space construction. In *International Symposium on Parallel and Distributed Computing (ISPDC 2011)*, page 8p., Cluj-Napoca, Romania, July 2011. Rapport LAAS 11460. 4, 43, 171, 172, 185

[Wil01] T. Wilke. Alternating tree automata, parity games, and modal m-Calculus. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 8(2):359, 2001. 36

[WL93] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-56922-7_6. 37, 40

[Wol86] P. Wolper. An automata-theoretic approach to automatic program verification. In *IEEE Symposium on Logic in Computer Science*, pages 322–331. Computer Society, 1986. 16, 29