Coping with the State Explosion Problem in Formal Methods: Advanced Abstraction Techniques and Big Data Approaches

A Thesis presented for the degree of

Doctor of Philosophy

presented by

Matteo Camilli

under the supervision of

Prof. Carlo Bellettini

and the co-supervision of

Prof. Mattia Monga

Dr. Lorenzo Capra

Dept. of Computer Science
Università degli Studi di Milano
Italy

February 2015

Dedicated to

Daniela

Coping with the State Explosion Problem in Formal Methods: Advanced Abstraction Techniques and Big Data Approaches

Matteo Camilli

Submitted for the degree of Doctor of Philosophy
February 2015

Abstract

Formal verification of dynamic, concurrent and real-time systems has been the focus of several decades of software engineering research. Formal verification requires high-performance data processing software for extracting knowledge from the unprecedented amount of data containing all reachable states and all transitions that systems can make among those states, for instance, the extraction of specific reachable states, traces, and more. One of the most challenging task in this context is the development of tools able to cope with the complexity of real-world models analysis. Many methods have been proposed to alleviate this problem. For instance, advanced state space techniques aim at reducing the data needed to be constructed in order to verify certain properties. Other directions are the efficient implementation of such analysis techniques, and studying ways to parallelize the algorithms in order to exploit multi-core and distributed architectures. Since cloud-based computing resources have became easily accessible, there is an opportunity for verification techniques and tools to undergo a deep technological transition to exploit the new available architectures. This has created an increasing interest in parallelizing and distributing verification techniques. Cloud computing is an emerging and evolving paradigm where challenges and opportunities allow for new research directions and applications. There is an evidence that this trend will continue, in fact several companies are putting remarkable efforts in delivering services able to offer hundreds, or even thousands, commodity computers available to customers, thus enabling users to run massively parallel jobs. This revolution is already started in different scientific fields, achieving remarkable breakthroughs through new kinds of experiments that would have been impossible only few years ago. Anyway, despite many years of work in the area of multi-core and distributed model checking, still few works introduce algorithms that can scale effortlessly to the use of thousands of loosely connected computers in a network, so existing technology does not yet allow us to take full advantage of the vast array of compute power of a “cloud” environment. Moreover, despite model checking software tools are so called “push-button”, managing a high-performance computing environment required by distributed scientific applications, is far from being considered such, especially whenever one wants to exploit general purpose cloud computing facilities.

The thesis focuses on two complementary approaches to deal with the state explosion problem in formal verification. On the one hand we try to decrease the exploration space by studying advanced state space methods for real-time systems modeled with Time Basic Petri nets. In particular, we addressed and solved several different open problems for such a modeling formalism. On the other hand, we try to increase the computational power by introducing approaches, techniques and software tools that allow us to leverage the “big data” trend to some extent. In particular, we provided frameworks and software tools that can be easily specialized to deal with the construction and verification of very huge state spaces of different kinds of formalisms by exploiting big data approaches and cloud computing infrastructures.

February 13, 2015

Declaration

The work in this thesis is based on research carried out at the Department of Computer Science of the Università degli Studi di Milano, Italy. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

First and foremost, I would like to thank my supervisor Carlo Bellettini for his teaching, guidance and sound advice during my PhD program. He has been my long-time advisor since 2008, when I was just at my third year of Bachelor degree. He devoted much of his time engaging me in various conversations and working side by side with me throughout these years. I want to say him "thank you so much", for his support, his continuous feedback, constant inspiration and his enthusiasm. I have learnt a lot from him about many aspects of software engineering, programming, research, academia, and more. His teaching and discussions have always been a source of inspirations. He gave me also the opportunity to be the teaching assistant for his "Progettazione del software" course for four consecutive years. I really enjoyed both preparing and doing lectures, edition by edition.

I would like to thank my co-supervisors Lorenzo Capra and Mattia Monga for their guidance and their inspiring discussions we have had in the last years. They have been always available for helping me in solving issues and revising my own work. I look forward to working closely together again in the near future. I want to thank Mattia for bringing a little bit of security awareness into my habits. He has been of great help during my unfortunate inception of PhD program, supporting my research activities and giving me a cozy accommodation at the "LaSER" laboratory. He gave me also the opportunity to do some lectures in different editions of his "Security" course. I want to thank Lorenzo for his help in developing many ideas and his careful guidance through every step of my growth during my studies. I cannot count how many coffee breaks we spent together discussing about research and more.

In addition, I'd like to acknowledge other colleagues and professors from either the Computer Science department of the Università degli Studi di Milano and other institutes with whom I have had useful discussions not always on scientific matters. I'd like to thank all the external reviewer of my thesis for their useful advices, in particular I'd like to thank

Carlo Ghezzi and the all the members of the DEEP-SE group at Politecnico di Milano whom kindly invited me to do a seminar on my research activity followed by interesting discussions and useful insights.

Finally, I'd like to thank my family who have supported me throughout my studies. I want also to express my deepest gratitude to *Daniela*, this thesis is dedicated to you, because I cannot think about this very long journey without your help. Thank you so much for being supportive and always understanding me. Thank you for rearranging my priorities and my calendar, for cleaning up my mind, for helping me in preparing talks and for being at my side while I was far from home. You gave me the strength and often brought light in my moments of darkness.

February 13, 2015

Contents

Abstract **iii**

Declaration **v**

Acknowledgements **vi**

1 Introduction **1**

1.1 Formal Verification 1

1.2 The Rise of Big Data 4

1.3 Problem Statement and Research Goals 6

1.4 Contributions 7

1.4.1 Real-time Systems Reachability Analysis 7

1.4.2 Big Data Approaches to Formal Verification 8

1.5 Dissemination 8

1.6 Organization 10

2 Background **12**

2.1 Petri Nets 12

2.2 Time Basic Petri Nets 14

2.3 State Space and Abstract State Space 16

2.4 Petri Nets Coverability Graphs 19

2.5 The State Explosion Problem 22

3 Dealing With Infinite-states Real-time Systems **24**

3.1 State of the Art 25

3.2 Verification of Reachability Properties for Time Basic Petri Nets 27

3.2.1 Basic notions 28
3.2.2 Time-coverage graph construction 29
3.2.3 Time Anonymous 33
3.2.4 Formal Definitions 36
3.2.5 Property Evaluation 42
3.2.6 Tool Architecture 44
3.2.7 Use Case and Comparison with other tools 45
3.2.8 Summary 46
3.3 Constructing Coverability Graphs for Time Basic Nets 47
3.3.1 Preliminaries 48
3.3.2 Coverability Tree Algorithm 50
3.3.3 Related Work 56
3.3.4 Summary 58

4 Big Data Approaches to Formal Verification 60

4.1 State of the Art 61
4.2 The MapReduce Programming Model 63
4.3 Symbolic State Space Exploration of RT Systems in the Cloud 69
4.3.1 Sequential and Parallel algorithms 70
4.3.2 Distributed implementations 74
4.3.3 Evaluation 79
4.3.4 Summary 81
4.4 Simplified Building of Reachability Graphs on Large Clusters 82
4.4.1 MaRDiGraS 83
4.4.2 Use Cases 87
4.4.3 Evaluation 89
4.4.4 Related Work 90
4.4.5 Summary 92
4.5 CTL Model Checking Using MapReduce 93
4.5.1 Computation Tree Logic 94
4.5.2 Fixed-Point Algorithms 96
4.5.3 Distributed Algorithms 97
4.5.4 Model-checking of Abstract Transition Systems 102

4.5.5 Complexity 104
4.5.6 Evaluation 105
4.5.7 Related Work 108
4.5.8 Summary 111

5 Conclusion 112
5.1 Contributions 113
5.2 Open Issues 115
5.3 Future Work 115

Appendix 132

A Petri Nets Models 132
A.1 The Gas Burner 132
A.2 The Shared Memory 134
A.3 The Dekker 136
A.4 The Simple Load Balancing 137

B Proofs 140
B.1 Correctness of the $EX\phi$ Algorithm 140
B.2 Correctness of the $EG\phi$ Algorithm 141
B.3 Correctness of the $E[\phi U \psi]$ Algorithm 142

List of Figures

2.1 Simple Petri net modeling a Two-process Semaphore example. 13
2.2 TB net running example. 16
2.3 Edges types of an abstract state space. 18
2.4 State space of the Petri net example shown in Figure 2.1. 19
2.5 Petri net coverability graph example. 21

3.1 Sample reachability graph. 29
3.2 Simple TB net example generating a "dead" token. 34
3.3 Infinite (a) and finite (b) representations of the reachability graphs extracted from the model shown in Fig. 3.2. 34
3.4 Simple excerpt of a TB net model used for TA heuristics examples. 38
3.5 Critical case for path feasibility. 43
3.6 Reference architecture. 44
3.7 State creation advancement. 45
3.8 Simple example showing an unbounded TB net model. 48
3.9 Portion of the infinite reachability tree associated to the TB net model presented in Figure 3.8. 48
3.10 Deactivations of symbolic states in the TBCT Algorithm 53
3.11 Coverability tree constructed from the TB net example presented in Figure 3.8. 53
3.12 Coverability graph constructed from the coverability tree presented in Figure 3.11. 54
3.13 Unbounded TB net model A. 55
3.14 Coverability tree/graph of example A. 56
3.15 Unbounded TB net model B. 56

3.16 Coverability tree/graph of example B. 57

3.17 Unbounded TB net model C. 58

3.18 Coverability tree/graph of example C. 59

4.1 The MapReduce model: the keys are in **bold**. 66

4.2 MapReduce execution overview. Adapted from [46]. 67

4.3 `Workers` model. 73

4.4 `Mappers` & `Folders` model. 73

4.5 Operations on edges during the `Fold` phase. 74

4.6 Distributed JavaSpaces model. 76

4.7 Hybrid Iterative *Map-Reduce* model. 78

4.8 Expansion front over time. 79

4.9 State space partitioning among 32 `Workers`. 81

4.10 Class diagram of the MaRDiGraS framework. 83

4.11 Reachability graph computation of the simple-lbs model. 91

4.12 (a) $T \models_s AF\phi$; (b) $T \models_s EF\phi$; (c) $T \models_s EG\phi$; (d) $T \models_s E[\phi U \psi]$ 96

4.13 Execution time, cheat and efficiency graphs. 109

A.1 The Shared memory Colored net model. 135

A.2 The Dekker P/T model with parameter $N = 2$. 136

A.3 The Simple load balancing Colored net model. 137

A.4 The Gas burner Time Basic Petri Net model. 139

List of Tables

3.1 Use case analysis results. 46

4.1 TRG Building Experiments Report 80
4.2 MaRDiGraSExperiments report 89
4.3 Shared memory ($\sim 10^6$ states, $\sim 10^7$ transitions) analysis report 106
4.4 Dekker model (10^7 states, 10^8 transitions) analysis report 107
4.5 Simple load balancing model ($\sim 10^8$ states, $\sim 10^9$ transitions) analysis report 108

A.1 Size of the Shared memory derived P/T net model instances. 135
A.2 Size of the Dekker P/T net model. 136
A.3 Size of the Simple load balancing P/T net model. 138

Chapter 1

Introduction

1.1 Formal Verification

Ensuring the correctness of software and hardware products is an extremely important issue, because failures could be catastrophic, since today information systems are deployed in safety-critical or life-critical settings. Furthermore, it is well-known that, even when the systems are not safety-critical or life-critical, errors could still result in a substantial loss of money or productivity. This has led to an increased interest in applying formal methods and verification techniques in order to ensure the correctness of the developed systems. For instance, *model checking* [43] is one of the most successful techniques that are widely used in both research and industry. Broadly speaking, in order to check that a system satisfies a certain property, we first create a model $\mathcal{M}$ (usually as a finite state transition system) that describes how the system evolves, and we express the property as a formula ϕ in some logical language apt to predicate on model entities (usually some temporal logic). This reduces the initial problem to checking whether $\mathcal{M}$ satisfies ϕ. In addition to model checking, many other state space methods were developed in order to support the computer-aided analysis and verification of the behavior of systems.

Formal verification of dynamic [1], concurrent [2] and real-time systems [3] has been the focus of several decades of software engineering research. One of the most challenging

[1]Systems characterized by their behavior over time [59].
[2]Different partially autonomous components which run in parallel and influence each other by interactions.
[3]System that must operate within the confines of certain temporal deadlines.

tasks in this context is the development of tools able to cope with the complexity of real world models' analysis. In fact, the main obstacle that model checking, and more generally formal verification techniques, faces is the *state explosion* problem [108]: the overall number of states of a concurrent system with multiple processes can be enormous. It grows up exponentially both in the number of processes and in the number of per-process components. Moreover, taking into account real-time or time-critical systems, this condition is even more complicated. In fact, for those systems, functionalities are defined with respect to time and their correctness can only be assessed by taking time into consideration. Therefore, such a systems reach an infinite or even uncountable number of states depending on the time domain adopted during the modeling phase. Abstraction techniques aim at constructing a finite contraction of the model $\mathcal{M}$ by removing some irrelevant details, which preserves properties of interest.

The research for efficient ways to implement abstraction techniques and verification algorithms started many years ago, in the late seventies and early eighties. Since then, the computational power of desktop machines grew up impressively. Around every eighteen months, speed and available memory doubled resulting directly into comparable increases in the efficiency and power of the available formal verification software tools. These breakthroughs came along with the algorithmic improvements that could be made to verify efficiently such as the use of partial order reduction and bitstate hashing techniques in software model checkers [75], effective/implicit state-space representations and symbolic model checking techniques in hardware model checkers [31], bounded model checking [90], model checking modulo theories [68], advanced state space methods for real-time systems [4, 20], and so forth. These breakthrough techniques enabled the development of tools that can tackle problems of a fairly big size that were unimaginable before. It has made it possible for us to continue to achieve further and further with the hope that our algorithms and machines would be able to cope with the potentially very large computational complexity of these problems.

Approximately fifteen years ago, though, manufacturers changed the trend of shrinking transistors in order to pack more of them onto a chip. Thus the trend started to depart from density driven approaches moving into parallelization: placing larger number of CPU cores onto a single chip. Large numbers of independent threads of execution can now all be executed in parallel, with mostly limited competition for shared resources. Instead of continuing to double the raw speed of CPUs, the chip makers now plan to double the number of cores on a chip with each new generation. Curiously, although the raw speed of CPUs has stalled at roughly 2002 levels, the size of RAM memory that is available on standard desktop systems continues to follow Moore's law [97]. Clearly, in multi-core systems the need for memory increases at least linearly with the number of CPU cores used, so the trend is understandable. But the growing divergence between memory size and the basic speed of a single CPU has important consequences for our work.

Multi-core verification techniques have been an active area of research for at least a decade (e.g., [103, 78, 9]). Verification techniques based on explicit representation of reachable states gained much more advantages exploiting computational power of multi-core computers. In fact, despite the impressive effectiveness of symbolic breakthrough techniques, which enabled the analysis of systems with a fairly big number of states, there are still valid reasons to use explicit approaches. It is widely accepted that explicit state model checking is better for verifying software systems [50], due to the intrinsic complexity of the state notion. Moreover symbolic approaches do not permit the computation of certain quantitative properties (e.g., state probabilities in stochastic Petri nets models [13]). Parallel model checking procedures, utilize all available processor cores to realize an exhaustive exploration of different partitions of the systems behavior independently. Thus, obtaining, in some cases, a speedup proportional to the number of cores. Although the exploitation of parallel threads or processes, these algorithms do not always give predictable performance, and they do not necessarily scale well to the use of very large numbers of CPUs (e.g., hundreds or thousands). There are several issues to face making scalability difficult to achieve. A first issue consists of the communication overhead that is needed in coordinating a search across multiple CPUs. Sometimes significant amounts of data must be exchanged between computational units to avoid overlapping work. This data exchange can be expensive, for instance in shared memory machines, a high number of concurrent access, from different parallel processes, becomes costly. Moreover, the presence of multiple threads in an application opens up potential issues regarding safe access to resources from multiple threads of execution. Different threads modifying the same resource might interfere with each other in unintended ways. Thus, communication often needs synchronization among independent processes in order to reach an agreement or commit a certain sequence of actions. Synchronization tools are often based on the definition of mutual exclusion regions of executions, which definitely increase the complexity of the application and often have a negative impact on performance.

1.2 The Rise of Big Data

Likewise the recent change of chip manufacturers from the development of single-core to multi-core CPUs, for desktop systems, a new trend came into sight applying Internet both as a development environment and execution infrastructure. and more in general people lives. A new technological trend is rising up: "Big data". Big data approaches start with the fact that there is a lot more information floating around nowadays than ever before. Streams of data come from our everyday life: from phones, credit cards, televisions, computers, from the infrastructure of cities, from sensor-equipped buildings or living beings, cars, factories and so forth. In this Big Data World information is unbelievably large in scale, scope, distribution, heterogeneity, and supporting technology. The exploding world of Big Data poses, more than ever, two challenge classes: efficiently managing data at unimaginable scale; and meaningfully combining information that is relevant to your concern. Industry analysis companies like to point out there are challenges not just in *Volume*, but also in *Variety*, *Velocity*, and *Veracity*. *Variety* refers to heterogeneity of data types, representation, and semantic interpretation. *Velocity* denotes both the rate at which data arrive and the time frame in which they must be processed. *Veracity* refers to the fact that data can be either uncertain, inaccurate or spoiled. In fact, every analytic exercise, spend a large amount of time on removing duplicates, fixing partial entries, eliminating null/blank entries, concatenating data, collapsing columns or splitting columns, aggregating results, and more. In addition to these challenges, other concerns, such as privacy and usability, still remain.

Different names have been used to describe this emerging trend. The term "Grid computing" first appeared in the nineties [57]. More recent terms are: "Cloud computing" and "Network centric" computing. Cloud computing is still an emerging and evolving paradigm where challenges and opportunities allow for new research directions and applications. Companies such as Amazon, Microsoft, and Google are all working on services that offer hundred or even thousands commodity compute nodes available to customers, thus enabling users to run massively parallel jobs. The evidence shows that this trend will continue. Once reached maturity, it could dramatically change the way scientific computing tasks can be performed. This revolution is already started in different scientific fields, achieving remarkable breakthroughs [80] through new kinds of experiments that would have been impossible only a decade ago. As many scientific application domains, different formal verification approaches require high performance data processing tools for extracting knowledge from the unprecedented amount of information coming from analyzed systems. Many problems in formal methods require the analysis of the state space, for instance, we may require the extraction of reachable states or paths satisfying certain constraints or we may compute specific structural properties and so forth. Since cloud based computing resources are becoming more and more accessible, there is a great opportunity for verification techniques and tools to undergo a deep technological transition in order to exploit the new available architectures. Anyway, despite many years of work in the area of multi-core and distributed model checking, there are few works introducing algorithms that can scale effortlessly to the use of thousands of loosely connected computers in a network, so existing technology does not yet allow us to take full advantage of the vast array of compute power of a "cloud" environment.

In the context of explicit reachability analysis and explicit-state model checking, taking advantage of a distributed environment is important to cope with real-world cases. The idea is simple: increasing the computing power and storage availability, by using a cluster of distributed computers. The use of networks of computers can provide the resources required to verify complex systems' models. Unfortunately, this approach requires several skills which – while common in the "big data" community – are rather unusual in the "formal methods" community. In a distributed setting, we should find an efficient distributed representation of both data to be analyzed, in order to minimize communication among compute units, and the solution for a problem, in a way that can it be easily retrieved and further analysis can be performed. For instance, each node of a distributed state space can hold pointers to all/one of its outgoing/incoming edges depending on the specific algorithm adopted for distributed analysis. Moreover, the number of edges having the source state stored in a component and the target in another component generally have a heavy impact on the overall number of messages sent over the network during analysis. Other challenges of primary importance are *load balancing* and *fault tolerance*. Since faults are much likely to occur in a distributed environment, the latter issue is particularly severe in the context of formal verification that must ensure a correct answer to a problem.

The connection between formal methods in software engineering and big data approaches were recently studied in our recent works [18, 17, 33, 34, 35]. The analysis of complex systems certainly falls in this context, although applying big data approaches to solve formal verification problems has been so far poorly explored [72, 77]. We believe, however, that the challenges to be tackled in formal verification can benefit a lot from the recent achievements in big data access and management. In fact formal approaches require several different skills: An adequate background is required in order to manage specific formalisms and abstraction techniques both in modeling and analysis interpretation. Moreover, these techniques should be deployed into software tools able to analyze large amount of data very reliably and efficiently, similarly to "big data" projects. Recent approaches have shown the convenience of employing distributed memory and computation to manage generation/exploration of large state-spaces. Unfortunately exploiting these frameworks requires further skills in developing complex applications with knotty communication and synchronization issues. In particular, tailoring applications so that they conveniently scale on available cloud computing facilities, might be a daunting task without a proper knowledge of the subtleties of data-intensive and distributed analysis.

1.3 Problem Statement and Research Goals

State space methods are among the most important approaches to computer-aided analysis and verification of the behavior of dynamic, concurrent, or even real-time systems. Basically, they consist of enumerating and analyzing the set of the reachable states of the system. Unfortunately, the number of reachable states is often far greater than can be handled in a realistic computer. This is a well known problem so called *state explosion problem*. Many advanced state space methods alleviate the problem by using a subset or an abstraction of the set of states that often restrict the set of problems that can be solved.

This statement leads to the definition of the following overall research goal: To alleviate the state explosion problem by exploring advanced abstraction techniques and data-intensive computational models in cloud computing infrastructures that allow us to deal with very large state spaces by either reducing the exploration space or increasing the computational power.

Thus, the overall research goal can be decomposed into two smaller research goals, belonging to two different branches of formal methods in software engineering:

Advanced abstraction techniques We focused on the analysis of real-time systems, in particular the reachability analysis of real time systems modeled with Time Basics (TB) nets [66] is still recognized as an open problem [86]. Moreover there is a lack of software tools supporting the analysis of TB nets. This calls for new advanced state space methods supported by software tools to overcame the major limitations of the currently available analysis techniques.

Big data approaches to formal verification Despite many years of work in the area of multicore and distributed formal verification, existing software tools do not yet allow us to take full advantage of the vast array of compute power of a “cloud” environment. Thus, there exists the opportunity to benefit from the recent achievements in big data access and management in the area of formal verification. This calls for new frameworks and software tools able to run massively parallel computations in the cloud to tackle the state explosion in many formal verification problems. Such a frameworks should re-enable a “push-button” mode into the distributed verification context even when these (complex on themselves) computing resources are involved.

1.4 Contributions

The thesis contains contributions in two branches within the area of formal methods in software engineering. These correspond to the two different parts of the thesis. Both contributions aim at alleviating the state explosion problem, but using two different complementary approaches. The first main contribution lies in the introduction of advanced state space methods able to deal with infinite-states real-time systems. The second main contribution focuses on the connection between formal methods in software engineering and big data approaches. In particular we outline approaches that will allow verification techniques and tools to undergo the recent technological transition in order to exploit the new available architectures.

1.4.1 Real-time Systems Reachability Analysis

The reachability analysis of real time systems modeled with Time Basics (TB) nets [66] is still recognized as an open problem [86]. Available analysis techniques and tools (e.g., [86, 65]) are based on inspecting a finite portion of the potentially infinite reachability-tree generated by a TB net. Thus, only time-bounded properties can be inferred from the state-space exploration of TB nets by using this kind of analyzers. The technique described in this thesis overcomes this major limitation. It relies on a symbolic reachability graph algorithm, which is in turn based on a relative notion of time and a procedure verifying inclusion between symbolic states. A particular state normalization, able to recognize and eliminate timestamp symbols actually not influencing the model evolution, permits in many cases the building of a sort of time coverage finite graph. This abstraction gave us a means to develop also an algorithm able to construct the coverability tree of a TB net model, which is a finite representation of some over-approximation of the reachable markings. This allows to deal with topologically unbounded TB nets models and decide different properties such as *coverability*, *boundedness*, *place-boundedness*, *semi-liveness*.

1.4.2 Big Data Approaches to Formal Verification

We try to benefit from the recent achievements in big data access and management in the area of formal verification. In particular we try to further bridge the gap between these different areas of expertise by providing frameworks and software tools that can be easily specialized to deal with the construction and verification of very huge state spaces of different kinds of formalisms (e.g., different kinds of Petri Nets, Process Algebras etc.) by exploiting cloud computing infrastructures. In particular we take advantage of the MapReduce programming model [46] and its related implementation Hadoop MapReduce [106], simplifying the task of dealing with a large number of reachable states by exploiting large clusters of machines.

In particular we studied and compared two different approaches, relying on distributed and cloud frameworks, respectively, to explore symbolic state-spaces of TB net models. Moreover, we introduced MaRDiGraS, a generic framework aimed at simplifying the construction of very large state transition systems on large clusters and cloud computing platforms. Finally, we enabled the verification of Computation Tree Logic (CTL) formulas on very large state spaces by adopting computational models relying on cloud computing facilities.

Our evaluations report that our approaches can be used effectively to build and analyze state spaces of different orders of magnitude. In some cases we have shown a potential for a super-linear speedup.

1.5 Dissemination

The research work carried out during my three years P.h.D. program has been disseminated through different publications. This section lists them together with a brief explanation about their contribution in this thesis.

Conference and Workshop papers

- Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. 2012. Symbolic State Space Exploration of RT Systems in the Cloud. In Proceedings of the 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '12). IEEE Computer Society, Washington, DC, USA, 295-302. DOI=10.1109/SYNASC.2012.18
  http://dx.doi.org/10.1109/SYNASC.2012.18

- Matteo Camilli. 2012. Petri nets state space analysis in the cloud. In Proceedings of the 34th International Conference on Software Engineering (ICSE '12). IEEE Press, Piscataway, NJ, USA, 1638-1640.

  These two papers introduce a study and comparison between two different approaches, relying on distributed and cloud frameworks, respectively. These approaches were designed and implemented following the same computational schema, a sort of map & fold. They are applied on symbolic state-space exploration of real-time systems specified by TB Nets. The outcome of different tests performed on a benchmarking specification are presented, thus showing the convenience of distributed approaches. Section 4.3 introduces this work.

- Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Mardi- gras: Simplified building of reachability graphs on large clusters. In Parosh Aziz Abdulla and Igor Potapov, editors, Reachability Problems, volume 8169 of LNCS, pages 8395. Springer Berlin Heidelberg, 2013.

  This paper introduces MaRDiGraS, a generic framework aimed at simplifying the construction of very large state transition systems on large clusters and cloud computing platforms. Through a simple programming interface, it can be easily customized to different formalisms, for example Petri Nets, by either adapting legacy tools or implementing brand new distributed reachability graph builders. This work is presented in section 4.4.

- Matteo Camilli, Carlo Bellettini, Lorenzo Capra, and Mattia Monga. CTL Model Checking in the Cloud using MapReduce. In Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on, pages 333-340, Los Alamitos, CA, USA, Sept 2014. IEEE CS Press. doi: 10.1109/SYNASC.2014.52.

- Matteo Camilli. 2014. Formal verification problems in a big data world: towards a mighty synergy. In Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 638-641. DOI=10.1145/2591062.2591088
  http://doi.acm.org/10.1145/2591062.2591088

  These two works are the basis of section 4.5. They introduces a distributed approach which exploits techniques typically used by the “big data” community to enable verification of Computation Tree Logic (CTL) formulas on very large state spaces. A computational model relying on cloud computing facilities is used.

Technical Reports

The following two works are currently technical reports and they are presented in chapter 3.

- Matteo Camilli: Verification of Reachability Problems for Time Basic Petri Nets. CoRR abs/1409.2778 (2014) `http://arxiv.org/abs/1409.2778`

  This is an extended version of [14]. It introduces a technique that enables the verification of reachability properties of RT systems modeled with TB nets. It relies on a finite symbolic reachability graph construction algorithm, which is in turn based on a relative notion of time and a procedure verifying inclusion between symbolic states. This extended version is enriched with a more in depth section about the Time Anonymous concept and some relevant new core definitions and heuristics.

- Matteo Camilli: Constructing Coverability Graphs for Time Basic Petri Nets. CoRR abs/1409.6253 (2014) `http://arxiv.org/abs/1409.6253`

  This latter work introduced a technique able to compute coverability graphs of real-time of TB net models. This technique extends the previous one, further exploiting the time anonymous concept in order to deal with topologically unbounded nets.

1.6 Organization

The thesis contains four main chapters.

The first one introduces a background needed to comprehend the following chapters. In particular it provides information about different formalisms for modeling concurrent and real-time systems. Moreover, it supplies information about labeled state transition systems which stands on the basis of the notion of state space and abstract state space. The background concludes with a brief overview on the state explosion problem.

Chapter 3 introduces two different works: the first one introduces a technique for reachability analysis of TB nets. The technique enables the building a finite symbolic reachability graph relying on a sort of time coverage notion, and overcomes the limitations of available analyzers for TB nets, based in turn on a time-bounded inspection of a (possibly infinite) reachability-tree; The latter work of this chapter, introduces a technique able to cope with topologically unbounded TB nets in order to determine different important properties also for such a systems. The technique exploits the abstraction introduced in the previous work, and builds upon it an algorithm able to compute coverability trees/graphs.

Chapter 4 focuses on the connection between formal methods in software engineering and big data approaches. This part of the thesis tries to overcome the major limitation of the software tools introduced in the previous chapter. In particular we outline approaches that allow verification techniques and tools to undergo a technological transition in order to exploit the new available architectures. After reasoning about the opportunity in parallelizing the TB nets analysis techniques, we generalize such a discussion resulting in the realization of the MaRDiGraS generic library. This framework is built on top of Hadoop MapReduce and can be easily specialized to deal with the construction of very large state spaces of different kinds of formalisms. Another work presented in this chapter outlines a distributed CTL (Computation Tree Logic) model checker, which implements iterative MapReduce algorithms based on the fixed-point characterization of the basic temporal operators of CTL.

Chapter 5, draws a conclusion and a brief overview on future works and research directions.

Appendix A describes all the benchmarking models used for experimentation. Appendix B presents the proofs of correctness of the algorithms introduced in section 4.5.

Chapter 2

Background

This chapter introduces an adequate background needed to comprehend the following chapters. In particular it provides information about Petri Nets and Time Basic Petri Nets formalisms to model concurrent and real-time systems, respectively. Moreover, it supplies information about labeled state transition systems which stands on the basis of the notion of state space and abstract state space. The background concludes with a brief overview on the state explosion problem. It also introduces some notation which will be used in the current and following chapters.

2.1 Petri Nets

Petri nets or Place/Transition (P/T) nets are a compact and elegant models of distributed and asynchronous systems because such models support the notion of distributed state combined with synchronization through shared transitions. For instance two generic concurrent interacting processes can be modeled by the Petri net in Fig. 2.1. In the Petri net, the presence of *tokens* (small black disks) in states (called *places* according to Petri net terminology) indicates that the component process is currently in that state, and hence ready to perform the action represented by the transition connected to the state (graphically represented by a thick bar). When an action is shared (for example t_1 and t_2 transitions), it can be carried out only if all participating processes are ready to execute it, that is, they are in the state connected to the transition (by means of a directed arrow from the corresponding place to the transition).

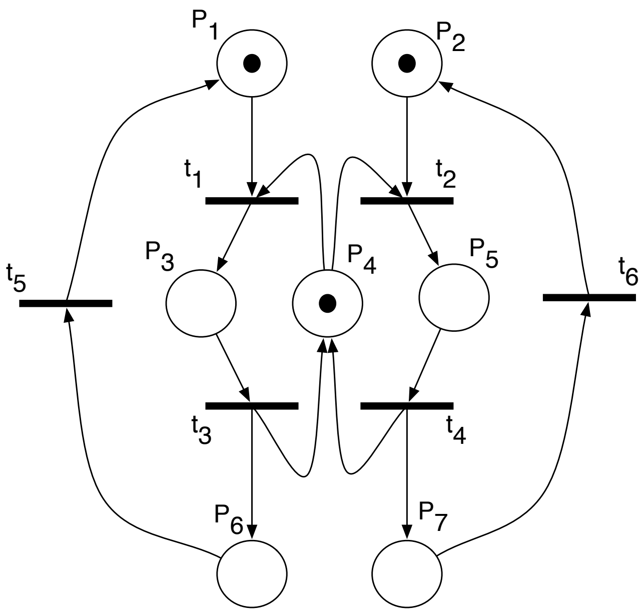


Figure 2.1: Simple Petri net modeling a Two-process Semaphore example.

More formally, a Petri net is a triple $\langle P, T, F\rangle$, where P is a finite set of places, T a finite set of transitions such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ a set of arcs (or flows) connecting places to transitions and transitions to places. The graphic representation of a Petri net represents places as circles, transitions as thick line segments, and arcs as directed arrows. The *preset* $^\bullet t$ of a transition t is the set of places directly connected to t by an arc. The *preset* $^\bullet p$ of a place p is the set of transitions directly connected to p by an arc.

$$^\bullet t = \{p \in P : (p,t) \in F\};\ ^\bullet p = \{t \in T : (t,p) \in F\}$$

The *postsets* of transitions and places are defined similarly.

$$t^\bullet = \{p \in P : (t,p) \in F\};\ p^\bullet = \{t \in T : (p,t) \in F\}$$

The example in Fig. 2.1 shows a Petri net with six transitions, $\{t_1, t_2, \ldots, t_6\}$ seven places $\{P_1, P_2, \ldots, P_7\}$, and sixteen arcs. The postset of transition t_3 is $\{P_4, P_6\}$. The preset of place P_6 is $\{t_3\}$.

The behavior of a Petri net is defined by the location changes of tokens in the places, occurring as a consequence of the firing of transitions. Places can store one or more tokens; a place is marked when it contains at least one token. The number of tokens stored in each place defines the state of the net, also called marking. A marking m is a function $m : P \to \mathbb{N}$ that associates a number of tokens to each place. Given a finite set of states P, a *marking* on P is an element of the set $Mark(P) = \mathbb{N}^P$. Given a marking $m \in Mark(P)$, we represent it either as $\langle m(p_1), m(p_2), \ldots, m(p_{|P|})$ (vector notation), or as $\{m(p_1)p_1, m(p_2)p_2, \ldots, m(p_k)p_k\}$ (multiset notation). For instance, $(1, 0, 0, 2)$ on $P = (p_1, p_2, p_3, p_4)$ is represented by the multiset $\{p_1, 2p_4\}$.

When all the places in the preset of a transition are marked, the transition is enabled. An enabled transition can fire; the firing removes a token from each place in the transitions preset and deposits one into each place in the transitions postset. If more than one transition is enabled in a given marking, the choice of which transition will fire is *nondeterministic*. Notice that a place p may be both in the preset and in the postset of the same transition t. When a transition such as t fires, a token in p is consumed and replaced by a new one immediately.

2.2 Time Basic Petri Nets

Time Basic nets belong to the category of Petri nets in which system time constraints are expressed as numerical intervals associated to each transition, representing possible firing instants computed since transition's enabling. Tokens, atomically produced by the firing of a transition, are thereby associated to time-stamps with values ranging over a determined set. With respect to the well-known representative of this category, (i.e., Time Petri nets [19]), interval bounds in TB nets are linear functions of timestamps in the enabling marking, rather than simple numerical constants. TB nets thus represent a much more expressive formal model for real-time systems than Time Petri nets.

TB nets are Petri nets where each token is associated with a time-stamp representing the instant at which it has been created. The domain of timestamps is $\mathbb{R}^+$. The structure of a Time Basic net extends the P/T net structure (P, T, F). A (time-stamp) *tuple* of $t \in T$ is an association $en : {}^\bullet t \to \mathbb{R}^+$. Each transition t is associated with a *time function* f_t which maps a tuple en of t to a (possibly empty) set of $\mathbb{R}^+$ values. A *marking* (state) is a mapping $m : P \to Bag(\mathbb{R}^+)$, $Bag(A)$ being the set of multiset over A. A tuple en of t is said to be *enabling* in m, in accordance to a *weak* semantics (as explained next), if $\forall p \in {}^\bullet t$ $en(p) \in m(p)$ and $f_t(en) \neq \emptyset$. $f_t(en)$ represents the possible firing times for en. Letting en be an enabling tuple of t in m, a pair (en, τ), $\tau \in f_t(en)$, is said a *firing instance* of t (in m).

The firing of (en, τ) produces the new marking m', s.t. $\forall p \in {}^\bullet t \setminus t^\bullet \; m'(p) = m(p) - en(p)$, $\forall p \in t^\bullet \setminus {}^\bullet t \; m'(p) = m(p) + \tau$, $\forall p \in t^\bullet \cap {}^\bullet t \; m'(p) = m(p) - en(p) + \tau$; for all remaining places, $m'(p) = m(p)$. This will be as usual denoted $m[(en, \tau) > m'$.

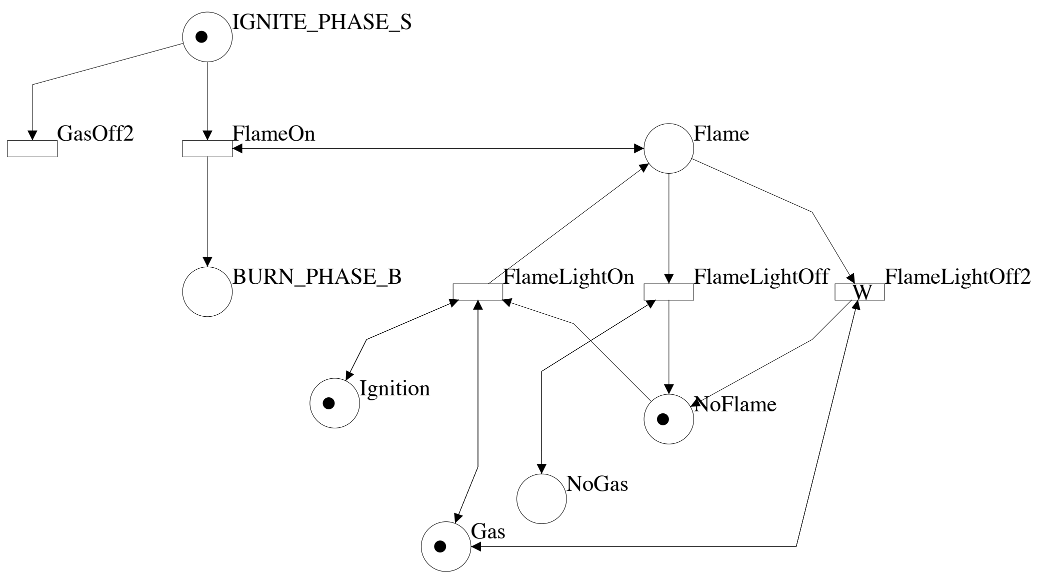
Hereafter a time function f_t is defined by a pair of linear functions $[lb_t, ub_t]$, denoting parametric interval bounds. lb_t, ub_t are in turn formally expressed in terms of (a non empty set of) places in ${}^\bullet t$: $lb_t(en)$, $ub_t(en)$ are the numerical expressions obtained by replacing each place occurrence p with $en(p)$. Time-functions must be monotonic, i.e., $\forall en \; lb_t(en) \geq enab \equiv max(\{en(p)\}, p \in {}^\bullet t)$. We will keep such assumption implicit in their formal notations.

The set of firing times $f_t(en)$ can be interpreted in at least two different ways, leading to different time semantics for *each* transition t. A first interpretation states that an enabling tuple en of t *can* fire at any instant $\tau \in f_t(en)$. Transitions with one such semantics are referred to as *weak*. A second interpretation states that an enabling tuple *must* fire at an instant $\tau \in f_t(en)$, unless it is disabled by the firing of any conflicting enabling tuple at an instant no greater than the latest firing time of t. Transitions with one such semantics are referred to as *strong*. Thereby the enabling condition previously given must take into account also the possible presence of other strong enabling tuples [66]. Notice that the only possible semantics for Time Petri Nets [19] is strong.

In order to meet an intuitive notion of time, TB net firing sequences are restricted to the set of firing sequences whose firing times are monotonically non decreasing with respect to the firing occurrences. However, the time of a firing may be equal to the enabling time of the tuple that belongs to the firing. Intuitively this means that an effect (the firing) can occur with no delay after the cause (that enables it) is fulfilled. Therefore, it is possible to have sequences of firings where the time does not change. In practice, it is useful to restrict the attention to a subclass of TB nets, such that there exist no infinitely long firing sequences which take a finite amount of time (non Zenonicity).

Consider the excerpt from the use case, depicted in Fig. 2.2. It relates to the *Ignite Phase*, just after the ignition transformer has been started and the gas valve has been opened. In this phase the controller must check if the flame has been lighted within a specific deadline, otherwise a recovery procedure that brings the system to *Idle* has to be activated. All transitions are strong, but *FlameLightOff2*. This permits us to express the *possibility* that an event occurs within a given time interval.

The flame turns on if there are *Ignition* and *Gas* (transition *FlameLigthOn*), but it can turn off if no gas is supplied (transition *FlameLigthOff*) or due to a failure, caused e.g. by



Initial marking	$IGNITE\_PHASE\_S\{T_0\}$ $Ignition\{T_0\}$ $Gas\{T_0\}$ $NoFlame\{T_0\}$
Initial constraint	$0 \leq T_0 \leq 10$
FlameOn	$[IGNITE\_PHASE\_S + 0.01, max(\{Flame + 0.1, IGNITE\_PHASE\_S + 0.01\})]$
FlameLightOn	$[enab + 0.5, enab + 0.5]$
FlameLightOff	$[enab, NoGas + 0.1]$
FlameLightOff2	$[enab, enab + 100]$ with weak time semantic
GasOff2	$[enab + 2, enab + 2]$

Figure 2.2: TB net running example.

wind (transition *FlameLigthOff2*). The time function associated with transition *FlameOn* (representing the system passing to *burnstate* after recognizing that the flame has turned on) can be interpreted as follows: *FlameOn* cannot fire before 0.01 time units elapse since the appearance of a token in place $IGNITE\_PHASE\_S$ (the minimum permanence time in *ignitestate*) and implicitly not before the timestamp in place *Flame*. The firing time cannot exceed the maximum between the timestamp of the token in place $IGNITE\_PHASE\_S$ plus 0.01 time units and the time-stamp of the token in place *Flame* plus 0.1 (i.e., the system recognizes the presence of a flame within this 0.1 units). Noticeably, this is an example of constraint that cannot be directly expressed using Time Petri Nets formalism [19].

2.3 State Space and Abstract State Space

The behaviour of a discrete-event dynamic, concurrent or real-time system is formally given in terms of a labeled state transition system: its *state space* (or concrete state space). $(S, \Lambda, \rightarrow)$

where S is the set of system's states, Λ is a set of labels, and $\rightarrow \subseteq S \times \Lambda \times S$: $(s, \lambda, s') \in \rightarrow$ if and only if s' is reachable from s through the occurrence of λ (s' is said to be a *successor* of s and it is written as $s \overset{\lambda}{\rightarrow} s'$).

The transition system associated to a Petri net model with initial marking s_0 is such that $s_0 \in S$, and $s \overset{\lambda}{\rightarrow} s'$ if and only if λ is a transition instance enabled in s, whose firing leads to s'. $S = \{s | \sigma_0 \overset{*}{\longrightarrow} s\}$ is the set of reachable states ($\overset{*}{\longrightarrow}$ is the reflexive and transitive closure of the relation $\rightarrow$ defined above). This structure can be represented with a state-transition graph $\langle S, E, s_0 \rangle$, where the set of edges E represents the relation $\rightarrow$ and the set of nodes S represents the set of reachable states. The node s_0 represents the initial state of the system. Sometimes, especially in the case of time extensions of PNs, the set S may be infinite, or even uncountable, for instance due to the dense nature of the time domain, we cannot enumerate each reachable concrete state of a RT system. Thus, a common way to face the fact that in general S may be infinite, or even uncountable, (like in some time PN extensions) consists of building a finite contraction of the original (concrete) state transition system. Different techniques are employed for that, depending on the formalisms. A non exhaustive survey regarding high-level PNs may be found in [81].

In general, $(A, L, \Rightarrow)$ is an abstraction of $(S, \Lambda, \rightarrow)$ if each $a \in A$ represents a set of concrete states, A is a coverage of S, i.e., $\bigcup_{a \in A} a \supseteq S$, and, letting f be a morphism $\Lambda \rightarrow L$, relation $\Rightarrow \subseteq A \times L \times A$ satisfies condition `EE` (exists-exists)[23, 17]:

`EE`-(1) if $a \overset{l}{\Rightarrow} a'$, then $\exists s \in a, s' \in a', \lambda \in f^{-1}(l) : s \overset{\lambda}{\rightarrow} s'$

`EE`-(2) if $s \overset{\lambda}{\rightarrow} s'$, then $\forall a \in A$ s.t. $s \in a$, $\exists a' \in A$ s.t. $s' \in a' : a \overset{f(\lambda)}{\Rightarrow} a'$

The first part of condition `EE` avoids two abstract states from being connected if no corresponding concrete states are. The second part ensures that each concrete path corresponds to some abstract path.

Depending on the particular abstraction technique, and the properties one is interested to check [17], it is possible/necessary to further refine condition `EE`, either locally or globally, as informally shown in Fig. 2.3. For example, condition `EA` (exists-for all) imposes that any abstract edge between states a, a' must correspond to a set of concrete ones, between some $s \in a$ and *each* $s' \in a'$. Any (abstract) state-transition system can thus be described by annotating edges with additional information, indicating which kind of connectivity among `EE`, `EA`, `AE` (for all-exists), and `AA` (for all-for all) is locally met. According to this convention, a concrete state space can be represented using only edges labeled `AA`. Edges, as well as

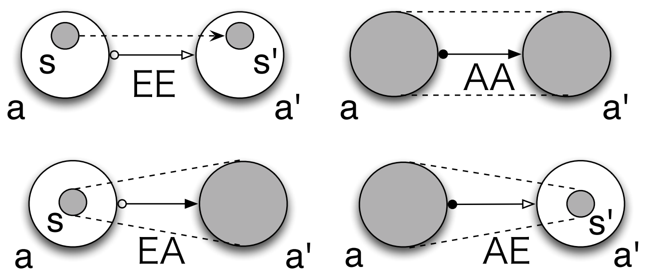


Figure 2.3: Edges types of an abstract state space.

nodes, usually carry other annotations that are specific to the particular formalism one is using. Therefore we can describe the behavior of a PN model with a directed graph, where both nodes and edges can be annotated with additional meta-data. As an example, in time PN extensions, edges are labeled by (symbolic) transition firing time of the transition that leads the system into another state [14, 23]. Nodes, instead, normally hold information on tokens creation times (expressed by linear constraints) [14].

Independently of the formalism used in the modelling phase (PNs, in their several extensions, process-algebras, etc.), we can reformulate most of the algorithms for building (abstract) state-transition systems in terms of an elementary iterative schema, whose essential points are outlined below:

(a) For each unexplored state a, we calculate the set of successors $succ(a)$, identifying which connectivity conditions are met. Then we mark a as explored.

(b) For each $a' \in succ(a)$, we try to identify equivalence/inclusion relationships between a' and any state a''.

- If a' has been shown equivalent to/included in a'', it is discarded and all existing edges towards a' are redirected to a'' (in the inclusion case the edges of kind `*A` are relabelled as `*E`).
- If $a' \supset a''$, a'' and all its outgoing edges are discarded. All existing edges towards a'' are redirected to a', and the outgoing edges from a'' will be replaced by outgoing edges from a', since $\bigcup_{s\in a'} succ(s) \supseteq \bigcup_{s\in a'' succ(s)}$.

Typically, such schema cycles until there are no unexplored states, using states coming from the previous iteration as input to the next one. The operations which depend on the adopted formalisms are the calculation of the successors of a state, the evaluation of relationships

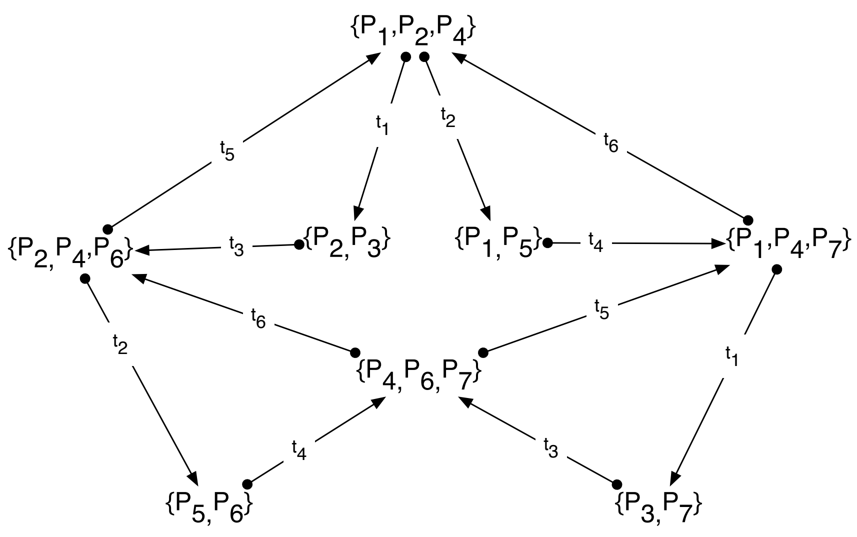


Figure 2.4: State space of the Petri net example shown in Figure 2.1.

between states (often the more computationally expensive operation), and the identification of connectivity conditions. The complexity of evaluating equivalence/inclusion relationships between abstract states can be alleviated by identifying any *syntactical* feature which defines a necessary condition for states' overlapping, e.g., the merely numerical distribution of tokens in a Coloured Petri net [81] marking. Examples of algorithms that could be rephrased according to the schema above are presented in [14] for TB nets, in [19] for time Petri nets, and in [39] for Well-Formed Coloured Petri nets. The construction of the reachability graph for classic Place/transition nets trivially falls in this category. For instance, the example shown in Fig. 2.1, from the initial marking $m_0 = \{P_1, P_2, P_4\}$ can evolve into two different states: either $m_1 = \{P_3\}$ or $m_2 = \{P_5\}$ respectively by firing transition t_1 or t_2. Neither m_1 nor m_2 are equal to m_0, hence we compute the successors of both states, and so forth, until the set of states to expand become empty. Fig. 2.4 shows the whole state space of the net. Note that all the edges are of type `AA` because states are concrete and we do not have to deal with inclusion relationships.

2.4 Petri Nets Coverability Graphs

The state space of a Petri net is potentially infinite. Nevertheless, many interesting verification problems are decidable on Petri nets. Among these decidable problems are the *coverability* problem (to which many safety verification problem can be reduced); the *boundedness* problem (is the number of reachable markings finite?); the *place boundedness* problem (is the maximal reachable number of tokens bounded for some place p?); the *semi-liveness* problem (is there a reachable marking in which some transition t is enabled?). In order to decide the aforementioned problems, one can use the coverability set (CS), which is a finite representation of some over-approximation of the reachable markings.

A marking M' covers a marking M, denoted by $M' \geq M$, if and only if $M'(p) \geq M(p)$ foreach place p. The notation $M' > M$ means that $M' \geq M \wedge M' \neq M$. If $M' \geq M$ and M_1 is reachable from M by firing the transition t ($M \xrightarrow{t} M_1$), then there exists a marking M_1' such that $M' \xrightarrow{t} M_1' \wedge M_1' \geq M_1$. Moreover $M_1' - M_1 = M' - M$, where $M_x - M_y$ denotes the function from places to integers such that $\forall p, (M_x - M_y)(p) = M_x(p) - M_y(p)$. This implies that if $M_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_1$ and $M_1 \geq M_0$ then $M_1 \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_2 \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_3 \ldots$ where $M_k = M_0 + k(M_1 - M_0)$. Moreover, if p is a place such that $M_1(p) > M_0(p)$, then the number of tokens in p grows without limit and the sequence of reachable markings is infinite.

If a Petri net has a finite number of places and transitions but an infinite number of reachable markings, then it can be proven that there exists an infinite execution with infinitely many different markings. Such an execution reaches a marking M and later on a marking M' shuck that $M' > M$.

Let us introduce the notion of ω-marking. An ω-marking is a function from the set of places to the set $(\mathbb{N} \cup \{\omega\})$ that associates a number of tokens to each place, where $\mathbb{N}$ is the set of natural numbers including 0 and ω is a symbol that means “any natural number”. The notion of coverage can be extended to ω-markings assuming that $\omega \geq \omega > c$, $\forall c \in \mathbb{N}$. An ω-marking M_ω denotes a set $[[M_\omega]]$ of ordinary markings such that:

- $\forall p \in P \; \forall M \in [[M_\omega]]$, if $M_\omega(p) \neq \omega$, then $M(p) = M_\omega(p)$.
- For every ordinary marking M such that $M \leq M_\omega$, $[[M_\omega]]$ contains an ordinary marking M' such that $M \leq M'$.

The set of ordinary markings $[[M_\omega]]$ is certainly infinite: there exist infinite markings M such that $M \leq M_\omega$ and for every M, $\exists M' \in [[M_\omega]] : M \leq M'$. An enabled transition in an ω-marking, and the result of its firing are defined likewise ordinary marking, except that a place marked with ω always contains enough tokens, and is marked with ω also after the transition occurrence. This implies that if M_ω contains at least one ω symbol, then $M_\omega \xrightarrow{t} M_\omega'$ represents an infinite number of occurrences of t from a marking in $[[M_\omega]]$ to a marking in $[[M_\omega']]$. A very simple algorithm for constructing the *cover ability graph* works

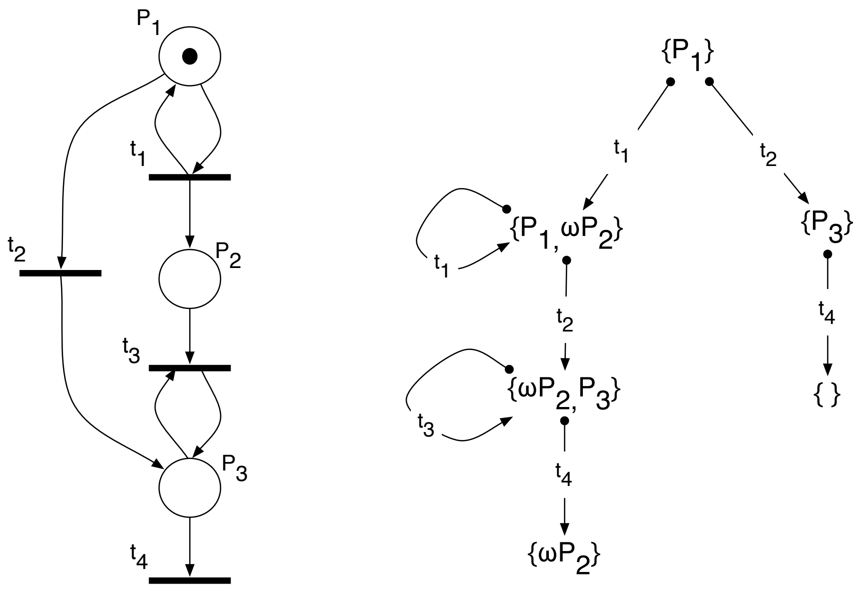


Figure 2.5: Petri net coverability graph example.

like the state space construction with the following exception: whenever a new reachable ω-marking M' is constructed, if M' covers and is reachable from an older ω-marking M, then M' is replaced with M_ω such that $M_\omega(p) = \omega$ if $M'(p) > M(p)$, $M_\omega(p) = M'(p)$ otherwise. In order to guarantee termination it is sufficient to compare each new ω-marking M' with all ω-markings along which M' was found. Figure 2.5 shows a simple Petri net example along with its cover ability graph.

Coverability graphs can be effectively used to detect unbounded places of a Petri net. Anyway the set represented by an ω-marking is not unique, thus coverability graphs cannot be used for checking the reachability of a marking. For example, regarding the model in Figure 2.5, if the weight of the arc from t_1 to P_2 were changed to 2, then the coverability graph would not change but the marking $\{P_1, P_2\}$ would become unreachable. That means there are simple properties that cannot be verified from coverability graphs. Anyway, they can be used for checking reachability of states where a given transition is enabled, this is sufficient for checking a large number of action-based safety properties. Another issue is that coverability graphs discard important information for verifying liveness properties. Despite those limitation such a technique represent an effective way to deal with infinite state spaces.

The classical algorithm to compute the coverability graphs is the "Karp&Miller" (K&M) tree [83]. Unfortunately the K&M tree is often huge and cannot be constructed in reasonable time, even for small nets, making it often useless in practice. Moreover, the definition of coverability graph makes it possible to obtain different version of this structure from the same Petri net model. An improvement of this K&M algorithm is the Minimal Coverability Tree (MCT) algorithm [54], which has been introduced twenty years ago, and implemented since then in several tools such as PEP [70]. Unfortunately, it can be shown that the MCT is not complete [62]: it might compute an under-approximation of the reachable markings. The MP algorithm [101] overcame this problem. It can be viewed as the MCT algorithm with a slightly more aggressive pruning strategy which ensures completeness. Experimental results show that this algorithm is a strong improvement over the K&M algorithm as it dramatically reduces the exploration tree.

2.5 The State Explosion Problem

State space methods aim at automatically analyzing the behavior of systems. Basically, they are based on the construction of the entire structure containing all reachable states and all transitions that the system can make among those states. The construction of the state space can be fully automated. Moreover, many verification and analysis questions can be answered by means of practical algorithms, given in input the state space of a system. Unfortunately, state space methods are very expensive, so that the belief was that state space methods would never work for the analysis of large-scale real world systems. In fact, the number of states reachable from any system of interest is huge. As an example, let us consider few rather simple systems:

- The system composed of n non-interacting processes, each with k possible local configurations, reaches k^n states.
- The classic dining philosophers systems with n philosophers, each with 4 states, reaches $3^n - 1$ states [108].
- The simple token ring protocol described in [69], reaches $9n2^{n-2}$ states, where n is the number of stations [108].

These examples show a common trend: the number of states reachable by a system, increases exponentially in the "size" of the system. The parameters that describe the system's size are: the number of processes (n in the above examples) and the number of per-process variables. The base of the exponentiation depends on the number of local configurations of each process, the number of values a variable can store, and some kind of “tightness” among different components. Such a “tightness” determines the ability of local states of components to influence the local states of other components.

It is easy to imagine how fast grows this number for models describing real world examples. It grows so fast that it seems to make state space methods unfeasible for analysis and verification of systems in practice. Anyway, the great power and advantages of such a methods motivated researchers to try to find ways of alleviating this problem. Many methods have been proposed that aim at reducing the number of states needed to be constructed in order to verify certain properties. The algorithms for constructing the reduced state space takes advantage of some details of the property to be verified in order to avoid the construction of the overall state space, if not needed. These advanced state space methods increased substantially the size of analyzable systems, while preserving many advantages of state space methods. Unfortunately, most state space reduction techniques disable some advantages. In fact, most of those techniques are able to perform only certain kind of analysis without losing the ability to reduce the number of states [108]. Because the information on states and transitions is somehow implicit, in a reduced state space, it is sometimes hard to extract knowledge and to answer verification questions. This leads to complicated and slow algorithms for certain verification questions. Thus methods based on reduced state spaces usually work well only for certain types of analysis questions, even if the reduction contains full information. The *symmetry* method [52], the *unfolding* method [95], and *BDD*s [30] are examples of reduced state space methods that preserve full information. Petri net coverability graphs, as discussed above, preserve incomplete information, and allow the verification of only a restricted class of properties. Anyway, the effectiveness of a reduced state space that contains full information on the behavior of a system, relies on some kind of regularity in such a behavior, otherwise the size of the reduction could be very similar to the explicit representation. For instance, the symmetry method relies on the assumption that the system is made up by several identical (or very similar) components [52].

Chapter 3

Dealing With Infinite-states Real-time Systems

This chapter introduces algorithms and related tools able to deal with infinite-states real-time systems. In particular, section 3.2 addresses the problem of constructing a finite representation of an infinite state space generated from Time-Basic (TB) Petri nets models. TB nets [66] (introduced in section 2.2) belong to the category of Petri nets in which system time constraints are expressed as numerical intervals associated to each transition, representing possible firing instants, computed since transition's enabling time. Tokens atomically produced by the firing of a transition are thereby associated to time-stamps with values ranging over a determined set. With respect to the well-known representative of this category, i.e., Time Petri nets [19], interval bounds in TB nets are linear functions of timestamps in the enabling marking, rather than simply numerical constants. TB nets thus represent a much more expressive formal model for real-time systems. The technique described in section 3.2 tries to overcome the major limitation of the existing analysis techniques and tools able to verify only time-bounded properties by inspecting a finite portion of the potentially infinite reachability-tree generated by a TB net model. Such a technique relies on a symbolic reachability graph algorithm, which is in turn based on a relative notion of time and a procedure verifying inclusion between symbolic states. A particular state normalization, able to recognize and eliminate timestamp symbols actually not influencing the model evolution (the *time anonymous* concept), permits in many cases to build a sort of *time coverage* finite graph. The symbolic graph construction has been automated by a tool-set written in Java. The output is a structure enriched with information on edges which might be exploited during property evaluation. The tool-set currently includes a module for the automatic verification of properties expressed as conditions on markings. As use case we will use the gas burner example (section A.1), that is widely used in literature as a representative of a small real system. A complete and formal description can be found in [100], and the corresponding TB net model was introduced in [15].

Section 3.3, introduces an algorithm (along with its implementation) able to compute coverability trees of real-time systems modeled with TB nets. This technique extends the previous one, further exploiting the time anonymous concept in order to deal with topologically unbounded nets. Despite coverability trees preserve incomplete information (section 2.5), such a technique gives us a means to decide several different important properties also for this formalism such as boundedness, and place-boundedness properties. Our proposal takes inspiration from the Monotone-Pruning (MP) algorithm introduced in [101], and extends it in order to deal with dense-time information associated with reachable symbolic states.

Different small TB net models will be used as running examples to explain in a rather informal way the essential points of both the symbolic graph and the coverability tree construction. Moreover, some relevant new core definitions are formally given.

3.1 State of the Art

Time dependent systems, *i.e.*, systems whose behavior and correctness depends on time, are important in the every day life. Formal verification of time dependent systems is an active research area since very long time [59], because the frequent use of such systems for critical applications increased the need of tools and techniques that guarantee high degree of correctness and reliability of the final product. Time Petri nets are very common formal models for the specification and the verification of systems where the explicit consideration of time is crucial. One of the main extensions of Petri nets with time are time Petri nets [96]. In this formalism, a transition can fire within a time interval and tokens, in the input places of the corresponding transitions, are meant to spend that time. Several variants of time Petri nets exist: time is either associated with places, with transitions or with arcs [27, 102]. Verification of real-time systems is complicated by the dense time model (time is considered in the domain $\mathbb{R}^+$). This raises the problem of handling an infinite number of states. In fact, the set of reachable states of time extension of Petri nets is generally infinite due to the infinite number of time successors a given state could have. The two main approaches, used to handle such a state space, are *region graphs* [4] and the *state class approach* [20]. Other refinements and improvements of these basic approaches were introduced in [19, 21, 61, 92, 24, 84]. The objective of these representations is to create a state space partition that groups concrete states into sets of states with similar behavior with respect to the properties to be verified. These sets of states must cover the entire state space and must be finite in order to ensure the termination of the verification process.

Despite many years of work on time extension of Petri nets, still few analysis techniques were proposed for Time Basic (TB) Petri nets [66]. With respect to the well-known time Petri nets, interval bounds in TB nets are functions of timestamps in the enabling marking, rather than simply numerical constants. Moreover, transitions can have either a *weak* or *strong* time semantics. Therefore, TB nets represent a much more expressive formal model for real-time systems. The reachability analysis of TB nets is still recognized as an open problem [86]. Available analysis techniques and tools (e.g., [86, 65]) are based on inspecting a finite portion of the potentially infinite reachability-tree generated by a TB net. Thus only time-bounded properties can be inferred from TB nets state-space exploration by using this kind of analyzers.

Another very useful, and well studied, advanced state space method for concurrent systems, is *coverability analysis*. Concerning the coverability analysis of classic P/T nets, Karp and Miller (K&M) introduced an algorithm for computing the *minimal coverability set* (MCS) [83]. This algorithm builds a finite tree representation of the (potentially infinite) reachability graph of the given unbounded P/T net. The K&M Algorithm has been also extended to other classes of well-structured transition systems [55, 56]. Anyway, the K&M Algorithm is not efficient in analyzing real-world examples and it often does not terminate in reasonable time. The MCT algorithm [54] introduces clever optimizations, but it has been proven that it is flawed [62]: it computes an incomplete *forward* reachability set (*i.e.*, all the markings reachable from the initial markings). In [62], the CoverProc algorithm, is proposed for the computation of the MCS of a Petri net. This algorithm follows a different approach and is not based on the K&M Algorithm. In [101], the MP algorithm is proposed. Experimental results show that the MP algorithm is a strong improvement over both the K&M and the CoverProc algorithms. Anyway, coverability analysis techniques for real-time systems remain rather unexplored. For time Petri nets, although the set of *backward* reachable states (*i.e.*, all the markings from which a final marking is reachable) is computable [1], the set of *forward* reachable states (*i.e.*, all the markings reachable from the initial markings) is in general not computable. Therefore any procedure for performing forward reachability analysis on time Petri nets is incomplete. In [2], an abstraction of the set of reachable markings of unbounded time Petri nets is proposed, but the termination of the forward analysis by means of this abstraction is not guaranteed. Coverability analysis techniques able to deal with TB nets unbounded models, have not been addressed, as far as we know.

3.2 Verification of Reachability Properties for Time Basic Petri Nets

The analysis technique presented in this section extends the capability of the existing analyzer for TB nets [16], which uniquely permits the verification of bounded invariance and response properties, through the inspection of a time-bounded symbolic reachability tree generated from a TB net.

The new technique aims at building a finite graph instead of an infinite tree for a wide category of TB nets. A combination of three complementary ideas is exploited. First, reachable symbolic states (i.e., infinite sets of ordinary reachable states) are compared to check subset relationships. Identifying subset relations between generated symbolic states is necessary for recognizing cyclic paths, but it is not enough in many situations. As time progresses, periodic occurrences of equivalent conditions may be unrecognizable simply due to their different offsets with respect to system's time zero. This observation leads us dealing with the second aspect. In the very common case a TB model contains no reference to *absolute times* (i.e., not as offset respect to enabling timestamps) in transition time functions, it is possible to remove any references to the "absolute zero" from symbolic states. This permits a periodic equivalent behavior to be recognized. The cost is a lossy information about state displacement along absolute time. We will discuss this aspects in section 3.2.5. Let us only point out that this kind of information could be recovered, if necessary, in a second step by retracing only the path(s) leading to the state of interest, or (at least partially) by combining the information on edges. The third key feature of the technique is the introduction of the *time anonymous* (*TA*) concept. This relates to the fact that in a symbolic state there may exist tokens whose timestamp values can be *forgotten*, as not influencing the evolution of a model. Several heuristics have been implemented, based on a mix of structural and state-dependent patterns, each characterizing one such situation. This enhances the ability of merging states, and permits facing situations where the presence of dead tokens could reintroduce a sort of *symbolic* absolute zero, nullifying the achievements at the previous points. Again, the cost to pay is a minor loss of information, as discussed later. There is some resemblance with the approach used in the construction of (topological) coverage graphs: the missing information is the exact timestamp of tokens instead of their exact number. *TA* recognition might be also exploited to introduce a topological notion of coverage for TB nets (section 3.2.8).

3.2.1 Basic notions

In order to understand the rationale behind the symbolic reachability graph construction technique for TB nets, we shall use once again the running example in Fig. 2.2. Let us only introduce a few basic notions used in the sequel, referring to [67] (where the symbolic reachability tree for TB nets is defined) for a full formalization.

Let $TS = \{T_i\}$, $i \geq 0$, be the set of time-stamp symbols. A *symbolic state* S is a pair $\langle M, C\rangle$, where $M : P \to Bag(TS)$, C is a (satisfiable) constraint formed by linear inequalities involving TS symbols occurring in M (so called symbolic marking).

Unless otherwise specified, we shall refer to a *normal form*: if k different TS symbols occur in M, they are $T_0, \ldots, T_{k-1}$, such that $\forall i : 0 \ldots k-2$, $C \Rightarrow T_i \leq T_{i+1}$.

An ordinary marking m is represented by $S : \langle M, C\rangle$ if and only if m is obtained from M by a numerical replacement $\sigma : TS \to \mathbb{R}^+$, σ being a solution of C. We say that S is contained in S' ($S \subseteq S'$) if and only if the corresponding represented ordinary markings are.

A mapping $en_s : {}^\bullet t \to TS$ is said a *symbolic tuple* of t. The notation (en_s, t) will be sometimes used. The *symbolic evaluation* of a time function f_t, denoted $f_t(en_s)$, is obtained by replacing each occurrence of $p \in {}^\bullet t$ in the formal expressions lb_t, ub_t, with $\tau = en_s(p)$.

According to a (monotonic) weak time semantics, (en_s, t) is said a *symbolic enabling* in S if $\forall p \in {}^\bullet t$ $en_s(p) \in M(p)$ and C': $C \wedge lb_t(en_s) \leq T_k \leq ub_t(en_s) \wedge T_{k-1} \leq T_k$ is satisfiable, i.e., there exists at least one numerical substitution (tuple) en for en_s that makes C satisfiable and $f_t(en)$ non empty. As already said the symbolic enabling condition is a bit more complex to take into account strong enablings: an example will be provided in Sect. 3.2.2.

The firing of a symbolic enabling (en_s, t) produces the new symbolic state $S' : \langle M', C'\rangle$, where M' is obtained from M by removing $en_s(p)$ from each place $p \in {}^\bullet t$, and putting the new symbol T_k in all places in $t^\bullet$, in full analogy with the ordinary firing rule. That is denoted $M[(en_s, t) > M'$. S' represents all the possible ordinary markings reachable from any marking represented by S, by means of any firing instance corresponding to (en_s, t).

3.2.2 Time-coverage graph construction

The time-coverage symbolic reachability graph (TRG) generated by the running example, composed by 14 symbolic states, is presented in Fig. 3.1.[1]

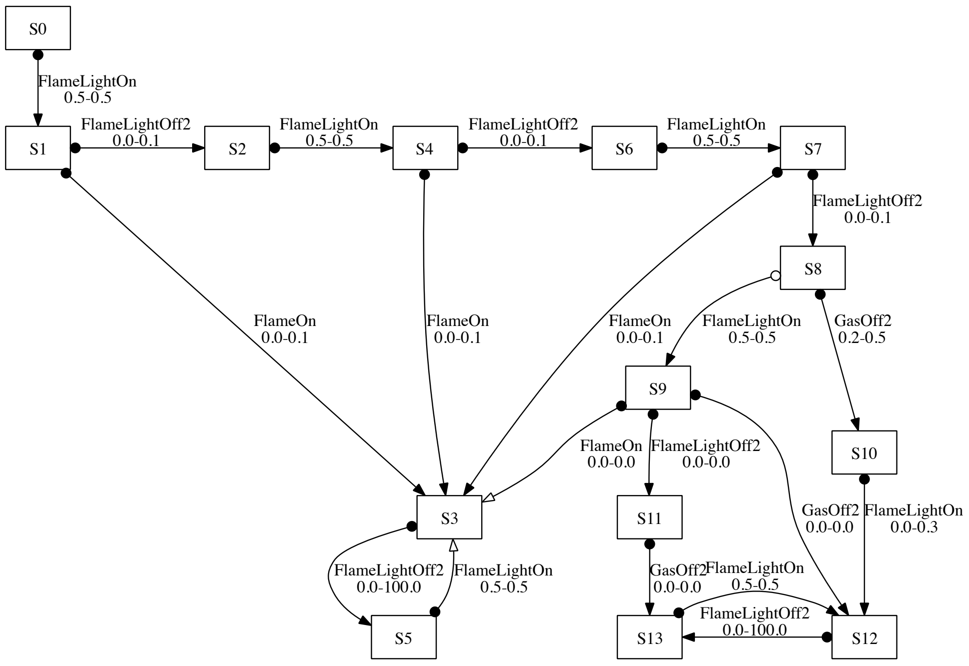


Figure 3.1: Sample reachability graph.

The adopted notation for states is: a square for symbolic states, a double square for symbolic states containing some deadlocks. Concerning edges (i.e., symbolic enablings), the format of head and tail specifies the kind of relation between source and target.

The normal case is black head and tail, e.g., from $S0$ to $S1$: considering any marking represented by $S0$ it is always possible to follow that edge and to reach all the markings represented by $S1$.

[1]This picture has been automatically obtained by using GraphViz visualization software [71] on the output generated from the tool-set.

Let us consider the symbolic state $S8$, formally described as follows:

$$\begin{aligned} M8 \quad &: \quad Gas\{T_1\}\ IGNITE\_PHASE\_S\{T_0\} \\ & \quad\quad Ignition\{TA\}\ NoFlame\{TA\} \\ C8 \quad &: \quad T_1 \geq T_0 + 1.5 \wedge T_1 \leq T_0 + 1.8 \end{aligned}$$

We can observe that, with respect to the original definition of symbolic state, a first extra time-stamp symbol is present, TA (time anonymous). This new symbol can occur only on the marking. Postponing an intuitive explanation of when and how symbol TA is introduced in a symbolic state representation, we can think of it as a token carrying on an unspecified time-stamp, which has been shown unessential for the computation of transition firing times.

The "candidates" for symbolic enabling in $S8$ are:

- $(\langle T_0\rangle, GasOff2)$
- $(\langle TA, T_1, TA\rangle, FlameLightOn)$.

Firing times are computed by (symbolically) evaluating transition time functions, as explained above. For *GasOff2* the (only) inferred firing time is $\{T_0 + 2\}$. Time function evaluation is slightly different for *FlameLightOn*, due to the occurrence of TA in the pre-set tuple. This symbol is *erased* during symbolic evaluation: $enab = max(\{TA, T_1, TA\}) \equiv max(\{T_1\}) = T_1$. The inferred firing time in this case is $\{T_1 + 0.5\}$.

Since both transitions have a strong semantics, there are two additional constraints specifying that the firing time of one cannot be greater than the (maximum) firing time of the other. They are $C_{\text{GO2}} : T_0 + 2 <= T_1 + 0.5$ and $C_{\text{FLO}} : T_1 + 0.5 <= T_0 + 2$, respectively.

Since both $C8 \wedge C_{\text{GO2}} \wedge T_2 = T_0 + 2$ and $C8 \wedge C_{\text{FLO}} \wedge T_2 = T_1 + 0.5$ are satisfiable, $(\langle T_0\rangle, GasOff2)$ and $(\langle TA, T_1, TA\rangle, FlameLightOn)$ are in fact symbolic enablings in $S8$. It is important to note that $C8 \Rightarrow C_{\text{GO2}} \wedge T_2 = T_0 + 2$, i.e., all the markings represented by $S8$ enable the transition *GasOff2*. Instead $C8 \not\Rightarrow C_{\text{FLO}} \wedge T_2 = T_1 + 0.5$, i.e., only a subset of the markings expressed by $S8$ enable the transition *FlameLightOn*. This is highlighted in the graph by the white tail of the edge from $S8$ to $S9$.

Consider now the firing of $(\langle T_0\rangle, GasOff2)$: it only consumes tokens. In such cases the symbolic firing rule slightly differs from the original one. A second special symbol, TL (Time Last), is introduced. TL can occur only on the constraint of a symbolic state and has an intuitive meaning: it stands for the last firing time of the TB net and it permits a correct interpretation of the model's time semantics.[2] The reached symbolic state $S10$ is:

$$\begin{aligned} M10 &: Gas\{T_1\}\ Ignition\{TA\}\ NoFlame\{TA\} \\ C10 &: C8 \wedge T_2 = T_0 + 2 \wedge TL = T_2 \end{aligned}$$

The normalization step eliminates symbols T_2 (the symbolic firing time) and T_0, as they occur only in $C10$, instead it leaves symbol TL. That results in (after a timestamp renaming):

$$\begin{aligned} M10 &: Gas\{T_0\}\ Ignition\{TA\}\ NoFlame\{TA\} \\ C10 &: TL \geq T_0 + 0.2 \wedge TL \leq T_0 + 0.5 \end{aligned}$$

Another circumstance that causes the introduction of TL symbol in a symbolic state representation is when the maximum timestamp symbol T_k is replaced with TA. The identification of a Time Anonymous in a given symbolic state is briefly introduced below and deeply treated in the next section.

The graph in Fig. 3.1 contains two looping paths: between states $S3$ and $S5$, and between $S12$ and $S13$ respectively. That happens because in the model represented by Fig. 2.2, no expected actions are activated after the system exits the *ignition phase* (e.g., closing the gas valve in the event of fail, or stopping ignition), so that an unbounded sequence of *FlameLightOff2*;*FlameLightOn* is possible.

The white head of the edge from $S5$ to $S3$ means that at least one of the ordinary markings represented by $S3$ is not reachable by following that edge. This happens when a newly built symbolic state is recognized to be strictly contained in an existing one. What permits recognizing inclusion between states in this specific case is the recognition of time anonymous timestamps. $S3$ is formally defined as:

$$\begin{aligned} M3 &: Gas\{TA\}\ BURN\_PHASE\_B\{TA\} \\ &\ Ignition\{T_0\}\ Flame\{T_1\} \\ C3 &: T_1 \geq T_0 \wedge T_1 \leq T_0 + 0.1 \end{aligned}$$

[2]In this paper, when TL is left implicit, it coincides with the "last" generated timestamp T_k.

Without using *TA*s, its original definition ($S3'$) would be:

$$\begin{aligned} M3' &: Gas\{T_0\}\ BURN\_PHASE\_B\{T_1\} \\ &\quad Ignition\{T_0\}\ Flame\{T_1\} \\ C3' &: T_1 \geq T_0 \wedge T_1 \leq T_0 + 0.1 \end{aligned}$$

Let us figure out what would be the model evolution from $S3'$, without introducing *TA*. After the firing sequence *FlameLightOff2*;*FlameLightOn*[3] a state $S3''$ would be reached, defined in turn as:

$$\begin{aligned} M3'' &: Gas\{T_1\}\ BURN\_PHASE\_B\{T_0\} \\ &\quad Ignition\{T_1\}\ Flame\{T_1\} \\ C3'' &: T_1 \geq T_0 + 0.5 \wedge T_1 \leq T_0 + 100.5 \end{aligned}$$

Since $S3'' \not\subseteq S3'$ and $S3' \not\subseteq S3''$, there is no possibility to merge them and in fact the analysis tool would produce an infinite firing sequence.

Back to $S3$, we note it corresponds to $S3'$ but for holding *TA* symbols in places $BURN\_PHASE\_B$ and Gas instead of T_1 and T_0, respectively. Token T_1 in $BURN\_PHASE\_B$ however is not (and will never be) involved in any symbolic enabling because $BURN\_PHASE\_B$ has an empty postset (Heuristic 3.1.0 in the following section), so it is immediately marked as *TA*. Token T_0 in Gas instead is in the preset of transitions *FlameLightOn* and *FlameLightOff2*. As for *FlameLightOn*, the tokens in place $Ignition$ and in place Gas carry on the same timestamp, so either of them is enough to correctly evaluate transition's time function. As for *FlameLightOff2*, the token in place Gas carries on redundant information due to the simultaneous presence of T_1 in $Flame$, that superseded it (Heuristic 3.2.1).

$S3''$ seems really different from $S3$, but nearly the same heuristics permits us to replace $T_0 : BURN\_PHASE\_B$ ($T_i : p$ denotes the occurrence of a timestamp in a place) and $T_1 : Gas$ with *TA*s. That eliminates all the occurrences of T_0 from the marking. After timestamp renaming, we obtain the normal form:

$$\begin{aligned} M3'' &: Gas\{TA\}\ BURN\_PHASE\_B\{TA\} \\ &\quad Ignition\{T_0\}\ Flame\{T_0\} \\ C3'' &: true \end{aligned}$$

[3]We omit in this description symbolic enablings, the TB net being safe.

However there is still a difference with respect to $S3$: places *Ignition* and *Flame* hold the same timestamp, but this boils down to a condition already represented by $S3$ ($T_1 = T_0 \Rightarrow C3$), so $S3''$ is recognized as a state contained in $S3$.

Notice that the other cycle on the graph, between $S12$ and $S13$, is due to the adoption of a relative notion of time, i.e., it does not depend on the introduced *TA* concept.

An important setting of the legacy tool [65] was the *time limit*, a positive interval time that guaranteed the finiteness of the symbolic reachability tree of a TB net. Upon elimination of absolute time references it has been substituted by a *relative time limit*. This positive interval specifies the maximum admissible distance between different timestamps in a state, and allows one to deal with possibly infinite reachability graph. The tool-set checks whether a symbolic state includes any ordinary states for which the distance between TL and T_0 (the oldest meaningful timestamp) exceeds the time limit, marking that state as *not to be expanded*. The rationale behind is that reaching such a user defined limit might be a symptom of the presence of unrecognized "dead tokens", reintroducing absolute time references. If we analyzed the running example disabling *TA* recognition, the resulting graph would be infinite, unless a time limit is set. For example, setting this limit to 3 (time units), 25 symbolic states would be generated: 13 already included in the presented graph, the others corresponding to a partial unrolling of the loop between $S3$ and $S5$.

The output generated by the tool-set associates a couple of numerical values to edges of the graph, corresponding to the minimum and maximum time distances from the source node to the target node. This permits us to partially recover time relations between nodes that were lost due to the removal of absolute times references from constraints. In the following section we will show how to exploit them.

3.2.3 Time Anonymous

The notion of time anonymous relies on the fact that in a symbolic state there may exist tokens whose timestamp values can be *forgotten*, as not influencing the evolution of the model. The adopted symbol to denote a time anonymous timestamp is *TA*, and it represents an undefined time value in the past chosen between the initial time and the time limit *TL*. The *TA replacement* (formally defined in the next section) allow us to build, in many cases, a finite reachability graph. In fact, the presence of "dead" tokens (i.e., those tokens that cannot be consumed) during the evolution of a model. by firing transitions, reintroduce a sort of *absolute time* that would prevent the identification of equality/inclusion relationships among states.

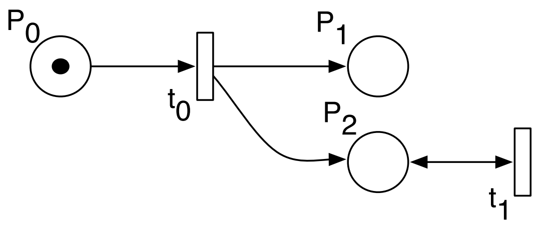


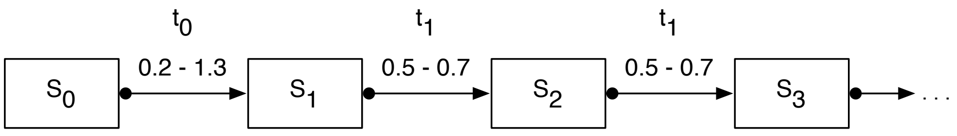
Figure 3.2: Simple TB net example generating a “dead” token.

Initial marking	$P_0\{T_0\}$
Initial constraint	$0 \le T_0 \le 1$
t_0	$[enab + 0.2, enab + 0.3]$
t_1	$[enab + 0.5, enab + 0.7]$

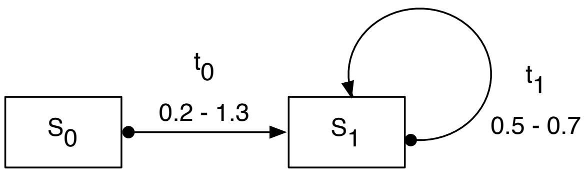
As a simple example, let us consider the model described in Fig. 3.2. Transition t_0 is enabled in the time lapse $[T_0 + 0.2, T_0 + 0.3]$. Its firing produces two new tokens, respectively into P_1 and P_2 with a timestamp T_1 representing a value chosen in such a time interval. This new configuration enables t_1 which can fire infinitely many times, by consuming and immediately after creating a token in P_2, each time with a new timestamp. Although the erasure of absolute times, the presence of a “dead” token in P_2, creates a sort of time marker which would make the reachability graph infinite, as we can see in Fig. 3.3a.

Figure 3.3: Infinite (a) and finite (b) representations of the reachability graphs extracted from the model shown in Fig. 3.2.

(a) Reachability graph without TA replacement.



(b) Reachability graph with TA replacement.



After the initial state S_0, reachable states are all equal in terms of symbolic marking: $P_1\{T_0\}P_2\{T_1\}$ but they have different constraints:

- $C_{S_1} = 0.2 \leq T_0 \leq 1.3 \wedge T_0 + 0.5 \leq T_1 \leq T_0 + 0.7$
- $C_{S_2} = 0.2 \leq T_0 \leq 1.3 \wedge T_0 + 1.0 \leq T_1 \leq T_0 + 1.4$
- $C_{S_3} = 0.2 \leq T_0 \leq 1.3 \wedge T_0 + 1.5 \leq T_1 \leq T_0 + 2.1$

and so forth, departing T_1 from T_0 further and further. Anyway, it is worth noting that T_0 does not influence the evolution of the model, thus we can forget about this value replacing it with an anonymous timestamp TA. The TA replacement cause the erasure of T_0 from constraints enabling the identification of equality relationships among states. In fact, a TA timestamp does not have any relationships with other symbolic values because it represents any time value in the past. Therefore, all the states after the initial one, would have the same constraint: $C_{S_1} = TRUE$. The finite reachability graph, resulting from the analysis of Fig. 3.2, using TA replacements, is shown in Fig. 3.3b.

We identified three different typologies of tokens disclosing a negligible symbolic time:

- The first category is composed of "dead" tokens. A token t_k is dead if belongs to a place with an empty postset. Therefore such a token will be never consumed by firing transitions. It is possible to statically identify places that may contain dead tokens.
- The second category contains all tokens t_k such that t_k belongs to a place p with a non empty postset, and t_k cannot be consumed by firing transitions. I.e. foreach $t \in p^{\bullet}$, any symbolic tuple (en_s, t), such that $en_s(p) = t_k$ is not an symbolic enabling. It is not possible to statically evaluate places containing such a tokens.
- This latter category regards all tokens t_k such that t_k can be consumed by a firing transition, but its firing time is not evaluated in terms of the timestamp associated with t_k. As the previous category, we must search for such a tokens dynamically, during the graph construction.

It is worth noting that, a symbolic enabling (en_s, t) such that $lb_t(en_s) = TA$ makes the lower bound $lb_t(en_s)$ equals to TL, in fact a TA lower bound means that TL exceeds the minimum enabling time. Anyway, in case the preset of a transition t contains only "TA tokens", t cannot fire because both the lower bound and the upper bound of t_f would be any time value in the past, thus we cannot determine whether it represents an empty set. The reason of a TA replacement of all tokens belonging to ${}^{\bullet}t$ could be that foreach symbolic tuple (en_s, t), $TL > ub_t(en_s)$. Thus, if such a tokens does not contribute to the evaluation of possible firing times of other transitions, we can forget about all their symbolic times.

The next section introduces a formal definition of a “TA replacement” and all the adopted heuristics in order to find time anonymous timestamps during the graph building.

3.2.4 Formal Definitions

Let us formalize some core concepts previously outlined, focusing in particular on *TA* and state inclusion. For the sake of readability, definitions involving transitions refer to the weak semantics.

Definition 3.2.1 (symbolic state) A symbolic state S is a pair $\langle M, C\rangle$, where M is a function $P \to \mathbf{Bag}(TS \cup \{\mathrm{TA}\})$, and C is a (satisfiable) linear constraint defined on $TS_{\mathrm{M}} \cup \{\mathrm{TL}\}$, $TS_{\mathrm{M}} \subset TS$ being the finite set of symbols T_i occurring on M, such that $\forall T_i \in TS_{\mathrm{M}}$, $C \Rightarrow TL \geq T_i$.

Definition 3.2.2 (well-defined erasure) Let g_t be the formal expression of a linear function. The *erasure* of a set of symbols $E \subset {}^{\bullet}t$ from g_t, denoted $g_{t[\neg E]}$, is well-defined if it doesn’t violate the ariety of any operators occurring in g_t.

Consider for instance t, s.t. ${}^{\bullet}t = \{p_1, p_2\}$, and $f_t : [max(\{p1, p2\}), p2 + 0.5]$, where, $max : 2^{\mathbb{R}^+} \setminus \emptyset \to \mathbb{R}^+$, $+ : \mathbb{R}^+, \mathbb{R}^+ \to \mathbb{R}^+$. Then, the erasure $f_{t[\neg\{p_1\}]}$ is well-defined and results in $[p2, p2 + 0.5]$, instead $f_{t[\neg\{p_2\}]}$ is not well-defined.

A symbolic instance of t is a mapping $en_s : {}^{\bullet}t \to TS \cup \{TA\}$.
Let $en_s^{-1}(\tau) = \{p\}$, $en(p) = \tau$.

Definition 3.2.3 (symbolic enabling) (en_s, t) is said a symbolic enabling in $S = \langle M, C\rangle$ if and only if:

i $\forall p \in {}^{\bullet}t$, $en_s(p) \in M(p)$

ii $f_{t[\neg en_s^{-1}(\mathrm{TA})]}$ is well-defined

iii $C \wedge lb_{t[\neg en_s^{-1}(\mathrm{TA})]}(en_s) \leq ub_{t[\neg en_s^{-1}(\mathrm{TA})]}(en_s)$ is satisfiable

Let $C \setminus X$ denotes the constraint obtained by eliminating variable X from C, in such a way that the solutions of $C \setminus X$ are “projections” of the solutions of C.

Definition 3.2.4 (symbolic firing) Let (en_s, t) be a symbolic enabling in $S = \langle M, C\rangle$, $k = |TS_{\mathrm{M}}|$. The firing of (en_s, t) produces the new symbolic state $S' : \langle M', C'\rangle$, where

- $\forall p \in {}^\bullet t \setminus t^\bullet$, $M'(p) = M(p) - en_s(p)$
- $\forall p \in t^\bullet \setminus {}^\bullet t$, $M'(p) = M(p) + T_k$
- $\forall p \in t^\bullet \cap {}^\bullet t$, $M'(p) = M(p) - en_s(p) + T_k$
- for all remaining places, $M'(p) = M(p)$
- $C' = C \backslash TL \wedge lb_{t_{[\neg en_s^{-1}(\mathrm{TA})]}}(en_s) \leq T_k \wedge T_k \leq ub_{t_{[\neg en_s^{-1}(\mathrm{TA})]}}(en_s) \wedge T_k \geq T_{k-1} \wedge TL = T_k$

C' may contain some symbols T_i that have been withdrawn from M'. After eliminating redundant variables, and (possibly) renaming left symbols, the reached state meets definition 3.2.1 and is in normal form.

Let $\mathbf{R}(S)$ be the set of symbolic states reachable from S

Definition 3.2.5 (valid *TA*-replacement) Given a state S, a timestamp occurrence $T_i : p$ is replaceable with $TA : p$ if and only if for each $S' = \langle M', C' \rangle \in \mathbf{R}(S)$ in which token $T_i : p$ is left (modulo timestamp renaming), for each symbolic enabling (en_s, t) in S' s.t. $en_s(p) = T_i$, $f_{t_{[\neg\{p\}]}}$ is a well-defined erasure and

$$C' \wedge max(\{TL, lb_t(en_s)\}) \leq ub_t(en_s) \Leftrightarrow C' \wedge max(\{TL, lb_{t_{[\neg\{p\}]}}(en_s)\}) \leq ub_{t_{[\neg\{p\}]}}(en_s)$$

The new semantics of a symbolic state is provided by the following coverage notion.

Definition 3.2.6 (symbolic state inclusion) Let $S = \langle M, C \rangle$ be a symbolic state. An ordinary marking m is included in S if and only if it corresponds to a numerical substitution σ of symbols occurring in M, s.t. σ satisfies C and for each ordinary enabling en of t in m, for each symbolic tuple (en_s, t) in S s.t. en is a numerical substitution of en_s,

- $lb_{t_{[\neg en_s^{-1}(\mathrm{TA})]}}$, $ub_{t_{[\neg en_s^{-1}(\mathrm{TA})]}}$ are well defined
- $lb_{t_{[\neg en_s^{-1}(\mathrm{TA})]}}(en) = lb_t(en) \wedge ub_{t_{[\neg en_s^{-1}(\mathrm{TA})]}}(en) = ub_t(en)$

The next lemma sets the relationship between ordinary and symbolic instances (state transitions).

Lemma 3.2.7 Let m be included in S. If $m[(en, \tau) > m'$, then there exists a symbolic enabling en_s, s.t. en is a numerical substitution of en_s, $S[(en_s, t) > S'$ and m' is included in S'

Let us finally report all the heuristics used by the algorithm to identify the TA replacements commented in the previous sections. The idea behind the eleven heuristics is to find situations during the evolution of the model, where timestamp associated with tokens can be forgotten because they do not contribute to the evaluation of any possible firing time of any firing transition. They identify, precisely speaking, a valid replacement of a timestamp occurrence $T_i : p$ with $TA : p$, in $S = \langle M, C \rangle$, according to definition 3.2.5. At least one of the following heuristic, must be verified foreach $t \in p^\bullet$. Note that if $p^\bullet = \emptyset$ (Heuristic 3.1.0), this condition is trivially true.

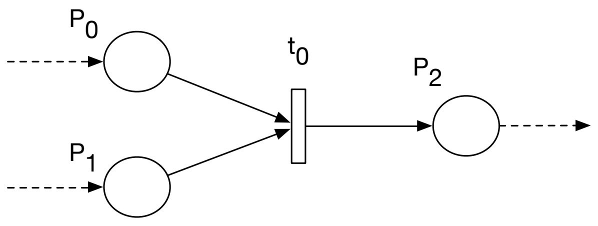


Figure 3.4: Simple excerpt of a TB net model used for TA heuristics examples.

Heuristic 3.2.1 $\forall p' \in {}^\bullet t,\ M(p') \neq \emptyset$

$\wedge\ f_t$ is in the form $[enab + c, enab + c']$

$\wedge\ \exists p' \in {}^\bullet t\ (\forall T_j \in M(p')\ C \Rightarrow T_j \geq T_i)$

All places belonging to ${}^\bullet t$ are marked, f_t is in the form $[enab + c, enab + c']$, but there exist another place containing only newer tokens. Thus tokens belonging to p won't be used to compute the enabling time.

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[enab + 1.0, enab + 2.0]$, the symbolic marking $M = P_0\{T0, T0\}P_1\{T1, T2\}$ and the symbolic constraint $C = T2 > T1 \wedge T1 > T0$. The enabling time of t_0 is computed using either $T1$ or $T2$ because they are both greater than $T0$. Thus all the tokens in place P_0 are recognized as TAs.

Heuristic 3.2.2 $\forall p' \in {}^\bullet t,\ M(p') \neq \emptyset$

$\wedge\ f_t$ does not contain p

$\wedge\ f_t$ does not contain $enab$

All places belonging to $^\bullet t$ are marked, but p will not be used to compute possible firing times of f because f_t does not contain both the variable p and $enab$.

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[P_2 + 1.0, P_2 + 3.0]$, the symbolic marking $M = P_0\{T0\}P_1\{T1\}$ and the symbolic constraint $C = T1 \geq T0$. The enabling time interval of t_0 is $[T1 + 1.0, T1 + 3.0]$, thus the token in place P_0 are recognized as TAs.

Heuristic 3.2.3 $\forall p' \in {}^\bullet t,\ M(p') \neq \emptyset$
$\wedge$ f_t is in the form $[max(\ldots) + c, max(\ldots) + c']$
$\wedge\ \forall(en_s, t)$ symbolic enabling, $lb_{t[\neg\{p\}]}(en_s) = lb_t(en_s) \wedge ub_{t[\neg\{p\}]}(en_s) = ub_t(en_s)$

All places belonging to $^\bullet t$ are marked, f_t is in the form $[max(\ldots) + c, max(\ldots) + c']$, but foreach enabling tuple en_s, $f_t(en_s)$ equals $f_t[\neg\{p\}](en_s)$ (well defined erasure). Thus neither $lb_t(en_s)$ nor $ub_t(en_s)$ refers to T_i.

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[max(P_0 + 1.5, P_1 + 1.0), max(P_0 + 2.5, P_1 + 2.0)]$, the symbolic marking $M = P_0\{T0, T0\}P_1\{T1\}$ and the symbolic constraint $C = T1 > T0 + 1.0$. According to C, the minimum firing time is dominated by $T_1 + 1.0$, while the maximum firing time is dominated by $T_1 + 2.0$, thus all the tokens in place P_0 are recognized as TAs. In fact, the evaluation of neither lb_t nor ub_t refers to $T0$.

Heuristic 3.2.4 $\forall p' \in {}^\bullet t,\ M(p') \neq \emptyset$
$\wedge\ \forall(en_s, t)$ symbolic enabling, $C \Rightarrow (TL > ub_t(en_s) \wedge TL \geq lb_t(en_s))$

All places belonging to $^\bullet t$ are marked, but t is not enabled $(TL > ub_t(en_s))$ and tokens in p won't be used to compute the lower bound of f_t even if t would be re-enabled by other tokens $(TL \geq lb_t(en_s)))$.

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[P_0 + 1.0, P_1 + 2.0]$, the symbolic marking $M = P_0\{T0\}P_1\{T1\}$ and the symbolic constraint $C = T1 > T0 + 2.0 \wedge TL = T1 + 2.5$. According to C, transition t_0 is disabled, because the variable TL is greater than the maximum firing time. Moreover, in case of a new enabling, $T0$ won't be used to compute lb_t because it is lesser than TL. Thus $T0$ is recognized as TA.

Heuristic 3.2.5 $\forall p' \in {}^\bullet t,\ M(p') \neq \emptyset$

$\wedge\ \forall (en_s, t)$ symbolic enabling, $C \Rightarrow (lb_t(en_s) > ub_t(en_s) \wedge (TL \geq lb_t(en_s) \vee lb_{t[\neg p]}(en_s) = lb_t(en_s)))$

All places belonging to ${}^\bullet t$ are marked, but t is not enabled ($lb_t(en_s) > ub_t(en_s)$) and tokens in p won't be used to compute the lower bound of f_t even if t would be re-enabled by other tokens, in fact $TL \geq lb_t(en_s)$) or p does not contribute to the evaluation of $lb_t(en_s)$.

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[enab, P_0 + 1.0]$, the symbolic marking $M = P_0\{T0\}P_1\{T1\}$ and the symbolic constraint $C = T1 > T0 + 1 \wedge TL = T1 + 2.0$. According to C, transition t_0 is disabled, because the maximum firing time is greater than the minimum firing time. Moreover, in case of a new enabling, $T0$ won't be used to compute lb_t, thus $T0$ is recognized as TA.

Heuristic 3.2.6 $\exists p' \in {}^\bullet t :\ M(p') = \emptyset$

$\wedge\ f_t$ does not contain p

t is disabled in S and p does not contribute to the evaluation of f_t foreach possible future symbolic enabling.

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[enab + 1.0, enab + 1.5]$, the symbolic marking $M = P_0\{T0\}$ and the symbolic constraint $C = TL = T0$. According to M, transition t_0 is disabled and in case of a new enabling, $T0$ won't be used to compute both lb_t and ub_t, because all new tokens in the enabling would have a timestamp greater than $T0$. Thus it is recognized as TA.

Heuristic 3.2.7 $\exists p' \in {}^\bullet t :\ M(p') = \emptyset$

$\wedge\ lb_t$ contains p

$\wedge\ ub_t$ does not contain p

$\wedge\ \forall T_x \geq TL\ \forall en \in Bag(TS)\ s.t.\ T_x \in en$ if (en, t) is a symbolic enabling then $(C \wedge T_x \geq TL) \Rightarrow TL \geq lb_t(en)$

t is disabled in S, ub_t does not contain the variable p, and foreach possible future symbolic enabling (en, t), the lower bound $lb_t(en)$ will be lesser or equal to TL.

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[P_1, enab]$, the symbolic marking $M = P_1\{T0\}$ and the symbolic constraint $C = TL = T0$.

According to M, transition t_0 is disabled and in case of a new enabling, $T0$ won't be used to compute lb_t, because TL would be greater than lb_t. Thus $T0$ is recognized as TA.

Heuristic 3.2.8 $\exists p' \in {}^{\bullet}t:\ M(p') = \emptyset$

$\wedge$ f_t is in the form $[max(\ldots) + c, max(\ldots) + c']$

$\wedge\ \forall T_x \geq TL\ \forall en \in Bag(TS)\ s.t.\ T_x \in en$ if (en, t) is a symbolic enabling then $lb_{t[\neg\{p\}]}(en) = lb_t(en) \wedge ub_{t[\neg\{p\}]}(en) = ub_t(en)$

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[max(P_0 + 1.0, P_1 + 1.5), max(P_0 + 2.0, P_1 + 2.5)]$, the symbolic marking $M = P_0\{T0\}$ and the symbolic constraint $C = TL = T0$. According to M, transition t_0 is disabled and in case of a new enabling, $T0$ won't be used to compute both lb_t and ub_t, because any of the new tokens would have a timestamp greater than $T0$. Since both lb_t and ub_t would be dominated by the second part containing the place P_1, $T0$ is recognized as TA.

Heuristic 3.2.9 $\exists p' \in {}^{\bullet}t:\ M(p') = \emptyset$

$\wedge\ \forall T_x \geq TL\ \forall en \in Bag(TS)\ s.t.\ T_x \in en$ if (en, t) is a symbolic enabling then $(C \wedge T_x \geq TL) \Rightarrow (TL > ub_t(en) \wedge TL \geq lb_t(en))$

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[P_1 + 1.0, P_1 + 2.0]$, the symbolic marking $M = P_1\{T0\}$ and the symbolic constraint $C = TL > T0 + 2.0$. According to M, transition t_0 is disabled and in case of a new enabling involving $T0$, TL would be greater than both lb_t and ub_t. Therefore, $T0$ is recognized as TA.

Heuristic 3.2.10 $\exists p' \in {}^{\bullet}t:\ M(p') = \emptyset$

$\wedge\ \forall T_x \geq TL\ \forall en \in Bag(TS)\ s.t.\ T_x \in en$ if (en, t) is a symbolic enabling then $(C \wedge T_x \geq TL) \Rightarrow (lb_t(en) > ub_t(en) \wedge TL \geq lb_t(en))$

As an example, consider the model depicted by Fig. 3.4 having the following temporal function $t_0[P_0 + 1.0, enab]$, the symbolic marking $M = P_0\{T0\}$ and the symbolic constraint $C = TL > T0 + 2.0$. According to M, transition t_0 is disabled and in case of a new enabling involving $T0$, ub_t would be greater than lb_t. Therefore, $T0$ is recognized as TA.

Heuristics 3.2.7, 3.2.8, 3.2.9 are respectively conceptually similar to 3.2.3, 3.2.4, 3.2.5 except they refer to future symbolic enablings, being t disabled within S.

Heuristic 3.2.11 Given a place p' and a symbolic tuple en_s, let $\phi_{TA}(en_s, p')$ be a new symbolic tuple such that:

$$\phi_{TA}(en_s, p')(p) = \begin{cases} en_s(p) & \text{if } p = p' \\ TA & \text{otherwise} \end{cases}$$

$\forall (en_s, t)$ symbolic enabling,

$C \Rightarrow (TL > ub_t(\phi_{TA}(en_s, p)) \wedge TL \geq lb_t(\phi_{TA}(en_s, p')))$

This heuristic assesses whether the symbolic time T_i influences the evaluation $f_t(en_s)$. To this end, we consider T_i as the last produced token by replacing each timestamp of en_s, except T_i, with a TA. If T_i does not contribute to evaluate $f_t(en_s)$, even if this condition holds, we can replace it with a TA timestamp.

3.2.5 Property Evaluation

The symbolic (time coverage) reachability graph contains several exploitable information.

The tool recognizes deadlocks even if they are topologically hidden by the presence of outgoing edges. In fact if all the outgoing edges have a white tail, it is still possible that a proper subset of the corresponding symbolic state is composed by deadlock marking. In the running example however no deadlock marking is reachable.

Disregarding time specification (i.e., considering only the number of tokens distributed over places), the graph nodes exactly identify all the reachable (topological) markings: if a marking matches a symbolic node then there exists at least one path from the initial state to such a marking, conversely if a marking matches no symbolic nodes, it is not reachable. It is thereby possible to verify P-invariants from a specified marking. In case of finite graph, it is possible to answer questions about maximum (minimum) number of tokens in some (combinations) of places.

In general, due to TA introduction, the set of ordinary markings included (Definition 3.2.6) by the states of the symbolic graph built from a TB net is a superset of the reachable ordinary markings of the TB net. Given a symbolic state $S = \langle M, C \rangle$, each numerical substitution of $\{T_i\}$ symbols occurring in M and satisfying C corresponds to the projection of reachable ordinary states. If we are interested in checking timing relations between token's timestamps on the states of the graph we can get three different answers upon graph inspection: a positive one (e.g., there exists a node that satisfies the condition), a negative one (e.g., no nodes satisfy the condition), or a possibly positive. For example, if we are looking for a state where a token in place *Flame* carries on a timestamp greater than the one in place *IGNITION_PHASE_S*, state $S9$ provides us with a positive answer. Instead, if we are checking whether places *Gas* and *Ignition* can ever hold the same timestamp the answer is may be (the presence of *TA* in either places *covers* that condition).

As for timing relations between token's timestamps in different markings, or between firing times in a transition firing sequence, the symbolic graph permits identifying critical paths by combining the information on edges. In particular, conservative bounds can be established. In the case they are not enough to exclude incorrect timing behaviors, it is possible to carry out a more accurate analysis by rebuilding a portion of the graph, retracing some critical paths and reintroducing absolute time references. For example, looking at the time information on edges, it is possible to establish that state $S10$ is not reachable from $S0$ in less than 1.7 time units. We cannot directly infer that $S10$ is reachable in exactly 1.7 time units.

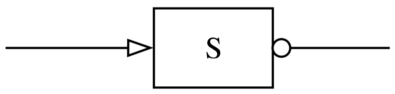


Figure 3.5: Critical case for path feasibility.

Concerning feasibility of firing sequences (Lemma 3.2.7), the symbolic graph expresses all the possibilities (an ordinary firing sequence is matched by any firing sequence on the graph). A possible critical situation is a white-arrow edge (meaning that we reach only a subset of the target state) is followed by a white-tail edge as shown in Fig. 3.5 (meaning that the transition is enabled only in a subset of the ordinary states represented by the node). In this case there is still the possibility that this path actually is not feasible. Also such critical paths could be retraced. Let us stress (back to the reachability problem) that by construction, for every node on the graph there exists a path from the initial state to such a node formed exclusively by black-arrow edges.

The available tool's evaluation component is still very simple, its integration with some existing model checking engines is currently under investigation. However it already permits examining the input graph looking for interesting properties on topological definition of markings:

- existence of a state with a marking satisfying a constraint (i.e., a boolean combination of condition on the number of tokens in places)
- maximum (minimum) value of an expression involving the number of tokens in places (possibly restricting the evaluation to markings satisfying a given constraint)

3.2.6 Tool Architecture

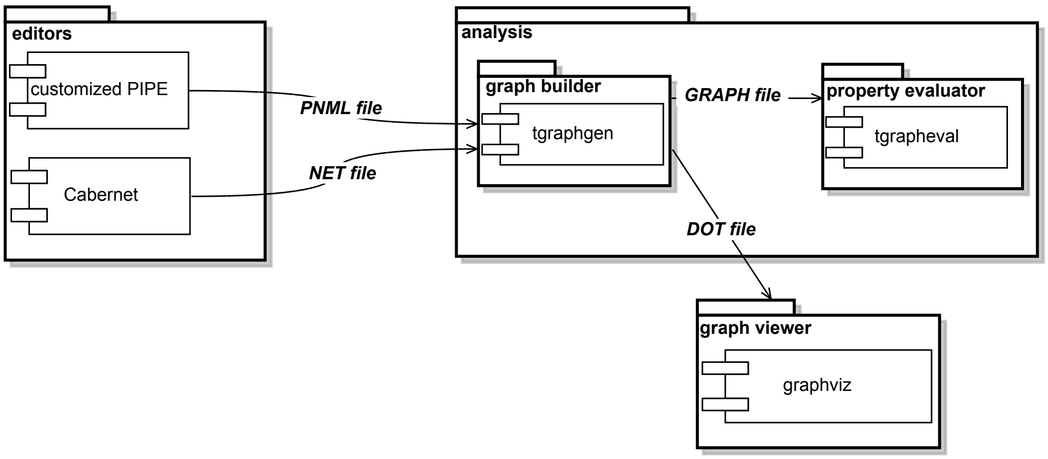


Figure 3.6: Reference architecture.

The analysis technique described in this paper has been implemented as a command line tool written in Java called Graphgen. The tool architecture depicted in Fig. 3.6 presents the various components that communicate by means of files. The *tgraphgen* module receives as input a Time Basic Petri net (either in the legacy file format used by the Cabernet tool, or in a PNML format generated, for example, by a customized version of PIPE2 open source tool[48]. It generates as outputs the graph in binary format (used by the property verification module *tgrapheval*), and in an annotated DOT text format (used by the GraphViz tool). The tool is also integrated as an analysis module in the customized PIPE2 open source tool. That will permit accessing all the functions by means of menu, and exploiting in an integrated environment consolidated structural analysis algorithms for the verification of the untimed part of TB nets (e.g., P/T nets invariant analysis). Both the command line tool and the customized version of PIPE2 are available for download at `http://camilli.di.unimi.it/graphgen`, together with a brief user guide and some running examples.

3.2.7 Use Case and Comparison with other tools

In order to make a comparison with the available analysis techniques and tools for TB nets, we consider now the complete gas burner example analyzed in [15], also reported in Fig. A.4) for completeness.

The main critical parameter of the system was identified in the concentration value of unburned gas. With the old analyzers it was only possible to do an approximate analysis, by verifying the safety requirement within a fixed time threshold [15], or by empirically guiding the construction of a portion of the reachability tree looking for a state invalidating the property [32]. These techniques were only able to verify the unsatisfiability of the time bounded safety property by ending the construction of the tree after reaching a state with a concentration exceeding a critical value (i.e., according the specification, one second of unburned gas). A significant improvement is that our technique computes the graph representing the complete behavior of the system, and thus for example permits calculating the actual concentration upper bound.

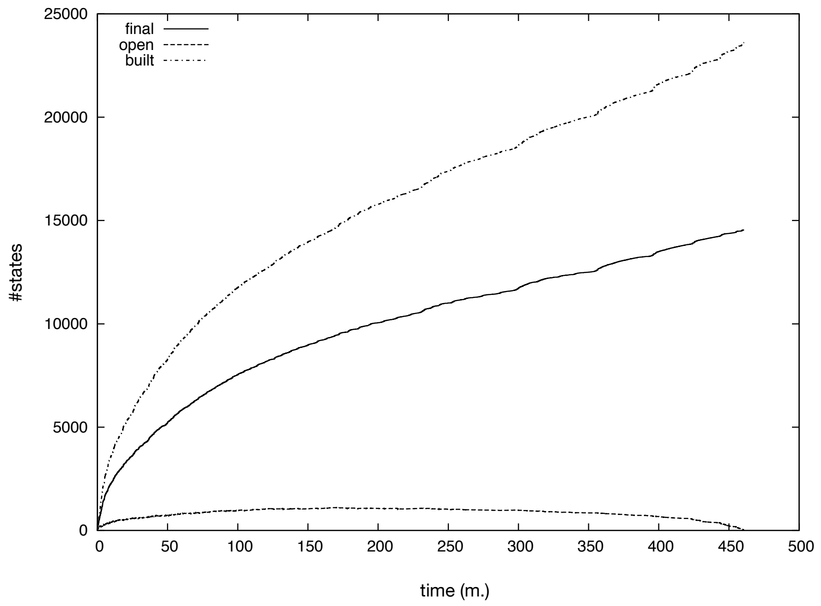


Figure 3.7: State creation advancement.

Table 3.1 reports the outcomes of the analysis on the use case. In particular the considered parameter has been measured with three versions of the net. They differ in the time granularity used for the unburned gas process, i.e., the time function of the transition

Inc_Conc. The first thing to note is however that the analysis result is coherent in the various situations, identifying the maximum amount of unburned gas as corresponding to a leaking period of two seconds.

The test has been performed on a Toshiba Notebook with 2.4Ghz Intel Core 2 Duo processor and 4GB of memory. The operating system is Ubuntu 10.10 and the Java Virtual Machine is OpenJDK IcedTea6 1.9.5.

On the table we report also the number of states of the final reduced graph against the overall number of states generated by the algorithm, and the execution times.

In Fig. 3.7 some profiling data – relating the 0.1 time granularity version of the model – are presented. On the x axis there is the execution time expressed in minutes, on the y axis there are the number of built nodes, of reduced (final) nodes, and of nodes ready to be processed, respectively. This picture is important for two reasons: first it shows that the performance degradation of state construction process is very small (the number of states created is pretty much constant in time after an initial burst); second, it supports the idea that a parallel (distributed) version of the graph builder, introduced in [17, 33, 18] should substantially improve the performances, in fact, the front of expansion remaining consistently wide.

Table 3.1: Use case analysis results.

Inc_Conc gran.	max(Conc)	# [final/built] states	exec. time
0.5	4	865/1217	$\approx 75secs$
0.25	8	2233/2983	$\approx 400secs$
0.1	20	14563/23635	$\approx 7.5hrs$

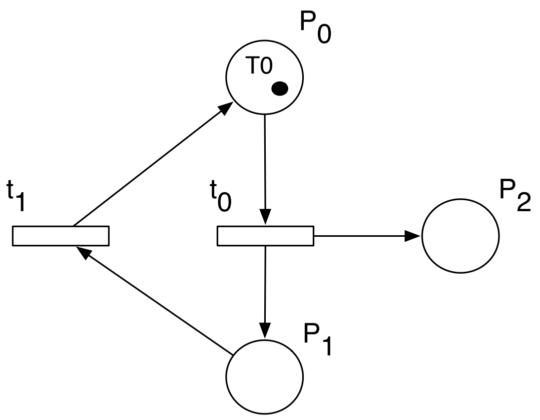
3.2.8 Summary

The analysis technique presented in this section overtakes the existing available analysis technique for Time Basic Nets (a very expressive timed version of Petri nets) because it permits the building of a sort of (symbolic) time-coverage reachability graph keeping interesting timing properties of the nets. In particular the introduction of the concept of *time anonymous* timestamps, enables a major factorization of symbolic states and allows, in many cases, to building a finite representation of the underling infinite state space. An extension of the technique that further exploits the time anonymous concept in order to deal with topologically unbounded nets exploits the concept of a coverage of *TA* tokens, i.e., a sort of ω anonymous timestamps. The next section addresses this latter topic.

3.3 Constructing Coverability Graphs for Time Basic Nets

When analyzing a Petri net, a very common question is whether or not the net is bounded. If it is bounded, the net is theoretically analyzable, and its state space is finite. However the net may be unbounded and classic state space methods generates an infinite number of reachable states from these kind of models. TB net models, as classic Place/Transition nets, may be topologically unbounded. The unboundedness happens whenever there exists a place in the net, that accumulates an infinite number of tokens during the execution. Coverability graph algorithms overcome this issue and allow us to decide several important problems: the *boundedness* problem (BP), the *place-boundedness* problem (PBP), the *semi-liveness* problem (SLP) and the *regularity* problem (RP) [83, 107]. Anyway, for TB nets, this task is complicated by the time domain. In fact, tokens come along with temporal information and, in general, it is not possible to cluster them into an ω symbol without loosing important information about the system's behavior. However, the time anonymous concept, introduced in the previous section (sec. 3.2), allow us to overcome this issue. In fact, time anonymous timestamps do not carry, by definition, any temporal information. Therefore, an infinite number of TA tokens can be clustered together into a TA^{ω} symbol without loss of information. The technique introduced in the current section, gives us a means to deal with topologically unbounded TB net models, where the unboundedness refers to places having an infinite number of TA tokens. Such a limitation is actually reasonable, in practice. In fact, this restricts the analyzable models to systems which do not exhibit *Zeno* behavior and do not express actions depending on "infinite" past events.

As a simple example, consider the model described in Figure 3.8. The behavior of the system is very simple: from the initial state, the transition t_0 must fire in the time interval $[T0, T0 + 2.0]$. Its firing consumes $T0$ and produces two new tokens in places P_1 and P_2, respectively. In this new state, t_1 is the only enabled transition, and its firing brings the system in the initial topological marking. It is worth noting that every time T_0 fires, a new token is placed into P_2 which cannot be consumed by any firing transition. Therefore, the abstraction technique introduced in section 3.2 applied to this example, generates an infinite



Initial marking	$P_0\{T_0\}$
Initial constraint	$T_0 \geq 0$
t_0	$[enab + 1.0, enab + 2.0]$
t_1	$[enab + 1.0, enab + 2.0]$

Figure 3.8: Simple example showing an unbounded TB net model.

number of reachable symbolic states because the number of tokens in place P_2 grows without limit. Figure 3.9 shows a portion of the infinite reachability tree.

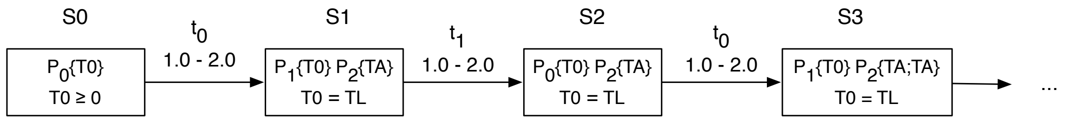


Figure 3.9: Portion of the infinite reachability tree associated to the TB net model presented in Figure 3.8.

As we can see, the number of TA tokens in place P_2 grows indefinitely, thus the execution of the software tool GRAPHGEN (introduced in section 3.2), on such a input, does not terminate. The current section, introduces an extension of the previous analysis technique able to build the coverability graph of unbounded TB nets, exploiting the concept of TA coverage tokens. Our proposal takes inspiration from the Monotone-Pruning (MP) algorithm introduced in [101], for P/T nets, and extends it to deal with TB net models, thus supplying a means, also for real-time systems, to solve the above mentioned problems.

3.3.1 Preliminaries

A quasi order $\geq$ on a set S is a reflexive and transitive relation on S. Given a quasi order $\geq$ on S, a state $s \in S$ and a subset X of S, we write $s \geq X$ iff there exists an element $s' \in X$ such that $s \geq s'$.

Given a finite set of places P, the marking M (section 3.2) on P is a function $P \rightarrow Bag(TS \cup \{TA\})$ which supplies foreach place, timestamps associated with tokens. The symbolic ω-marking M^ω on P is a function $P \rightarrow Bag(TS \cup \{TA, TA^\omega\})$. The TA^ω symbol represents, in this case, any number of TA symbols (∞ included). Given the set $U(P) = \mathbb{N}^{|P|}$, an u-marking $\bar{u}$, is an element of $U(P)$ which associates foreach place, the number of non-TA tokens. Given the set $V(P) = (\mathbb{N} \cup \{\omega\})^{|P|}$, an v-marking $\bar{v}$, is an element of $V(P)$ which associates foreach place, the number of TA tokens. Given a symbolic state S, we denote with $\bar{u}(S)$, and $\bar{v}(S)$ the u-marking and the v-marking associated with S, respectively.

Given an element $\bar{u} \in U(P)$, $\bar{v} \in V(P)$, and a place p, we denote with $\bar{u}_p$ the number of non-TA tokens in place p, and with $\bar{v}_p$ the number of TA tokens in place p. Since the ω symbol represents an infinite number of TA tokens, the component $\bar{v}_p = \omega$ if and only if $TA^\omega \in M^\omega(p)$.

For instance, if $P = \{p_1, p_2, p_3, p_4\}$ and the symbolic ω-marking is $\{p_1\{T0, TA\}, p_3\{T0, T1, TA^\omega\}\}$, the corresponding u-marking, and v-marking are $\{1, 0, 2, 0\}$, and $\{1, 0, \omega, 0\}$, respectively.

The set $V(P)$ is equipped with a partial order $\geq$ naturally extended by letting $n < \omega, \forall n \in \mathbb{N}$ and $\omega \geq \omega$.

In the current section, when referring to symbolic states, we consider an extended version of the definition 3.2.1, where the marking is represented by the function M^ω rather than M.

Definition 3.3.1 (TA erasure) Given a symbolic state $S = \langle M^\omega, C\rangle$, $S_{[\neg TA]}$ is a symbolic state composed of $\langle M^{\omega\prime}, C\rangle$, where $M^{\omega\prime}$ is a symbolic ω-marking obtained from the erasure of all TA symbols from M^ω.

Definition 3.3.2 (state coverage) Given two symbolic states $S = \langle M^\omega, C\rangle$ and $S' = \langle M^{\omega\prime}, C'\rangle$, the u- and v- markings of S $\bar{u}$, $\bar{v}$, and the u- and v- markings of S' $\bar{u}'$, $\bar{v}'$, S covers S' ($S \geq S'$) iff $\bar{u} = \bar{u}' \wedge \bar{v} \geq \bar{v}' \wedge C \equiv C'$.

That means that S differs from S' only in the number of TA tokens in places. In particular, the number of TA tokens foreach place in S is greater or equal to those ones foreach place in S'. Formally, $\forall p \in P, \bar{v}_p \geq \bar{v}'_p$. Whenever $S \geq S'$ and $\bar{v} \neq \bar{v}'$ we say that S properly covers S', and we denote it with $S > S'$.

Definition 3.3.3 (Coverability Tree) Given a TB net $\mathcal{R} = \langle P, T, F\rangle$, a coverability tree is a tuple $\mathcal{T} = \langle N, n_0, E\rangle$, where N is a set of symbolic states, $n_0 \in N$ is the root state, $E \subseteq N \times T \times N$ is the set of edges labeled with firing transitions. Where:

1. foreach reachable symbolic state S in $TRG(\mathcal{R})$ (section 3.2), there exists $S' \in N$ s.t. either $S' \supseteq S'$ or $S' \geq S$.

2. foreach symbolic state $S = \langle M^\omega, C\rangle \in N$, having u-marking $\bar{u}$ and v-marking $\bar{v}$, there exists either a reachable state s of $\mathcal{R}$ s.t. $s \in S$, or a an infinite sequence of reachable symbolic states in $TRG(\mathcal{R})$, $(S_n)_{n\in\mathbb{N}}$ s.t. $\forall n, C_n \equiv C$ and $\forall n$, $\bar{u}(S_n) = \bar{u}$ and the sequence $(\bar{v}(S_n))_{n\in\mathbb{N}}$ is strictly increasing converging to $\bar{v}$.

Given a symbolic state $S \in N$, we denote by $Ancestor_{\mathcal{T}}(S)$ the set of ancestors of S in $\mathcal{T}$ (S included). If S is not the root of $\mathcal{T}$, we denote by $parent_{\mathcal{T}}(S)$ its first ancestor in $\mathcal{T}$. Finally, given two symbolic states S and S' such that $S \in Ancestor_{\mathcal{T}}(S')$, we denote by $path_{\mathcal{T}}(S, S') \in E^*$ the sequence of edges leading from S to S' in $\mathcal{T}$.

3.3.2 Coverability Tree Algorithm

This section presents the algorithm able to construct coverability trees of TB nets. We call it $TBCT$ (Algorithm 1) and it is inspired by the Monotonic pruning (MP) algorithm introduced in [101], able to build minimal coverability sets for classic P/T nets. Our proposal involves the acceleration function *Acc*, first introduced in the Karp and Miller algorithm [83]. Such a function aims at computing the limit of repeating any number of times some sequences of transitions that strictly increase the number of tokens in certain places. However, in our context, it is defined and also applied in a slightly different manner. It actually modifies the symbolic ω-marking M^ω of a symbolic state by inserting proper TA^ω symbols, accordingly to the following:

$$Acc : 2^N \times N \to N, Acc(W, S)(p) = S' \text{ s.t.}$$

$$\forall p \in P, \bar{v}(S')_p = \begin{cases} \omega & \text{if } \exists S'' \in W : S'' < S \wedge \bar{v}(S'')_p < \bar{v}(S)_p \wedge S'' \rhd S \\ \bar{v}(S')_p & \text{otherwise} \end{cases} \tag{3.3.1}$$

Where $S'' \rhd S$ iff there exists $\sigma = path_{\mathcal{T}}(S'', S)$, such that σ is feasible from S. Such a condition holds whenever, either:

1. $C_{S''} \implies C_S$, meaning that, $S''_{[\neg TA]} \subseteq S_{[\neg TA]}$. In this case, all the paths starting from S'' are feasible from S.

2. $C_S \implies C_{S''}$ and the first component of σ is of type `A*` (section 2.3). In this case $S_{[\neg TA]} \subseteq S''_{[\neg TA]}$, therefore not all paths starting from S'' are feasible from S, but since σ starts from all ordinary states of S'', σ is feasible also from S.

For instance, considering the example in Figure 3.9, the evaluation of the *Acc* function on $S2$ and its ancestors: *Acc(*$\{S0, S1\}$*, S2)*, causes the insertion of the TA^ω symbol into P_2 because $S2 > S0$, $\bar{v}(S2)_{P_2} > \bar{v}(S0)_{P_2}$ and the path from $S0$ to $S2$ is feasible from $S2$. This way, we recognize that TA tokens into place P_2 can grow without limit.

Algorithm 1 TBCT Algorithm.

```
Require: A TB net $\mathcal{R} = \langle P, T, F \rangle$
Ensure: A coverability tree $\mathcal{T} = \langle N, n_0, E \rangle$, $N = Act \cup Inact$
 1: function TBCT($\mathcal{R}$)
 2:     $n_0 = BuildRoot(\mathcal{R})$
 3:     $N = \{n_0\}$; $Act = N$; $Wait = N$; $E = \emptyset$;
 4:     while $Wait \neq \emptyset$ do
 5:         $s = Pop(Wait)$;
 6:         if $s \in Act$ then
 7:             for $t \in EnabledTransitions(s, \mathcal{R})$ do
 8:                 $m = Successor(s, t)$;
 9:                 $n = Acc(Ancestors_{\mathcal{T}}(m) \cap Act, m)$;
10:                 $N+ = \{n\}$; $E+ = \{\langle s, t, n \rangle\}$;
11:                 if $\nexists a \in Act : a \supseteq n$ then
12:                     if $\exists a \in Act : a \subset n$ then
13:                         $Act- = \{x : a \in Ancestors_{\mathcal{T}}(x)\}$;
14:                     end if
15:                     if $\nexists a \in Act : a \geq n$ then
16:                         $Act- = \{x : \exists y \in Ancestors_{\mathcal{T}}(x)$ s.t $y \leq n$
17:                                   $\wedge\ (y \in Act \vee y \in Ancestors_{\mathcal{T}}(n))\}$;
18:                         $Act+ = \{n\}$; $Wait+ = \{n\}$
19:                     end if
20:                 end if
21:             end for
22:         end if
23:     end while
24: end function
```

Likewise both the Karp and Miller and the MP Algorithms, the *TBCT* algorithm builds a coverability tree, but nodes, in the current context, are symbolic states containing symbolic ω-markings and edges are labeled by transitions of the analyzed TB net. Therefore it proceeds by exploring the reachability tree of the net, as shown in section 3.2, and accelerating along branches to reach “limit” symbolic ω-markings (containing proper TA^ω symbols). In addition, during the exploration, it can prune branches that are covered by nodes on other branches. Therefore, nodes of the tree are partitioned in two subsets: *active* nodes, and *inactive* ones. Intuitively, active nodes will form the coverability set of the TB net, while inactive ones are not part of the final coverability set, because they are dominated by other active nodes.

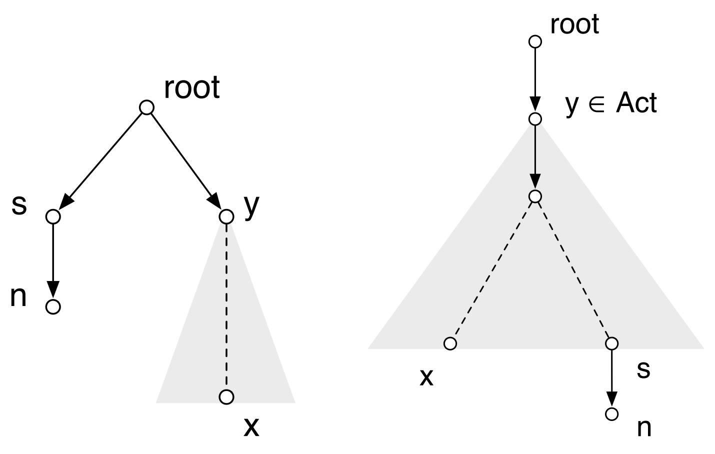
The Algorithm 1 proceeds in the following steps to decide how to change the structure $\mathcal{T}$ according to new computed reachable symbolic states:

1. The symbolic state s, popped from $Wait$ should be active (test of Line 6).
2. The algorithm iterates through all the enabled transitions and computes one by one all the successor symbolic states: $m = Successor(s, t)$; (Line 8).
3. The state m is accelerated w.r.t. its active ancestors. A new symbolic state n is created by this operation: $n = Acc(Ancestors_{\mathcal{T}}(m) \cap Act, m)$; (Line 9).
4. If the new symbolic state n is not included or equal to one of the existing active nodes, then it is candidate to be active (test of Line 11).
5. If the new symbolic state n includes an existing active node a, then the sub-tree with root a is deactivated (Lines 12-13).
6. The new symbolic state n is declared as active iff it is not covered by any existing active nodes (test of Line 15 and Line 18).
7. If n is not covered, some symbolic states are deactivated (Line 17).

The update of the set *Act*, complies with the following rules. Intuitively, nodes (and their descendants) are deactivated if they are included or covered by other nodes. This would lead to deactivate a node x iff it owns an ancestor y dominated by n, i.e. such that either $y \subset n$ (Lines 12- 13) or $y \leq n$ (15-17). Concerning the latter case, whenever a new node n (obtained from *Wait*) covers a node y ($y \leq n$), then y can be used to deactivate nodes in two ways:

- if $y \notin Ancestors_{\mathcal{T}}(n)$, then no matter whether y is active or not, all its descendants are deactivated (Figure 3.10a).
- if $y \in Ancestors_{\mathcal{T}}(n)$, then y must be active ($y \in Act$), and in that case all its descendants are deactivated, except node n itself as it is added to Act (Line 18). We require $y \in Act$ to avoid useless operations. In fact, descendants of n dominates descendants of y (Figure 3.10b).

For example, considering the example in Figure 3.9, the insertion of $S2$ accelerated causes the deactivation of both $S0$ and $S1$ because of the execution of line 17. In particular, such a



(a) $y \notin Ancestors_{\mathcal{T}}(n)$ (b) $y \in Ancestors_{\mathcal{T}}(n)$

Figure 3.10: Deactivations of symbolic states in the TBCT Algorithm

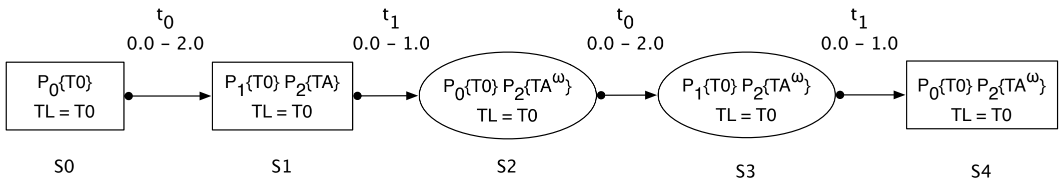


Figure 3.11: Coverability tree constructed from the TB net example presented in Figure 3.8.

situation corresponds to Figure 3.10b, because $S2 \geq S0$ and $S0$ (*active* node) belongs to $Ancestors_{\mathcal{T}}(S2)$.

Figure 3.11 depicts the coverability tree constructed from the TB net example presented in Figure 3.8. Elliptic symbolic states form the final coverability set (*active* nodes), while the squared ones are symbolic states deactivated during the analysis. As we can see, the $TBCT$ algorithm builds a finite tree structure from an unbounded TB net model. In particular, as shown before, the algorithm is able to identify that the number of TA tokens in place P_2 can grow without limit.

As we can see in Figure 3.11, edges carry information about their type (either `AA`, `EE`, `AE` or `EA`, introduced in section 2.3), and about the local minimum-maximum firing time, as introduced in the previous section 3.2. In the following, given an edge e, we refer to these information with *type(e)* and *time(e)*, respectively. In particular we refer to the source type with *type(e)*$_{src}$ and to the target type with *type(e)*$_{trgt}$.

It is also possible to construct a coverability graph $\mathcal{G}$ rather than a tree. This task, starting from the tree structure $\mathcal{T} = \langle N, n_0, E \rangle$, executes the following steps:

1. All *inactive* nodes are erased from N.
2. $\forall a \in Act, \forall \langle a, t, b \rangle \in E$, if b is *inactive*, we search for $a' \in Act$ so that $a' \supseteq b$ or $a' \geq b$, thus we remove $\langle a, t, b \rangle$ from E and we insert $\langle a, t, a' \rangle$.
3. All covered edges (Definition 3.3.4) are removed from E.

Definition 3.3.4 (edge coverage) Given a coverability tree $\mathcal{T} = \langle N, n_0, E \rangle$ and two edges $e = \langle a, t, b \rangle$, $e' = \langle a', t', b' \rangle \in E$, e covers ($\geq$) e' iff:

i $a = a' \wedge t = t' \wedge b = b'$

ii $time(e) \supseteq time(e')$

iii $type(e)_{src} \geq type(e')_{src} \wedge$ $type(e)_{trgt} \geq type(e')_{trgt}$, being `A` $>$ `E`

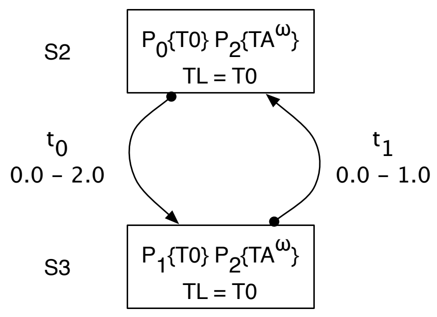
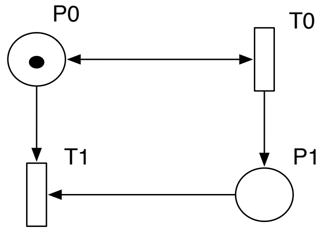


Figure 3.12: Coverability graph constructed from the coverability tree presented in Figure 3.11.

Figure 3.12 shows the coverability graph resulting from the coverbility tree presented in figure 3.11. Such a structure contains only active symbolic states and gives us a more intuitive overview on the system's behavior. For instance, by observing Figure 3.12, it's easy to figure out that the system alternates two symbolic states where P_0 and P_1 are marked with a single token, while place P_2 can accumulate TA tokens without limit.

The rest of this section reports some simple examples of unbounded TB net models analyzed by the software tool implementing the $TBCT$ algorithm. All the coverability trees/graphs have been automatically obtained by using GraphViz visualization software [71] on the output generated from the tool-set. The TW notation used into symbolic ω-markings, stands for TA^{ω}.

Example A Figure 3.13 depicts an unbounded TB net model with two places ($P0$, $P1$) and two transitions ($T0$, $T1$). It represents a simple synchronous system, where an operation occurs at each time unit (e.g. production/consumption). Produced units are stored into a infinite buffer. After the first consumption the system stops.



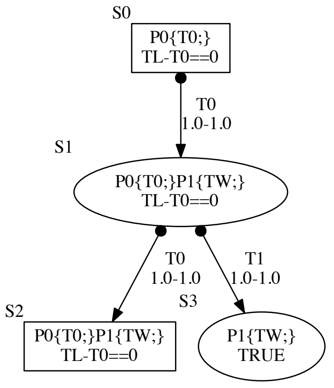
Initial marking	$P0\{T0\}$
Initial constraint	$T0 \geq 0$
$T0$	$[enab + 1.0, enab + 1.0]$
$T1$	$[enab + 1.0, enab + 1.0]$

Figure 3.13: Unbounded TB net model A.

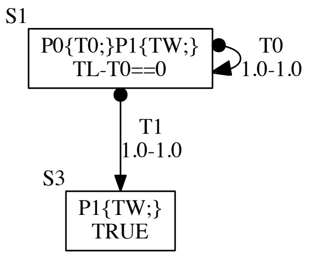
Figure 3.14a shows the coverability tree of A. As we can see, the introduction of $S1$ causes the deactivation of $S0$ ($S1 \geq S0$). From $S1$ the system can evolve either into $S2$ which is inactive ($S2 = S1$), or $S3$ which is a final state. Such a behavior is also shown by the coverability graph (Figure 3.14b): the system loops into $S1$, by the firing of $T0$ transition, until the firing of $T1$ which leads into the final state $S3$.

Example B This model (Figure 3.15) is analogue to model A, except for an additional arc and a different initial marking. It represents two synchronous tasks, where each task can either produce or consume. An infinite buffer stores produced units. Figure 3.16a and 3.16b show its coverability tree and coverability graph, respectively. It is worth noting that the firing of $T0$ from $S1$ produces an additional token into place $P1$ and because of the recognition of both tokens of $P1$ as TA, the acceleration of $S2$ recognizes the TA^ω into $P1$. Therefore, $S2$ deactivates both $S0$ and $S1$. Successors of both $S3$ and $S4$ are identified equal to $S2$.

Example C This model (Figure 3.17) represents an unbounded TB net with four places ($P0$, $P1$, $P2$, $P3$), two strong transitions ($T0$, $T2$) and a weak transition ($T3$). Transition $T0$ acts as a sort of timer, in fact, whenever enabled, it must fire in 10 time units from its previous firing time. Figure 3.18a and 3.18b show its coverability tree and coverability graph, respectively.

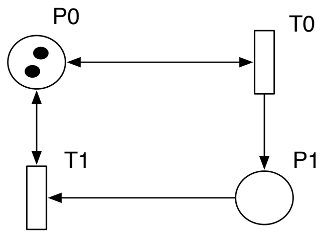


(a) Coverability tree of A



(b) Coverability graph of A

Figure 3.14: Coverability tree/graph of example A.



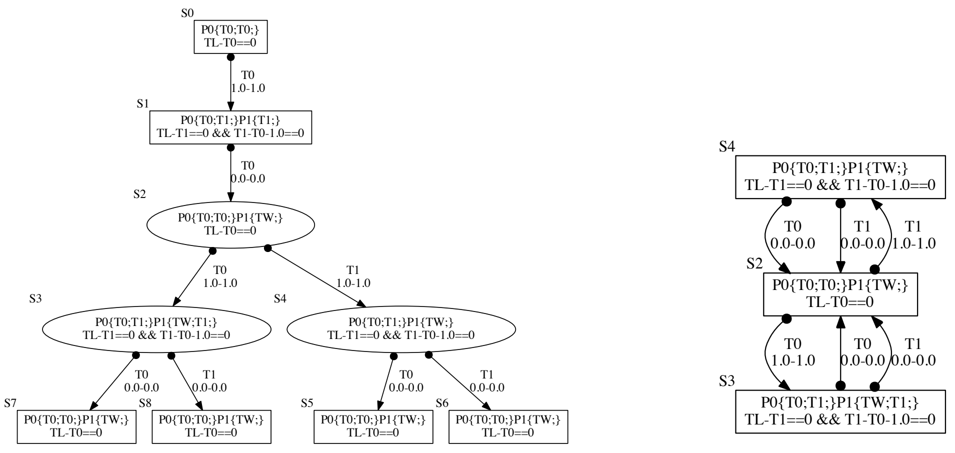
Initial marking	$P0\{T0, T0\}$
Initial constraint	$T0 \geq 0$
$T0$	$[enab + 1.0, enab + 1.0]$
$T1$	$[enab + 1.0, enab + 1.0]$

Figure 3.15: Unbounded TB net model B.

Concerning the current example, it is worth noting that before the introduction of $S4$, all the symbolic states were *active*. The acceleration of $S4$ leads to the recognition of a TA^{ω} into place $P3$, and thus the identification of the coverage $S4 \geq S1$. This causes the deactivation of both $S1$ and its descendants $S2$ and $S3$. The successors of $S4$ are $S5$ and $S6$. In this case, since $S5 \subset S6$, $S5$ is deactivated. Finally, $S7$ is recognized to be equal to $S4$.

3.3.3 Related Work

As introduced in section 3.1, Karp and Miller (K&M) introduced an algorithm for computing the *minimal coverability set* (MCS) in [83]. It uses acceleration techniques to collapse branches of the tree and ensure termination. Anyway, the K&M Algorithm is not efficient in

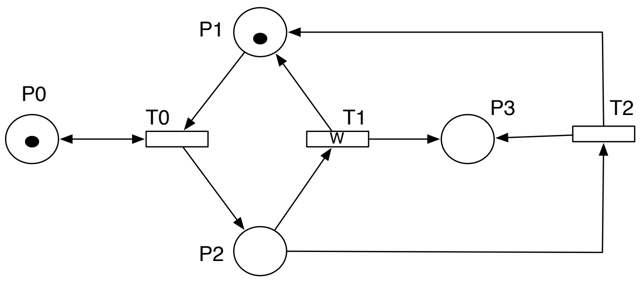


(a) Coverability tree of B (b) Coverability graph of B

Figure 3.16: Coverability tree/graph of example B.

analyzing real-world examples and it often does not terminate in reasonable time. One reason is that in many cases it will compute several times a same subtree. The MCT algorithm [54] introduces clever optimizations: a new node is added to the tree only if its marking is not smaller than the marking of an existing node. Then, the tree is pruned: each node labelled with a marking that is smaller than the marking of the new node is removed together with all its successors. The idea is that a node that is not added or that is removed from the tree should be covered by the new node or one of its successors. However, the MCT algorithm is flawed [62]: it computes an incomplete *forward* reachability set. The CoverProc algorithm, is proposed for the computation of the MCS of a Petri net. This algorithm follows a different approach and is not based on the K&M Algorithm. In [101], the MP algorithm is proposed. This algorithm can be viewed as the MCT algorithm with a slightly more aggressive pruning strategy. Experimental results show that the MP algorithm is a strong improvement over both the K&M and the CoverProc algorithms. The *TBCT* algorithm, introduced in the current section, is somehow inspired by the MP algorithm, and is able to construct coverability graphs of real-time systems modeled with TB nets.

For timed Petri nets (TPNs), although the set of *backward* reachable states is computable [1], the set of *forward* reachable states is in general not computable. Therefore any procedure for performing forward reachability analysis on TPNs is incomplete. In [2], an abstraction of the set of reachable markings of TPNs is proposed. It introduces a symbolic representation



Initial marking	$P0\{T0\}P1\{T0\}$
Initial constraint	$T0 \geq 0$
$T0$	$[enab, P0 + 10.0]$
$T1$	$[enab + 2.0, enab + 3.0]$
$T2$	$[enab + 1.0, enab + 4.0]$

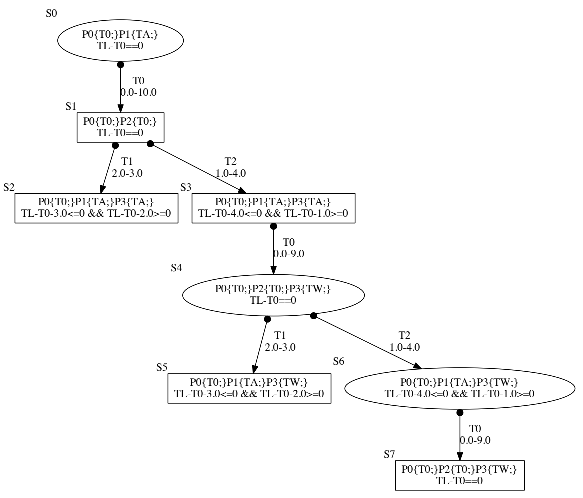
Figure 3.17: Unbounded TB net model C.

for downward closed sets, so called region generators (i.e. the union of an infinite number of regions [3]). Anyway, the termination of the forward analysis by means of this abstraction is not guaranteed.

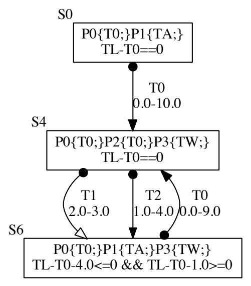
In the current section, we addressed unbounded TB nets, which represent a much more expressive formalism for real-time systems than TPNs (interval bounds in TB nets are linear functions of timestamps in the enabling marking, rather than simple numerical constants). Other coverability analysis techniques for such a formalism, have not been proposed yet, as far as we know.

3.3.4 Summary

The current section introduces a coverability analysis technique able to construct a coverability tree/graph for unbounded TB net models. The termination of the $TBCT$ algorithm is guaranteed as long as, within the input model, tokens growing without limit, can be anonymized. This means that we are able to manage models that do not exhibit *Zeno* behavior and do not express temporal functions depending on “infinite” past events. This is actually a reasonable limitation because, in general, real-world examples do not exhibit such a behavior.



(a) Coverability tree of C



(b) Coverability graph of C

Figure 3.18: Coverability tree/graph of example C.

Chapter 4

Big Data Approaches to Formal Verification

This chapter focuses on the connection between formal methods in software engineering and big data approaches. This part of the thesis tries to overcome the major limitation of the software tools introduced in the previous chapter. In particular we outline approaches that will allow verification techniques and tools to undergo a technological transition in order to exploit the new available architectures. The idea is simple: increasing the computational power and storage availability, by using a cluster of distributed computers. The use of networks of computers can provide the resources required to verify complex systems' models. Unfortunately, this approach requires several skills which—while common in the "big data" community—are rather unusual in the "formal methods" community. Our recent works focused on the connection between formal methods in software engineering and big data approaches [18, 17, 33]. The analysis of complex systems certainly falls in this context, although formal verification has been so far poorly explored by big data scientists [72]. We believe, however, that the challenges to be tackled in formal verification can benefit a lot from the recent achievements in big data access and management. In fact formal approaches require several different skills: on one hand, an adequate background is required in order to manage specific formalisms and abstraction techniques both in modelling and analysis interpretation; on the other hand, these techniques should be supported by tools able to analyse large amount of data very reliably and efficiently, similarly to "big data" projects. Recent approaches have shown the convenience of employing distributed memory and computation to manage generation/exploration of large state-spaces. Unfortunately exploiting these frameworks requires further skills in developing complex applications with knotty communication and synchronization issues. In particular, tailoring applications so that they conveniently scale on available cloud computing facilities, might be a daunting task without a proper knowledge of the subtleties of data-intensive and distributed analysis.

This chapter is composed of three sections that aim at further bridging the gap between these different areas of expertise by providing different techniques, frameworks and tools, built on top of HADOOP MAPREDUCE [106], which is based in turn on the MapReduce programming model [46] (described in section 4.2). This programming model, which has become the *de facto* standard for large scale data-intensive applications, has provided researchers with a powerful tool for tackling big-data problems in different areas [94, 18, 104]. We firmly believe that explicit state model checking could benefit from a distributed MapReduce based approach, but this topic has not yet been sufficiently investigated as far as we know.

Section 4.3 discusses about the parallelization of the TB nets analysis techniques introduced in section 3.2. In particular, we study and compare two different approaches, relying on distributed and cloud frameworks, respectively. Section 4.4 starts from the results obtained from the study carried out parallelizing the TB nets state space building, and introduced a generic framework, formalism independent so called MARDIGRAS. This framework can be easily specialized to deal with the construction of very large state spaces of different kinds of formalisms (e.g., different kinds of Petri Nets, Process Algebras etc.). Section 4.5 outlines a distributed CTL (Computation Tree Logic) model checker, which implements iterative MapReduce algorithms based on the fixed-point characterization of the basic temporal operators of CTL. It can be easily specialized to deal with verification of CTL formulas on huge state spaces generated from different formalisms, for example by means of MARDIGRAS based tools.

4.1 State of the Art

Approximately fifteen years ago, the trend of multi-core and distributed computing brought to multi-core and distributed verification algorithms. This has made it possible, in many cases, to achieve better performance although the potentially very large computational complexity of these problems. In fact, for real world models, the state space size may easily exceed the memory capacity of a single computer, hence sequential formal verification tools could be very slow or even crash as soon as memory is exhausted. Therefore, the use of parallel/distributed processing platforms to tackle state space explosion in explicit-state verification techniques gained a growing interest in recent years. Among other works, we may cite [36, 41, 42, 88, 25]. However, most of these works are related to a specific formalism, and they do not consider new emerging distributed solutions like Big data approaches. Works presented in [93, 51] describe large-scale graph processing application reformulated in terms of MapReduce programming model, but unfortunately, this large class of graph algorithms doesn't fit well with the state explosion problem in large-scale graph building, which remains rather unexplored. Works presented in [91, 28, 8] discuss parallel/distributed verification of *Linear Temporal Logic* (LTL) formulas. They aim at increasing memory availability and reducing the overall computation time by employing distributed search techniques of accepting cycles in Büchi automata. Distributed and parallel approaches to CTL model checking have been proposed in [29, 13, 25] These message-passing based algorithms work by splitting the model state space into a number of "partial state spaces". Each node involved in the computation owns a partial state space and performs a (partial) model checking on this incomplete structure. This is in truth the main idea that most of existing distributed approaches for both LTL and CTL model checking rely upon. The differences stay in the way the state space is partitioned (through a *partition function*), which is a crucial issue. In order for a parallel/distributed model checking to be effective, a good load balancing among machines must be achieved. Ideally each computation unit should manage nearly the same number of states. The performances of distributed approaches also depend on the number of cross-border transitions of the partitioned state space (*i.e.,* transitions having the source state stored in a component and the target in another component). This number should be kept as small as possible, since it heavily impacts on the overall number of messages sent over the network during analysis [26]. As for LTL model checking, some probabilistic partitioning techniques have been defined, e.g., [91, 103]. A different approach based on the structural properties of the formula to be verified has been proposed in [10].

Anyway the effectiveness of big data approaches in formal verification, has been poorly addressed as far as we know. In fact, despite many years of work in the area of multi-core and distributed model checking, still few works introduce algorithms that can scale effortlessly to the use of thousands of loosely connected computers in a network, so existing technology does not yet allow us to take full advantage of the vast array of compute power of a "cloud" environment. This revolution is already started in different scientific fields, achieving remarkable breakthroughs through new kinds of experiments that would have been impossible only few years ago. The use of big data approaches in the context of formal methods is a new emerging trend, and to our knowledge, a few other techniques and tools have been introduced: [22] presents a MapReduce approach to check specifications expressed in a metric temporal logic over large execution traces, with aggregation modalities; [77] attempts (in a quite different context, *i.e.*, Swarm Verification) to exploit massively parallel jobs running test randomization techniques to verify the correctness of mission critical software; in the context of run-time verification, [11] introduces an algorithm for the automated verification of LTL formulas on event traces by processing multiple, arbitrary fragments of the trace in parallel, and compute its final result through a cycle of runs of MapReduce instances; while [12] proposes a MapReduce based approach to monitoring systems offline, where system actions are logged in a distributed file system and subsequently checked for compliance against policies formulated in an expressive temporal logic.

4.2 The MapReduce Programming Model

MapReduce is a programming model and an associated implementation for processing and generating very large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in [46]. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. This allows programmers with little experience with parallel and distributed systems to easily utilize the resources of a large distributed system . MapReduce is designed to run on a large cluster of commodity machines and is highly scalable. It relies on the observation that many information processing activities have the same basic design: a same operation is applied over a large number of records (*e.g.,* database records, or vertices of a graph) to generate partial results, which are then aggregated to compute the final output. The MapReduce model consists of two functions: The "map" function turns each input element into zero or more key-value pairs.

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

A “key” is not unique, in fact many pairs with a given key could be generated from the Map function: the “reduce” function is applied to the list of values associated to the same key. The result is a set of key-value pairs consisting of whatever is produced by the Reduce function applied to the list of values.

$$reduce(k_2, list(v_2)) \rightarrow list(k_3, v_3)$$

Between these two main phases, the system sorts the key-value pairs by key, and groups together values with the same key. This two-step processing structure is presented in Figure 4.1. Users create their own applications through a “map” function which specifies per-record computations, and a “reduce” function which specifies the aggregation of map computations: both operate in parallel on key-value pairs which represent the input of the problem. The mapper is applied to every input key-value pair to generate an arbitrary number of intermediate key-value pairs. The reducer is then applied to all values associated with the same intermediate key to generate an arbitrary number of final key-value pairs as output.

As an example of the MapReduce programming model, consider the problem of counting the number of occurrences of each word in a large collection of documents (several Gigabytes or even Terabytes of data). The user would write code like the following pseudocode (Algorithm 2):

Algorithm 2 WordCount Procedure

```
function MAP(String $key$, String $value$)
    // $key$: document name
    // $value$: document contents
    for $word \in value$ do
        EmitIntermediate($word, 1$)
    end for
end function
function REDUCE(String $key$, Iterator $values$)
    // $values$: a list of counts
    int result = 0
    for $v \in values$ do
        $result+ = v$
    end for
    Emit($result$)
end function
```

The *map* function emits each word plus an associated count of occurrences (just 1 in this simple example). The *reduce* function sums together all counts emitted for a particular word.

Other few simple examples of interesting programs that can be easily expressed as MapReduce computations are for instance:

- *Distributed Grep.* The map function takes as input lines of text and emits a line whether it matches a supplied pattern. The reduce phase is just an identity function: the reducers just copy the supplied intermediate text lines to the output.
- *Count of URL Access Frequency.* A mappers takes as input a set of logs of web page requests and outputs $\langle URL, 1\rangle$ foreach processed URL. The reduce function adds together all values for the same URL and emits the pair $\langle URL, totalcount\rangle$.
- *Reverse Web-Link Graph.* The map function parses a web page source and outputs a $\langle target, source\rangle$ pair for each link to a target URL. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle target, list(source)\rangle$.
- *Inverted Index.* The map function emits a set of $\langle word, documentID\rangle$ pairs, found in a document. The reduce function takes as input all pairs for a given word, sorts the corresponding document IDs and emits the pair $\langle word, list(documentID)\rangle$. The set of all output pairs forms a simple inverted index. It is also possible to modify a little this computation to keep track of word positions. For instance the map function could emit $\langle word, (documentID, offset)\rangle$, and the reduce function can thus emit the pair $\langle word, list(documentID, offset)\rangle$.

The Map invocations are distributed across multiple machines by automatically partitioning the input data. into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function:

$$hash(key)\ mod\ R$$

The number of partitions (R) and the partitioning function are specified by the user. Figure 4.2 shows the overall flow of a MapReduce job in its original implementation, as introduced in [46]. When the user program launches a MapReduce job, the sequence of actions reported

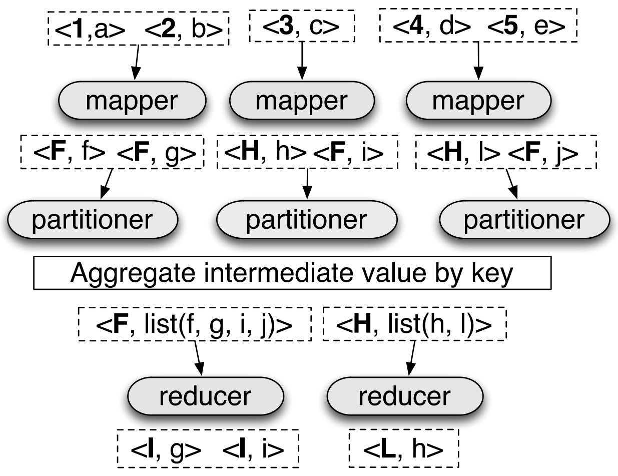


Figure 4.1: The MapReduce model: the keys are in **bold**.

below, take place (the numbered labels in Figure 4.2 correspond to numbers in the following list):

1. The user program uses the MapReduce library to split the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via optional parameters). Later on, it starts up different copies of the program among the machines of the cluster.

2. Among those copies of the program, one is special and it is called "master". Other copies are workers that receive and perform tasks assigned assigned by the master. There are M map tasks and R reduce tasks to assign. The master assigns them to idle workers.

3. After a map task is assigned, a worker reads the contents of the corresponding input split. It parses key/value pairs from the input data and foreach pair, invokes the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

4. Intermediate key/value pairs are periodically written into local disks, partitioned into R regions by means of the user-defined partitioning function. At the end of the Map phase, the mappers communicates the locations of these buffered pairs to the master, which is responsible for forwarding these locations to reducers.

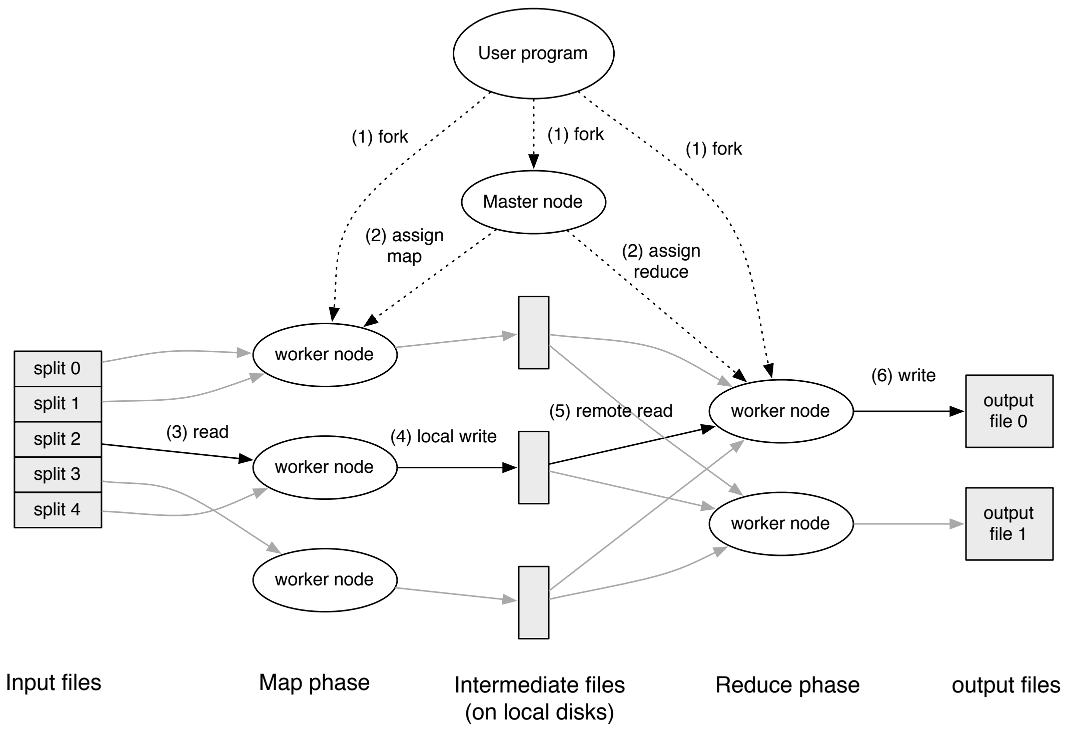


Figure 4.2: MapReduce execution overview. Adapted from [46].

5. Notified reduce workers, use remote procedure calls to read the buffered data from the local disks of the map workers. When all the intermediate data has been read by a reduce worker, it is sorted it by the intermediate keys so that all values having the same key are grouped together. The sorting phase is needed because typically many different keys map to the same reduce task.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key, it calls the user-defined Reduce function passing as input the key and the corresponding set of intermediate values. The output of the Reduce function is written into a final output file for this reduce task.

7. After the end of all map tasks and reduce tasks, the master terminates its its job, by waking up the user program. Thus, the execution flow comes back to the user code.

After successful completion, the output of the MapReduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

Under the MapReduce programming model, a developer needs to provide implementations of the mapper and reducer. On top of a distributed file system [64], The execution framework handles transparently all non-functional aspects of execution on big clusters. It is responsible, among other things, for scheduling (moving code to data), handling faults, and the large distributed sorting and shuffling needed between the map and reduce phases since intermediate key-value pairs must be grouped by key.

The "partitioner" is responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. The users specify the number of reducers/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. The default partitioner computes a hash function on the value of the key modulo the number of reducers (i.e. $hash(key)\ mod\ R$). This does not guarantee good load balance because the distribution of values associated with the same key may be highly skewed. In some cases, it might be useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using $hash(Hostname(UrlKey))\ mod\ R$ as the partitioning function causes all URLs from the same host to end up in the same output file. Another example is graph construction: sometimes we may want to group together intermediate values (graph nodes) ensuring that nodes potentially related belong to the same partition, in order to merge equal nodes. Thus we can use $hash(getFeatures(NodeKey))\ mod\ R$, where $getFeatures$ computes specific features such that the equality of the evaluation of such a features is a necessary condition for equality relationship among graph nodes.

As an optimization, MapReduce supports the use of "combiners", which are similar to reducers except that they operate on the output of single mappers. Combiners operate in isolation on each node in the cluster after a mapper and cannot use partial results from other nodes. They allow a programmer to aggregate partial results (i.e., intermediate key-value pairs), thus reducing network traffic. In cases where an operation is both associative and commutative, reducers can directly serve as combiners, although in general they are not interchangeable. A good example of this is the *WordCount* previously introduced (Algorithm 2). Since word frequencies tend to follow a Zipf distribution [110], each map task will produce hundreds or thousands of records of the form $\langle the, 1\rangle$, $\langle be, 1\rangle$, $\langle and, 1\rangle$ and so forth. All of these counts will be sorted and then sent over the network to a single reduce task which adds together the values by the user defined reduce function to produce one number. A Combiner function allows partial merging of this data before it is sent over the network. The Combiner function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task. Partial combining significantly speeds up certain classes of MapReduce operations [46].

Input data can be supplied in several different formats, in fact the framework provides different extendable *reader*s. A very common input format is “text”. Text input reader splits each line as a key/value pair: the key is the offset (in Bytes) in the file and the value is the contents of the line. Another common supported format is “Sequence” input, for reading particular binary file formats. Each input type implementation defines how to split files into meaningful pieces for processing, for example, splitting in text mode ensures that input splits occur only at line boundaries. Users can add support for a new input type by providing an implementation of the reader interface, A reader can also provide data from different sources. For example, it could be defined a reader that reads records from a database, or from data structures mapped in memory. In a similar fashion, MapReduce supports a set of output types for producing data in different formats by means of *writer*s, and users can extend the code to add support for new output types.

4.3 Symbolic State Space Exploration of RT Systems in the Cloud

The growing availability of distributed and cloud computing frameworks makes it possible to face complex computational problems in a more effective and convenient way. A notable example is state-space exploration of discrete-event systems specified in a formal way. The exponential complexity of this task is a major limitation to the usage of consolidated analysis techniques and tools. Several techniques for addressing the state space explosion problem within this context have been studied in the literature. One of these is to use distributed memory and computation to deal with the state space explosion problem. In this section we study and compare two different approaches, relying on distributed and cloud frameworks, respectively. These approaches were designed and implemented following the same computational schema, a sort of map & fold. They are applied on symbolic state-space exploration of real-time systems specified by Time-Basic Petri Nets, by re-adapting a sequential algorithm implemented as a Java tool. The outcome of several tests performed on a benchmarking specification are presented, thus showing the convenience of distributed approaches.

4.3.1 Sequential and Parallel algorithms

The TRG construction has been automated by means of a Java tool called Graphgen, introduced in section 3.2.6. The corresponding algorithm follows a very simple sequential schema described by the Algorithm 3.

The `remaining` list contains the reachable nodes of the graph not yet examined, *i.e.,* the *expansion front* of the graph. The graph builder takes one node at a time from the expansion front and executes two main phases that we call `Map` and `Fold`. These operations derive from a well known programming model in which a Map instance takes as input a sequence of values and computes a given function for each value. Then, a Fold instance combines in some way the elements of the sequence using an associative binary operation.

In the TRG builder, the `Map` generates the successors of a node, the `Fold` combines them with the already existing nodes by identifying possible inclusion relationships. Whenever the `Fold` phase identifies a relation between a new node a (just computed by the `Map`) and an old node a' (already expanded), different operations must be performed on the adjacent edges depending on the relation between a and a'.

- If $a \subseteq a'$, the incoming edges of a are redirected to a'. The outgoing edges are not yet calculated
- If $a \supset a'$, the incoming edges of a' are redirected to a and the outgoing edges of a' are removed.

At the end of the `Fold` phase the nodes computed by the `Map` which are not included in any old nodes, are placed into the `remaining` list. The `Map` phase and the `Fold` phase are repeated until the expansion front becomes empty.

Algorithm 3 GRAPHGEN(a_0)

```
Require: The root state $a_0$.
Ensure: The state space $\langle A, E, a_0\rangle$ of the model.
 1: Stack $remaining$
 2: Set $A, E, N$
 3: $t$ ::= EQUALS | INCLUDES | INCLUDED | NONE
 4: $Push(remaining, a_0)$
 5: $Add(A, a_0)$
 6: while $remaining \neq \emptyset$ do
 7:     $s = Pop(remaining)$
 8:     $N = ReachableStatesFrom(s)$
 9:     for $n \in N$ do
10:         $Add(E, \langle s, n\rangle)$
11:         for $o \in A$ do
12:             $t = IdentifyRelationship(n, o)$
13:             if $t ==$ EQUALS $\vee$ $t ==$ INCLUDED then
14:                 $Replace(E, \langle s, n\rangle, \langle s, o\rangle)$
15:             else if $t ==$ INCLUDES then
16:                 $Remove(A, o)$
17:                 $Add(A, n)$
18:                 $Push(remaining, n)$
19:                 $Replace(E, \langle *, o\rangle, \langle *, n\rangle)$
20:                 $Remove(E, \langle o, *\rangle)$
21:             else if $t ==$ NONE then
22:                 $Add(A, n)$
23:                 $Push(remaining, n)$
24:             end if
25:         end for
26:     end for
27: end while
```

Note that while discussing the sequential algorithm we never referred explicitly to the TB net formalism. In fact, by specializing the `Map` and the `Fold` concepts we could exploit it for computing the state space of models expressed by other formalisms.

Since the sequential TRG builder execution takes several hours on a single commodity hardware machine, even for relatively small examples (e.g. the Gas Burner analysis introduced in section 3.2), we identified independent computational sequences in order to be able to deploy the TRG builder algorithm on distributed environments. Considering the `Map` and `Fold` phases as the building blocks of GRAPHGEN, we could combine them in different ways, obtaining different parallel versions of the sequential algorithm. For example, we can pack together the two phases into a single block called *worker*, and then exploit different workers to implement a classical worker-based algorithm. Otherwise, if we consider the *mapper*s and the *folder*s as different stand alone entities, we can conceive a *Map-Reduce* based algorithm.

These two different ways of organizing parallel computations are described below.

Workers model

This model parallelizes the processing of nodes in the expansion front. A set of independent computational units (`Worker`s, see Fig. 4.3) locally executes the `Map` and `Fold` phases. Each `Worker` computes a portion of the final graph by examining a set of *similar* nodes. The whole state space is partitioned among the `Worker`s by applying to each reachable state a the following function

$$h(a) = \mathrm{Hash}(f(a)) \bmod n \tag{4.3.1}$$

where n is the number of `Worker`s, and f extracts some features from symbolic states ensuring that $f(a) = f(a')$ is a necessary condition for a and a' to be included into one another. More precisely, in our implementations f is an easy to compute abstraction of M, called *soft marking*. As discussed in section 4.4.3, different definitions of soft marking can be helpful to achieve better load balancing of the workload among computational units.

The first definition we used disregards the identity of time-stamp symbols. Another definition will be discussed in Section 4.4.3. Let $|M(p)|$ be the number of tokens in the place p. The soft marking of a state a is defined as:

$$f(a) = \langle |M(p_1)|, ..., |M(p_k)| \rangle \ \in \mathbb{N}^k \tag{4.3.2}$$

where $p_1, ..., p_k$ are the places of the TB net.

Thus, any two nodes possibly related by inclusion are assigned to the same `Worker`. Therefore, each `Worker` is able to locally accomplish the fold operation. Then it sends the mapped nodes for which it is not responsible to the appropriate peers. Fig. 4.3 shows the overall architecture of this model: each `Worker` has its own `remaining` list, which contains nodes not yet examined. The expansion front is now the overall union of all local `remaining` lists.

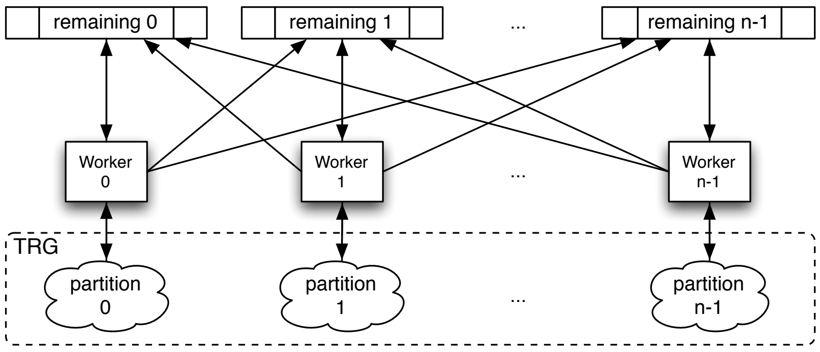


Figure 4.3: `Workers` model.

Mappers & Folders model

The second model specializes the `Workers` in `Mappers` and `Folders` (see Fig. 4.4). A `Mapper` computational unit takes nodes from the expansion front, it maps them to their successors, and assigns the map outcome to the proper `Folders` by means of the Hash function (4.3.1), where n is the number of `Folders`; they in turn identify possible inclusion relationships, and build partitions of the whole final graph.

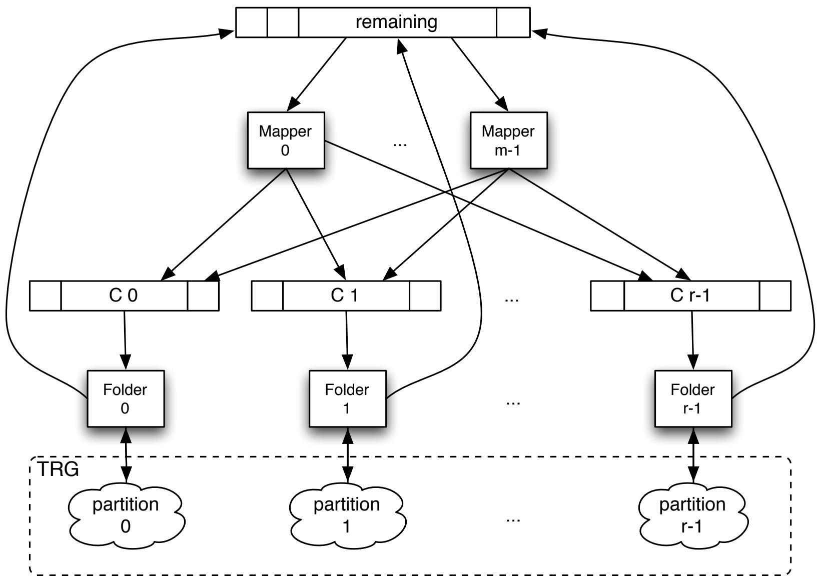


Figure 4.4: `Mappers` & `Folders` model.

It is worth noting that with respect to ordinary state-space exploration techniques, both parallel models incur in additional overheads due to extra communication and synchronization, that may greatly affect speed-up. The main overheads are due to the frequent locking of the data structure recording symbolic nodes (usually implemented by hash tables), and to the load imbalance deriving from the asymmetric computations of `Worker`s.

A conceptually global symbolic structure (the TRG) is partitioned among several computational units, according to the rule that each unit stores a set of nodes and the associated *incoming* edges. This choice makes easier the distributed management of the TRG: the only synchronization point occurs when an already expanded node (with outgoing edges) needs to be erased, as it is absorbed by another (new) one. The required information are not locally present because outgoing edges are stored in the target nodes, which are (usually) assigned to other units.

To further minimize the communications between computational units, we perform a delayed removal of pending edges (outgoing edges of removed nodes) at the end of the global computation. For instance, the node a_0 represented in Fig. 4.5 is included in a_1. The redirection of incoming edges ($a_2 \to a_0$, $a_3 \to a_0$) is locally performed because a_0 and a_1 belong to the same partition. The removal of outgoing edges ($a_0 \to a_4$, $a_0 \to a_5$) instead, cannot be performed locally, being a_4 and a_5 outside the partition i.

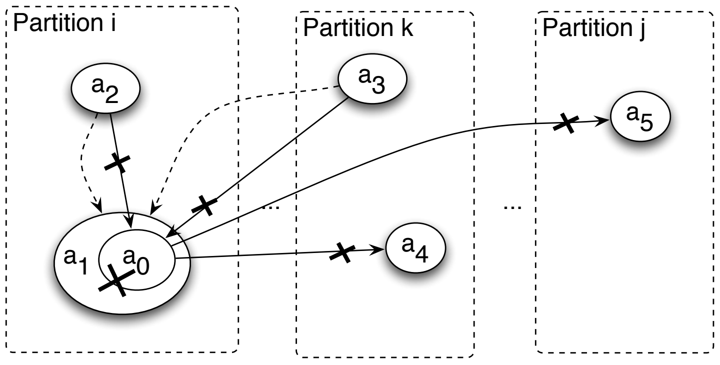


Figure 4.5: Operations on edges during the `Fold` phase.

4.3.2 Distributed implementations

In order to be able to scale our application to a large number of computational units we considered different distributed architectures. In particular we used two consolidated frameworks: JavaSpaces [58] and Hadoop MapReduce [106]. This way we focused on the functional aspects of the application, while leaving to the frameworks the management of fault tolerance and low-level communication. While the JavaSpaces implementation has been designed to run on local networks, *Map-Reduce* has the possibility to be deployed "in the cloud" to exploit the *horizontal scaling*: the dynamic allocation or releasing of resources of the same type. This way, we could exploit a larger number of commodity hardware machines to run massively parallel computations.

JavaSpaces Tool

JavaSpaces technology is a high-level tool for building distributed applications, and it can also be used as a coordination tool. It has its roots in the Linda coordination language [63]. Departing from more traditional distributed models that rely on message passing or RMI, the JavaSpaces model views a distributed application as a collection of processes that use a persistent storage (one or more *spaces*) to store objects and to communicate. Processes coordinate actions by exchanging objects through spaces by means of four primary operations:

- `write()`: Writes new objects into a space.
- `take()`: Retrieves objects from a space.
- `read()`: Makes a copy of objects in a space
- `notify()`: Notifies a specified object when entries that match the given template are written into a space.

By using this framework we have implemented the first parallel model presented in Section 4.3.1 (Fig. 4.3). Each `remaining` list is represented as a space where `Worker` processes exchange states not yet examined. One coordinator process starts the overall computation by producing the initial state, then it is kept waiting for the termination of all `Worker`s in order to merge the computed partition into the final TRG. The whole architecture is presented in Fig. 4.6. Workers iterate `Reduce` and `Map` phases until their own expansion fronts (stored in appropriate spaces) become empty. Worker i `take`s states from the expansion front located on its own space, one at a time:

$$a = \texttt{take}(Space_i)$$

If the `Reduce` phase does not identify any inclusion relationships involving a, the set $\{a'_k\}_{k=1..m}$ of states reachable from a is computed. Workers responsible for their examination (and related spaces) are easily identified by means of the static hash function defined in (4.3.1), thus the correct `write`s can be performed:

$$\texttt{write}(a'_k, Space_{h(a'_k)}),\ k = 1...m$$

After the computation of each worker is completed, each single partition of the state space is written into the coordinator's space. Dashed arrows in Fig. 4.6 represent communications between computing units. They have a different meaning, depending on their direction: an arrow from a space s to a computation entity e means that e can perform `read`/`take` operations on s. An arrow from e to s means that e can perform `write` operations on s.

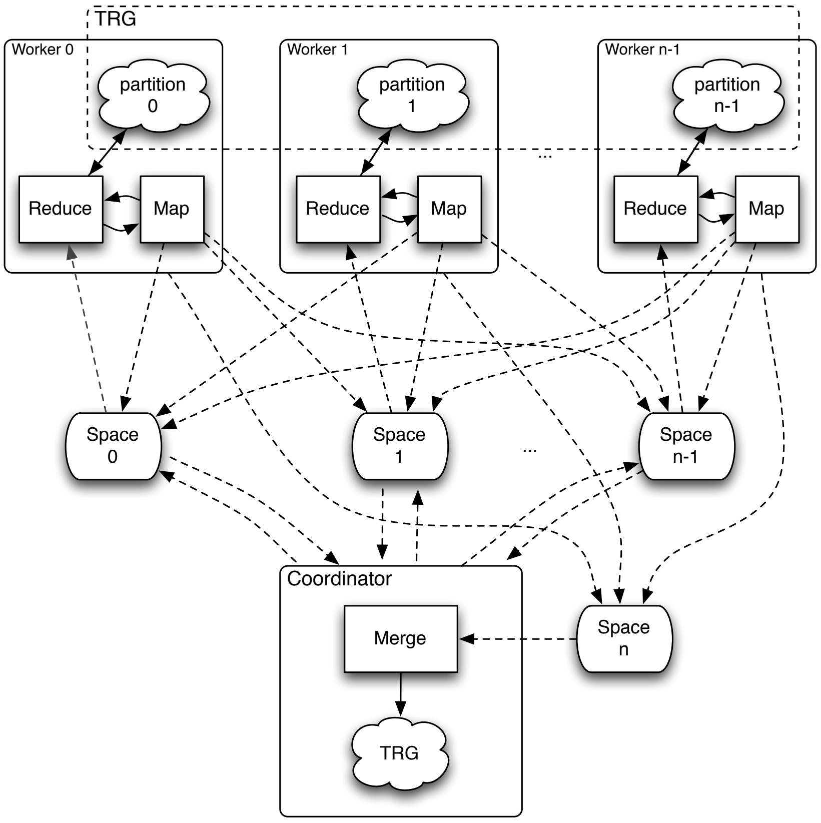


Figure 4.6: Distributed JavaSpaces model.

Hybrid Iterative *Map-Reduce*

This is a distributed implementation of the second parallel model presented in Section 4.3.1 (Fig. 4.4). In order to exploit this programming model we represent our data set as pairs $\langle f(a), a \rangle$, where a is a node of the symbolic TRG with associated incoming edges, $f(a)$ is the soft marking defined in (4.3.2).

We actually used an extended version of the original *Map-Reduce* model introduced in [46]. With respect to such a model, *Map-Reduce* jobs are iterated until the *expansion front* becomes empty. This is called "Iterative *Map-Reduce*" [51]. Each iteration maps all nodes in the expansion front, then it reduces the new nodes by identifying possible inclusion relationships. Note that the reduce phase requires all the TRG nodes in order to identify each potential inclusion relationship between them. For this reason, the input of each iteration is made up by a set of *new* nodes (the expansion front) and a set of *old* nodes (the TRG portion till now computed).

A `Map` takes a pair $\langle f(a), a \rangle$ as input. If it corresponds to an *old* node it is just passed to the reduce phase, without being processed. Otherwise, the set $\{\langle f(a'), a' \rangle\}$ of the states directly reachable from a is computed, and it is passed to the reduce phase together with $\langle f(a), a \rangle$. After the map phase is concluded, an intermediate `shuffle` phase brings together pairs with the same soft marking $f(a)$ and it gives each group to a different `Reduce`. A `Reduce` erases the values (states) that are shown to be included in any others, and it produces in output a set of values forming a partition of the TRG.

The original *Map-Reduce* model also permits one to define a `Combine` function that performs a sort of local reduce on each `Map`'s output, before the actual, distributed reduce phase. A `Combine` runs on the same machine as the related `Map` and it tries to partially aggregate intermediate data in order to improve the overall system performance. In our application we have chosen to discard this optimization because in TB nets context it is very unlikely that symbolic states generated by the same parent share the soft marking [14]. A combine phase before the reduce phase could even affect the application performances. By the way, using other formalisms this observation might be no more valid, and the `Combine` phase could reveal helpful.

Since the *Map-Reduce* model is not the best choice for elaborating a relatively small input, we introduced the possibility of dynamically changing the computational model, depending on the size of analyzed data set. Since the expansion front varies considerably during the TRG construction, it is convenient using a sequential model on a single machine as long as it remains below a given threshold T. When the expansion front exceeds T, an Iterative *Map-Reduce* model on a large cluster of machines is employed. We call this approach, sketched in Fig. 4.7, *Hybrid Iterative Map-Reduce* (himapred in Table 4.1). A hysteresis (H) is programmed, in order to react with some delay in front of possible swings of the expansion front within T.

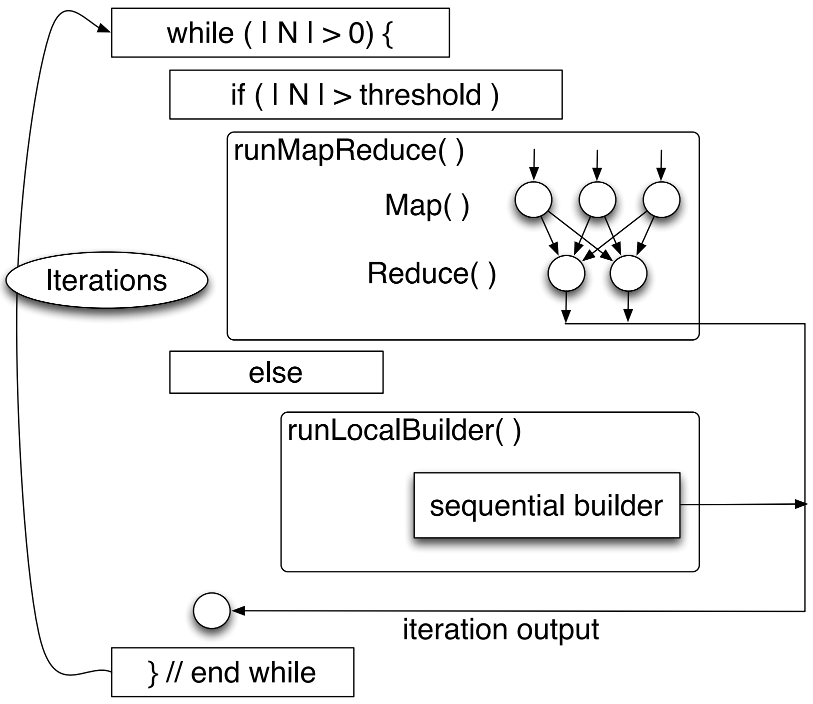


Figure 4.7: Hybrid Iterative *Map-Reduce* model.

Fig. 4.8 shows the expansion front of the Gas Burner analysis over time. The trend line clearly shows how the execution time of a single *Map-Reduce* iteration depends on the TRG size, denoted as $|TRG|$. Since a `Map` processes single sates, its execution time is independent from $|TRG|$ and in many cases it may be neglected. Conversely, a `Reduce` works on a partition of the TRG (checking relationships between any pairs of nodes), thus its complexity is $\mathcal{O}(|TRG|^2)$. The worst case occurs when all nodes in the TRG have the same feature $f(a)$: in that case a single `Reduce` has to process the whole graph. Although the worst case is very unlikely, a common situation is the presence of large clusters of nodes that share the same key $f(a)$. This leads to a computational load imbalance among the reducers often resulting in a significant degradation of performances.

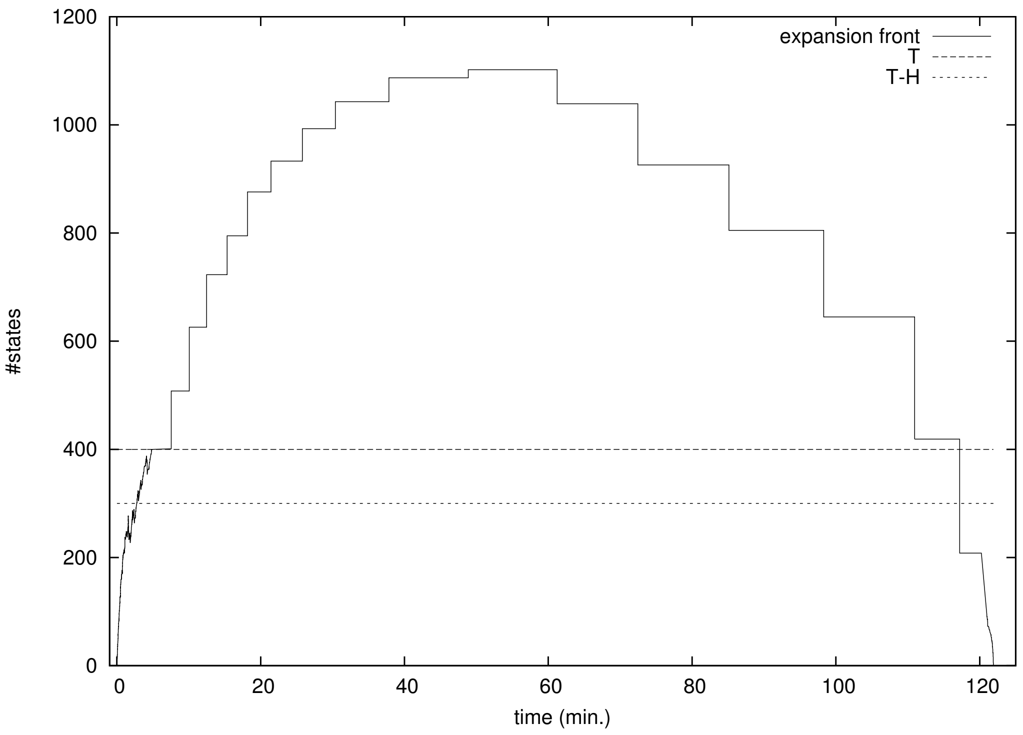


Figure 4.8: Expansion front over time.

4.3.3 Evaluation

The sequential builder produces a graph with 14563 nodes for the Gas Burner example (versus 23635 symbolic states generated during computation), and takes about 7.5 hours on a notebook with a 2.4Ghz Intel Core 2 Duo processor and 4GB of RAM (the operating system is Ubuntu 10.10 and the JVM is OpenJDK IcedTea6 1.9.5). This section adopts the Gas Burner example as a well known benchmark.

Testing activities on the JavaSpaces tool have been performed on a local network (33 computers over a 100Mb Ethernet LAN). Preliminary experiments in this setting show that although performances are much better than the single-thread program on the same environment (the execution time is reduced by a factor ~ 7), there is a major bottleneck preventing further improvements: the state space partitioning among the `Workers` set is not uniform. This means that some computation units are much more loaded than others, which remain idle for most of the time. In order to alleviate this problem, we conceived a different

Table 4.1: TRG Building Experiments Report

architecture	# CPUs	tool version	compute model	T	H	f	exec. time
2.4Ghz Intel Core 2 Duo, 2GB RAM	1×2 cores	sequential	local (single machine)	-	-	(4.3.2)	∼7.5 hrs
3Ghz Intel Pentium 4, 2GB RAM	33×1 cores	JavaSpaces	local (distributed)	-	-	(4.3.2)	1h55m40s
3Ghz Intel Pentium 4, 2GB RAM	33×1 cores	JavaSpaces	local (distributed)	-	-	(4.3.4)	1h2m0s
m2.2xlarge, 13 EC2 [5]	3×4 cores	himapred	cloud	200	50	(4.3.2)	1h35m33s
m1.xlarge, 8 EC2 [5]	8×4 cores	himapred	cloud	200	50	(4.3.2)	1h43m19s
m2.2xlarge, 13 EC2 [5]	8×4 cores	himapred	cloud	200	50	(4.3.2)	1h0m0s
m2.2xlarge, 13 EC2 [5]	8×4 cores	himapred	cloud	200	50	(4.3.4)	39m33s

partitioning policy that allows for a higher degree of parallelism. We used the function defined in (4.3.1) with a different f, called *discriminant soft marking.* Let dm be a function:

$$dm : P \to \mathbb{N}^2,\ dm(p) = \langle i, j\rangle \tag{4.3.3}$$

where p is a place of the analyzed TB net, j is the number of *anonymous* time-stamps in p, and i is the number of *other* time-stamps in p. The discriminant soft marking of a is now defined as:

$$f(a) = \langle dm(p_1), ..., dm(p_k)\rangle\ \in \mathbb{N}^{2k} \tag{4.3.4}$$

This new definition comes from the observation that, even if two states have the same soft marking (4.3.2), they cannot be included into one another if the distribution of *anonymous* time-stamps in the corresponding markings is different.

Fig. 4.9 shows the state space partitioning among 32 `Worker` processes using the two different partitioning policies. The execution time with this new setting is ~ 14 times faster of the sequential one in the same environment.

The last *Map-Reduce* tool has been deployed "in the cloud" by means of the Amazon Elastic MapReduce web service [5] that employs the Amazon Elastic Compute Cloud (EC2) infrastructure. Table 4.1 summarizes the outcomes of the Gas Burner analysis carried out using different distributed frameworks with varying configurations. The results point out the different factors that contribute to improve the performances of our distributed applications: the computational model, the number of computational units, the hardware of each cluster machine, and the partitioning policy. In particular the latter one turns out to be a key factor for the possibility of conveniently scaling the available computation resources.

Because Amazon EC2 is built on commodity hardware, over time there may be several different types of physical hardware underlying EC2 instances. However, the amount of

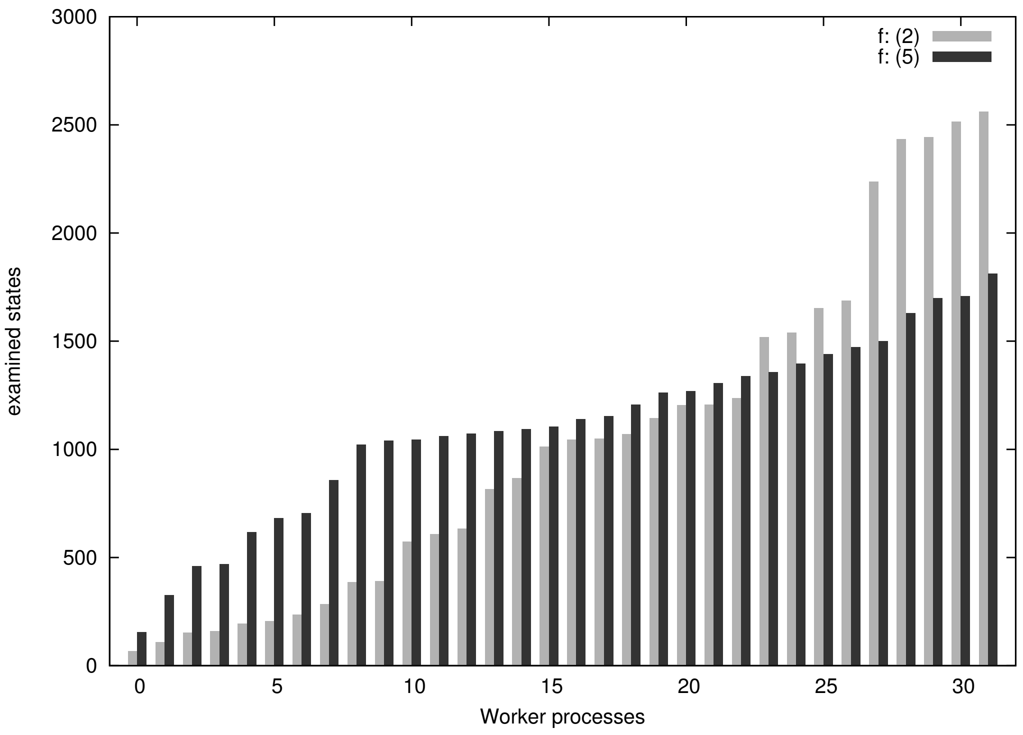


Figure 4.9: State space partitioning among 32 `Workers`.

CPU that is allocated to a particular instance is expressed in terms of EC2 Compute Units: One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. So, there are evident troubles in managing the consistency and the predictability of the performance of an EC2 Compute Unit, and there are also difficulties in understand the overheads introduced by the cloud environment. Thus, is quite difficult to diagnose performance problem in our MapReduce based implementation. Anyway, we obtained the minimum execution time by running this implementation in the cloud over 8 quad-core instances with the capacity of 13 EC2 compute units. But in this case, the execution time is reduced only by a factor ~ 5 (the execution time of the sequential GRAPHGEN tool on the same environment is 2h55m).

4.3.4 Summary

This section presented and discussed two approaches to face the state-space explosion in discrete-event system analysis, based on exploitation of distributed/cloud computing frameworks. These approaches have been experienced on a timed, symbolic reachability analysis of Time Basic (TB) Petri nets. The proposed implementations extend the sequential builder of TB nets' time reachability graph. Standing on a common basic computational schema (a sort of Map & Fold), our approach is general enough to be made parametric to different formalisms, by simply specializing the concepts of *state*, `Map`, `Reduce`, and "soft marking" f. The outcomes of tests performed on a benchmarking RT model clearly show how distributed implementations can be conveniently used to increase the performances of the sequential builder. Although the parallel workers model has shown an higher speed-up with our benchmarking example, the cloud environment can be conveniently used to exploit a large cluster of machines which we may dnot have at our disposal locally. Moreover, in the latter case, we don't need any setup phase of our environment. In fact, concerning the JavaSpaces based tool, we spent several hours setting up the execution environment, while the Hadoop MapReduce based tool allowed the execution in a "push button" like mode, exploiting cloud computing services.

Examples and binaries of the tools described in this section can be found at: `http://camilli.di.unimi.it/graphgen/distributed_computing.html` and `http://camilli.di.unimi.it/graphgen/cloud_computing.html`.

4.4 Simplified Building of Reachability Graphs on Large Clusters

Dealing with complex systems often requires to build of huge reachability graphs, thus revealing all the challenges associated with big data access and management. Thus we require high performance data processing tools that would allow scientists to build very large data structures coming from these analyzed systems. In this section we present MaRDiGraS, a generic framework aimed at simplifying the construction of very large state transition systems on large clusters and cloud computing platforms. Through a simple programming interface, it can be easily customized to different formalisms, for example Petri Nets, by either adapting legacy tools or implementing brand new distributed reachability graph builders. The outcome of several tests performed on benchmark specifications are presented.

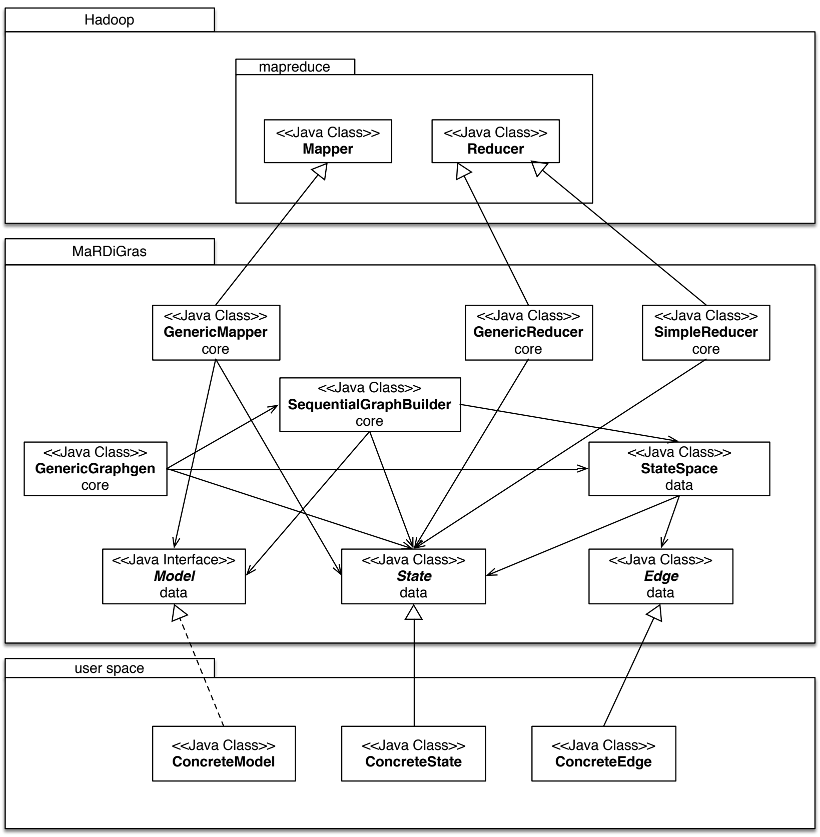

Hadoop
mapreduce
<<Java Class>> Mapper
<<Java Class>> Reducer
MaRDiGras
<<Java Class>> GenericMapper core
<<Java Class>> GenericReducer core
<<Java Class>> SimpleReducer core
<<Java Class>> SequentialGraphBuilder core
<<Java Class>> GenericGraphgen core
<<Java Class>> StateSpace data
<<Java Interface>> Model data
<<Java Class>> State data
<<Java Class>> Edge data
user space
<<Java Class>> ConcreteModel
<<Java Class>> ConcreteState
<<Java Class>> ConcreteEdge

Figure 4.10: Class diagram of the MaRDiGraS framework.

4.4.1 MaRDiGraS

MaRDiGraS follows the *Hybrid Iterative MapReduce* model sketched in Figure 4.7. Computation starts by considering the initial state of the system under analysis and goes on with a sequential state-space building phase until the set N of states not yet explored becomes *large enough*: in other words there is a configurable *threshold* (in terms of number of states) below which a (all-in-RAM) sequential approach is considered more efficient than the distributed one. Once we go above the threshold, an *iterative* MapReduce algorithm runs over a cluster of machines. We carried out several experiments to determine a good setting of the threshold. Experimental evidences suggest that this parameter is strictly related to the number of new nodes created at each iteration: this makes us confident in a possible run-time setting of the threshold.

The *map* step (computation of new states) and the *reduce* step (identification of equivalence/inclusion relationships), iterate until $|N|$ remains above the threshold. Between them, the default partitioner splits the intermediate key set, ensuring that all possibly related states belong to the same partition. This is done by using as intermediate keys a function g such that if two states s_1, s_2 are related then $g(s_1) = g(s_2)$. This way the partitioner delivers all possibly related states to the same reducer. Whenever $|N|$ goes back below the threshold the output of all reducers is merged in order to proceed with the sequential algorithm. This operation might cause a memory overflow in some cases, due to the huge size of the state-space computed until that point (potentially many GB or TB). This is why the user can choose not to switch to the sequential algorithm anymore, after the first exceeding of the threshold. Once the set of unexplored states becomes empty, the entire-state space is supplied as output either in a single file, or distributed over different files.

A simplified class diagram of the MaRDiGraS framework is sketched in figure 4.10. The code base is made up by two main packages which split logically the framework into two different parts: the `data` package and the `core` package. The `data` package contains all entities concerning the data of our framework: the state space, with states and edges, and the model. These entities must be extended in order to be customized to a specific formalism: for example, in the case of time PN extensions, one may want to attach specific meta-data to nodes and edges holding timing properties. The `core` package contains all the algorithms of the framework. They implement, together with the user defined building blocks, the *Hybrid Iterative MapReduce* model.
The `data` package contains the *Model*, the *State* and the *Edge* entities.

Model The *Model* is an interface which should be implemented by the class representing the model under analysis. It contains two methods which must be implemented in order to correctly interpret the user input model and to build the root state of the reachability graph.

- `void buildFromFile(InputStream in)`
  the framework invokes this method to get the internal representation of the model under analysis from the specified file (e.g., XML based representation of the model like PNML format [99]).

- `State buildRoot()`
  this method results in the root state of the system under analysis, given the internal representation of the model; it is invoked by the framework, to initialize the computation

of the entire state space. It returns a `State` object, which is an abstract class extended by the actual implementation of the state concept.

State The *State* is an abstract class which should be extended to instatiate the state concept in a particular formalism. The user can also add properties to this entity, other than the standard ones supplied by the framework: an identifier and a list of incoming edges. The framework largely uses and manipulates these objects during the computation through a few user-implemented methods.

- `List<State> createSuccessors()`
  This method returns a list of new `State` objects representing the states directly reachable from the subject of the call. The framework supplies some relevant information that could be used within the implementation of this method: a unique identifier, which the user could assign to newly created states in order to assign them a unique name; and the representation of the model, needed to compute the successors from the current state, respectively. MaRDiGraS calls this method during the “map” phase in order to compute all new reachable states from the unexplored ones.

- `Relationship identifyRelationship(State s)`
  This method evaluates the actual relationship between (abstract) states sharing some specific features. Possible returned values are: `NONE, EQUALS, INCLUDED, INCLUDES`. It is invoked during the “reduce” phase. Depending on the returned value, the framework discards from the state space those states evaluated included or equal to other ones, modifying all incoming edges of the remaining state, as explained in Section 4.3.

- `String getFeatures()`
  This method provides some state features so that the equality of these features must be a necessary condition for equivalence/inclusion relationships between states. MaRDi-GraS uses this method to compute the key of each intermediate key-value pair. This way the default partitioner assigns all possibly related states to the same reducer. Whenever the equality of computed state features is also a sufficient condition for state equivalence, one should more conveniently use an optimized version of the *reducer* called `SimpleReducer`.

Edge The *Edge* is an abstract class which should be extended to represent the edge concept. The extending class should implement all the properties that the user want to attach to edges.

- `void addLabel(State source, State target)`
  MARDIGRAS invokes this method to initialize an edge between two states. During this stage, we can change the default edge type (`EE` type), and we can attach additional information to the edge, in order to supply the label concept.

The main components of the `core` package work together with the user supplied building blocks to implement the *Hybrid Iterative MapReduce* schema described above. The `GenericMapper` component is in charge of creating new reachable states from unexplored ones; the `GenericReducer` and the optimized version `SimpleReducer` are in charge of identifying relationships between states; the `GenericGraphgen`, manages the entire computation, in particular by deciding when the computational model should be switched from the sequential one to the distributed one, and vice versa. The user supplies all the building blocks by initializing a `GenericGraphgen` object with the following parameters: the `Class` which extends the `State`, the `Class` which extends the `Edge`, and the `Class` which implements the `Model` interface.

SimpleReducer The *reduce* phase can be performed in two different ways: the standard reducer works by evaluating the user-supplied `identifyRelationship` method for each pair of states potentially related. This is a very expensive task and it must be done whenever the actual relationship between two states is unknown, because we supplied a necessary condition, but not sufficient for evaluating the relationship between states. But, if the implementation of the `getFeatures` method gives also a sufficient condition for evaluating state equivalence, the framework already knows that all states sharing a key are equal. In that case `SimpleReducer` should be used, which performs the reduce phase much more efficiently: it simply returns one among the input states, redirecting all incoming edges of the others into that state.

It is worth noting that no particular knowledge on MapReduce and the HADOOP framework is required in order to use MARDIGRAS. The user only cares about the functional aspects of the application, leaving to the framework the management of all other aspects of execution on big clusters. A tool based on MARDIGRAS will produce a set of binary files containing the representation of the state transition system computed from the user's model. The input file format is chosen by the user. In fact, the user implements also the `buildFromFile` method that is in charge to translate the user supplied file into a `Model` instance. The output could be used in turn to extract the knowledge from the analyzed systems: for example to model-check it or to verify particular structural properties on the graph.

The MaRDiGraS framework can be found at `http://camilli.di.unimi.it/mardigras` together with the API description, installation instructions and a working application.

4.4.2 Use Cases

Time Basic PNs As extensively discussed in chapter 3, a classical application area of state-space exploration is the validation of Real-Time systems, that require intensive verification before deployment. Time-Basic (TB) nets [66] (introduced in section 2.2) belong to the category of PNs in which time dependencies are expressed as numerical intervals associated to each transition, denoting the possible firing instants since enabling. Tokens atomically produced by a firing are thereby associated to time-stamps in a given domain (e.g., $\mathbb{R}+$). Transition interval bounds are functions of time-stamps in transition presets and each transitions may be assigned either a weak or a strong semantics. In order to exploit MaRDiGraS to compute the associated abstract state transition system (called TRG) [14], we have extended `State`, `Edge`, and `Model` classes. In particular, TRG states are defined as pairs $\langle M, C \rangle$, where M is an association between *places* and a multi-set of symbols denoting time-stamps, C is a predicate formed by linear inequalities involving such symbols. Labels on edges include the firing transition and the minimum-maximum firing times. Once created all data structure, the application logic has been supplied to the framework by implementing the abstract methods described above. The `createSuccessors` individuates all transition instances enabled in the current TRG state, and for each of those computes a new reachable state; `identifyRelationship` figures out the actual relationship between given states, according to the following sufficient condition for $a \subseteq a'$: $M = M'$ and $C \wedge \neg C' \equiv false$. Depending on the actual computed relationship, the framework modifies the incoming edges' type (either EE, AA, AE or EA). The `getFeatures` method just returns the topological part (M) of a TRG state.

P/T Nets In order to prove the effectiveness of using MaRDiGraSto improve legacy tools, we adapted an existing P/T nets tool: PIPE [48], an open source Java tool (∼82400 lines of code) [1]. PIPE supports the design and analysis of P/T nets with priorities, and their stochastic extension (GSPN). In particular a module is in charge of computing the reachability graph (without any particular smart technique such as decision diagrams, use of structural information, partial order techniques, etc.). For this reason, in such a situation, the memory consumption and the execution time become heavy even during the analysis of relatively small models. In order to exploit the MaRDiGraS framework to overcome these troubles, we first simply identified all those parts of PIPE representing our needed building blocks described in section 4.4.1. Then we encapsulated these blocks with proper *adapter* classes. To adapt the sequential algorithm of PIPE into a distributed one, we just needed 290 lines of code: a very small number also if compared with the dimension of the effectively used PIPE modules (∼6500 lines of code). In this particular implementation, `State`s correspond to reachable *marking*s, `Edge`s are of the type *AA* and they carry on information about firing transitions. The `createSuccessors` method simply identifies all reachable states from a given one, by making all enabled transitions fire. The `getFeatures` method just returns a compact representation of the actual *marking* of the `State`, and because the equality of the *marking* is a necessary and sufficient condition for equality between states, the application can exploits the *SimpleReducer* version.

Well-formed Nets Well-formed Nets (WN) [39] are a power-retaining version of Colored Petri nets characterized by a structured syntax that permits the construction of a quotient graph, called Symbolic Reachability Graph (SRG). The SRG relies on the notion of Symbolic Marking (SM). SMs provide a syntactical equivalence relation on the set of concrete markings. They are formally expressed using dynamic subclasses instead of colors, representing parametric partitions of static subclasses in which WN color classes are in turn partitioned. The SRG is directly built from a given SM, through a symbolic firing rule. By using the *canonical representation* of a SM, the equivalence between SMs boils down to the syntactical identity. In order to exploit the MapReduce based framework for the SRG construction we first need a `SymbolicMarking` extension of `State`, in which `createSuccessors` (according to the symbolic firing rule) simply returns the list of successor SMs of the current SM, in a non-canonical form. Each SRG edge is by construction of kind AA. The `getFeatures` method should return the canonical representation of the current SM. In such a case the reduce phase is similar to the P/T nets case, thus we can exploit the *SimpleReducer* to

[1]The source code of PIPE is available at `http://pipe2.sourceforge.net/`

Table 4.2: MaRDiGraSExperiments report

model	# machines	machine-type	# states	# reducers	threshold	time (m)
gas-burner	4	m2.2xlarge	14563	16	200	95
gas-burner	8	m2.2xlarge	14563	32	200	39
shared-memory	2	m2.2xlarge	1.831×10^6	2	1000	325
shared-memory	4	m2.2xlarge	1.831×10^6	4	1000	163
shared-memory	8	m2.2xlarge	1.831×10^6	4	1000	100
shared-memory	16	m2.2xlarge	1.831×10^6	4	1000	74
simple-lbs	20	m2.2xlarge	4.060×10^8	40	1000	530

fold the incoming lists of equivalent SMs. A possible adaptation of modules of GreatSPN package [38] (written in C), that natively supports the analysis of WN models, is currently under investigation.

4.4.3 Evaluation

The experiments described in this section are executed using the Amazon Elastic MapReduce [5] on the Amazon Web Service cloud infrastructure and are partially supported by "AWS in Education Grant award" [6].

Gas Burner The Gas Burner [14], previously introduced in this thesis, is a benchmark real-time system model specified with a TB Petri net. We specialized MaRDiGraS to obtain the same MapReduce based distributed application introduced in section 4.3. We obtained substantially the same results during the analysis of this example, thus the MaRDiGraS layer does not introduce additional complexity during computation which negatively affects performance. The MaRDiGraS based tool, executed on the input model, generates a graph with 14563 nodes (23635 states are generated during computation) and it takes only 39 minutes, over 8 *m2.2xlarge* machines. Despite the generated graph is quite small, the execution time is 80% faster than the sequential approach running on the same environment (2 hours and 55 minutes). It is worth noting that with this formalism we choose to implement a $getFeature$ function that returns a necessary but not sufficient condition for the inclusion relationships between states, thus since we cannot exploit the *SimpleReducer*, the reduce phase becomes very expensive. For this reason, we gain substantial benefits by increasing the number of reducers, as shown in table 4.3.

Shared Memory This model, introduced in section A.2, is taken from the GreatSPN benchmarks [37, 85]. This P/T net models a system composed of 10 processors competing for the access to a shared memory using a unique shared bus. The PIPE tool fails after more than 20 hours of computation on a *m2.2xlarge* due to an out of memory error (Garbage Collector overhead limit exceeded). In such a situation the benefits deriving from using the adapted tool, as shown in table 4.3, are clear. As we can see, the construction is scalable even for this relatively small state space.

Simple Load Balancing This P/T net, introduced in section A.4, represents a simple load balancing system composed of 10 clients, 2 servers, and between these, a load balancer process. In order to analyze this model, we implemented the building blocks of MaRDiGraS from scratch, to overcome some inefficiencies introduced by PIPE.

As shown in table 4.3, the reachability graph generated is very large: 4.060×10^8 states and 3.051×10^9 arcs for a total size of 120 GB of data. Thus this computation goes clearly beyond the capacity of a single machine. Fig. 4.11 shows the state space dimension over different MaRDiGraS iterations. As we can see, it explodes very quickly, but the computation slows at the end because the number of new states foreach iteration becomes very small. This condition could be tackled for example by considering different optimizations coming from the big data community: In particular we are evaluating the possibility of splitting old and new states into different files, and applying the *schimmy pattern* [94]. This would allow to highly decrease the time required by the last iterations.

4.4.4 Related Work

As described in section 4.1, Several works, in the literature, describe tools and techniques for generating the state space associated to discrete-event systems in a parallel/distributed fashion [36, 41, 42, 88, 25]. However, most of these works are related to a specific formalism, and they do not consider new emerging distributed solutions. Moreover, we considered another important aspect: we wanted to completely remove the costs of deploying our framework into an end-to-end solution, for this reason we developed our software on top of the consolidated Hadoop MapReduce framework. Works presented in [93, 51] describe large-scale graph processing application reformulated in terms of MapReduce programming model, but unfortunately, this large class of graph algorithms doesn't fit well with the state explosion problem in large-scale graph building, which remains rather unexplored. As a common point, both iterate a number of times, using graph states from the previous iteration as input to the next one, until some stopping criterion is met. Thus both use an Iterative

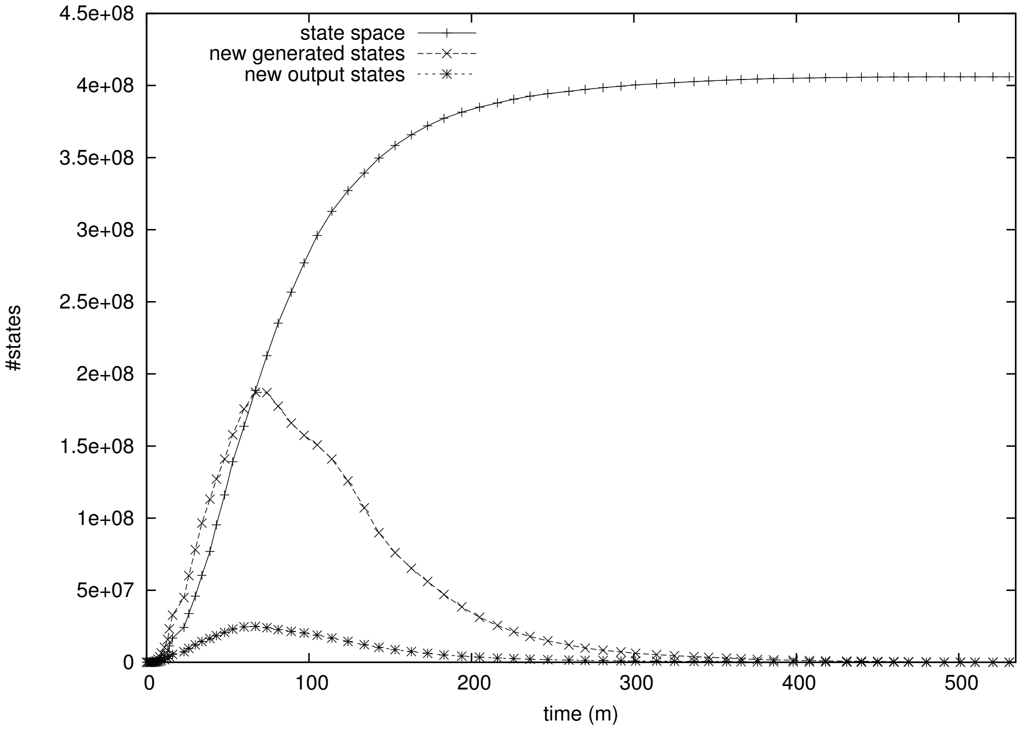


Figure 4.11: Reachability graph computation of the simple-lbs model.

MapReduce approach [51, 17]. But there are also significant differences: first of all, we have to deal with large graph building, not with large graph processing. Second, the input of each iteration is different: in graph processing, it is the internal status of all nodes of the graph; in graph building, it is a portion of the final graph. As a direct consequence, in the latter case, the input dimension greatly varies at each iteration making a standard iterative MapReduce approach ineffective. The input, in graph building, is also partitioned into two different classes of states: "explored states" and "unexplored states" and Mappers must act differently depending on the membership class. Moreover, some key points of graph building algorithms depend on the specific adopted formalisms, thus they must become user defined parameters.

Concerning the formalism independence aspect, some effort has been already shown in few other works. [49] introduces a library that supplies some building blocks which can be combined or replaced at will by users to perform LTL model checking by means of a transition-based generalized Büchi automata approach. It should be noted that the state space generation has been left out of the library, as it is expected to be carried out by third party tools. Thus such a library may be used to build a software tool able to check LTL formulas on state spaces constructed using the MaRDiGraS framework. [73] introduces instead a library to solve reachability problems in a distributed fashion. Similarly to MaRDiGraS it proposes a generic environment dedicated to distribution of any type of state space construction. Anyway it does not support the verification of reachability problems upon abstract state space structures having different relationships among classes of infinite states (introduced in section 2.3). Moreover, the proposed approach does not exploit big data approaches but were designed to run upon distributed local environment or multi-core machines. Experiments shows that, using 22 machines equipped with a dual Xeon hyper-threaded at 2.8GHz processor and 2GB of RAM, more than 40 hours were needed to construct a state spaces sized with a 10^7 order of magnitude. As shown in section 4.4.3, this is far higher with respect to the time required by a MaRDiGraS based software tool running on a similar environment to deal with larger state spaces.

4.4.5 Summary

This section introduced MaRDiGraS: a generic framework which can easily adapted for tackling the state explosion problem within the computation of the reachability graph associated to different formalisms. This framework exploits techniques typically used by the big data community and so far poorly explored for this kind of issues. Thanks to its very simple programming interface, it provides a powerful tool for constructing high-performance distributed applications without the need to deal with the complex communication and synchronization issues required for exploiting a computation distributed on large clusters. Our experiments report that MaRDiGraS can be used effectively to compute state spaces sized with different orders of magnitude. We believe that this work could be a first step towards a meeting between two very different, but related communities: the "formal methods" community and the "big data" community. Exposing this issue to scientists with different backgrounds could stimulate the development of new interesting and more efficient solutions. We believe MaRDiGraS, thanks to its very simple programming interface, provides a powerful tool for constructing distributed applications: indeed it was easy to use it for implementing a distributed version of an existing sequential tool (PIPE) that was able to analyze a model beyond the capacity of a single machine. MaRDiGraS is flexible enough to be used with rather different formal models.

It is worth noting that this framework can be exploited as a basic component of a generic library for distributed model checking. In particular we developed a software tool which exploits the MaRDiGraS computed graphs by applying iterative map-reduce algorithms based on fixed point characterizations of the basic temporal operators of CTL (Computation Tree Logic). This software tool is described in the next section.

Anyway, several questions remains open and require further investigation: for example, could a dynamic programming approach help in choosing partitions and/or thresholds? How the proposed computational model can be optimized when the number of new states gets very small? Are there classes of formalisms for which this approach cannot be used? And how can we adapt it to these classes?

4.5 CTL Model Checking Using MapReduce

In this section we continue the exploration of big data approaches to formal verification. Given a very large state space, we are now interested in extracting the knowledge from these very large data structures in order to verify specific properties on the analyzed models. In particular we introduce a framework to ease the adoption of a distributed approach to verification of Computation Tree Logic (CTL) formulas on very large state spaces. The approach exploits/integrates the parametric state-space builder MaRDiGras (section 4.4).

The whole framework adopts MapReduce as core computational model, and can be tailored to different modeling formalisms. The outcomes of several tests performed on (Petri-nets based) benchmark specifications are presented, thus showing the convenience of the proposed approach.

4.5.1 Computation Tree Logic

CTL [44] is a branching-time logic which models time evolution as a tree-like structure where each moment can evolve in several different possible ways. In CTL each basic temporal operator (X, F, G) must be immediately preceded by a path quantifier (either A or E). If AP is the set of atomic propositions, and $p \in AP$, CTL formulas are inductively defined as follows:

$$\begin{aligned} \phi &::= & p \mid \neg\phi \mid \phi \vee \phi \mid A\psi \mid E\psi & \qquad \text{(state formulas)} \\ \psi &::= & X\phi \mid F\phi \mid G\phi \mid \phi U \psi & \qquad \text{(path formulas)} \end{aligned}$$

The universal path operator (A) and the existential path operator (E) express that a property is valid for all paths and for some paths, respectively. The temporal operators *next* (X) and *until* (U) express that a property is valid in the next state, and until another property becomes valid, respectively; moreover the operators *finally* (F) expresses that a property becomes eventually valid in a future state and *globally* (G) expresses that a property is valid along the entire subsequent path. The interpretation of a CTL formula is defined over a *Kripke structure*, (*i.e,* a *state transition system*). A Kripke structure is made up by a finite set of states, a set of transitions (*i.e.,* a relation over the states), and a labelling function which assigns each state the set of atomic propositions that are true in that state. Such a model describes the system at any instant corresponding to a state; the transition relation describes how the system evolves from a state to another in a single time step.

Definition 4.5.1 (Kripke structure) A Kripke structure T is a tuple $\langle S, S_0, R, L\rangle$, where:

1. S is a finite set of states.
2. S_0 is the set of initial states.
3. $R \subseteq S \times S$ is a total transition relation, that is: $\forall s \in S\ \exists s' \in S$ s.t. $(s, s') \in R$

4. $L : S \to 2^{AP}$ labels each state with the set of atomic propositions that hold in that state.

The totality in the third point imposes the *seriality* of the transition relation. This means that the system cannot have deadlock states. In case of deadlocks, this condition can be always ensured by adding an "error" livelock state (with one outgoing transition directed to the state itself).

A path σ in T from a state s_0 is a infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots$ where $\forall i \geq 0,\ (s_i, s_{i+1}) \in R$.

Definition 4.5.2 (Satisfiability) Given a CTL formula ϕ and a state transition system T with $s \in S$, T satisfies ϕ in the state s (written as $T \models_s \phi$) if:

- $T \models_s p$ iff $p \in L(s)$.
- $T \models_s \neg\phi$ iff $T \not\models_s \phi$.
- $T \models_s \phi \wedge \psi$ iff $(T \models_s \phi \wedge T \models_s \psi)$.
- $T \models_s \phi \vee \psi$ iff $(T \models_s \phi \vee T \models_s \psi)$.
- $T \models_s EX\phi$ iff $\exists t$ s.t. $R(s,t) \wedge T \models_t \phi$.
- $T \models_s EG\phi$ iff $\exists$ a path $s_0 s_1 s_2 \ldots$ s.t.: $\forall i \geq 0, T \models_{s_i} \phi$.
- $T \models_s E[\phi U \psi]$ iff $\exists$ a path $s_0 s_1 s_2 \ldots$ s.t.:
  $\exists i \geq 0, (T \models_{s_i} \psi) \wedge (T \models_{s_j} \phi\ \forall j < i)$.

We can also write $T \models \phi$ which means that T satisfies ϕ in all the initial states of the system.

It can be shown that any CTL formula can be written in terms of $\neg, \vee, EX, EG$, and $E[\phi U \psi]$, for example $AX\phi$ is $\neg EX \neg\phi$, $EF\phi$ is $E[True\ U\ \phi]$, and so forth. The possible combinations are only eight: $AX, EX, AF, EF, AG, EG, AU, EU$.

The semantics of some widely used CTL operators is exemplified in Figure 4.12.

Definition 4.5.3 (Model Checking) Let T be a Kripke structure and let ϕ be a CTL formula. The model checking problem is to find all the states $s \in S$ such that $T \models_s \phi$.

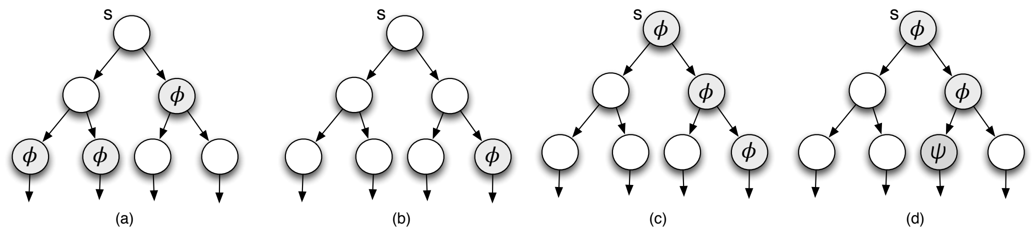


Figure 4.12: (a) $T \models_s AF\phi$; (b) $T \models_s EF\phi$; (c) $T \models_s EG\phi$; (d) $T \models_s E[\phi U \psi]$

4.5.2 Fixed-Point Algorithms

One of the existing model-checking algorithms is based on fixed-point characterizations of the basic temporal operators of CTL [43] (similar ideas can be used for model checking Linear Temporal Logic). Let $T = \langle S, S_0, R, L\rangle$ be a Kripke structure. The set $\mathcal{P}(S)$ of all subsets of S forms a lattice under the ordering by set inclusion. For convenience, we identify each state formula with the set of states in which it is true. For example, we identify the formula *false* with the empty set of states, and we identify the formula *true* with S (the set of all states). Each element of $\mathcal{P}(S)$ can be viewed both as a set of states and as a state formula (a predicate). Formally, given a CTL formula ϕ we can define:

$$[\![\phi]\!]_T := \{s \in S \ : \ T \models_s \phi\}$$

This way, we can associate set operators to boolean connectors:

$$[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!],\ [\![\phi_1 \vee \phi_2]\!] = [\![\phi_1]\!] \cup [\![\phi_2]\!], [\![\neg\phi]\!] = S \setminus [\![\phi]\!]$$

The set of states identified by the temporal operator EX, can be defined trivially if we consider the preimage with respect to the relation R.
Given $W \in \mathcal{P}(S)$:

$$R^-(W) := \{s \in S \ : \ \exists s'(R(s,s') \wedge s' \in W)\}$$

$$[\![EX\phi]\!]_T = R^-([\![\phi]\!]_T) \tag{4.5.1}$$

Let's now consider a function $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ called *predicate transformer*.

Definition 4.5.4 (Fixed-Point) We say that a state formula X is the *least fixed-point* μ_X (or respectively the *greatest fixed-point* ν_X) of a predicate transformer τ iff (1) $X = \tau(X)$, and (2) for all state formulas X', if $X' = \tau(X')$, then $X \subseteq X'$ (respectively $X \supseteq X'$).

Definition 4.5.5 (Monotonic Predicate Transformer) A predicate transformer τ is *monotonic* iff for all $X, X' \in \mathcal{P}(S)$, $X \subseteq X'$ implies $\tau(X) \subseteq \tau(X')$.

A monotonic predicate transformer on $\mathcal{P}(S)$ always has a least fixed-point and a greatest fixed-point (by Tarski's Fixed-Point Theorem [105]). The temporal operators EG and EU can each be characterized respectively as the greatest and the least fixed-point of two different monotonic predicate transformers:

$$[\![EG\phi]\!]_T = \nu_X([\![\phi]\!]_T \cap R^-(X)) \tag{4.5.2}$$

$$[\![E[\phi U \psi]]\!]_T = \mu_X([\![\psi]\!]_T \cup ([\![\phi]\!]_T \cap R^-(X))) \tag{4.5.3}$$

We can calculate the least fixed-point of a monotonic predicate transformer: $\mu_X(\tau(X))$ as follows. We define $X_0 = \emptyset$ and $X_i = \tau(X_{i+1})$ for $i \geq 1$. We first compute X_1, then X_2, then X_3, and so forth, until we find a k such that $X_k = X_{k-1}$. It can be proved that the X_k computed in this manner is the least fixed-point of τ. To compute the greatest fixed-point, we follow a similar procedure but starting by setting X_0 as the whole S set. Pseudocode for this procedure is shown by Algorithm 4.

Algorithm 4 Least Fixed-Point Procedure

```
1: function LFP($\tau$)
2:     $X := \emptyset$
3:     while $X \neq \tau(X)$ do
4:         $X := \tau(X)$
5:     end while
6:     return $X$
7: end function
```

4.5.3 Distributed Algorithms

This section introduces our distributed approach that enables formal verification to exploit distributed and cloud computing facilities. The distributed algorithms compute formulas of type EX, EG, and $E[\phi U \psi]$ since any other CTL formula can be reformulated in terms of these three basic operators. Algorithms described in this section were implemented on top of the Hadoop MapReduce framework [106]. The proofs of correctness of the algorithms are reported in appendix B.

Distributed State Space Generation

The idea underlying a distributed algorithm for state space exploration is that of using multiple computational units to explore different parts of the whole state space in parallel. This task is typically performed by using *Worker*s-based algorithms [88]: states are partitioned among workers by means of a static hash function; workers explore successor states and assign them to the proper computational units. Communication among different machines is usually implemented through message passing. Since state space partitioning is a known critical issue, different load balancing techniques and compact state representations [89, 60] were set up.

Recent studies have also shown the convenience of exploiting big data approaches and cloud computing facilities to accomplish this task. A framework (called MaRDiGraS) [18] was recently developed to ease implementing distributed state space builders for different formalisms. Given a cluster of n machines, a MaRDiGraS-based application generates n files $F_1, F_2, ..., F_n$ storing a partition of the whole state space. MaRDiGraS supports *symbolic* state representations, thus a single state can actually represent an aggregate. Let S be the set of reachable states: the set of states emitted by the ith computational unit is $S_i = \{s \in S : \text{Hash}(f(s)) = i\}$, where $f : S \to D_S$ is a user supplied function such that,

$$\forall s, s' \in S \; s \subseteq s' \vee s \supset s' \Rightarrow f(s) = f(s')$$

I.e., f associates states with specific features (represented by the domain D_S) that must coincide for each pair of states related by inclusion.

What makes developing a distributed model checker on top of MaRDiGraS easy and convenient is its particular implementation of transition relation (R). Each state in fact stores locally all incoming transitions as a list of state identifiers (ids). Therefore, given $W \subseteq S$, the backward (or predecessor) set $R^{-}(W)$ can be very efficiently computed. It is composed by all the states whose ids are in:

$$\bigcup_{i} \bigcup_{s \in W_i} R^{-}_{id}(s)$$

where R^{-}_{id} gives the set of ids associated with predecessors states, and W_i represents the partition of W processed by the i-th mapper. All the predecessors ids are then emitted in parallel and sorted during the shuffle phase, along with the associated states.

This is a key point of our distributed approach, because the MapReduce based algorithms compute the R^{-} function very often during the map phase. The fact that the set of predecessors is locally available in each state, without the need of communication among computational units, lowers both network traffic and synchronizations, thus speeding up the computation.

The fixed-point algorithms work on transition systems preserving the *seriality* of the transition relation (Section 4.5.1). Otherwise, the MaRDiGraS framework produces a complementary output file containing a single “error” livelock state, where the list of incoming transitions includes the error state and all deadlocks.

EX **Formulas**

The computation of $[\![EX\phi]\!]_T$ relies on the assumption that the set of states satisfying ϕ has already been computed: ϕ can be either a formula locally evaluable or a complex sub-formula previously evaluated. This task can be performed in a single MapReduce job where the predecessor states of $[\![\phi]\!]_T$ are evaluated in parallel. The input is composed of two separated sets of files. One storing all the states belonging to $S \setminus [\![\phi]\!]_T$, the other storing all the states belonging to $[\![\phi]\!]_T$. This way the mappers can evaluate and emit in parallel the identifiers of the states belonging to $R^{-}([\![\phi]\!]_T)$: as shown in Algorithm 5 (lines 2-4), the `Map` function associates the identifiers of these states with a particular “empty” value $\perp$ (line 4). In addition, each mapper emits its input (line 7). After the shuffle phase, all the values with the same identifier are brought together so that the `Reduce` function can emit the final output by just checking for the occurrence of the empty value in the input list (lines 10-11). The Hadoop MapReduce framework transparently handles the *emitIntermediate* function in order to produce all the intermediate key-value pairs forming shuffler’s input. The Reduce function instead uses the *emit* routine to produce the final output in the form of binary files of the Hadoop Distributed filesystem [106]. Each *reducer* produces its own output file that can be either retrieved by the user or re-processed by the framework in order to evaluate a more complex formula.

Algorithm 5 MapReduce-based $EX[\phi]$ evaluation

```
function MAP($k, s$)
    if $s \in [\![\phi]\!]_T$ then
        for $e \in R^{-}(s)$ do
            $emitIntermediate(e, \bot)$
        end for
    end if
    $emitIntermediate(k, s)$
end function
function REDUCE($k, list(states)$)
    if $(\bot \in list) \ \wedge \ (s \neq \bot \in list)$ then
        $emit(k, s)$
    end if
end function
```

EG Formulas

The operator $[\![EG\phi]\!]_T$ is likewise computed. The evaluation of the final result is just a bit more complex than in the previous case. Our approach is based on the greatest fixed-point characterization of the monotonic predicate transformer (4.5.2). Thus an iterative MapReduce algorithm is used, where at each iteration the predicate transformer is computed on the output of the previous iteration, until a fixed-point is reached. Algorithm 6 shows the `Map` and the `Reduce` functions employed during job iterations. The input of each MapReduce job is made up by a set of files containing $[\![\phi]\!]_T$ and another set of files X representing the current evaluation of the formula. Since at the beginning $X = S$ and $R^{-}(S) = S$, and we know the result of the first evaluation of the predicate transformer (4.5.2), we directly start iterating by setting $X = [\![\phi]\!]_T$. As shown in Algorithm 6, the map phase emits in parallel all the predecessor states of X (lines 2-4) and all the states belonging to $[\![\phi]\!]_T$ (lines 7-8). This way, the reduce phase can verify and emit in parallel all the predecessor states belonging to $[\![\phi]\!]_T$ (lines 12-13). The iteration goes on until either we reach the empty set or the number of output key-value pairs of two consecutive jobs is equal.

$E[\phi U \psi]$ Formulas

The approach to compute $[\![E[\phi U \psi]]\!]_T$ is similar to the previous one(s). It is based on the least fixed-point characterization of the monotonic predicate transformer (4.5.3). As usual, we assume that the states corresponding to ϕ and ψ have been pre-computed. The iterative fixed-point algorithm uses the MAP and the REDUCE functions presented in Algorithm 7. The input of each iteration is made up by a set of files storing the current evaluation of the

Algorithm 6 MapReduce-based $EG[\phi]$ evaluation

```
 1: function MAP($k, s$)
 2:     if $s \in X$ then
 3:         for $e \in R^{-}(s)$ do
 4:             $emitIntermediate(e, \bot)$
 5:         end for
 6:     end if
 7:     if $s \in [\![\phi]\!]_T$ then
 8:         $emitIntermediate(k, s)$
 9:     end if
10: end function
11: function REDUCE($k, list(states)$)
12:     if $(\bot \in list) \ \wedge \ (s \neq \bot \in list)$ then
13:         $emit(k, s)$
14:     end if
15: end function
```

formula (X) and another set storing $[\![\psi]\!]_T$. Since at the beginning $X = \emptyset$, and the predicate transformer (4.5.3) results in $[\![\psi]\!]_T$, iteration is initialized by setting $X = [\![\psi]\!]_T$. The map phase computes in parallel all predecessor states of X (lines 2-4) and forwards $[\![\phi]\!]_T \cup [\![\psi]\!]_T$ to reducers (lines 7-8). The reduce phase emits in parallel all predecessor states satisfying $[\![\phi]\!]_T$ and all states satisfying $[\![\psi]\!]_T$ (lines 12-14).

As an optimization, the map phase actually computes $R^{-}(X_i \setminus X_{i-1})$ because it can be easily shown that $R^{-}(X_{i-1}) \subseteq R^{-}(X_i)$. In fact, all the states belonging to X_{i-1} belong also to X_i, being the predicate transformer in 4.5.2, monotonic increasing.

Algorithm 7 MapReduce-based $E[\phi U \psi]$ evaluation

```
 1: function MAP($k, s$)
 2:     if $s \in X$ then
 3:         for $e \in R^{-}(s)$ do
 4:             $emitIntermediate(e, \bot)$
 5:         end for
 6:     end if
 7:     if $s \in [\![\phi]\!]_T \vee s \in [\![\psi]\!]_T$ then
 8:         $emitIntermediate(k, s)$
 9:     end if
10: end function
11: function REDUCE($k, list(states)$)
12:     if $(s \neq \bot \in lis)$ then
13:         if $(\bot \in list \ \wedge \ s \in [\![\phi]\!]_T) \vee (s \in [\![\psi]\!]_T)$ then
14:             $emit(k, s)$
15:         end if
16:     end if
17: end function
```

4.5.4 Model-checking of Abstract Transition Systems

We have till now implicitly assumed that CTL formulas are checked on ordinary Kripke structures: let us shortly discuss about how and whether we can handle abstract structures.

Let $T_s : \langle S, s_0, R \rangle$ be an ordinary state-transition system. We call $\langle A, a_0, \Gamma \rangle$ an abstract representation of T_s if A is finite and each state $a \in A$ symbolically represents a (possibly infinite) aggregate of ordinary states sharing some features: more precisely, if $s_0 \in a_0$ (normally $a_0 = \{s_0\}$), $\bigcup_{a \in A} a \supseteq S$, and the abstract transition relation Γ satisfies condition $\exists\exists$ (or EE, introduced in section 2.3).

The first part of condition EE avoids two abstract states from being connected, if no corresponding ordinary states are. The second part ensures that each ordinary transition path has an abstract representative.

If the abstract states form a partition of S we speak of quotient graph. Instead, if they globally represent a superset of ordinary reachable states we speak of coverage graph.

If we are interested in properties other than state reachability, however, we should put stronger requirements on abstract transitions (edges) $\langle a, a' \rangle \in \Gamma$:

$$\begin{aligned}
\exists\forall\ (EA) &\iff \forall s' \in a', \exists s \in a, \langle s, s' \rangle \in R \\
\forall\exists\ (AE) &\iff \forall s \in a, \exists s' \in a', \langle s, s' \rangle \in R \\
\forall\forall\ (AA) &\iff (\forall s \in a, \exists s' \in a', \langle s, s' \rangle \in R)\ \wedge\ (\forall s' \in a', \exists s \in a, \langle s, s' \rangle \in R)
\end{aligned}$$

The default MaRDiGraS version tries to achieve the maximum state aggregation by checking for inclusion relationships between generated abstract states, inferring conditions on edges during the building phase itself. For instance, if a is an already expanded node and a' is a newly generated one such that $a' \supset a$, then incoming edges of a (previously inferred) of kind $?A$ are redirected to a', and relabelled as $?E$. Conversely, if we check that $a' \subset a$, a new edge $\langle a'', a \rangle$ of kind $?E$ is added to the graph, a'' being the predecessor of a.

Resulting final graphs may thus have edges differently labelled as EE, EA, AE, or AA. As we will explain, it is possible to configure the MaRDiGraS algorithm so that *all* abstract edges fulfill a condition other than the default one (EE). By the way this may affect the achieved state aggregation, and introduces a computation overhead, strictly dependent on the adopted formalism, due to the need of checking for even partial overlapping relationships between abstract states, not just inclusions.

Let $K : \langle A, A_0, \Gamma, L\rangle$ be a Kripke structure corresponding to an abstract state-transition system generated by MaRDiGraS. Two extra conditions are necessary so that the map-reduce approach to CTL model checking continues working.

- labelling L, which associates abstract states with (atomic) formulas holding in them, must be such that $\phi \in L(a) \Leftrightarrow \forall s \in a, \phi_s$, where ϕ_s is the interpretation of formula ϕ in ordinary (concrete) state s. This very intuitive requirement just points out that formulas defined on symbolic states may involve variable symbols in turn. An example will be given next.

- Abstract edges must be of kind AE (by the way $AA \Rightarrow AE$). This requirement has two, related, motivations. First, it extends the *seriality* constraint to the transition relation of the underlying ordinary state-transition system. Not surprisingly, it also ensures that the map-reduce algorithms to compute basic temporal operator EX, EG, and $E[\phi U \psi]$ remain valid: in fact each round (given a pre-computed set of states corresponding to a sub-formula) just relies on exactly determining the inverse image Γ^{-1} of $Y \subseteq A$. Under the assumption above $\Gamma^{-1}(Y) \equiv \bigcup_{a \in Y, s \in a} R^{-1}(s)$, i.e., the abstract set of predecessors actually corresponds to the concrete set of predecessors. An abstract state space which natively meets condition AA, and that could be easily reproduced in a distributed version using MaRDiGraS, is the the Symbolic Reachability Graph of Symmetric Nets [79], a particular quotient graph which outlines/exploits behavioural system symmetries.

Since in general, modelling formalisms, especially when they include time specifications, don't guarantee by default property AE on their abstract contractions, and final users are likely to be interested in checking arbitrary state formulas, one must be able to configure the MaRDiGraS generic builder so that both requirements above are met in generated graphs.

Just to give an idea of what splitting an abstract state means, let us consider the state-transition graph associated with Time-Basic Petri nets [14], whose nodes are pairs $\langle M, C\rangle$, where M (marking) is the topological description of a system state, formally defined by a finite set of *places*, each associated with a multi-set of symbols denoting time-stamps; C is a set of linear inequalities involving such symbols, *e.g.,* $T_2 - T_0 \leq 1.5 \wedge T_0 \leq T_1$. A concrete state corresponds to any assignment of $\{T_i\}$ with values in $\mathbb{R}^+$ which makes C true. Assuming for simplicity that time-stumps' subscripts refer to the holding places (i.e., using a Petri nets, parlance, the model is 1-safe), a formula which is valid in that state is

$T_2 - T_1 \leq 1.5$. But if we consider the formula $\phi : T_1 \leq T_2$, we argue that it is not implied by C. If we want to check this formula, we need to split $\langle M, C\rangle$ in $\langle M, C \wedge T_1 \leq T_2\rangle$ and $\langle M, C \wedge T_1 > T_2\rangle$.

A quite different, yet helpful, approach that we are evaluating consists of refining a previously generated abstract state space, so that it become CTL model-checkable. Such an approach would have some resemblance with [23], where a technique for time Petri nets is proposed, based on applying consolidated partition refinement techniques on a (preliminarily built) compact contraction of the ordinary state-space. In our case this task is complicated by the fact that we have to deal with a distributed abstract representation of the state space.

4.5.5 Complexity

A formal representation of the MapReduce protocol has been introduced in [82], from the complexity theory perspective. In particular a R rounds M_R machine is defined as an alternating list of mappers and reducers $(\mu_1, \rho_1, \dots, \mu_R, \rho_R)$, whose execution follows these steps:

1. Letting U_{r-1} be the list of key-value pairs processed from the last round (or the input pairs when $r = 1$), the map phase applies μ_r in parallel to each key-value pair of U_{r-1} to produce the multi-set $V_r = \bigcup_{\langle k,v\rangle \in U_{r-1}} \mu_r(k, v)$.

2. The shuffle phase sorts and gathers intermediate values by key, producing a set of intermediate *key-list of values* pairs $\langle k, V_r^k = \{v_1, v_2, \dots\}\rangle$.

3. The reduce phase applies ρ_r in parallel foreach V_r^k. The output U_R is $\bigcup_k \rho_r(k, V_r^k)$.

we say that a R rounds M_R machine accepts the input $\langle x\rangle$ if in the final round $U_R = \emptyset$. Moreover, M_R decides a language L if it accepts $\langle x\rangle$ if and only if $x \in L$. A language L is in $\mathcal{MRC}[f(n), g(n)]$ if there are a constant $0 < c < 1$, two $\mathcal{O}(n^c)$-space and $\mathcal{O}(g(n))$-time Turing machines M, M', and $R = \mathcal{O}(f(n))$, such that $\forall x \in \{0,1\}^n$ the following holds.

1. Given $\mu_r = M$, $\rho_r = M'$, the machine $M_R = (\mu_1, \rho_1, \dots, \mu_R, \rho_R)$ accepts x iff $x \in L$.

2. Each μ_r outputs $\mathcal{O}(n^c)$ distinct keys.

Function $f(n)$ represents the number of times global synchronization has to be performed, $g(n)$ represents the computing time spent by each processing unit (mapper/reducer), finally space bounds ensure that the size of data on each processing unit is smaller than the full input.

Denoting the class $\mathcal{MRC}[poly(n), poly(n)]$ with $\mathcal{PMRC}$, it can be proven that $\mathcal{PMRC} \subseteq \mathcal{P}$ [82]. In fact, a polynomial time Turing machine can trivially simulate a M_R machine. It should just perform the intermediate grouping by key manually, and sequentially run the μ_r and ρ_r functions as subroutines. Thus intuitively, the $\mathcal{PMRC}$ class represents those problems in $\mathcal{P}$ that can be efficiently solved by a M_R machine. In any case, it is unknown whether $\mathcal{P} \subseteq \mathcal{PMRC}$ or not. Similarly, the relationship between $\mathcal{PMRC}$ and $\mathcal{NC}$ [87] has not yet been established. Only a partial answer has been given to this question by showing that a large class of problems in $\mathcal{NC}$ are in $\mathcal{PMRC}$ [82].

Concerning the algorithms for EG and EU, the total number of rounds is bounded by the number of backward steps performed during the fixed point computation ($\mathcal{O}(|S|)$), even if we experimentally verified that in most practical cases $f(n) << n$. The time spent by each computing unit instead is given by the *in-degree* of the state-space. In the worst case it is $\mathcal{O}(n^2)$: this situation occurs only when the state-space contains global "hubs" of incoming transitions. Although this case is feasible, especially in the context of free networks [7], it is very unlikely. In fact, state spaces are very similar to random graphs, which have Poisson distribution of degrees. The average degree is very small, typically around three, or even smaller than two, since there are many vertices with degree one [98, 76]. Thus, in the general case, both the algorithms fall in the $\mathcal{MRC}[n, 1]$ class. The EX falls instead in the $\mathcal{MRC}[1, 1]$ class, since a single backward step is required.

All these observations are not surprising at all. Since both μ_r and ρ_r are $\mathcal{O}(1)$ in the general case, the overall complexity of a polynomial Turing machine simulating M_R in verifying a composite formula φ is given by the number of backward steps foreach sub-formula in φ. This is consistent with the known CTL model checking complexity, $\mathcal{O}(|S| \times |\varphi|)$ [43], which corresponds to the time bound of a sequence of M_R runs, one foreach sub-formula.

4.5.6 Evaluation

The experiments described in this section were executed using the Amazon Elastic MapReduce [5] on the Amazon Web Service cloud infrastructure. They were supported by an "AWS in Education Grant award" [6]. In particular all runs have been performed on clusters of *m2.2xlarge* computational units [5] of varying size.

Table 4.3: Shared memory ($\sim 10^6$ states, $\sim 10^7$ transitions) analysis report

property	$\lvert [\![property]\!]_T \rvert$	# machines	time *(s)*
$EX[\phi]$	2.135×10^5	1	70
$EX[\phi]$	2.135×10^5	2	67
$EX[\phi]$	2.135×10^5	4	50
$EX[\phi]$	2.135×10^5	8	38
$EG[\psi]$	0	1	67
$EG[\psi]$	0	2	55
$EG[\psi]$	0	4	58
$E[\omega\ U \rho]$	1.831×10^6	1	1898
$E[\omega\ U \rho]$	1.831×10^6	2	1124
$E[\omega\ U \rho]$	1.831×10^6	4	839
$E[\omega\ U \rho]$	1.831×10^6	8	564
$E[\omega\ U \rho]$	1.831×10^6	16	509

As a proof of concept, we generated three different state spaces of different orders of magnitude. Then we ran our distributed algorithms to verify three different CTL formulas (on each state space): $EX[\phi]$, $EG[\psi]$ and $E[\omega\ U \rho]$, where ϕ, ψ, ω and ρ are atomic propositions evaluable in every single state. Models and formulas used for the experiments are reported in [85]. The models are three known Petri nets (more precisely, P/T nets) benchmarks, whose state spaces have been generated by a MaRDiGraS instance.

Shared Memory This model (introduced in A.2) is taken from the GreatSPN benchmarks [38]. This P/T net models a system composed of 10 processors which compete for the access to a shared memory by using a unique shared bus. The number of reachable states of this model is 1.831×10^6. Despite the generated state space is relatively small, the benefit gained from our distributed approach grows as the number of states involved in the verification grows (as shown by the table 4.3): indeed, the verification of the last formula $E[\omega\ U \rho]$ scales better than the previous two.

Dekker This model (introduce in A.3) represents a 1-safe P/T net of a variant of the Dekker's mutual exclusion algorithm [47] for $N = 20$ processes. The state space generated by this model is an order of magnitude higher than the previous example: 1.153×10^7 reachable states. As shown by the table A.2 and by the graph shown in fig. 4.13a, the benefits deriving from our distributed approach are clearer. In fact, the evaluation of both the three formulas

Table 4.4: Dekker model (10^7 states, 10^8 transitions) analysis report

property	$\lvert[\![property]\!]_T\rvert$	# machines	time *(s)*
$EX[\phi]$	1.153×10^7	1	660
$EX[\phi]$	1.153×10^7	2	532
$EX[\phi]$	1.153×10^7	4	241
$EX[\phi]$	1.153×10^7	8	144
$EX[\phi]$	1.153×10^7	16	120
$EG[\psi]$	7.405×10^6	1	1567
$EG[\psi]$	7.405×10^6	2	1356
$EG[\psi]$	7.405×10^6	4	517
$EG[\psi]$	7.405×10^6	8	391
$EG[\psi]$	7.405×10^6	16	287
$E[\omega\ U\rho]$	5.767×10^6	1	1357
$E[\omega\ U\rho]$	5.767×10^6	2	1063
$E[\omega\ U\rho]$	5.767×10^6	4	585
$E[\omega\ U\rho]$	5.767×10^6	8	454
$E[\omega\ U\rho]$	5.767×10^6	16	372

gets faster by increasing the number of computational units. The graph shown by fig. 4.13c (and fig. 4.13d for the next model), plots the function cheat defined as follows:

$$\text{cheat}(n) = \frac{\mathit{time}(\text{parallel version with 1 node})}{\mathit{time}(\text{parallel version with } n \text{ nodes})} \tag{4.5.4}$$

However, as shown by fig. 4.13e, the efficiency is quite poor. Concerning this model, the graph tells us the application scales well when using no more than four compute nodes. This represents the optimal number of worker nodes in terms of efficiency.

Simple Load Balancing The simple load balancing system (introduced in A.4) is composed of 10 clients, 2 servers, and among these, a load balancer process. The reachability graph generated is very large: 4.060×10^8 states and 3.051×10^9 arcs for a total size of 120 GB of data. As shown by the table 4.5 and by Figure 4.13b, benefits deriving from our distributed approach are greater with respect to both previous examples. This points out a clear trend: the more is the complexity of the model, the more is the scalability of our distributed algorithm. In fact, both the cheat (figure 4.13d) and the *efficiency* (figure 4.13f) gained during the analysis of this last example greatly overcome the ones gained during the analysis of the Dekker model. In particular, we reached a super-linear speedup during the evaluation of $EG[\psi]$. The comparison of the results obtained analyzing the two last systems reveals how the proposed approach behaves better when increasing the amount of data to be

Table 4.5: Simple load balancing model ($\sim 10^8$ states, $\sim 10^9$ transitions) analysis report

property	$\lvert [\![property]\!]_T \rvert$	# machines	time *(s)*
$EX[\phi]$	1.716×10^8	1	2908
$EX[\phi]$	1.716×10^8	2	2401
$EX[\phi]$	1.716×10^8	4	937
$EX[\phi]$	1.716×10^8	8	693
$EX[\phi]$	1.716×10^8	16	251
$EG[\psi]$	4.060×10^8	1	21678
$EG[\psi]$	4.060×10^8	2	17147
$EG[\psi]$	4.060×10^8	4	6525
$EG[\psi]$	4.060×10^8	8	2983
$EG[\psi]$	4.060×10^8	16	1226
$E[\omega\ U\rho]$	7.524×10^7	1	1821
$E[\omega\ U\rho]$	7.524×10^7	2	1714
$E[\omega\ U\rho]$	7.524×10^7	4	602
$E[\omega\ U\rho]$	7.524×10^7	8	377
$E[\omega\ U\rho]$	7.524×10^7	16	203

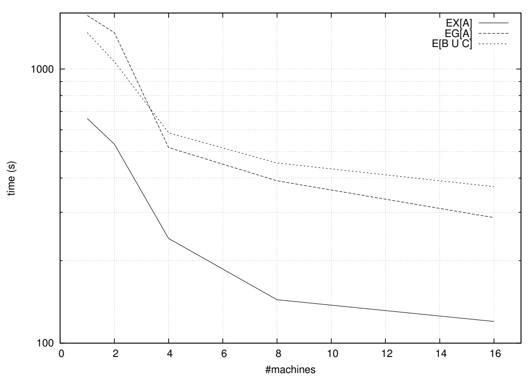
analyzed: the maximum cheat found verifying the *Dekker* system was just 5.5 against 17.7 verifying the *Simple load balancing* system. Another worth noting aspect is the *efficiency*: our experiments show how we can better exploit a greater number of compute units when increasing the amount of data. In fact the average *efficiency* found using 16 machines during the analysis of the *Dekker* model was 0.3 against 0.8 during the analysis of the *Simple load balancing* model. Concerning the maximum value of the *efficiency*, it was 0.78 for the *Dekker* model against 1.11 for the *Simple load balancing* model.

4.5.7 Related Work

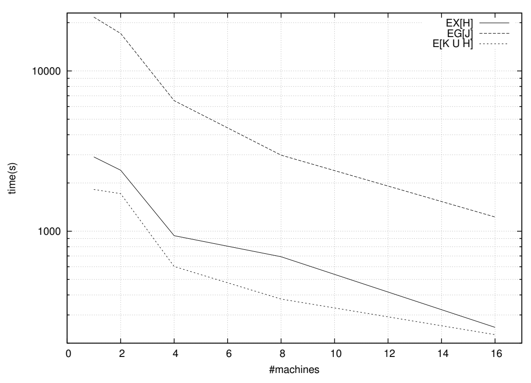
The MapReduce-based framework for CTL model checking we have presented in this chapter is quite different from distributed approaches relying on message passing. The only synchronization point among computational units is the shuffle phase, where key-value pairs emitted by mappers are sorted and moved to reducers. The number of cross-border transitions is not actually a critical issue. We know that a small number of cross-border transitions usually means a reduced network traffic due to data exchange. This is partially achieved with the map phase: the shuffling starts up as soon as data become available from single mappers, without waiting for the entire map output. Furthermore, there is an experimental evidence that the time required by shuffling is not dominating w.r.t. the overall computation time. Thus adding a (dynamic) partitioning phase between MapReduce iterations might not really

Figure 4.13: Execution time, cheat and efficiency graphs.

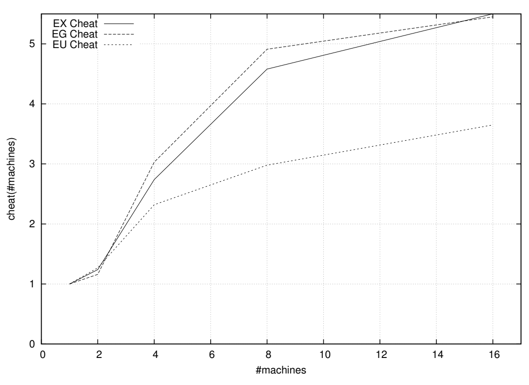
(a) Dekker model checking time



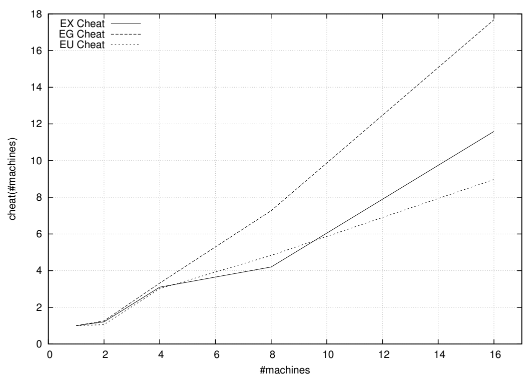
(b) Simple load bal. model checking time



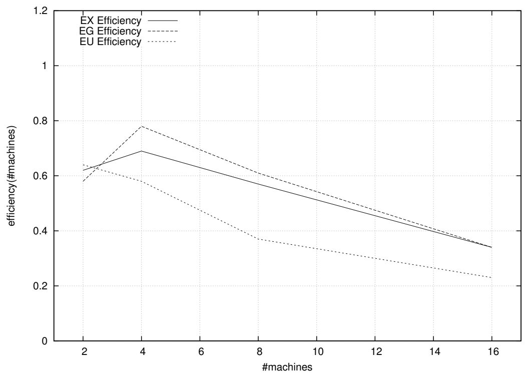
(c) Dekker *cheat*



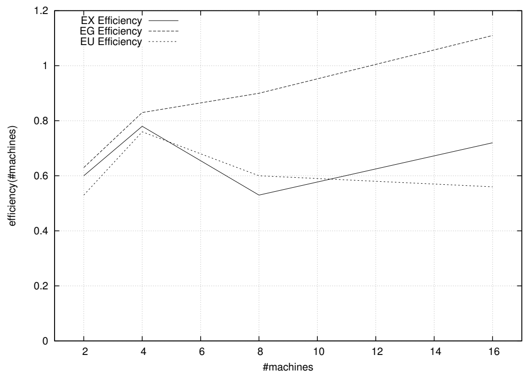
(d) Simple load balancing *cheat*



(e) Dekker efficiency



(f) Simple load balancing efficiency



be advantageous, or even hurt performances. We plan to deepen this point in order to better understand how a more sophisticated partitioning could impact on the performances of a MapReduce-based approach.

A comparison between our framework and some tools representing the state of the art of Petri nets' distributed analysis [13, 26, 25] highlights the effectiveness of big data approaches in formal verification. In fact, it seems that these tools can efficiently manage state spaces of magnitude up to 10^7, while there are no experimental evidences about their successfully usage for greater magnitude orders. Works presented in [73, 74] introduce a library to distribute existing model checkers and its evaluation with the GreatSPN model checker [38], respectively. The library, similarly to the MaRDiGraS framework, proposes a generic environment dedicated to distribution of any type of state space construction and were designed to create software tools deployable upon distributed local environment or multi-core machines. Anyway, as far as we know, it has been experimented to solve only reachability problems that are easier than causal properties (expressed by means of temporal logic formulas). Concerning the state of the art of sequential Petri nets tools [85] (based on explicit approaches), such as *LoLa (Low Level Petri Net Analyzer)* [109], it turns out they perform very well on small/medium size state spaces, or on models exhibiting specific features such as symmetry and/or a high degree of concurrency. But carried out experiments, [85] have shown they are, in many cases, unable to verify CTL formulas on the *Dekker* and the *Simple load balancing* models used in this thesis.

Our main goal and (hopefully) contribution, however, has been to provide users with a model checking framework (rather than a tool), which can be easily deployed in the cloud. In fact, departing from the current literature on distributed CTL model checking, we considered an important, sometimes understated, aspect: we have enabled a "push-button" operating mode in the context of distributed formal verification to remove, or dramatically lower, the costs of deploying applications into end-to-end solutions. Think, e.g., of the intrinsic complexity of grids and high-performance computing clusters. We have provided a way to run complex scientific applications on Cloud Computing infrastructures, meeting compute-intensive and data-intensive challenges of formal verification. To our knowledge, a few other techniques and tools have been introduced, with similar aims. E.g, [22] presents a MapReduce approach to check specifications expressed in a metric temporal logic over large execution traces, with aggregation modalities; [77] attempts (in a quite different context, i.e., Swarm Verification) to exploit massively parallel jobs running test randomization techniques to verify the correctness of mission critical software.

4.5.8 Summary

Cloud computing is an emerging and evolving paradigm where challenges and opportunities allow for new research directions and applications. Companies such as Amazon, Microsoft, and Google are putting remarkable efforts in delivering services able to offer hundreds, or even thousands, commodity computers available to customers, thus enabling users to run massively parallel jobs. There is an evidence that this trend will continue. Once reached maturity, it could dramatically change the way software verification tasks are performed. This section presents a framework for model checking very complex systems, based on iterative MapReduce algorithms that use a fixed-point characterization of temporal operators of CTL. Despite model checking software tools are often called “push-button” technologies, managing the high-performance computing environments required by scientific applications is far from being considered such, especially if one wants to exploit general purpose cloud computing facilities. Our framework has been designed for re-enabling a “push-button” operating mode in the context of distributed formal verification. We have reported some experiments showing the convenience of using the framework to effectively check CTL formulas on huge state spaces. In some particular cases a super-linear speedup has been achieved. We believe that this work could be a further step towards reducing the distance between different, but related communities: the “formal methods” one and the “big data” one. Exposing this issue to scientists with different skills could stimulate the development of new interesting and effective solutions.

Chapter 5

Conclusion

The thesis focused on two complementary approaches to deal with the state explosion problem for dynamic, concurrent, and real-time systems.

On the one hand, we explored advanced abstraction techniques in order to deal with infinite-states systems. These techniques aim at reducing the number of states needed to be constructed in order to verify certain properties. The algorithms for constructing the reduced state space take advantage of some details of the property to be verified in order to avoid the construction of the overall state space, if not needed. In particular we addressed several different open issues for real-time systems modeled with Time Basic Petri Nets.

On the other hand, we introduced distributed approaches which exploits techniques typically used by the big data community to enable verification of very complex systems using big data approaches and cloud computing facilities. Despite many years of work in the area of multi-core and distributed model checking, still few works introduce algorithms that can scale effortlessly to the use of thousands of loosely connected computers in a network, so existing technology does not yet allow us to take full advantage of the vast array of compute power of a "cloud" environment. Cloud computing is an emerging and evolving paradigm where challenges and opportunities allow for new research directions and applications. There is an evidence that this trend will continue, in fact several companies are putting remarkable efforts in delivering services able to offer hundreds, or even thousands, commodity computers available to customers, thus enabling users to run massively parallel jobs. This revolution is already started in different scientific fields, achieving remarkable breakthroughs through new kinds of experiments that would have been impossible only few years ago.

5.1 Contributions

The major contributions lie in two different branches of formal methods in software engineering. Both contributions aim at coping with the state explosion problem, but using two different complementary approaches. The first main contribution consists in the introduction of algorithms and related tools able to deal with infinite-states real-time systems. The second main contribution focuses on the connection between formal methods in software engineering and big data approaches.

Advanced State Space Methods The reachability analysis technique for TB net models (section 3.2) overtakes the existing available analysis technique for TBasic nets because it allows the building of a sort of symbolic time-coverage reachability graph keeping interesting timing properties of the nets. In particular the introduction of the concept of time anonymous timestamps, enables a major factorization of symbolic states and allows, in many cases, to building a finite representation of the underling infinite state space.

An extension of this technique that further exploits the time anonymous concept, in order to deal with topologically unbounded nets, exploits the concept of a coverage of TA tokens, i.e., a sort of TA^{ω} (section 3.3). Such a coverability analysis technique is able to construct coverability trees/graphs for unbounded TB net models. The termination of the algorithm is guaranteed as long as, within the input model, tokens growing without limit can be anonymized. This means that we are able to manage models that do not exhibit Zeno behavior and do not express actions depending on "infinite" past events. This is actually a reasonable limitation because, generally, real-world examples do not exhibit such a behavior.

Other coverability analysis techniques for such a formalism, have not been proposed yet, as far as we know.

Big Data Approaches to Formal Verification Work presented in section 4.3 discusses two approaches to face the state-space explosion in discrete-event system analysis. These approaches try to combine abstraction techniques and parallel algorithms to exploit distributed/cloud computing frameworks. These approaches have been experienced on the timed, symbolic reachability analysis of TB nets. The outcomes of tests, performed on a benchmarking real-time system model, clearly show how the combination of these techniques can be conveniently used to deal with real world examples. Moreover it has been shown how distributed versions of the state space builder increase performances of the sequential one.

Although the parallel workers model has shown a higher speed-up with our benchmarking example, the cloud environment can be conveniently used to exploit a big cluster of machines with recent hardware which we might not have at disposal locally. Moreover, in the latter case, we do not need any setup phase of our environment.

Section 4.4 introduced MaRDiGraS: a generic framework which can easily adapted for tackling the state explosion problem during the computation of reachability graphs of systems modeled by different formalisms. This framework exploits techniques typically used by the big data community and so far poorly explored for this kind of problems. Thanks to its very simple programming interface, it provides a powerful tool for constructing high-performance distributed applications without the need to deal with the complex communication and synchronization issues required for distributing a computation on large clusters. Indeed, it was easy to use it for implementing a distributed version of an existing sequential tool that was able to analyze a model beyond the capacity of a single machine. Our experiments report that MaRDiGraS can be used effectively to compute state spaces sized with different orders of magnitude. We believe that this work could be a first step towards a meeting between two very different, but related communities: the "formal methods" community and the "big data" community.

Section 4.5 introduced a framework for model checking very complex systems, using iterative MapReduce algorithms based on the fixed-point characterization of temporal operators of CTL. Despite model checking software tools are often called "push-button" technologies, managing the high-performance computing environments required by scientific applications is far from being considered such, especially if one wants to exploit general purpose cloud computing facilities. We reported some experiments showing the convenience of using the framework to effectively check CTL formulas on huge state spaces. In some particular cases a super-linear speedup has been achieved. Departing from the current literature on distributed CTL model checking, we considered an important, sometimes understated, aspect: we enabled a "push-button" operating mode in the context of distributed formal verification to remove, or dramatically lower, the costs of deploying applications into end-to-end solutions. Think, e.g., of the intrinsic complexity of grids and high-performance computing clusters. We have provided a way to run complex scientific applications on Cloud Computing infrastructures, meeting compute-intensive and data-intensive challenges of formal verification.

5.2 Open Issues

Concerning the reachability analysis of TB nets, the evaluation component is still very simple: it permits to examine the input graph looking for interesting properties on topological definition of markings. Therefore, its integration with some existing model checking engines is currently under investigation. For instance, we might be interested in checking whenever some state-based formula ϕ is satisfied within a time interval $[d, D[$, with $d \in \mathbb{R}^+$ and $D \in (\mathbb{R}^+ \cup \infty)$ starting from the initial symbolic state. Anyway, only conservative bounds can be established by combining the information on edges. In the case they are not enough to exclude incorrect timing behaviors, it is possible to carry out a more accurate analysis by rebuilding a portion of the graph, retracing some critical paths and reintroducing absolute time references This task is also complicated in the presence of paths containing `EE` edges. This means that there exist some edges leading to a subset of the target state from a subset of the ordinary states represented by the source node. In this case there is the possibility that the path actually is not feasible.

Concerning the distributed technique, adopted by the MaRDiGraS framework, several questions remain open and require further research: The optimal threshold and partitioning should be automatically chosen by the framework rather than by the user. Furthermore, the proposed computational model should be optimized when the number of not expanded states gets very small, as illustrated in section 4.4. Finally, a study on how the framework can be adapted to deal with other classes of formalisms, should be performed.

Regarding the distributed CTL framework, the problem of integrating distributed verification algorithms in the presence of an abstract state space (section 2.3), is still open. In fact, CTL model checking requires all the edges to be `AA`. Therefore, the framework supports the verification of models that can reach a finite number of concrete states. While the general case, where the presence of `EE` edges makes the task more complicated, has not yet been addressed.

5.3 Future Work

Regarding reachability problems for TB nets, our ultimate goal is to allow the verification of timed reachability properties, for instance properties expressed in TCTL logic [59]. Therefore, we are currently investigate the feasibility of integrating the reachability graph construction with some existing model checking engines.

The cloud computing ecosystem, along with platforms and services, is an emerging and evolving area. Therefore, it is quite natural that in the near future new mechanisms for sorting, analyzing, and storing data will emerge beside the MapReduce model. For instance, *Google Cloud Dataflow* [45] was recently announced. *Cloud Dataflow* is described as a successor of MapReduce, to implement advanced, multi-step processing pipelines to extract deep insight from datasets of any size, free from the burden of deploying clusters, tuning configuration parameters, and optimizing resource usage. We plan to explore also such a new advances in the Big data area, to leverage them also in the context of distributed formal verification.

Bibliography

[1] ParoshAziz Abdulla and Aletta Nylén. Timed petri nets and bqos. In *Applications and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 53–70. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42252-5. doi: 10.1007/3-540-45740-2_5. URL http://dx.doi.org/10.1007/3-540-45740-2_5.

[2] ParoshAziz Abdulla, Johann Deneux, Pritha Mahata, , and Aletta Nylén. Forward reachability analysis of timed petri nets. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 343–362. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23167-7. doi: 10.1007/978-3-540-30206-3_24. URL http://dx.doi.org/10.1007/978-3-540-30206-3_24.

[3] Rajeev Alur and David Dill. Automata for modeling real-time systems. In MichaelS. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer Berlin Heidelberg, 1990. ISBN 978-3-540-52826-5. doi: 10.1007/BFb0032042. URL http://dx.doi.org/10.1007/BFb0032042.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126 (2):183–235, April 1994. ISSN 0304-3975. doi: 10.1016/0304-3975(94)90010-8. URL http://dx.doi.org/10.1016/0304-3975(94)90010-8.

[5] Amazon Web Services. Elastic MapReduce. http://aws.amazon.com/documentation/elasticmapreduce/, 2013. Last visited: September 2014.

[6] Amazon Web Services. AWS in Education. http://aws.amazon.com/grants/, 2013. Last visited: September 2014.

[7] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, 2003. ISBN 0452284392.

[8] J. Barnat, L. Brim, M. Češka, and P. Ročkai. Divine: Parallel distributed model checker. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 4–7, 2010. doi: 10.1109/PDMC-HiBi.2010.9.

[9] Jiri Barnat, Lubos Brim, and Petr Rockai. Scalable multi-core ltl model-checking. In Dragan Bosnacki and Stefan Edelkamp, editors, *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 187–203. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73369-0. doi: 10.1007/978-3-540-73370-6_13. URL `http://dx.doi.org/10.1007/978-3-540-73370-6_13`.

[10] Jiśrı́i Barnat, Luboś Brim, and Ivana Černá. Property driven distribution of nested DFS. In *M. Leuschel and U. Ultes-Nitsche (Eds.): Proc. of the 3rd International Workshop on Verification and Computational Logic*, pages 1–10, Pittsburgh, PA, USA, 2002. Dept. of Electronics and Computer Science, U. of Southampton.

[11] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hall. Mapreduce for parallel trace validation of ltl properties. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35631-5. doi: 10.1007/978-3-642-35632-2_20. URL `http://dx.doi.org/10.1007/978-3-642-35632-2_20`.

[12] David Basin, Germano Caronni, Sarah Ereth, Mat Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In Borzoo Bonakdarpour and ScottA. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 31–47. Springer International Publishing, 2014. ISBN 978-3-319-11163-6. doi: 10.1007/978-3-319-11164-3_4. URL `http://dx.doi.org/10.1007/978-3-319-11164-3_4`.

[13] 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, 2005. ISSN 1433-2779. doi: 10.1007/s10009-003-0129-2. URL `http://dx.doi.org/10.1007/s10009-003-0129-2`.

[14] Carlo Bellettini and Lorenzo Capra. Reachability analysis of time basic Petri nets: A time coverage approach. *2011 13th International Symposium on Symbolic and*

Numeric Algorithms for Scientific Computing, 0:110–117, 2011. doi: http://doi.ieeecomputersociety.org/10.1109/SYNASC.2011.16.

[15] Carlo Bellettini, Miguel Felder, and Mauro Pezzè. A tool for analysing high-level timed Petri nets. IPTES Esprit Project 5570 PDM-41, Politecnico di Milano, September 1993. URL `http://unimi.academia.edu/CarloBellettini/Papers/741343`.

[16] Carlo Bellettini, Miguel Felder, and Mauro Pezz. Merlot: a tool for analysis of real-time specifications. In *Software Specification and Design, 1993., Proceedings of the Seventh International Workshop on*, pages 110–119, Dec 1993. doi: 10.1109/IWSSD.1993.315507.

[17] Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Symbolic state space exploration of RT systems in the cloud. In *Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC 2012, pages 295–302, Los Alamitos, CA, USA, 2012. IEEE CS Press. ISBN 9780769549347. doi: 10.1109/SYNASC.2012.18.

[18] Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Mardigras: Simplified building of reachability graphs on large clusters. In ParoshAziz Abdulla and Igor Potapov, editors, *Reachability Problems*, volume 8169 of *LNCS*, pages 83–95. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41035-2. doi: 10.1007/978-3-642-41036-9_9. URL `http://dx.doi.org/10.1007/978-3-642-41036-9_9`.

[19] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Softw. Eng.*, 17:259–273, March 1991. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.75415.

[20] Bernard Berthomieu and Miguel Menasche. An enumerative approach for analyzing time petri nets. In *IFIP Congress*, pages 41–46, 1983.

[21] Bernard Berthomieu and Franois Vernadat. State class constructions for branching analysis of time Petri nets. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 442–457. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-00898-9. doi: 10.1007/3-540-36577-X_33. URL `http://dx.doi.org/10.1007/3-540-36577-X_33`.

[22] Domenico Bianculli, Carlo Ghezzi, and Srdjan Krstić. Trace checking of metric temporal logic with aggregating modalities using mapreduce. In Dimitra Giannakopoulou and Gwen Salaun, editors, *Software Engineering and Formal Methods*, volume 8702 of *LNCS*, pages 144–158. Springer International Publishing, 2014. ISBN 978-3-319-10430-0. doi: 10.1007/978-3-319-10431-7_11. URL `http://dx.doi.org/10.1007/978-3-319-10431-7_11`.

[23] H. Boucheneb and R. Hadjidj. CTL* model checking for time Petri nets. *Theor. Comput. Sci.*, 353(1):208–227, March 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.11.002. URL `http://dx.doi.org/10.1016/j.tcs.2005.11.002`.

[24] Hanifa Boucheneb, Guillaume Gardey, and Olivier H. Roux. Tctl model checking of time petri nets. *J. Log. and Comput.*, 19(6):1509–1540, December 2009. ISSN 0955-792X. doi: 10.1093/logcom/exp036. URL `http://dx.doi.org/10.1093/logcom/exp036`.

[25] Mohand Cherif Boukala and Laure Petrucci. Distributed model-checking and counterexample search for CTL logic. *Int. J. Crit. Comput.-Based Syst.*, 3(1/2):44–59, January 2012. ISSN 1757-8779. doi: 10.1504/IJCCBS.2012.045076. URL `http://dx.doi.org/10.1504/IJCCBS.2012.045076`.

[26] M. Bourahla. Distributed CTL model checking. *Software, IEE Proceedings -*, 152(6): 297–308, Dec 2005. ISSN 1462-5970. doi: 10.1049/ip-sen:20050001.

[27] M. Boyer and O.H. Roux. Comparison of the expressiveness of arc, place and transition time petri nets. 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 63–82. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73093-4. doi: 10.1007/978-3-540-73094-1_7. URL `http://dx.doi.org/10.1007/978-3-540-73094-1_7`.

[28] Luboš Brim, Ivana Černá, Pavel Moravec, and Jiří Šimša. Accepting predecessors are better than back edges in distributed LTL model-checking. In AlanJ. Hu and AndrewK. Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*, pages 352–366. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23738-9. doi: 10.1007/978-3-540-30494-4_25. URL `http://dx.doi.org/10.1007/978-3-540-30494-4_25`.

[29] Luboš Brim, Karen Yorav, and Jitka Žídková. Assumption-based distribution of ctl model checking. *International Journal on Software Tools for Technology Transfer*,

7(1):61–73, 2005. ISSN 1433-2779. doi: 10.1007/s10009-004-0163-8. URL `http://dx.doi.org/10.1007/s10009-004-0163-8`.

[30] RandalE. Bryant and Christoph Meinel. Ordered binary decision diagrams. In Soha Hassoun and Tsutomu Sasao, editors, *Logic Synthesis and Verification*, volume 654 of *The Springer International Series in Engineering and Computer Science*, pages 285–307. Springer US, 2002. ISBN 978-1-4613-5253-2. doi: 10.1007/978-1-4615-0817-5_11. URL `http://dx.doi.org/10.1007/978-1-4615-0817-5_11`.

[31] J. R. Burch, E.M. Clarke, K. L. McMillan, D.L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, 1990. doi: 10.1109/LICS.1990.113767.

[32] F. Calzolari and M. Pezzè. Property decomposition to speed up analysis. *Real-Time Systems, Euromicro Conference on*, 0:147, 1995. ISSN 1068-3070. doi: http://doi.ieeecomputersociety.org/10.1109/EMWRTS.1995.514305.

[33] Matteo Camilli. Petri nets state space analysis in the cloud. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1638–1640, June 2012. doi: 10.1109/ICSE.2012.6227217.

[34] Matteo Camilli. Formal verification problems in a big data world: Towards a mighty synergy. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 638–641, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591088. URL `http://doi.acm.org/10.1145/2591062.2591088`.

[35] Matteo Camilli, Carlo Bellettini, Lorenzo Capra, and Mattia Monga. Ctl model checking in the cloud using mapreduce. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, pages 333–340, Los Alamitos, CA, USA, Sept 2014. IEEE CS Press. doi: 10.1109/SYNASC.2014.52.

[36] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In Giorgio De Michelis and Michel Diaz, editors, *Application and Theory of Petri Nets 1995*, volume 935 of *Lecture Notes in Computer Science*, pages 181–200. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60029-9. doi: 10.1007/3-540-60029-9_40. URL `http://dx.doi.org/10.1007/3-540-60029-9_40`.

[37] G. Chiola and G. Franceschinis. Colored GSPN models and automatic symmetry detection. In *Petri Nets and Performance Models PNPM89*, pages 50–60, 1989. doi: 10.1109/PNPM.1989.68539.

[38] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. Greatspn 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Perform. Eval.*, 24(1-2): 47–68, November 1995. ISSN 0166-5316. doi: 10.1016/0166-5316(95)00008-L. URL `http://dx.doi.org/10.1016/0166-5316(95)00008-L`.

[39] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. A symbolic reachability graph for coloured Petri nets. *Theor. Comput. Sci.*, 176(1-2):39–65, April 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(96)00010-2. URL `http://dx.doi.org/10.1016/S0304-3975(96)00010-2`.

[40] Giovanni Chiola and Giuliana Franceschinis. Colored GSPN models and automatic symmetry detection. In *The Proceedings of the Third International Workshop on Petri Nets and Performance Models*, PNPM '89, pages 50–60, Washington, DC, USA, 1989. IEEE Computer Society. ISBN 0-8186-2001-3. URL `http://dl.acm.org/citation.cfm?id=645811.670651`.

[41] G. Ciardo. Automated parallelization of discrete state-space generation. *J. Parallel Distrib. Comput.*, 47(2):153–167, December 1997. ISSN 0743-7315. doi: 10.1006/jpdc.1997.1409. URL `http://dx.doi.org/10.1006/jpdc.1997.1409`.

[42] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS J. on Comp.*, 10(1):82–93, January 1998. ISSN 1526-5528. doi: 10.1287/ijoc.10.1.82. URL `http://dx.doi.org/10.1287/ijoc.10.1.82`.

[43] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

[44] EdmundM. 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 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982. ISBN 978-3-540-11212-9. doi: 10.1007/BFb0025774. URL `http://dx.doi.org/10.1007/BFb0025774`.

[45] dataflow. Google Cloud Dataflow. http://googlecloudplatform.blogspot.it/2014/06/. Last visited: September 2014.

[46] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[47] EdsgerW. Dijkstra. Cooperating sequential processes. In PerBrinch Hansen, editor, *The Origin of Concurrent Programming*, pages 65–138. Springer New York, 2002. ISBN 978-1-4419-2986-0. doi: 10.1007/978-1-4757-3472-0_2. URL http://dx.doi.org/10.1007/978-1-4757-3472-0_2.

[48] Nicholas J. Dingle, William J. Knottenbelt, and Tamas Suto. Pipe2: a tool for the performance evaluation of generalised stochastic Petri nets. *SIGMETRICS Perform. Eval. Rev.*, 36(4):34–39, March 2009. ISSN 0163-5999. doi: 10.1145/1530873.1530881. URL http://doi.acm.org/10.1145/1530873.1530881.

[49] A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized b uuml;chi automata. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 76–83, Oct 2004. doi: 10.1109/MASCOT.2004.1348184.

[50] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In Dragan Bonaki and Stefan Leue, editors, *Model Checking Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 230–239. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43477-1. doi: 10.1007/3-540-46017-9_18. URL http://dx.doi.org/10.1007/3-540-46017-9_18.

[51] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proc. of Symp. on High Performance Distributed Computing*, pages 810–818, 2010. ISBN 978-1-60558-942-8. doi: http://doi.acm.org/10.1145/1851476.1851593. URL http://doi.acm.org/10.1145/1851476.1851593.

[52] E.Allen Emerson and A.Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1-2):105–131, 1996. ISSN 0925-9856. doi: 10.1007/BF00625970. URL http://dx.doi.org/10.1007/BF00625970.

[53] Sami Evangelista. High level petri nets analysis with helena. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets*, ICATPN'05, pages 455–464, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-26301-2, 978-3-540-26301-2. doi: 10.1007/11494744_26. URL `http://dx.doi.org/10.1007/11494744_26`.

[54] Alain Finkel. The minimal coverability graph for petri nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer Berlin Heidelberg, 1993. ISBN 978-3-540-56689-2. doi: 10.1007/3-540-56689-9_45. URL `http://dx.doi.org/10.1007/3-540-56689-9_45`.

[55] Alain Finkel and Jean Goubault-Larrecq. Forward analysis for wsts, part i: Completions. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*, volume 09001 of *Dagstuhl Seminar Proceedings*, pages 433–444. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009. ISBN 978-3-939897-09-5. doi: http://drops.dagstuhl.de/opus/volltexte/2009/1844.

[56] Alain Finkel and Jean Goubault-Larrecq. Forward analysis for wsts, part ii: Complete wsts. In *Proceedings of the 36th Internatilonal Collogquium on Automata, Languages and Programming: Part II*, ICALP '09, pages 188–199, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02929-5. doi: 10.1007/978-3-642-02930-1_16. URL `http://dx.doi.org/10.1007/978-3-642-02930-1_16`.

[57] I. Foster and C. Kesselman. *The Grid. Blueprint for a new computing infrastructure.* Morgan Kaufmann Publishers, San Francisco, USA, 1999.

[58] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1st edition, 1999. ISBN 0201309556.

[59] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi. Modeling time in computing: A taxonomy and a comparative survey. *ACM Computing Surveys*, 42:6:1–6:59, March 2010. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1667062.1667063. URL `http://doi.acm.org/10.1145/1667062.1667063`.

[60] 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. ISBN 978-3-540-42124-5. doi: 10.1007/3-540-45139-0_14. URL `http://dx.doi.org/10.1007/3-540-45139-0_14`.

[61] Guillaume Gardey, OlivierH. Roux, and OlivierF. Roux. Using zone graph method for computing the state space of atime petri net. In KimGuldstrand Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 246–259. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21671-1. doi: 10.1007/978-3-540-40903-8_20. URL `http://dx.doi.org/10.1007/978-3-540-40903-8_20`.

[62] Gilles Geeraerts, Jean-Franois Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set for petri nets. In KedarS. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 98–113. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75595-1. doi: 10.1007/978-3-540-75596-8_9. URL `http://dx.doi.org/10.1007/978-3-540-75596-8_9`.

[63] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35:97–107, February 1992. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/129630.129635. URL `http://doi.acm.org/10.1145/129630.129635`.

[64] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL `http://doi.acm.org/10.1145/945445.945450`.

[65] C. Ghezzi and M. Pezz. Towards extensible graphical formalisms. In *Software Specification and Design, 1993., Proceedings of the Seventh International Workshop on*, pages 69–77, Dec 1993. doi: 10.1109/IWSSD.1993.315511.

[66] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. Softw. Eng.*, 17:160–172, February 1991. ISSN 0098-5589. doi: 10.1109/32.67597.

[67] C. Ghezzi, S. Morasca, and M. Pezzè. Validating timing requirements for time basic net specifications. *J. Syst. Softw.*, 27:97–117, November 1994. ISSN 0164-1212. doi: 10.1016/0164-1212(94)90024-8.

[68] Silvio Ghilardi and Silvio Ranise. MCMT: A model checker modulo theories. In *Proc. of the 5th International Conference on Automated Reasoning*, IJCAR'10, pages 22–29, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14202-8, 978-3-642-14202-4. doi: 10.1007/978-3-642-14203-1_3. URL `http://dx.doi.org/10.1007/978-3-642-14203-1_3`.

[69] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In EdmundM. Clarke and RobertP. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196. Springer Berlin Heidelberg, 1991. ISBN 978-3-540-54477-7. doi: 10.1007/BFb0023732. URL `http://dx.doi.org/10.1007/BFb0023732`.

[70] Bernd Grahlmann. The PEP tool. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 440–443. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63166-8. doi: 10.1007/3-540-63166-6_43. URL `http://dx.doi.org/10.1007/3-540-63166-6_43`.

[71] graphviz. Graphviz. `http://www.graphviz.org/`. Last visited: June 2013.

[72] Feng Guo, Guang Wei, Mengmeng Deng, and Wanlin Shi. CTL model checking algorithm using MapReduce. In W. Eric Wong and Tinghuai Ma, editors, *Emerging Technologies for Information Systems, Computing, and Management*, volume 236 of *Lecture Notes in Electrical Engineering*, pages 341–348. Springer, 2013. ISBN 978-1-4614-7009-0. doi: 10.1007/978-1-4614-7010-6_39. URL `http://dx.doi.org/10.1007/978-1-4614-7010-6_39`.

[73] A. Hamez, F. Kordon, and Y. Thierry-Mieg. Iibdmc: a library to operate efficient distributed model checking. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007. doi: 10.1109/IPDPS.2007.370647.

[74] Alexandre Hamez, Fabrice Kordon, Yann Thierry-Mieg, and Fabrice Legond-Aubry. dmcg: A distributed symbolic model checker based on greatspn. In *Proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, ICATPN'07, pages 495–504, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73093-4. URL `http://dl.acm.org/citation.cfm?id=1769053.1769085`.

[75] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011. ISBN 0321773713, 9780321773715.

[76] Gerard J. Holzmann. Algorithms for automated protocol verification. *AT T Technical Journal*, 69(1):32–44, Jan 1990. ISSN 8756-2324. doi: 10.1002/j.1538-7305.1990.tb00101.x.

[77] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Trans. Softw. Eng.*, 37(6):845–857, November 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.110. URL `http://dx.doi.org/10.1109/TSE.2010.110`.

[78] G.J. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *Software Engineering, IEEE Transactions on*, 33(10):659–674, Oct 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70724.

[79] ISO/IEC. Systems and software engineering – High-level Petri nets – Part 2: Transfer format. Technical Report 15909-2:2011, ISO, 2011.

[80] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, July 2014. ISSN 0001-0782. doi: 10.1145/2611567. URL `http://doi.acm.org/10.1145/2611567`.

[81] Kurt Jensen and Grzegorz Rozenberg, editors. *High-level Petri Nets: Theory and Application*. Springer-Verlag, London, UK, UK, 1991. ISBN 3-540-54125-x.

[82] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics. ISBN 978-0-898716-98-6. URL `http://dl.acm.org/citation.cfm?id=1873601.1873677`.

[83] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969. ISSN 0022-0000. doi: 10.1016/S0022-0000(69)80011-5. URL `http://dx.doi.org/10.1016/S0022-0000(69)80011-5`.

[84] Kais Klai, Naim Aber, and Laure Petrucci. Verification of reachability properties for time petri nets. In ParoshAziz Abdulla and Igor Potapov, editors, *Reachability*

Problems, volume 8169 of *Lecture Notes in Computer Science*, pages 159–170. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41035-2. doi: 10.1007/978-3-642-41036-9_15. URL `http://dx.doi.org/10.1007/978-3-642-41036-9_15`.

[85] F. Kordon, A. Linard, M. Becutti, D. Buchs, L. Fronc, F. Hulin-Hubard, F. Legond-Aubry, N. Lohmann, A. Marechal, E. Paviot-Adet, F. Pommereau, C. Rodrígues, C. Rohr, Y. Thierry-Mieg, H. Wimmel, and K. Wolf. Web report on the model checking contest @ petri net 2013, available at http://mcc.lip6.fr, June 2013.

[86] A.N. Kovacs and S. Hudak. Time semantics in time basic nets. In *Applied Machine Intelligence and Informatics*, pages 315 –319, January 2010. doi: 10.1109/SAMI.2010.5423710.

[87] Dexter C. Kozen. The complexity of computations. In *Theory of Computation*, Texts in Computer Science. Springer London, 2006. ISBN 978-1-84628-297-3. doi: 10.1007/1-84628-477-5_1. URL `http://dx.doi.org/10.1007/1-84628-477-5_1`.

[88] LarsM. Kristensen and Laure Petrucci. An approach to distributed state space exploration for coloured petri nets. In Jordi Cortadella and Wolfgang Reisig, editors, *Applications and Theory of Petri Nets 2004*, volume 3099 of *Lecture Notes in Computer Science*, pages 474–483. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22236-1. doi: 10.1007/978-3-540-27793-4_28. URL `http://dx.doi.org/10.1007/978-3-540-27793-4_28`.

[89] Rahul Kumar and Eric G. Mercer. Load balancing parallel explicit state model checking. *Electronic Notes in Theoretical Computer Science*, 128(3):19 – 34, 2005. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2004.10.016. URL `http://www.sciencedirect.com/science/article/pii/S1571066105001659`. Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004) Parallel and Distributed Methods in Verification 2004.

[90] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi Junttila. Simple bounded LTL model checking. In AlanJ. Hu and AndrewK. Martin, editors, *Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*, pages 186–200. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23738-9. doi: 10.1007/978-3-540-30494-4_14. URL `http://dx.doi.org/10.1007/978-3-540-30494-4_14`.

[91] 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. ISBN 978-3-540-66499-4. doi: 10.1007/3-540-48234-2_3. URL `http://dx.doi.org/10.1007/3-540-48234-2_3`.

[92] Didier Lime and Olivier H. Roux. Model checking of time petri nets using the state class timed automaton. *Discrete Event Dynamic Systems*, 16(2):179–205, April 2006. ISSN 0924-6703. doi: 10.1007/s10626-006-8133-9. URL `http://dx.doi.org/10.1007/s10626-006-8133-9`.

[93] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In *Research and development in information retrieval*, SIGIR '09, pages 155–162. ACM, 2009. ISBN 978-1-60558-483-6. doi: 10.1145/1571941.1571970. URL `http://doi.acm.org/10.1145/1571941.1571970`.

[94] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Mining and Learning with Graphs*, pages 78–85, New York, 2010. ACM Press. ISBN 978-1-4503-0214-2. doi: 10.1145/1830252.1830263. URL `http://doi.acm.org/10.1145/1830252.1830263`.

[95] K.L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In Gregor von Bochmann and DavidKarl Probst, editors, *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer Berlin Heidelberg, 1993. ISBN 978-3-540-56496-6. doi: 10.1007/3-540-56496-9_14. URL `http://dx.doi.org/10.1007/3-540-56496-9_14`.

[96] Philip M. Merlin and David J. Farber. Recoverability of modular systems. *SIGOPS Oper. Syst. Rev.*, 9(3):51–56, January 1975. ISSN 0163-5980. doi: 10.1145/563905.810899. URL `http://doi.acm.org/10.1145/563905.810899`.

[97] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998. ISSN 0018-9219. doi: 10.1109/JPROC.1998.658762.

[98] Radek Pelnek. Typical structural properties of state spaces. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes*

in Computer Science, pages 5–22. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21314-7. doi: 10.1007/978-3-540-24732-6_2. URL `http://dx.doi.org/10.1007/978-3-540-24732-6_2`.

[99] pnml. The Petri nets markup language. `http://www.pnml.org/`. Last visited: June 2013.

[100] A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *Software Engineering, IEEE Transactions on*, 19(1):41–55, Jan 1993. ISSN 0098-5589. doi: 10.1109/32.210306.

[101] Pierre-Alain Reynier and Frédéric Servais. Minimal coverability set for petri nets: Karp and miller algorithm with pruning. *Fundam. Inf.*, 122(1-2):1–30, January 2013. ISSN 0169-2968. doi: 10.3233/FI-2013-781. URL `http://dx.doi.org/10.3233/FI-2013-781`.

[102] Joseph Sifakis. Use of petri nets for performance evaluation. In *Proceedings of the Third International Symposium on Measuring, Modelling and Evaluating Computer Systems*, pages 75–93, Amsterdam, The Netherlands, The Netherlands, 1977. North-Holland Publishing Co. ISBN 0-444-85058-9. URL `http://dl.acm.org/citation.cfm?id=647408.724424`.

[103] Ulrich Stern and DavidL. Dill. Parallelizing the murϕ verifier. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin Heidelberg, 1997. ISBN 978-3-540-63166-8. doi: 10.1007/3-540-63166-6_26. URL `http://dx.doi.org/10.1007/3-540-63166-6_26`.

[104] C. t. Chu, S. K. Kim, Y. a. Lin, Y. Yu, G. R. Bradski, A Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *Neural Information Processing Systems*, pages 281–288, 2006.

[105] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[106] The Apache Software Foundation. Hadoop MapReduce. `http://hadoop.apache.org/mapreduce/`, 2007. Last visited: June 2014.

[107] Rdiger Valk and Guy Vidal-Naquet. Petri nets and regular languages. *Journal of Computer and System Sciences*, 23(3):299 – 325, 1981. ISSN 0022-0000. doi:

http://dx.doi.org/10.1016/0022-0000(81)90067-2. URL `http://www.sciencedirect.com/science/article/pii/0022000081900672`.

[108] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-65306-6. doi: 10.1007/3-540-65306-6_21. URL `http://dx.doi.org/10.1007/3-540-65306-6_21`.

[109] Karsten Wolf. Generating petri net state spaces. 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 29–42. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73093-4. doi: 10.1007/978-3-540-73094-1_5. URL `http://dx.doi.org/10.1007/978-3-540-73094-1_5`.

[110] G.K. Zipf. *Selected Studies of the Principle of Relative Frequency in Language*. Harvard University Press, 1932. ISBN 9780674432048. URL `http://books.google.it/books?id=\_BubcQAACAAJ`.

Appendix A

Petri Nets Models

A.1 The Gas Burner

The gas burner example has been used as a representative of a real small system. [100] presents the complete and formal description of the example. Here, we only report the informal description given there. A gas burner is a safety-critical system as an accident may occur if an excessive amount of unburned gas leaks to the environment. Small gas leaks ignition cannot be avoided during ignition. A burning flame may also be blown out causing some gas to leak before the failure is detected. The gas burner is control led by a thermostat and the gas is ignited by an ignition transformer. The control law of the gas burner is composed by the following phases:

- **Idle**: Awaits heat request; no gas and ignition. It enters the Purge phase on heat request.
- **Purge**: Pauses for 30 seconds. and then Ignite1 is entered.
- **Ignite1**: Starts ignition and gas supply; enters the *Ignite2* phase after 1 second.
- **Ignite2**: Monitors the flame and enters the Burn phase if flame is sensed within 1 second.
- **Burn**: Ignition is switched off, but gas is still supplied. The Burn phase is stable until heat request goes off. The Idle phase is then entered and the gas is turned off.

A simple error recovery procedure of returning to IDLE is used. If a flame is not sensed within 2 sec. in Ignite2 (ignite failure), or if the flame disappears during the Burn phase (flame failure), then the Idle phase is entered and the gas is turned off. The 30 sec Purge pause ensures a sufficient distance between periods with leaking gas.

Figure A.4 presents the TB net specication of the gas burner. The states of the controller of the gas burner are modeled by places:

- IDLE_PHASE, representing the controller waiting for activation (the phase represented in the initial marking);
- IDLE_PHASE_bis, it is equivalent to place IDLE PHASE; it is used only to keep the time the phase is entered;
- PURGE_PHASE, entered as soon as the execution cycle is started (transition HrOn, representing the request of heat). It guarantees the starting of a new phase at least 30 seconds after the idle phase is entered;
- IGNITE_PHASE_B, (Ignite phase begin) entered when the command to start ignition is issued (transition IgnOn);
- IGNITE_PHASE_S, (Ignite phase stable) entered when the command to open the gas valve is issued (transition GasOn). In this phase the controller checks if the flame has been lighted;
- BURN_PHASE_B, (Burn phase begin) the flame is detected, and the normal functioning is started (transition FlameOn)
- BURN_PHASE_S, (Burn phase stable) entered when the command to stop ignition is issued (transition IgnOff). The gas burner is fully activated;
- STOP_PHASE_I, (Stop phase init) entered when the heating request is finished (transition HrOff);
- STOP_PHASE_F, (Stop phase final) entered when the command to close the gas valve is issued (transition GasOff); when the flame is off, the controller entered the idle phase (transition F lameOff);
- IGNIT_FAIL_PHASE, entered if a failure occurs in the ignite phase stable (transition GasOff2); once the exception has been handled the system returns to the idle phase (transition IgnOff2);

- FLAME_FAIL_PHASE, entered if a failure is detected in the burn phase stable (transition FlameOff2); once the exception has been handled the system returns to the idle phase (transition GasOff3);

The system can either require heat or not, by marked places *HeatReq* and *NoHeatReq*, respectively. The transitions of the embedding system are represented by transitions *switchHROff* and *switchHROn*. The gas valve can be closed or open, represented by places *NoGas*, *Gas* (and *Gas_bis*) , respectively. The gas actuator is represented by transitions *CloseValve* and *OpenValve*, and places *ValvActCloseReq* and *ValvActOpenReq*. Place *Ignition* and *NoIgnition* represent the ignition active and not active, respectively. The ignition actuator is represented by transitions *IgnLightOff* and *IgnLightOn*, and places *IgnActOffReq* and *IgnActOnReq*. Places *Flame*, *NoFlame* (and *NoFlame_bis*) represent the state of the flame. The flame is turned on if there are ignition and gas (transition *FlameLightOn*); it is turned off if no gas is supplied (transition *FlameLightOff*) or due to a failure, e.g. wind (transition *FlameLightOff2*). The gas concentration is measured by the number of tokens in place *Concentration*. New tokens are produced regularly in place *Concentration* when the gas is supplied with flame off (transition *Inc_Conc*). Transition *Dec_Conc* extracts from place *Concentration* token older than 30 seconds.

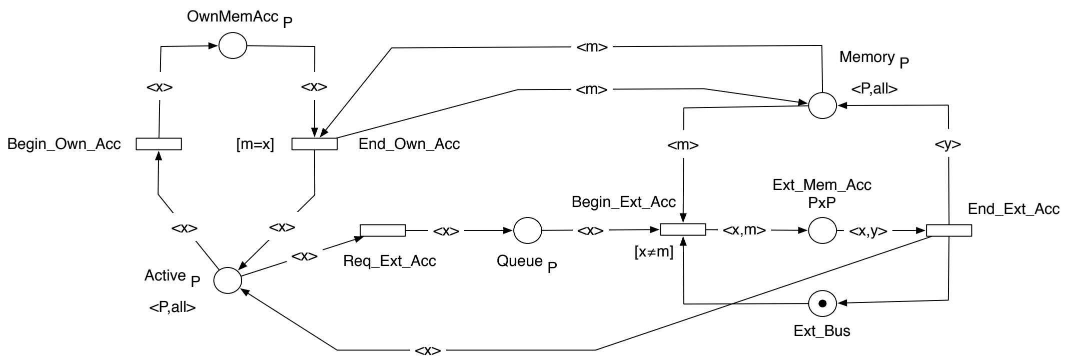
A.2 The Shared Memory

This model is taken from the GreatSPN benchmarks [37, 85, 40]. It models a system composed of P processors, each one with a local memory. Each processor can access its local memory using a dedicated local bus and the other memories using a unique shared bus. The processor accessing a remote memory have priority on those accessing their own memory. It is assumed that external access request causes preemption of the owner processor eventually accessing its local memory.

The model is depicted by Figure A.1. For the sake of clarity, the model is described by a Colored net [81]. Table A.1 shows the size of the model, accordingly to the parameter P. The size values refer to derived P/T net model instances.

Table A.1: Size of the Shared memory derived P/T net model instances.

Parameter	# places	# transitions	# arcs	reachable markings
$P = 5$	41	55	200	1863
$P = 10$	131	210	800	1.831×10^{6}
$P = 20$	100	440	3240	4.451×10^{11}
$P = 50$	2651	5050	20000	5.870×10^{26}
$P = 100$	10301	20100	80000	1.701×10^{51}
$P = 200$	40601	80200	320000	3.524×10^{99}



Class
 P is $1 \ldots 5$;
Domain
 PxP is$\langle P, P \rangle$;
Var
 x,y,m in P;

Figure A.1: The Shared memory Colored net model.

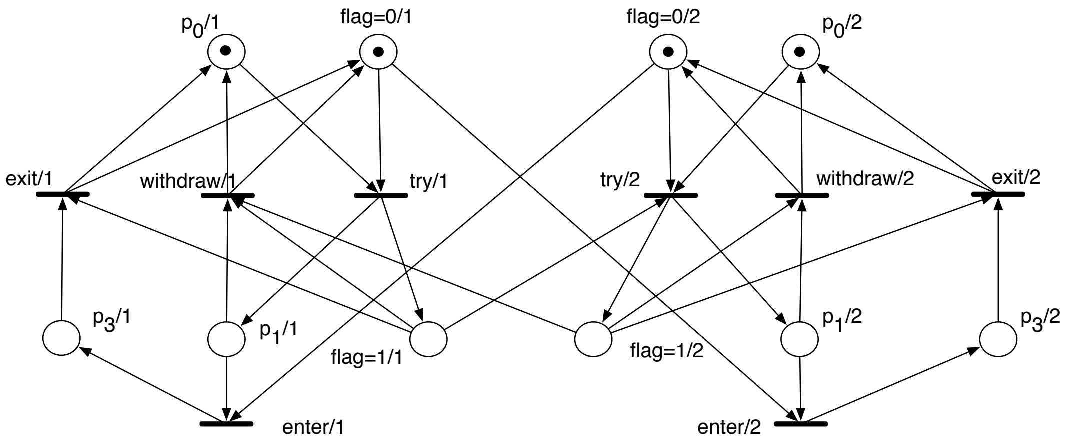


Figure A.2: The Dekker P/T model with parameter $N = 2$.

Table A.2: Size of the Dekker P/T net model.

Parameter	# places	# transitions	# arcs	reachable markings
N	$5N$	$N^2 + 2N$	$O(N^2)$	
$N = 10$	50	120	820	6144
$N = 15$	75	255	1830	278528
$N = 20$	100	440	3240	1.153×10^7
$N = 50$	250	2600	20100	?
$N = 100$	500	10200	80200	?
$N = 200$	1000	40400	320400	?

A.3 The Dekker

This model is a Place-Transition net representing a variant of the Dekkers mutual exclusion algorithm [47] for $N > 2$ processes. Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming. It allows two threads to share a single-use resource without conflict, using only shared memory for communication. Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation. The net models each process with three states, p_0, p_1, and p_3. p_0 is the initial state. From there, the process tries to enter the critical section and raises its flag, reaching p_1. In p_1, if at least one of the other process has a high flag, it withdraws its intent and goes back to p_0. In p_1, it enters the critical section if all other process flag is zero. From $p3$, the process can only exit the critical section.

The P/T net model is depicted by Figure A.2. Table A.2 shows the size of the model, accordingly to the parameter N.

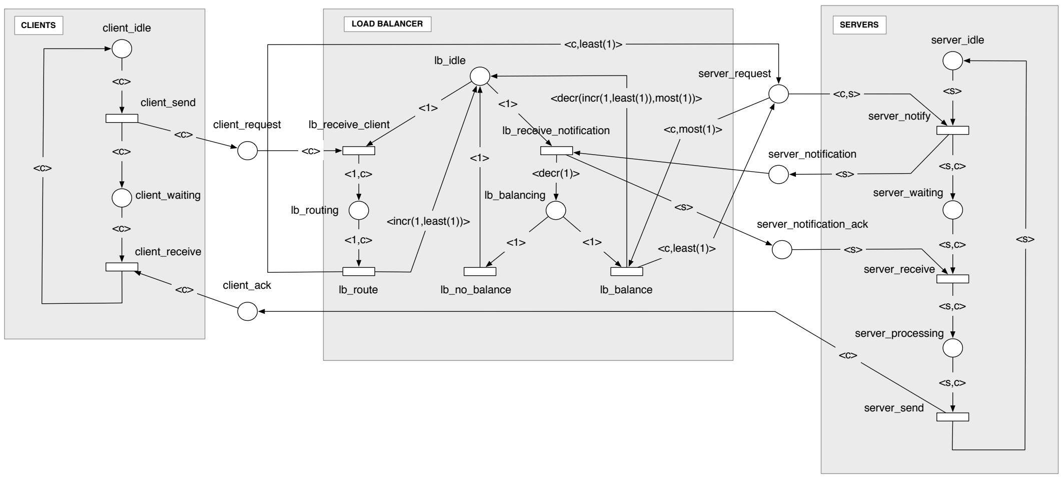


Figure A.3: The Simple load balancing Colored net model.

A.4 The Simple Load Balancing

This P/T net models is taken from the HELENA tool distribution [53]. It models a simple load balancing system made up by a set of N clients, two servers, and between these, a load balancer process so called *lb* process.

The role of *clients* is to send requests to servers (transition *client_send*), wait for an answer and get it (transition *client_receive*). Requests are sent to the *lb* process so that this one routes it to the appropriate server. Once the request is sent, the client waits for the answer. When the answer arrives, the client comes back to the idle state (place *client_idle*). We denote with c, the number of clients. Clients are numbered from 1 to c.

The *servers* waits for requests (i.e., tokens in place *server_request*) from clients sent via the *lb* process. When a server processes a request, it send a reply to the client (transition *server_process*). The server then notifies the *lb* process (transition *server_notify*) in order to rebalance requests among servers. Once the load balancer has acknowledged this notification, the server can go back to the idle state (transition *server_send*). We denote the number of servers with s. Servers are numbered from 1 to s.

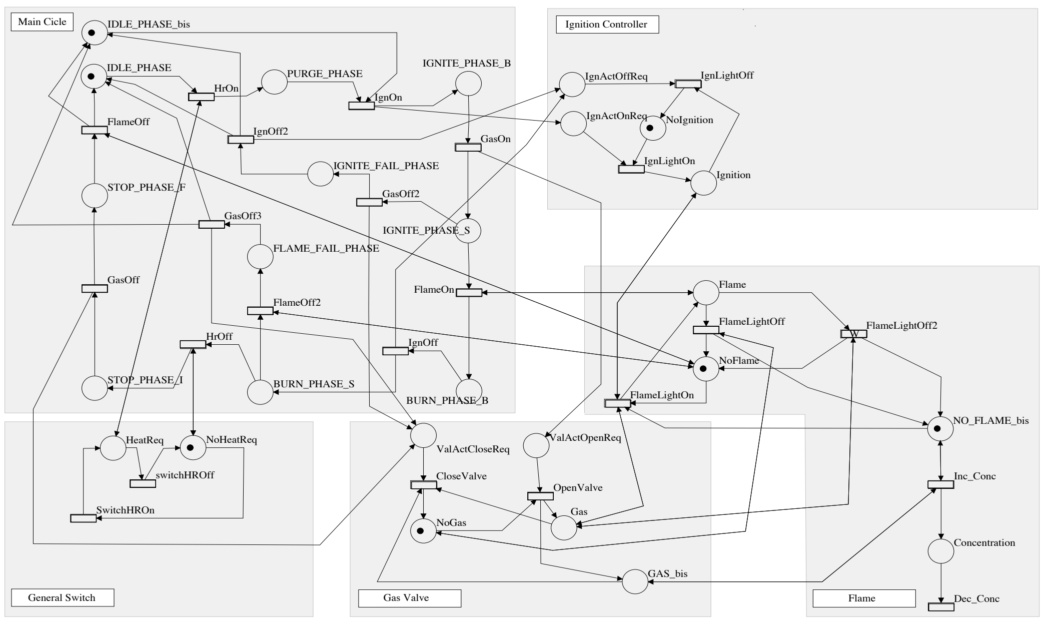
The *lb* process is the most complex component. It can perform two kinds of task. The first one is to redirect each client request to the least loaded server. The latter one, when a server accepts a request from a client the load balancer has to rebalance the pending requests. If these are already balanced, the load balancer has nothing to perform and can come back to its idle state (transition *lb_no_balance*). If the loads are not balanced, the load balancer

Table A.3: Size of the Simple load balancing P/T net model.

Parameter	# places	# transitions	# arcs	reachable markings
$N = 2$	32	45	252	832
$N = 5$	59	180	1158	116176
$N = 10$	104	605	4148	4.060×10^8
$N = 15$	149	1280	8988	1.374×10^{12}
$N = 20$	194	2205	15678	4.583×10^{15}

takes a pending request of the most loaded server and redirects it to the least loaded server (transition *lb_balance*). The load balancer has to maintain for each server the number of requests sent to this server.

The model is depicted by Figure A.1. As in the previous section, for the sake of clarity, the model is described by a Colored net. Table A.3 shows the size of the model, accordingly to the parameter N.



Initial marking:	$IDLE\_PHASE\{T_0\}$, $IDLE\_PHASE\_bis\{T_0\}$, $NoIgnition\{T_0\}$, $NoHeatReq\{T_0\}$, $NoGas\{T_0\}$, $NoFlame\{T_0\}$, $NO\_FLAME\_bis\{T_0\}$
Initial constraint:	$0 \leq T_0 \leq 10$

Time-Functions:

HrOn	$[IDLE\_PHASE + 0.01, max(\{IDLE\_PHASE + 0.01, HeatReq + 0.1\})]$
HrOff	$[BURN\_PHASE\_S + 0.01, max(\{BURN\_PHASE\_S + 0.01, NoHeatReq + 0.1\})]$
IgnOn	$[max(\{PURGE\_PHASE + 0.01, IDLE\_PHASE\_bis + 30\}), max(\{PURGE\_PHASE + 0.01, IDLE\_PHASE\_bis + 30\})]$
CloseValve	$[ValActCloseReq + 0.2, ValActCloseReq + 0.2]$
OpenValve	$[ValActOpenReq + 0.2, ValActOpenReq + 0.2]$
FlameOff	$[STOP\_PHASE\_F + 0.01, max(\{STOP\_PHASE\_F + 0.01, NoFlame + 0.1\})]$
FlameOff2	$[BURN\_PHASE\_S + 0.01, max(\{BURN\_PHASE\_S + 0.01, NoFlame + 0.1\})]$
FlameOn	$[IGNITE\_PHASE\_S + 0.01, max(\{BURN\_PHASE\_S + 0.01, NoFlame + 0.1\})]$
IgnLightOn	$[IgnActOnReq + 0.2, IgnActOnReq + 0.2]$
IgnLightOff	$[IgnActOffReq + 0.2, IgnActOffReq + 0.2]$
FlameLightOn	$[max(\{Gas, Ignition\}) + 0.5, max(\{Gas, Ignition\}) + 0.5]$
FlameLightOff	$[enab, NoGas + 0.1]$
FlameLightOff2	$[enab, enab + 100]$
GasOn	$[enab + 0.01, enab + 0.1]$
GasOff	$[enab + 0.01, enab + 0.1]$
GasOff2	$[enab + 2, enab + 2]$
GasOff3	$[enab + 0.01, enab + 0.1]$
IgnOff	$[enab + 0.01, enab + 0.1]$
IgnOff2	$[enab + 0.01, enab + 0.1]$
SwitchHROn	$[enab, enab + 10]$
switchHROff	$[enab + 120, enab + 120]$
Inc_Conc	$[enab + 0.1, enab + 0.1]$
Dec_Conc	$[enab + 30, enab + 30]$

Figure A.4: The Gas burner Time Basic Petri Net model.

Appendix B

Proofs

This chapter repots proofs of correctness of all the algorithms introduced in section 4.5. Broadly speaking, we prove that algorithms 5, 6, 7 are correct and compute $EX\phi$, $EG\phi$, $E[\phi U \psi]$, rispectively.

In the following proof of correctness, we refer to the semantics of the MapReduce programming model introduced in section 4.5.5.

B.1 Correctness of the $EX\phi$ Algorithm

The algorithm in Figure 5 (Alg_5 hereafter), employed to compute the $EX\phi$ formula, consists of a single *MapReduce* round. In order to prove its correctness we just need to show that Alg_5 computes $R^{-}(\llbracket\phi\rrbracket_T)$ (equation 4.5.1).

Proof: The *Map* function emits each input key-value pair (where key is a unique state identifier, and value is the state itself), and the result of the application of the inverse image R^{-}_{id} to the input values matching ϕ, associated with the empty value.

$$V_1 = \bigcup_{\langle k,v\rangle \in U_0} \mu_1(k,v) = \bigcup_{v\in S} \langle k_v, v\rangle \;\cup \bigcup_{k\in R^{-}_{id}(\llbracket\phi\rrbracket_T)} \langle k, \perp\rangle \tag{B.1.1}$$

After the shuffle, apart from repetitions, each V_1^k contains both values $\perp$ and v, $v \neq \perp$, if and only if v is a predecessor of a state in which ϕ is valid. Hence the set of values of the output of *Reduce* function, which is formally given by

$$U_1 = \bigcup_{k \in V_1} \rho_1(k, V_1^k) = \bigcup_{k \in V_1, v \in V_1^k : \perp \in V_1^k \wedge v \neq \perp} \langle k, v \rangle \tag{B.1.2}$$

actually corresponds to $R^-(\llbracket\phi\rrbracket_T)$. □

B.2 Correctness of the $EG\phi$ Algorithm

The algorithm in Figure 6 (Alg_6 hereafter), which computes the $EG\phi$ formula, is used in an iterative *MapReduce* run, where the output of the i^{th} iteration represents the input of the $i+1^{th}$ iteration. In order to prove its correctness we need to show that the execution of a single round on input X computes the same result as an application of the monotonic predicate transformer 4.5.2 (from now on simply G) on X, and that both Alg_6 and the fix-point evaluation stop at the same iteration.

Let $*Alg_6(S)$ denote an iterative map-reduce run which starts from $X = S$. Formally, $*Alg_6(S) = \nu_X(G(S)) = \nu_X(\llbracket\phi\rrbracket_T \cap R^-(S)) = \llbracket EG\phi\rrbracket_T$.

Proof: The proof follows the schema below:

1) Proof that $Alg_6(X) = G(X)$

2) Proof that $*Alg_6(S)$ and $\nu_X(G(S))$ stop at the same iteration.

By the way, $1) \wedge 2) \implies *Alg_6(S) = \nu_X(G(S))$

In order to prove point 1) we follow the same steps as in the previous proof: we evaluate the output of $Alg_6(X)$, then we verify that it equals $\llbracket\phi\rrbracket_T \cap R^-(X)$.

The *Map* function emits each input pair $\langle k, v\rangle$ such that ϕ is satisfied in state v. In addition it emits all the keys belonging to $R^-_{id}(v)$, associated with the empty value.

$$V_r = \bigcup_{\langle k,v\rangle \in U_{r-1}} \mu_r(k, v) = \bigcup_{v \in \llbracket\phi\rrbracket_T} \langle k_v, v\rangle \ \cup \ \bigcup_{k \in R^-_{id}(X)} \langle k, \perp\rangle \tag{B.2.3}$$

After the shuffle, the intermediate lists V_r^k contain both $\perp$ and v, $v \neq \perp$, if and only if v is a state in which ϕ is satisfied and has a successor in X. The output of *Reduce* function is:

$$U_r = \bigcup_{k \in V_r} \rho_r(k, V_r^k) = \bigcup_{k \in V_r, v \in V_r^k : \perp \in V_r^k \wedge v \neq \perp} \langle k, v \rangle \tag{B.2.4}$$

As a consequence, the values contained in U_r are exactly the same as those obtained by applying the predicate transformer G on X.

In order to prove point 2), we have to show that the condition of termination adopted by $*Alg_6$ (the outputs of two consecutive iterations must have the same size) coincides with the condition of termination of the fix-point evaluation, $G(X_i) = G(X_{i+1})$. For this aim, it is sufficient to prove that

$$|G(X_i)| = |G(X_{i+1})| \Longrightarrow G(X_i) = G(X_{i+1})$$

We proceed by contradiction, assuming that $G(X_i) \neq G(X_{i+1})$. Hence it should be $G(X_{i+1}) = [\![\phi]\!]_T \cap R^-(X_i) = (G(X_i) \setminus A) \cup B$, with $A \cap B = \emptyset$, $A \subseteq G(X_i)$, $B \cap G(X_i) = \emptyset$. Roughly speaking, that means the only way to obtain the set $G(X_{i+1})$ from $G(X_i)$ would be removing some states and adding new ones. The predicate transformer G being monotonic decreasing, this would imply $B = \emptyset$ and $A \neq \emptyset$, i.e., $|G(X_i)| \neq |G(X_{i+1})|$. $\square$

B.3 Correctness of the $E[\phi U \psi]$ Algorithm

Also the algorithm in Figure 7 (Alg_7), employed to compute the $E[\phi U \psi]$ formula, is used in an iterative *MapReduce* run. In order to prove its correctness we need once again to show that the execution of a single round of Alg_7 on input X computes the same output as an application of the monotonic predicate transformer 4.5.3 (L hereafter) on input X, and that both $*Alg_7(\emptyset)$ and the fix-point 4.5.3 stop at the same iteration. Formally,

$$*Alg_7(\emptyset) = [\![E[\phi U \psi]]\!]_T = \mu_X(L(\emptyset)) = \mu_X([\![\psi]\!]_T \cup ([\![\phi]\!]_T \cap R^-(\emptyset)))$$

The formal proof of correctness of the latter algorithm is omitted for the sake of space, due to its high similarity with the previous schema (section B.2).