Scalable Multi-Core Model Checking

Alfons Wilhelmus Laarman

Graduation committee:

Chairman:	Prof. dr. Peter M.G. Apers
Promotor:	Prof. dr. Jaco C. van de Pol
Members:	
Prof. dr. ir. Arend Rensink	University of Twente
Prof. dr. ir. Boudewijn R.H.M. Haverkort	University of Twente
Prof. dr. Wan J. Fokkink	VU University Amsterdam
dr. Keijo Heljanko	Aalto University
Referee:	
dr.rer.nat. Michael Weber	Qualcomm Research, Silicon Valley

CTIT Ph.D. Thesis Series No. 14-308
Centre for Telematics and Information Technology
University of Twente
P.O. Box 217 – 7500 AE Enschede, NL

IPA Dissertation Series No. 2014-06
The work in this thesis has be carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 978-90-365-3656-1
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 14-308)
DOI 10.3990/1.9789036536561
URL `http://dx.doi.org/10.3990/1.9789036536561`

Typeset with LaTeX
Printed by Gildeprint Enschede
Cover design by Iris Cousijnsen, Iris Ontwerpt, KvK 01666142

SCALABLE MULTI-CORE MODEL CHECKING

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof. dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended,
on Friday, May 9th, 2014 at 16:45

by

Alfons Wilhelmus Laarman

born on April 23th, 1983
in Zevenaar, The Netherlands

This dissertation has been approved by:

Prof. dr. Jaco van de Pol (promotor)

Acknowledgments

Prologue. Several circumstances contributed to the exciting and educative time that I enjoyed as PhD in the Formal Methods and Tool group at the University of Twente. I would like to mention a few to provide some more context and meaning to the subsequent thank yous.

As a Master student in the Software Engineering group, I developed an appetite for doing academic research. At that time, sifting through stacks of papers to find an optimal solution to the problem at hand, or simply solving a hard problem like a puzzle, each time felt like a welcome mental exercise, even though communicating the results still presented a difficult and often painful undertaking for me. I was therefore happy that others still considered me a suitable PhD candidate. It was Mariëlle Stoelinga who invited me to apply for a job offer in the Formal Methods and Tools group. I saw it as an excellent opportunity to sharpen my skills and also learn at least something about the art of presenting and lecturing.

Knowing little of the field of formal methods, I went to the job interview slightly intimidated. That made it quite exiting to learn that Jaco van de Pol and Michael Weber agreed to take me under their wings and offer me the job as PhD on the project "Multi-Core Model Checking". At that time, I was mainly preoccupied on becoming familiar with the field of formal methods, hence what I did not expect then, was that this job would offer such an enriching experience, for both my professional and personal life.

Before I could get started however, it was first decided by colleague Mark Timmer that I needed a holiday to rest from the hard work on my Master's thesis. Actually, I agreed, and luckily Jaco and Michael as well. This short holiday ensured that my first working day as a PhD coincided with the Dutch Model Checking Day 2009 (DMCD) organized that year by Jaco and Michael at the University of Twente. Attending the symposium certainly gave an inspiring start to my career.

Lucky enough to land in the same office as my daily supervisor Michael and direct colleague Stefan Blom, I was in an excellent position to learn the craft of the trade. I hope this thesis is indeed proof that I actually learned something; that your efforts were not in vain. For me, being able to work in the stimulating environment that the FMT group provided was already very rewarding.

Thank yous. Jaco, in your role as promotor and later also daily supervisor, you had the most influence in shaping my academic skills, for which I thank you. Our weekly meetings helped a lot to improve the precision of this and other works. Your enthusiasm to work on new parallel algorithms was quite infectious, and also in this area I learned a lot from your rigorous mathematical analysis and insights. I was most impressed by the amount of papers, theses, conference submissions and other work you always manage to take home and review. Even at late deadlines, you helped by proofreading the last revisions in our submission, always resulting in useful comments and suggestions, which I am certain improved our standing with the reviewers. Although somehow we never managed to avoid these scheduling issues, I did learn from the feedback. Indeed, when we both handed in our reviews of the same paper for a conference in 2014, you remarked that our reviews were similar and congratulated yourself on ‘raising’ me well. Quite a compliment!

Michael, thank you for taking me on and being my daily supervisor in those first years. You always provided an excellent overview of the state-of-art in the field of model checking and managed to point me to the interesting open problems. When I received your comments on this thesis, including the discussion of future work, I was reminded of those first years when our discussions often led to so many new ideas. Both you and Jaco gave me a lot of freedom to work out my own plans and pursue my own interests, making the project more interesting but also more challenging. Your way with language was funny (with pun often intended), but was also helpful for writing those early papers together. I also admired your skills in understanding (to a point) something as complex and extensive as the GNU build system, but also the whole process that comes with maintaining a large piece of software. Unfortunately, I cannot say that I learned the craft entirely, but I still hope that the latest LTSmin 2.0 release lives up to your standards. Judging from your feedback on this thesis, the same probably holds for my mastery of the English language, though I should note that currently as an American citizen, you hold an advantage. (On that subject, Michael, I understood that you had to abandon your PhD student in order to relocate with your family to the United States, although I have some difficulty with the fact that you also stole an excellent student and potential colleague, namely Freark, to work for you over there.)

I also thank the committee members for agreeing to participate in my PhD committee, and reviewing this work. Wan Fokkink, thank you for your detailed and useful feedback. As mentioned in Chapter 5, you had taken up an earlier interest in this work and already shared many deep insights before. Keijo Heljanko, Arend Rensink, and Boudewijn Haverkort, thank you for your time to discuss my work and communicate comments on this thesis, it has certainly led to many improvements. Keijo, thanks as well for the pointers to related work that I oversaw.

I like to thank several colleagues, foreign and domestic, for many fruitful collaborations, the results of which are included in this thesis.

Mads Chr. Olesen and Andreas Dalsgaard thanks for your work on the OPAAL frontend, which allowed the utilization of the parallel data structures and algorithms presented in this thesis for the domain of timed automata, and culminated in the publication of a paper at the Computer Aided Verification (CAV) conference, certainly a nice crown on our work. Mads, I especially remember us communicating via Skype those few late hours allowed by our hardly overlapping day schedules when you where in Australia. The papers that the three of us wrote are proudly presented in Part V.

I am als grateful to Kim Larsen for seeing the potential in our parallel algorithms and deciding to join forces to realize scalable multi-core LTL model checking of timed systems. I also value that you found the time in your busy schedule to write recommendation letters for me, this certainly helped.

Anton Wijs and Rom Langerak, thanks for your collaboration on the multi-core nested depth-first search algorithm in Chapter 5. Rom, you really saved my week when you cheerfully entered my office with the statement that you could repair the algorithm after Jaco discovered a counterexample to our proofs. I very much enjoyed your vivid recounts of our fights with the correctness of this algorithm. With you, Anton, I had the pleasure of visiting several remote conferences, in Taipei, Taiwan, and Salt Lake City and Los Angeles, United States, traveling through these countries for some weeks after the conference, or *post-conferenting* as you dubbed it. Indeed, there was not a path that we hiked without us consciously applying the search methods of our freshly invented algorithms and contemplating the different colors of the paths. Indeed, as my colleague Tri Minh Ngo would phrase it: post-conferenting is also work.

David Faragó thanks for your enthusiasm when I contacted you about a possible parallel variant of your DFS$_{\text{FIFO}}$ algorithm. Our resulting publication is included in Chapter 8.

I thank Sami Evangelista and Laura Petrucci for their fruitful collaboration on a new combined version of our respective earlier parallel algorithms, which are similar in some respects, but different enough to complement each other ideally. It was a convenient coincidence that our rivaling parallel algorithms were published simultaneously at the Automated Technology for Verification and Analysis 2011 (ATVA) conference (see [EPY11], and [Laa+11] or Chapter 5). And after Jaco and I showed how the two algorithms complement each other (see Chapter 6), you came up with an excellent idea to combine them. Together, we managed to publish the resulting CNDFS algorithm, as included in Chapter 7, at the ATVA conference the following year.

Elwin Pater, thanks for inviting me to collaborate on publishing your results on partial-order reduction. While these papers are not directly included in this thesis, compatibility with partial-order reduction plays an important role throughout the text. I enjoyed our philosophical discussions on computer science topics, and hope we can collaborate in the future on satisfiability.

I thank Henri Hansen for collaborating with Elwin, Jaco and me on partial-order reduction. While our joint work is not included, it helped me to improve Chapter 8.

Maintaining a toolset like the model checker LTSMIN, which implements the techniques presented in this thesis, is not an easy task. At the same time, having a piece of software to implement and experiment with new ideas, is essential for the kind of research behind this thesis. Other people and organizations were supportive when it came to developing and improving LTSMIN.

The ideas of Jaco and Michael and efforts laid the basis for creation of the LTSMIN toolset. Stefan Blom contributed to the distributed and symbolic backends, a backend to Mark's SCOOP tool for probabilistic systems, and many supporting components like the IO library. Jaco contributed to the μCRL/MCRL2 frontends, and made substantial contributions to the symbolic backends together with Jeroen Ketema. Elwin Pater contributed many features to the toolset in the short time that he was a Master student, including but not limited to: partial-order reduction, LTL, CTL and μ-calculus cross products, trace pretty printing, reordering, and the DiVinE frontend. Tom van Dijk added parallel BDD algorithms, and took over the maintenance of the toolset.

Freark van der Berg and Steven van der Vegt, thanks for your fruitful work on LTSMIN, and the contributions you delivered during your student projects, and afterward as student assistant. Freark laid the basis for the SpinS tool, which functions as a PROMELA frontend to LTSMIN. Extending this tool allowed me to perform extensive and direct comparisons to the state-of-the-art model checker SPIN, which are included in Chapter 11. Steven implemented the testing framework for LTSMIN, and wrote the code for the Cleary table.

I also want to thank Anton Starikov and the Computational Materials Science group at University of Twente for making their cluster available for our experiments. I thank the Linux kernel developers for rapidly providing patches remedying performance regressions on kernel releases from 2007. Anton Starikov was also instrumental in communicating our findings and working with the kernel developers to find and test a solution. This work was of direct importance to me, because it aided the publication at

FMCAD of my first paper on formal methods, which is presented in Chapter 2.

I thank Petr Ročkai and Jiří Barnat for their support on the DiVinE toolset, and Jeroen Keiren for the support on the connection with the mCRL2 toolset.

I thank these people, and the numerous students who helped improving LTSmin, for providing a good basis for my research.

Many people were kind enough to review early drafts of my papers. Here, I like to recount their efforts as acknowledged in those papers.

I thank Cliff Click and Gerard Holzmann for giving helpful comments on an early draft of Chapter 2. I thank Jiří Barnat and Keijo Heljanko for organizing the Parallel and Distributed Model Checking (PDMC) workshop in 2011, and inviting us to write a contribution. The result is now included in Chapter 6. Moreover, I thank Jiří Barnat for his comments on the time complexity of parallel randomized algorithms, and Keijo Heljanko for his pointer to the statistical model in [HJN08]. Stefan Blom provided thoughtful reflection for many ideas in Chapter 3. Elwin Pater proofread the early draft of Chapter 5, and was always around for an interesting discussion on various topics. Thank you both. Mark Timmer, I am grateful for your help with the statistics in Chapter 6, other mathematical questions and the proofread of an early draft of Chapter 8. I thank Mads Chr. Olesen and Christoph Scheben for your useful comments on an early draft of Chapter 8. I am grateful to Antti Valmari, Patrice Godefroid and Dragan Bošnački for their valuable discussions and useful feedback on our work on partial-order reduction, which is used in Chapter 11.

Finally, I am also grateful to the numerous anonymous reviewers that provided useful comments to the submitted drafts of my papers. We all know the review process can be a burden because a good review offers little or no direct rewards. Therefore, I always value those reviews that demonstrate the extra effort on the part of the reviewer to understand your work thoroughly.

The working environment at the FMT group was both stimulating, with many bright colleagues, and also enriching due to its international character. I greatly enjoyed the serious talks about computer science subjects, the philosophical debates during our weekly BOCOM reception, and the interesting stories and perspectives that the various colleagues from all over the world shared, but also the cheerful atmosphere during many events including the Floor Five Film marathons and other group outings. Thank you, Tri, Lesley, Marina, Arend, and Stefano for organizing many of these events.

Amir, Gijs, Mark, Tom, it was nice to get to know you better. Amir, it was a pleasure to share an office with you for a year. You were always talkative and I learned quite a bit from your experience as a postdoc and our discussions about geopolitics, life, and less serious stuff. Gijs, I enjoyed my final attendance of an IPA course, where we traveled together. I also valued your insights in many organizational structures, life and formal methods. When you manage to take a night off from being a busy family man, I always could count on you for taking me up on the offer for a next round of drinks. Mark, sorry I made you my go-to guy for so many things, but you know them so well as evidenced by your results as a PhD and school teacher. I very much enjoyed our joint trip together with my dear Laura through the United States' West Coast. You and Arnd were the necessary condition for the four of us to be able to see so many amazing things in such a short time. I wish you and Thijs the best of luck with the adoption. I am sure your child will have the best examples as parents. Tom, you conveniently may not have remembered that we already worked together on a Physics subject, but I do. I was therefore a pleasantly surprised when Jaco informed me that you were this guy in his class, who toke up the work to parallelize BDDs all by himself. I am happy to have been able to work with you on this subject.

Eduardo, Stefano, Gerjan, Bugra, Dennis, Steven, Afshin, Saeed, Waheed, Mohsin, Enno, Mariëlle, Hajo, Florian, Wojciech, Maarten, Matej, Paul, and Jeroen Ketema thanks for many good discussions and athmosphere during the BOCOM and our many group events. Axel, thanks for always lending a helping hand with my technical issues. Good luck with your thesis. I wish the same for the other PhD students included above. Stefan thanks for often taking care of the BOCOM's beer supply. Also for your always helpful attitude and patience in explaining so many things. I hope that the opportunity arises for us to finally also write a paper together to combine our results. Marieke, after all these years, you became like an aunt to me, but you seem to have this effect on your PhDs and postdocs as well. Joke Lammerink is the organizational force behind the FMT group, but I suspect also the real power broker (which, by the way, she admitted to me, after she had just send Jaco home to prevent him from overworking). Thanks Joke, for lending a helping hand, always friendly and cheerfully I should note, with so many organizational issues, especially concerning the Dutch Model Checking Day these last busy days.

Next to the educative experience, a PhD also provides a stimulating international experience. I had the incredible good fortune to be able to join many interesting conferences, attend great summer schools, meet many inspiring people, and also explore some remote countries where the conferences were hosted.

As mentioned above, Anton, Mark and Gijs made for excellent traveling partners. But I also want to mention Vadim Ryvchin, who I had the pleasure with of going on many exciting hiking trips, in Germany, Russia, and Italy. The Marktoberdorf Summer School, that year held in Bayrischzell, Bavaria, formed an excellent location for us to explore a different mountain every day. We also experienced the inconvenience of ending up on the wrong side of the mountain, exhausted, with the darkness setting in. That time we were saved by a friendly family who gave us a lift by car. Maybe they should not have done that, because apparently we did not learn our lesson. For the next year in Trento, Italy, after an exhilarating hike over a mountain with majestic views, we again ended up on the wrong side of the mountain, totally exhausted, with nightfall setting in. This time William Denman joined us on his flip-flops, or so it seemed (he really had very slippery sneakers, which made the way down over stones a horrible sliding experience for him). By some miracle, the locals saved us again by pointing us to the last open restaurant in their ancient village, which to my mind could have served to shoot the back story scenes of the Godfather. A lovely Italian-Iranian hostess provided us with a lot of great food, which at that moment tasted better than anything I ever ate. The brother of the hostess was called in to drive us 10km back around the mountain, for lack of operating taxi services. With such nice endings, how will we never learn our lesson?

Katharina Spies and Silke Müller, thanks for organizing the renowned Marktoberdorf summer school. Thanks also to Sergio Mover, Musard Balliu, Delphine Demange, Srivathsan Balaguru, Ghila Castelnuovo, Othman Rez, Fabrizio Montesi, Linna Pang, Siavash Soleimanifard, Inna Pereverzeva, and Filippo Del Tedesco for the great time there. And Lukas Bulwahn, thanks for organizing the nice hike in Austria.

With Vasilis Papavasileiou and Vadim, I explored the beautiful Saint Petersburg.

At the UPCRC summer school at the University of Illinois, I was lucky to meet and befriend Can Bekar, Sisu Xi, Verena Kuhlemann, Bryan Nehl, and Yuheng Long.

In Kerela, India, I met Grigory Fedyukovich and Petr Bulychev (again), and we explored a very small part of this magical country. During the social event, I had the pleasure of meeting Daniel Neider and Nils Jansen again. I am happy that I can count on you guys to always be at the most interesting conferences.

I had a great evening with good conversation and many beers in Lugano, after the FMCAD conference, for which I thank David L. Rager and Jakob Zwirchmayr.

My stay in London was a blast due to Mads, Tom, Andreas, and Delphine.

The IPA Spring Days, Autumn Days, and courses were always nice social and educative events. Thank you, Michel Reniers, Tim Willemse, Meivan Cheng for organizing these events.

I always enjoyed working with motivated students and was lucky enough to meet quite a few.

Freark, thank you for asking me to be a supervisor for your project on linking the LLVM to LTSmin. It was certainly an invigorating and very relevant subject. The process had its ups and downs, but the end result definitely worth it. I hope to be able to exploit it soon.

Ronald Brugman, thanks for explaining so many times to me how dynamic partial-order reduction works. I have a lot of respect for the job you did on this difficult subject.

Steven van der Vegt and Simon de Vries, thanks for being such enthusiastic Bachelor students. Steven, thanks for covering the MEMICS conference presentation for me while I was attending another in the United States. It does not always work out so well for Bachelor projects, but when it does the results can be very helpful. The compact hash table that Steven wrote, contributed to my research, and eventually led to a basis for Chapter 4. Simon, good luck with achieving your goals for the project.

Tom van Dijk, thanks for being my first Master student. The best of luck with achieving your goals to further parallelize symbolic model checking. I hope to attend a fierce thesis defense on the subject when the result is there.

Jeroen Meijer, I was not really a supervisor for your Master project, but I enjoyed the exchanges with you on the subject. Thanks for valuing my input enough to warrant it with a present at your graduation. I am looking forward to collaborate more once you assume your new role as LTSmin developer.

Iris Cousijnsen thanks a lot for taking the time in your active life to create such a beautiful cover for this thesis. I am sorry I had to expose you to some of the boring details trying to get an image across of what this thesis contains. You picked up the idea quickly, and I am very happy with the result, which was after all your first shot.

Thanks Koos, Gijs, Merijn, Arjan, Menno and Henk Skatoelakis for so many parties, especially those very memorable ones in Crete. Koos, please thank your mother and father from me for being such lovely hosts. Thank you Ellen, for putting up with us, and joining our trips as an excellent tourist guide.

Paul, Rüben, Nora, Nino, Julian, Franziska, Lisette, Katya, Sarah, Yonga, Marc, thank you for your support and enthusiasm.

Trinley, Ole, Sander, Michel, Michèle, Babet, Merijn, Johanneke, Marja, Xavier, Hans, Sjoerd, Paul, Jeroen, Tommy, Bram, thank you.

Ik wil natuurlijk graag mijn ouders bedanken voor hun ondersteuning en toewijding. Vaak genoeg was ik zo eigenwijs om het niet met jullie eens te zijn, maar ik kan me ook herinneren dat jullie advies om te studeren, zo goed als je kan, altijd veel indruk op mij heeft gemaakt. Dat is zeker overgekomen, ook als ik naar mijn broers kijk. Voor mij persoonlijk kan ik bevestigen dat het een hele waardevolle ervaring was, die me ver gebracht heeft en waar ik nog lang de vruchten van kan plukken. Mama, bedankt voor je hulp met het schilderen van onze schuur, dat waren drie gezellige dagen en de verf zit er nog goed op. Papa, bedankt voor het repareren van het afdakje boven onze voordeur.

Robbin en Fabian, bedankt dat jullie gelijk bereid waren om de rol van paranimf op je te nemen. Ik ben altijd blij te zien hoe goed het met jullie gaat. Ook bedankt dat jullie altijd bereid zijn om te helpen met die dingen waar ik minder handig in ben, zoals het zetten van een raam in de badkamer en het onderhoud van de auto. Toen ik begin 2014 op weg naar papa en mama van de Duitse snelweg afkwam met een rokende motor had ik de auto al afgeschreven. Gelukkig dat Robbin weer bereid was om de helpende hand te reiken en mijn eigen oordeel te overrulen met de legendarische woorden: “laat toch maar even zien dan”. Inderdaad bleek alleen de oliedop van de motor te zijn gesprongen. Sinds we het motorblok opnieuw hebben gevuld met verse olie die middag, rijdt de auto nog steeds prima. Martien, jij bedankt voor de vele gezellige avonden.

Oma, bedankt voor uw onaflatende interesse in mijn onderzoek en uw inspirerende reisverhalen. Als ik terugkom van een mooie vakantiebestemming, vind ik het altijd weer leuk om van u te horen over die keren dat u daar met opa was geweest. Hartstikke fijn dat u samen met Peter mijn verdediging kunt bijwonen om mij aan te moedigen.

To you Laura, I am thankful the most. You had to put up with me these last several months when I was not often available for the fun things in life. I am always amazed to find how well you take care of me when I need it most. Whenever you buy a thing that I did not even realize I needed, I am remembered of that fact. Your help made the strenuous work of completing a thesis more bearable, and also more worthwhile. Luckily, we also had the opportunity to still do many amazing things together. At the top of my list of favorites, are the trips to the USA, Canada, and more often France (we could of course go skiing somewhere else, but then we would miss out on ‘le raclette’). I very much enjoyed our visits to your parents, brother (and his lovely Mariana) and sister in Spain, to whom I am very thankful for their hospitality and engagement in our lives. I hope in the future we have more time and room to make many more exciting trips. Meanwhile, you have to complete your own thesis, so I will practice some patience. I am sure you can complete the work to satisfaction, and I’m looking forward to the following period in our lives together.

Epiloque. After these exciting, tough and educative years as a PhD, I am now looking forward to continue my career in academia as a postdoc in the FORTYSE group of the Technical University of Vienna. I will work with Georg Weißenbacher, whom I am thankful to for giving me an opportunity to work in the exciting field of satisfiability. In his project ‘Heißenbugs’, we will employ SAT and other techniques, to identify and track down bugs in multi-threaded code.

Before leaving, I have to arrange the last bits surrounding my defense and handle the small organizational issues of the Dutch Model Checking Day 2014, which will held on the same day as my PhD defense. In fact, this year’s Model Checking Day is organized by Jaco and me. It will simultaneously be my last working day at the University of Twente, and quite literally the last day at this well-organized, thriving university; the next day, I will depart for Vienna, hopefully a whole lot wiser than at that first working day as PhD.

Abstract

Our modern society relies increasingly on the sound performance of digital systems. Guaranteeing that these systems actually behave correctly according to their specification is not a trivial task, yet it is essential for mission-critical systems like auto-pilots, (nuclear) power-plant controllers and your car's ABS.

The highest degree of certainty about a system's correctness can be obtained via mathematical proof, a tedious manual process of formally describing and analyzing the system's behavior. Especially the latter step is tedious and requires the creativity of a mathematician to demonstrate that certain properties are preserved under the strict mathematical rule system. With the invention of "model checking", this part of this process became automated, by letting a computer exhaustively explore the behavior of the system.

However, the size of the systems that can be "model checked" is severely limited by the available computational resources. This is caused by the so called state explosion, a consequence of the fact that a machine can only perform small mechanized computations and does not exhibit the creativity to make generalizing (thinking) steps. Therefore, the goal of the current thesis is to enable the full use of computational power of modern multi-core computers for model checking. The parallel model checking procedures that we present, utilize all available processor cores and obtain a speedup proportional to the number of cores, i.e. they are "scalable".

The current thesis achieves efficient parallelization of a broad set of model checking problems in three steps, each described in one part of the thesis:

First, we adapt lockless hash tables for multi-core, explicit-state reachability, the underlying search method that realizes the exhaustive exploration of the system's behavior. With a concurrent tree data structure we realize state compression, and reduce memory requirements significantly. Incremental updates to this tree further ensure similar performance and scalability as the lockless hash table, while the combination with a compact hash table realizes small compressed sizes of around 4 bytes per state, even when storing more than 10 billion states. Empirical evidence shows that the compression rates most often lie within 110% of this optimal.

Second, we devise parallel nested depth-first search algorithms to support model checking of LTL properties in linear time. Building on the multi-core reachability, we let worker threads progress semi-independently through the search space. This swarm-based technique leverages low communication costs through the use of optimistic, yet possibly redundant work scheduling. It could therefore become more important in future multi-core systems, where communication costs rise with the increasing steepness of memory hierarchies. Experiments on current hardware already demonstrate little redundancy and good scalability.

Third, to support verification of real-time systems as well, we extend multi-core reachability and LTL checking to the domain of timed automata. We develop a lockless multimap to record time-abstracted states, and also present algorithms that deal with coarse subsumption abstraction for the verification of LTL for solving larger problem instances. The scalability, memory compression and performance are all maintained in the timed setting, and experiments therefore show great gains with respect to the state-of-the-art timed model checker UPPAAL.

The above techniques were all implemented in the model checking toolset LTSMIN, which is language-independent, allowing a direct comparison to other model checkers. We present an experimental comparison with the state-of-the-art explicit-state model checkers SPIN and DIVINE. Both implement multi-core algorithms, while DIVINE also heavily focuses on distributed verification. These experiments show that our proposed techniques offer significant improvements in terms of scalability, absolute performance and memory usage.

Current trends and future predictions tell us that the available processing cores increase exponentially over time (Moore's Law). Hence, our results may stand to gain from this trend. Whether our proposed methods will withstand the ravages of time is to be seen, but so far the speedup of our algorithms has kept up with the 3-fold increase in cores that we have witnessed during this 4-year project.

Contents

Part I

General Introduction

1 Introduction

The topic of the current thesis is the improvement of methods for establishing correctness and identifying faults in digital systems, both software and hardware. The current chapter provides an introduction to this topic which is mainly written for other computer scientists, but which should be understandable for a broader technically oriented audience. Section 1.1 illustrates the importance of correctly functioning digital equipment in our modern society, while Section 1.2 goes on to show that technological advancements make these systems rapidly more complex, thereby increasing the challenge to guarantee their correctness.

Next, Section 1.3 outlines the field of Formal Methods, which aims at establishing mathematically rigorous methods that guarantee dependability for (software and hardware) systems. The thesis focuses in particular on a technique called model checking, which given a formally stated requirement, *fully automatically* establishes correctness of a system, or if the system is *buggy*, returns a counterexample that can be used for reparations. Section 1.4 studies this method therefore in more detail.

The benefit of model checking is that it delivers mathematical proofs in a completely mechanical fashion: The procedure can be implemented as a (software) tool, a so-called *model checker*, which takes the system-under-development as input and can be run by any system engineer, whether expert mathematician or mathematical illiterate. The downside, on the other hand, is that the systems that a model checker can handle are severely limited in size by the available computational resources. Therefore, *the goal of the current thesis is to enable the full use of computational power of modern multi-core computers for model checking*. Section 1.5 outlines this goal and its subcomponents in more detail.

Multi-core processors are quickly becoming ubiquitous because efforts to speedup computers by increasing their clock frequencies have halted the past decade due to physical limitations. Section 1.6 explains this trend and studies the real challenge behind leveraging the power of multi-core processors. It identifies both conceptual difficulties, e.g. the model checking task has to be split up in multiple, more-or-less independent tasks, and technical ones, e.g. modern hardware provides limited memory bandwidth and is hard to program correctly and efficiently.

Nonetheless, the current thesis provides proven, scalable solutions for many important disciplines in the field of model checking. These contributions are summarized in Section 1.7. Finally, Section 1.8 provides an overview of the contents and a reading guide for the current thesis.

1.1 The Societal Impact of Failing Digital Systems

In 1994, Intel released its latest and fastest Pentium processor. Shortly after, the international media reported the discovery of a *bug* in its calculation of floating point numbers. After mounting public pressure, the company was forced to recall the chips, leading to an estimated write-down of almost half a billion dollars [Unk95], not to mention a loss of goodwill. This news event provides a good example of the financial stake that companies have in producing digital systems on a massive scale. Especially considering the fact that only a small percentage of customers decided to go through all the hassle of sending back their processors for the mere problem that it introduces an error in only 1 out of 9 billion floating point division calculations [Hal95] (a defect that probably only affects scientific experiments and not day-to-day office applications and probably not even computer games).

Four years later, in December 1998, NASA send its Mars Climate Orbiter onto a voyage through outer space of 9 months towards the red planet. Upon arrival the $125 million spacecraft promptly disintegrated in the planet's atmosphere. It turned out that the NASA crew communicated with the craft using US customary units, whereas its software "spoke" the international system of units [Ste+99]. This simple mistake not only wasted a lot of money, it also set back the clock on progress in space exploration by several years. Unfortunately, the example represents only a single failure in a long string of at least 10 space exploration missions that either failed or seriously under-performed due to software bugs [Joh13].

The worst examples on the societal cost of failing devices involves those that our life actually depends on, the so-called *safety-critical* systems. Luckily, the practice shows that usually we can depend on the most crucial systems, like autopilots in airplanes, air-traffic guidance systems at airports, board computers that control car engine acceleration and braking, etc. At the bottom-line these systems make our modern fast-paced lives safer. *However, as these electronic systems become more pervasive, our dependence on them rapidly increases.*

Several unfortunate examples remind us of the risks involved. A failing acceleration control system in cars of a certain maker, may have resulted in 37 deaths since 2000 [Hea11], forcing the manufacturer to pay over $1 billion in damages and recall over 8 million sold cars [Zal11]. Other horrendous examples resulting with fatal consequences include: X-ray machines delivered too high radiation dosages [LT93], a race condition triggered in an energy management system caused a two day power outage across large swaths of the north-eastern USA [Pou04], and a round-off error caused Patriot missiles to malfunction which then failed intercept an incoming Iraqi missile [Ske92].

This short historical review of malfunctioning digital equipment constitutes only a small portion of the accidents that became public. And as companies and governments often tend to hide such problems behind the curtains, we may reasonably expect that this is just the tip of the iceberg. Moreover, human behavior quickly adapts to the newly available technology. For example, we pack our bags according to the weather report on our smart phone, and few people ever still bring blankets on long (car) rides to guard for strong weather. *So not only are digital systems becoming omnipresent, we also tend to become more reliant on them in our day-to-day lives.*

All these developments, in conclusion, call for mathematically rigorous methods for the *verification of correctness* of digital systems.

1.2 Parallelism and Moore's Law

The expanding influence of digital systems also led to aggressive investment in their further development. Large companies, such as Intel™, AMD™, and ARM™, were able to manufacture ever faster processor microchips by reducing the sizes of the transistors on the chip's surface. Some economists maintain that these technological advances are at the basis of economic progress over the last decades [Hut09; MD13] and even that the death of the law could cause economic downturn [Dun11]. The flip-side of this development is that these processors become more complicated by the year, which in turn increases the difficulty of programming these devices and the likelihood of the presence of bugs in both hardware and software.

Moore's law [Moo65] stipulates that the number of transistors on a chip doubles every 18 months. This law has held for almost 5 decades after the Intel founder originally coined it. Recent news indicates however that processor manufacturers need to overcome ever larger problems because the structure of the transistors, measuring currently only 8 nanometer in extremes [Cou13], is reaching the physical limitations (a silicon

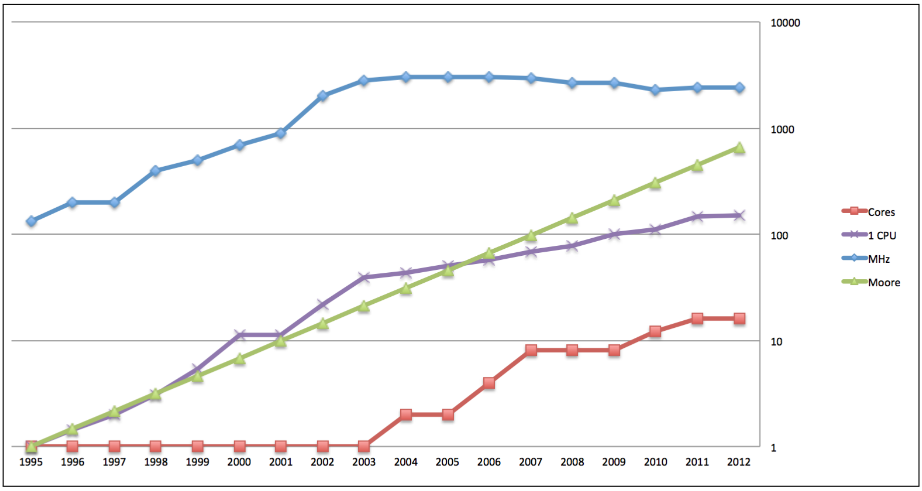


Figure 1.1: Moore's law in practice: Since 2002, CPUs stopped getting exponentially faster as shown by the the MHz line and sequential SPECint benchmark line (1 CPU). Instead, they only become linearly faster and only got 4x faster in 10 years. Also since 2002, however, the number of cores in the systems has increased exponentially. (Taken from [App13])

atom is 0.2 nm in diameter). However, according to many industry experts the law will hold at least for the following decade.

Some believe erroneously that Moore's law is already dead, due to the fact that processor frequencies have plateaued in the previous decade (see Figure 1.1). However, an increase in clock frequencies is merely a consequence of Moore's law. The increase in transistor counts can equally well be used for additional parallelism. Hence in the past decade, we also witnessed an exponential increase in the number of processor cores.

The downside of this development is that the free lunch is over, in the sense that our algorithms do no automatically profit from the next generation of processors (an exponential gain). *Therefore, these algorithms need to be parallelized.* However, parallel programming again adds more complexity. It is well known that the complexity of sequential computer programs can be daunting for even the best programmers, because she needs to consider all the possible *states* that her program can be in. Adding parallelism makes matters worse. As each parallel thread can be at any state in the computation, the number of different states exponentially increases with the number of threads. Often a few threads in a simple procedure already increase complexity beyond our immediate understanding. Depending on the experience of the programmer, project organization, and programmatic abstraction, software is known to contain at least a few bugs per thousand lines of code. Many systems consist of millions of lines of code.

1.3 Formal Methods

As a branch of computer science, formal methods is concerned with mathematical techniques for the specification, development and verification of software and hardware systems. Its primary aim is to establish ways to conceive these systems in such a way that they guarantee their stated requirements in all circumstances. A secondary aim is to establish certain quantifiable characteristics of the system in order to optimize them. For example, to determine probability of failure of the system, so that it can be optimized to achieve the highest possible *dependability* [Bai+03; BCS07; Bou+08]. But a system can also be analyzed using some cost metric, such as its power consumption, throughput, or memory use [AFS04; Tim13].

In the current thesis, we focus solely on verification of correctness, and do not treat the latter, so-called quantitative verification methods. In this case, correctness means the complete absence of errors, i.e. violations of the stated requirements. Verification therefore distinguishes itself from *testing* [MSB11; Luo01], which merely tries to identify errors in a system by trying as many of its execution paths as possible. Even formal approaches to testing [Tre99; BBS06] do not guarantee that errors cannot occur on some obscure, untested path that might happen in practice (for example if the system runs a very long time, or if its environment changes in some way unforeseen by the test cases). *Verification on the other hand provides a formal proof that the system is correct with respect to the stated requirement*, i.e. that all possible behavior of the system respects the requirement. But there is another, more subtle difference between testing and formal verification techniques. While testing allows for the inclusion of the system's *complete* environment, verification does so only in a limited sense, because it requires that all behavior is formally specified. However, this does make the verified behavior explicit, in contrast to testing where any input from the environment might be accidental [Rom99].

In the previous section, we saw the importance of verification for the users of mission-critical and safety-critical systems. For a further discussion of verification methods, it might be useful to discuss how a formal approach to correctness is just as important for the system developers themselves. As systems grow increasingly complex with the exponential growth rates in processor speeds and memory sizes (Moore's law was discussed in Section 1.2), it becomes less feasible for a programmer or circuit designer to maintain an understanding of the complete system that is being developed. A layman may view these engineers as wizards with magic abilities, research however suggests that they are just as limited by their cognitive abilities [SM79], which in turn is famously limited by short-term memory that can track around seven objects at a time [Mil56].

To overcome these cognitive limitations, there is a continuous trend to increase the level of abstraction in programming languages. Despite early objections from for instance von Neumann, the father of the modern stored-program computer architecture, who became infuriated at his student's attempt to create the first assembly language [LL95], imperative languages quickly gained ground with FORTRAN [Bac78]. Later, with Java and .NET, strictly typed languages have become more dominant. The same trend can be witnessed for hardware specification languages, e.g. [Baa+10]. One could argue that the *functional languages*, with their closer correspondence to a mathematical description, would be the logical next step. Indeed, in practice this shift seems ongoing with e.g. the introduction lambda expressions in the Java language.

From the Curry-Howard correspondence [CF58; How80], we know that type systems are a kind of proof systems, so in a sense programmers are already delivering limited proofs for the code they write: types deliver a proof that the computed value is of the correct kind. The above brief history therefore demonstrates the necessity of formal methods in system development.

The next step to a complete proof system would be to prove that the computed result has the right value. Such a full proof system is realized by the first verification method that we discuss now: In *proof carrying code* [Nec02] (PCC), types are in effect replaced by proofs, forcing a programmer to provide a mathematical specification of the computed value at each step of the computation (for each return value / for each assignment), which can be checked by the compiler. Making proofs a first-class citizen of the language of course has the downside that it puts the burden of writing these often long proofs completely on the shoulders of the programmer.

Static analysis, another verification technique, employs a similar, but coarser way, to include proofs in the source code. Code is often annotated (second-class citizen) on the level of functions. The variants of this approach are too numerous to list and evaluate here. *Assertion-based reasoning* could be considered an early version of static analysis, and includes the predicate logic suggested by Hoare [Hoa69], which could be used in assertions and also inspired Dijkstra to come up with a guarded-command language [Dij75]. More modern examples use separation logic [ORY01; IO01] or abstract interpretation [CC77]. Verifast [JP08] is an example of a successful static analysis tool.

Theorem provers, on the other hand, separate the proof obligations completely from the code by expressing them in a functional formalism, which allows an automated way to discharge them. These tools have a long history with early successes [DLL62], and recent tools like Isabelle/HOL [PW02] have proved valuable for the machine-based verification of a large set of algorithms and mathematical theorems [Sut09]. Recently, they are also used to generate executable code from the proof specification to automatically derive a correct functional program [Esp+13].

Finally, model checking could be considered the most automated method of verification. It operates under the assumption that the system under verification has a finite number of states, or configurations, which can be modified through (internal) execution steps, or state transitions, in the system. The method then explores all reachable states to find states or traces (execution paths) that violate the requirement in question. This way, it mechanically *checks* whether a system M is a *model* of a property φ, or stated mathematically: $M \models \varphi$. The requirement needs to be stated in some concise formalism. The system is often also expressed in a more mathematical formalism, e.g. a process algebra [Gro+08] or some domain-specific language [Hola]. However, in software model checking [HS99; JM09; BNR09], the implementation of the system is used directly in the verification process, lowering the entry threshold for users significantly.

> The exhaustive exploration of all reachable states, makes model checking a completely automated method for proof derivation. It is therefore the topic of the current thesis.

1.4 Model Checking

Figure 1.2 shows the workflow in model checking. The model checker tool is represented as a box. It takes as input a formal description of the requirement (φ), which we will call the *property*, and a formal description of the system (M), which can be modeled in some concise specification language. It is often pointed out that the mere task of *formalizing* property and system in this way already improves the engineer's understanding of both system and requirements (represented by the cloud-shaped nodes in the diagram), potentially eliminating existing inconsistencies upfront [BK08, Section 1.1]. In the literature, this formalization process is often referred to as "modeling" the system/property, though this somewhat inaccurately describes the originally intended meaning of the mathematical 'model of' relation as described in the preceding section [Cla08]. The formalization is often done manually, but can be automated for example by translating the system [Cla08] or by using its implementation directly as done in *software model checking*.

Depending on the nature of the system-under-verification, different languages are used to formalize its behavior. Software systems, such as communication protocols and controllers, can be expressed using (extended) state machines (to which about any programming language can easily be translated). If the system includes crucial timing

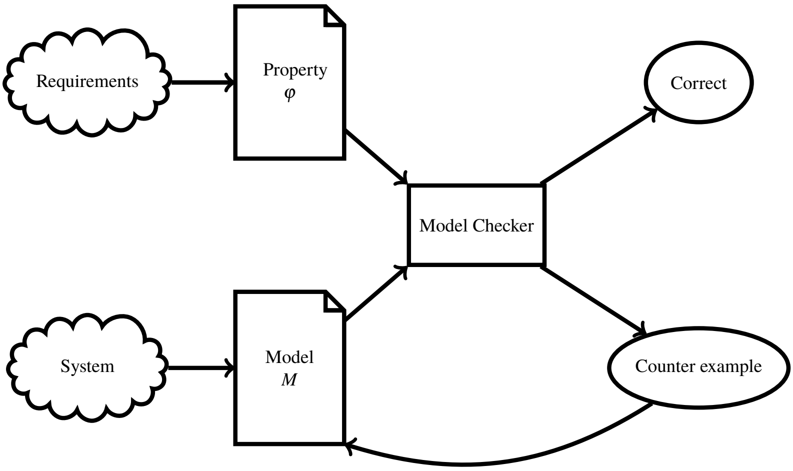


Figure 1.2: The workflow in modelchecking

behavior, e.g. a real-time system, or interacts with analog physical components, e.g. a thermostat, it can be modeled using *timed automata*, or their superset: *hybrid automata*, which model this behavior using continuous variables, e.g. *clocks*. Probabilistic and stochastic behavior can be expressed using *probabilistic automata*. For all of these types of systems, multiple alternative formalisms exist, e.g.: timed Petri nets [Ram74], timed process calculi [BB91] and probabilistic process algebras [Tim13].

Via exhaustive exploration of the system's state, while taking into account the semantics of the property, the model checker can prove the system's correctness, or more precisely that the system is a model of its requirements: $M \models \varphi$. If the opposite however is true, there exists some state in the system or some execution through the system, that violates a stated requirement in the form of a property. In this case, a nice feature of the model checker is that it is able to deliver a *counterexample* in the form of an execution trace. The counterexample can then be used to improve the system specification and/or the property (the latter is not drawn in the figure).

A distinction is often made between safety and liveness properties. Safety properties state properties of the kind: "nothing bad ever happens". While liveness properties *also* reason over (infinite) paths: "eventually something good happens" [BK08]. Since safety properties reason over individual states and actions, it suffices to find a finite trace in which the property is violated to demonstrate that the property does not hold. Many simple safety properties, such as *deadlocks* and invariants, can be checked by establishing reachability of states in which these properties are violated (which can be checked locally on a state). Liveness properties, on the other hand, require a more complicated analysis.

To express liveness properties several logics were developed. It is still up for debate which is the best suitable for model checking [Var01]. The *computational tree logic* (CTL) is a *branching-time logic* that expresses properties over *some or all* paths in the system [BK08, Chapter 6]. The problem of CTL model checking was shown to be linear time in both the size of the system $|M|$ and the size of the property $|\varphi|$ [EL87]. While *linear temporal logic* (LTL) on the other hand, expresses properties over *all* paths and is linear in $|M|$, but exponential in $|\varphi|$. However, many properties of interest can be expressed exponentially shorter in LTL [BK08, Chapter 6]. Moreover, the expressiveness of LTL and CTL are incomparable, i.e. both languages contain properties that are inexpressible in their counterpart [BK08].

To overcome the limitations of CTL and LTL, several other languages have been invented. CTL* is a branching-time logic that expresses a superset of both LTL and CTL. The modal μ-calculus expresses a far broader set of properties, but its general checking procedure is also more complex than LTL [Eme97]. Although the subset of the μ-calculus needed to express LTL properties can be checked as efficiently as LTL itself [CGR11] and also on-the-fly [MS03], there are still important advantages to LTL-based model checking: *compositionality* of formulae, understandability of formulae, and its *automata-theoretic* approach [Var01].

1.4.1 An Archeology of Model Checking

Recounting Edmund E. Clarke's "The Birth of Model Checking" [Cla08], we find that the earliest exhaustive state exploration techniques can be traced to Bochmann [Boc78], who used it for the verification of protocols. Around the same time, Holzmann also worked on similar methods for concurrent system and protocol verification [Holb], which were not implemented until 1980 [Hol81].

The novelty of the contribution of Clarke, Emerson and Sifakis [CE82; QS82], that eventually won them a Turing award[CES09], was their combination of exhaustive state exploration with Pnueli's [Pnu77] definition of *temporal logic*. The latter constitutes an ideal formalism for expressing all kinds of (liveness) properties over program executions. While Hoare logic from 1969 [Hoa69] only allowed the expression of functional properties over statements and functions in a program, temporal logic takes entire execution paths into account.

The EMC model checker, developed by Clarke [CES86], implements an algorithm that checks *computational tree logic* (CTL) in time linear to $|M|$ and $|\varphi|$. Later, the automata-theoretic approach for LTL checking was developed, replacing the property with an ω-automaton that expresses infinite paths [VW86]. Both approaches are called *explicit-state* model checking, since the state descriptors are represented as raw data, as opposed to a mathematical (symbolic) description of the system's states and transitions.

Further developments in model checking allowed larger systems to be verified by reducing or compressing the exponentially-sized state space. E.g., partial-order reduction [Ove81; Val88; KP88a; God90] was introduced to prune traces from the transition system that are not of interest to the property that is being verified. It can yield exponential reductions [Val98]. McMillan [McM92] used *binary decision diagrams* [Bry86] to symbolically represent the transition system (state space and transition relation) concisely. In Section 1.4.3, we detail these and other methods to combat state-space explosion. In the following section, we first illustrate some successful applications of mode checking.

1.4.2 Model Checking Successes

Model checking was hugely successful. The early EMC tool found bugs in existing published circuits [Cla08; QS08].

Holzmann is another pioneer in the development of model checkers. His early `pan` verifier [Hol81], a predecessor to SPIN [Hol11], was successful in identifying bugs in existing protocol specifications [Hol81]. SPIN was later used to verify the FireWire protocol [LRG03], subsystems of NASA's Mars rover [HJ04], and many communication protocols [Hol90; Hol91].

The model checker MURφ was used to verify cache coherence protocols [Che+07] and cryptographic protocols [MMS97]. While the process-algebraic model checkers μCRL [Blo+07] and mCRL2 [Gro+08; Cra+13] were used to verify industrial case studies and communication protocols.

PRISM [KNP11] solved a large set of (probabilistic) communication, network, and multimedia protocols. And UPPAAL was used verify industrial case studies and protocols. PAT [LSD11] showed successes in the verification of sensor networks and real-time systems.

SLAM [BR01; BR02] won considerable respect in the verification community for its successful application of symbolic verification techniques to solve the problem of checking device drivers for Microsoft™ Windows. Using related techniques, though oriented more towards testing, the tool SAGE [GLM+08] "fuzzes" for bugs in a wide array of Microsoft products. This technique symbolically generates different inputs for white box testing. It runs 24/7 on very large scale clusters to identify as many faults as possible in new software releases. This the investment in this expensive effort is quickly repaid, because each patch release that is avoided, saves the company millions of dollars.

1.4.3 Dealing with State-Space Explosion

The exhaustive state exploration technique discussed above, expands the system description M as a transition system or Kripke structure [Kri71]. Such a structure is the equivalent of the mathematical definition of a directed graph or *digraph*, annotated with additional labels at the vertices and/or arcs (directed edges). Computations of statements or functions in the system constitute the arcs/transitions in the graph/transition system. These transitions often lead to a new global state in the system under verification, with different program counters and data values than the source state of the corresponding transition. We simply refer to the full descriptor containing all these variable valuations: *the state* or *state descriptor*. This term is used as synonymous for the vertex/ node in the graph / transition system that it is part of.

We call all possible valuations of the variables in a system, i.e. program counters, communication channels and data variables, the *syntactic state space*. The number of reachable states is often only a small subset of the syntactic state space and can be obtained via the exhaustive state exploration. Therefore, we often simply call this procedure *reachability*. Reachability starts at the initial system state and searches for new states, by executing all possible transitions at a state. To this end, it needs only to store the stack of the search to avoid infinite repetition in the search as Savitch's algorithm for the STCON problem demonstrates [Sav70]. However, the different paths through the transition system are often so numerous that such an approach leads to exponential complexities (in the number of reachable states and transitions). For this reason, the search procedure often maintains a set containing all previously visited states. Upon completion of the reachability procedure, this set then contains all reachable states, which we will call the *semantic state space* or simply *state space*.

For efficient model checking, the state space needs to be stored in main memory. This memory is fast enough to verify systems in reasonable time, while at the same time large enough to explore systems of interest. A state space is know to be exponential in the size of the system, i.e. the number of (parallel) components, data variables and channel buffers. Moreover, a property can also be exponential in the size of the predicates it contains and is often combined with the state space using some cross product procedure, leading to even larger state spaces. *Therefore, dealing with state-space explosion is the most important problem in model checking.*

1.4.3.1 Explicit-State Techniques

The main dichotomy in model checking approaches is between explicit and symbolic techniques. In the explicit approach, the reachable states are stored in full, as vectors containing the data values of the variables in the system, while the symbolic approach uses mathematical equations to represent entire sets of states. The preferable approach depends on the nature of the input system: hardware systems and other systems with high parallelism tend to be handled effectively with symbolic approaches, while software systems usually are better verified using explicit approaches.

On-the-fly exploration. The main advantage of the explicit approach is that states are processed individually, so the search can be limited to relevant parts of the state space. The so-called *on-the-fly* exploration in automata-theoretic model checking [VW86] synchronizes the state-space exploration with the steps in the automaton representing the property. When the system contains bugs, the exploration can stop upon detecting a counterexample, saving computational resources to explore (and store) other parts of the state space.

State compression. To reduce the memory requirements for model checking systems with large state descriptors, we can compress states. One option is to apply lossy compression and only store one or multiple hash values per state in a hash table [GVR99; WL93; DM09] or a *Bloom filter* [DM04]. The downside however is that these approaches, i.e. supertrace/bit-state hashing [Hol98] and hash compaction [GVR99], sacrifice the typical completeness property of model checking. In the case of liveness verification, soundness is also problematic [BHR13].

State-space caching. To improve upon the previous methods, a model checker can further attempt to avoid the storage of states in memory completely by heuristically caching only parts of the state space. A trivial technique is to employ depth-first search (DFS) and only store the stack [Sav70] (as opposed to storing all visited states). This may result however in exponential runtime. Therefore, the DFS technique is often combined with state-space caching [GHP95]. Better heuristics can aid the effectiveness of the caching, for example by focusing on states that form entry points of cycles in the graph [BLP03]. With proper heuristics, the technique can even be used without its dependency on DFS [BLP03; MW09] (accomplishing completeness by guaranteeing progress).

Alternatively, to avoid state revisits, the visited states can also be off-loaded to the much slower hard disk [SD98]. By heuristically selecting states that are encountered less often, the performance penalty can be minimized [Pen+02].

Both techniques can also be combined [HW07].

1.4.3.2 Symbolic Techniques

With their ability to exponentially reduce the size of the state space by expressing large sets of states using simple mathematical formulae, symbolic techniques greatly contributed to solving the state-space explosion problem. However, they may also increase the size of a state space exponentially if its structure is not combinatorial, so they must viewed as an orthogonal approach to explicit-state model checking.

Binary Decision Diagrams. One way to mitigate the state-space explosion is to symbolically represent subsets of the syntactic state space using *(reduced and ordered) binary decision diagrams* (BDDs) [Bry86]. However, BDDs only efficiently support set operations, and have horrible performance for adding single states, as the model checking procedure as described above, demands. Therefore, the transition relation also needs to be expressed symbolically [McM92]. The downside is this somewhat limits the input language of the model checker to reflect the relational nature of the transitions. Moreover, some arithmetical operations, such as multiplication, are therefore more expensive. However, BDD-based symbolic model checkers [Bur+90], such as NuSMV [Cim+02] still enjoy huge successes due to their suitability for checking highly-parallel systems, such as hardware.

Despite slow BDD updates, some explicit-state model checkers still employ BDDs to compress the state space [Gre96; Vis96; HP99]. The model checker LTSmin uses the same approach, but mitigates the resulting runtime penalty by learning the *partitioned* transition function in a piecemeal fashion [BPW10; BPW09].

Boolean satisfiability. The contribution of Davis and Putman [DP60] in 1960 resulted in a revolution in (Boolean) satisfiability solvers (SAT). Eventually these solvers where used for model checking [Bie+99b; Bie+99a; Bie+03; DKW08], by generating a propositional formula that described executions of the system-under-verification up to a bounded length k. The method was later greatly improved using an inductive on-the-fly generation of the transition relation by Bradley [Bra11].

For software systems, it is often infeasible to express the state transitions as a Boolean formula because the use of large data variables, together with their arithmetic operations, are expensive to express using only Boolean connectives. For this purpose, researchers have sought to solve the "satisfiability modulo theory" (SMT) problem [NOT06], which is essentially the SAT problem extended with predicates from different, higher-level theory. The theory can be either (fixed-sized) bit-vectors, natural numbers, real numbers, etc. SMT solvers have successfully been used in the model checking procedure [AMP06].

While the successes of bounded model checking quickly replaced BDD-based methods, the methods should be considered complementary, as there are problem instances which can be solved efficiently with either BDDs or SAT, but not with both [GZ01].

1.4.3.3 Orthogonal Techniques

Some general state-space reduction methods can be applied, regardless of whether we employ symbolic or explicit techniques.

CounterExample-Guided Abstraction Refinement. *CounterExample-Guided Abstraction Refinement* (CEGAR) [Cla+00] uses over-approximating abstractions to mitigate the state-space explosion. Upon detection of a counterexample, its feasibility is checked in the original system. When the counterexample is infeasible, the abstraction is refined (possibly locally) and the checking process is reiterated. This technique is more naturally expressed using symbolic techniques [Cla+00; CGS04; Cla+02], but can also be used for explicit and hybrid systems [BL13b; Cla+03].

Partial-order and confluence reduction. Independence and commutativity between transitions in concurrent systems can be exploited with *partial-order reduction* (POR) [Ove81; Val88; Val89; KP88a; God90]. Exhaustive verification needs to consider only a subset of all possible concurrent interleavings, without losing the global behavior of interest to the verified property. In practice, the state space is pruned by considering a sufficient subset of successors in each state.

A a related approach is *confluence reduction* [BP02], which achieves the same goal. For probabilistic systems with branching-time logics [TSP11], confluence reduction was shown to deliver reductions at least as good as POR.

Partial orders have been shown to be crucial for feasible solutions to the model checking problem of systems with relaxed-memory semantics [AKT13].

Symmetry reduction. *Symmetry reduction* [NIPD96; Cla+96; ES96] prunes interleaving behavior of identical system components by using an equivalence relation over state descriptors induced by permutation groups (of the identical components). Partial-order reduction can be considered orthogonal to symmetry reduction, because both approaches leverage reductions from different aspects of the system [EJP97]. Wahl et al. [EW05] developed dynamic symmetry reduction to extend the approach to systems with imperfect symmetries.

1.4.3.4 Language-Specific Techniques

Some approaches to combat state explosion have been developed for specific specification languages. For example, McMillan [McM93] defined an unfolding technique for Petri nets, a formalism ideally suitable for specifying parallel systems, which avoids the exponential explosion caused by exhaustive exploration. The technique can be adapted for use in other formalisms, such as *synchronous products of labeled transitions systems*, although its details may depend on the formalism to which it is applied [EH08]. Other examples involve the symbolic relation induced by the time-abstracting constraint systems for finite representations of *timed automata* (TA) [Dal+11].

1.4.3.5 Techniques using Parallelism

Finally, we can also choose to increase the amount of hardware resources dedicated to solving a model checking problem. The availability of large- scale computing clusters makes it possible to distribute the problem over individual machines and communicate results via a network (LAN or WAN). The machines themselves can be scaled to contain multiple processors or even multiple processor cores on a single chip.

Distributed systems. Distributed systems have a long history of being used for model checking. The benefit of this approach is that the available memory increases linearly with the number of machines used. A difficulty is however to split the model checking problem in such a way, so that the communication among those machines is minimized.

Symbolic approaches have been distributed by Grumberg at al by splitting BDDs using a so-called window function [BD+00]. The added benefit was that in some cases the BDDs tend to reduce in size due to their partitioning, as other experiments seem to indicate [GHS01]. Other works investigate the distribution of *timed automata*, which express systems with real-time behavior [Beh05; BHV00].

Explicit-state model checking has been distributed with the work on SPIN [LS99], MURφ [SD97; Bin+10], CADP [Gar+07; GMS12], Groove [BKR10] and DiVinE [Bar+10; Bar+06]. Other (language-independent) distributed approaches that use state compression techniques to reduce network traffic [Blo+08a]. The size of the distributed state space can be further reduced using distributed bisimulation reduction algorithms [BO03; BO05].

Multi-core and multi-processor systems. More recently, multi-core and multi-processor systems are becoming more prevalent, as discussed in Section 1.2. These systems share the main memory subsystem between all the available processing cores.

Communication is therefore not as expensive as in the distributed case, but also less transparent because the memory is presented as a single virtual address space to the user-space programs (see Section 1.6). Model checkers can benefit from the performance of multi-core processors [BBR09b; HB07; Hol08; IB02; BR08]

> The current thesis focuses exclusively on methods to speedup the model checking procedure for multi-core and multi-processor systems. We do however also consider combinations with other techniques that deal with state-space explosion, such as partial-order reduction as discussed in section Section 1.5. Section 1.6 explains the intricacies of parallelism.

1.5 Scalable Multi-Core Model Checking

1.5.1 Problem Statement

As mentioned above, the model checking procedure is severely limited by the (exponential) state-space explosion. At the same time, the method does not exploit the increasing amount of parallelism of modern multi-core processors. To benefit from the exponential performance increase of each next generation of processors, model checkers need to be parallelized.

1.5.2 Limitations and Existing Contributions

Prior to the commencement of the research project that led to the current thesis, a few researchers had already recognized the importance of this approach and proposed parallel solutions for model checking on shared-memory machines.

Brim, Barnat and Ročkai [BBR09b] implemented a parallel model checker DiVinE for multi-processor systems. Their results were promising, but unfortunately the runtime of some of their parallel algorithms could become quadratic in the worst case, while their sequential counterparts remain strictly linear-time. Furthermore, the algorithms exhibit limited scalability on multi-core systems.

The spin model checker also saw several attempts to revise its algorithms for multi-core machines [HB07; Hol08]. A wise choice was made here to maintain as much backward compatibility with earlier implemented algorithms. Unfortunately, little speedup was obtained. Inggs and Barringer [IB02] present a way to parallelize reachability using an imprecise state store. The work resulted in reasonable speedups on older SGI machines, but the method is inherently unsuitable to support liveness algorithms like

OWCTY, because the correctness of such algorithms depends strongly on precise state counting arguments.

Specific solutions for shared memory machines with shared state store where not investigated to satisfaction. Some researchers therefore were convinced that scalable parallelization of model checking operations was limited to input with large data sizes (state vectors), long transition delays (next-state computation) and high branching factors [HB07; Hol08]. Others believed that the right algorithm/implementation has yet to be invented [BR08].

Researchers agreed that parallel linear-time algorithms for checking LTL properties remained an important open problem [HB07; BBR10b].

1.5.3 Research Questions

In the first place, we want to realize efficient procedures for parallel model checking on multi-core machines. The ideal to strive for is obviously a speedup that equals the number of cores used. If this a not attainable, at the least a linear speedup would provide some indication that the algorithm will also scale beyond the number of currently available cores. We therefore ask the following research question:

Main research question

Can the model checking procedure scale, linearly or ideally, on modern multi-core machines?

We interpret model checking in the broad sense and aim at supporting different specification languages and properties. Some specification languages, like timed automata, add symbolic properties to the state space and therefore require different algorithms. Furthermore, the verification of liveness properties requires different algorithms than the safety properties because these reason on paths in the state space. All these different algorithms need to be parallelized individually.

The inherent difficulty of parallelism demands that we require some proof of correctness for new algorithms and/or data structures. At the very least these proofs should be on the higher algorithmic level, where we can easily reason about mathematical properties. For data structures, a limited implementation with some abstraction could be model checked to provide some confidence in their correctness. We do not require that the implementation themselves is completely verified, as this is often infeasible. So concretely, we need to demonstrate correctness:

Subquestion 1

Are our proposed methods for multi-core model checking provably correct?

Finally, a parallel model checker that efficiently uses all the available processors/-cores is a great tool for dealing with state-space explosion considering the predicted technological advancements according to Moore's Law (see Section 1.2). However, without other techniques, such as partial-order reduction, on-the-fly model checking, state compression, etc, it can hardly compete with other sequential tools. Therefore, we should keep an eye on compatibility with these methods:

Subquestion 2

Are our parallel model checking procedures compatible with other existing approaches to tackle the state-space explosion problem?

1.5.4 Approach

Different model checking problems have different complexities. On the first axis comes the input specification, which might add complexity to the exhaustive exploration of the state space. For example, to obtain a finite state space for timed automata a symbolic abstraction is needed, which complicates the comparison and storage of states [BHV00]. The following enumeration shows different formalisms in their increasing complexity:

1. explicit-state formalisms (SPIN's PROMELA [Hola], DiVinE's DVE [Bar+10], mCRL2's process algebra [Gro+08], etc).
2. timed automata (UPPAAL's [LPY97] timed automata),
3. hybrid systems [Man+13], and
4. probabilistic systems (PRISM's probabilistic TAs [KNP11] and Scoop's MAPA [Tim11]).

Another axis of increased complexity we find in the property specification:

1. Reachability or safety properties,
2. *linear temporal logics* (LTL) and *branching-time logics* (CTL), and
3. the *modal μ-calculus*.

It is not feasible to try to parallelize all of the above problems at once. We can however start with finding a solution for the explicit-state exhaustive exploration problem, or reachability, which already solves many safety properties, such as deadlocks, for explicit-state systems. In fact, when no efficient solution can be found for parallel reachability, it is also unlikely that we can find efficient solutions for LTL checking or the checking of timed automata. Therefore, it is crucial that we initially focus on scalable parallel explicit-state reachability.

From there on, we can focus on more complex problems on both axes, such as LTL checking and the checking of timed automata. Linear-time solutions for parallel LTL checking are still an important open problem [HB07; BBR10b], and therefore would logically be the next goal on the list. From its advantages compared to CTL as outlined in [Var01] (see also Section 1.4), LTL also seems the logical first choice. Later, we could also investigate CTL and CTL* logics starting from the algorithms in [Fis+01].

Once the explicit-state case has been solved, the same methodologies can be applied for the parallel checking of timed automata [Li09], and possibly the more general hybrid automata. While distributed versions of UPPAAL exist [Beh05], their scalability was only established on decade-old machines, and likely is not preserved on modern multi-core machines (an implementation is also not available to check this). Furthermore, the UPPAAL tool is limited to checking a small subset of CTL. Moreover, the problem combining LTL checking with rigorous time-abstractions is still an open problem [TYB05].

Completely symbolic model checking can be considered last. Because SAT-based techniques are NP-complete, many (heuristic) approaches are equally valid for different sets of input problems, making a portfolio-solution a natural fit [Xu+11], as we will discuss in the following section. Nonetheless, it was recently shown that clause learning can be shared among the solvers [WH13a; WH13b].

BDD-based structures can be parallelized by distributing them using window functions as discussed at the end of the previous section. For multi-core machines however it could also be useful to parallelize the individual operations on the BDD structure. These operations traverse a large directed, acyclic graph (DAG). Therefore, the problem of parallelizing these BDD operations is similar to that of parallelizing the exhaustive (explicit) state-space search, though with many more constraints [Bry86]. Hence we can expect that a successful parallelization of BDD-based model checking depends on the success of parallelizing the explicit approach.

Due to the different nature of the symbolic approaches to model checking, the parallelization of these techniques is likely orthogonal to the parallelization of the explicit-state techniques, meaning that their solutions will have little in common and can be considered independently [DLP13].

One requirement for the parallel checking methods is compatibility with the other state-space reduction methods discussed in the previous section. Without e.g. partial-order reduction the gains of parallel model checking would diminish greatly. Also state compression and on-the-fly verification are important capabilities that should be preserved. These considerations should therefore be taken into account from the start.

Table 1.1 shows an overview of the large research area that we discussed in the order suggested in the current subsection. The first column shows the main formalisms, the second column the different kinds of properties that we discussed, and the first row the different approaches and some useful reduction techniques that can be applied for an approach. In the introduction of every thesis part, we will discuss which open questions are solved in that part based on Table 1.1. In the conclusions, we provide an overview of the solved open questions (see Chapter 12). To wit: explicit-state reachability and LTL checking, for explicit and timed specifications, combined with most state-space reduction methods (areas circled in the table).

Table 1.1: Open questions in the area of multi-core model checking.

Formalism	**Property**	**Explicit state**	**+ On-the-fly**	**+ Compression**	**+ POR**	**BDD-based**	**SAT-based**	**Distributed**
Explicit	Reachability	?	?	?	?	?	?	?
	LTL	?	?	?	?	?	?	?
	CTL	?	?	?	?	?	?	?
	μ-calculus	?	?	?	?	?	?	?
Timed	Reachability	?	?	?	?	?	?	?
	LTL	?	?	?	?	?	?	?
	CTL	?	?	?	?	?	?	?
	μ-calculus	?	?	?	?	?	?	?
Stoch.	Reachability	?	?	?	?	?	?	?
	LTL	?	?	?	?	?	?	?
	CTL	?	?	?	?	?	?	?
	μ-calculus	?	?	?	?	?	?	?

1.6 The Challenges of Parallel Computing

The irony of history is that many research on shared-memory parallel programming and architectures was already performed in the 80's but discontinued due to rapid advancement of efficient sequential architectures. Market-driven development of sequential chips mainly by Intel and AMD, pushed parallel architectures in the background and also off many research agendas. Therefore, interesting background materials can be already decades old (translating to centuries in the field of computer science), but are becoming more relevant again, e.g. [MCS91].

Recalling this history also gives great insight into the status quo of current parallel shared-memory architectures. While in the past these systems often had large bus interconnects, with often pieces of local memory available to each processor, we encounter nowadays almost exclusively *cache-coherent* architectures that provide a single consistent view of memory for each core, pushing the communication in the background by means of a cache coherence protocol. These concepts are discussed in the following sections.

The current situation can be explained by the availability and vast usage of fast sequential chips and the prevalence of sequential algorithms for these chips. In the old situation, the burden of parallel programming was entirely on the programmer, but its functioning was transparent in the form of explicit communication operations. This setting was much closer to the message-passing approaches for distributed programming. Nowadays, the processor manufacturer has taken on part of the burden of parallel programming by providing this single view of memory. Programming these systems is less involved but also much less transparent.

1.6.1 Parallelism is Inherently Complex

Parallelism is known for its inherent complexity.

The theory. Complexity theory, in the first place, suggests that "feasible" problems for sequential machines lie within the polynomial-time complexity bound (P), as a higher-than polynomial upper bound on computation time seems unreasonable given that computation power grows exponentially [Cob64; DC80]. A similar argument can be made for the complexity of a parallel algorithm using a polynomial amount of hardware, i.e. *circuit size*. This suggests that the efficiently parallelizable problems are a subset of P, since polynomial hardware running for polynomial amount of time can be simulated by a polynomial-time algorithm [DC80, Sec. 5].

Nick Pippinger [Pip81] came up with a characterization of the circuit size which turned out to be invariant for different circuit layouts. Since the width of the circuit is closely related to the amount of parallelism, it is believed that Nick's class (NC), a term coined by Cook [Coo79], describes those problems that are efficiently parallelizable. It is widely believed that $\mathsf{P} \neq \mathsf{NC}$. In other words, some problems exist that are *inherently sequential* (these should necessarily include all the P-complete problems).

In practice, the fact that a problem is in NC means that it may use a polynomial number of processors solving the problem in poly-logarithmic time. Several points of critique can be addressed with regard to such a theoretical model [Var11]. First of all, it is infeasible to scale the number of processors as the model suggests. Second, the definition of a parallel computer in this model is often assumed to be a Concurrent Read-/Concurrent Write (CRCW) parallel random-access memory (PRAM) machine, which can perform communication in a single cycle and access all memory equally cheaply. The practice is far from this model as we will see next. Worse yet, the practice is actually evolving such that communication becomes more expensive and memory latencies less constant. *Therefore, the current thesis does not emphasize the theoretical aspect of parallel computation.* Although sometimes, we draw from the theory to understand were the difficult problems may lie, e.g. in Part III, we discuss the inherent sequential nature of depth-first search (DFS).

Memory hierarchy, latency and bandwidth. A consequence of Moore's law (see Section 1.2) is that processor clock frequencies increase faster than memory latency. This has over time led to very steep memory hierarchies in computers, prompting processor manufacturers to include several layers of fast on-chip caches, referred to as L1, L2 and L3, to make up for a slower main memory (see Figure 1.3). The L2 and L3 caches are often shared among multiple cores in multi-core chips.

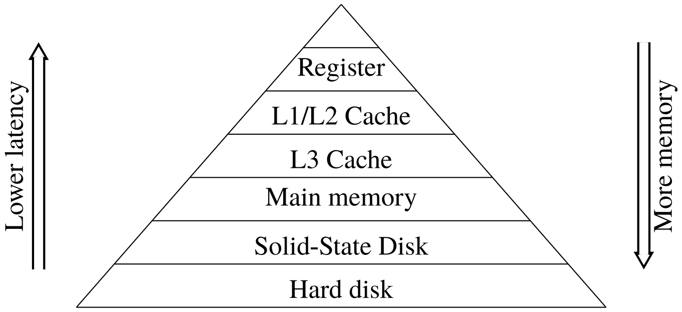


Figure 1.3: Memory hierarchies balance low latency and large memory.

The roofline model [WWP09; Asa+09] tries to predict an algorithm's parallel efficiency by modeling only its memory bandwidth usage and number of operations per second. The ratio of bandwidth and computation, or *algorithmic intensity*, is the main measure in this theory. When the algorithm itself performs fewer computations per byte of memory it loads than the processor can support, the algorithm will be limited by the algorithmic intensity that the processor can deliver. This performance model offers a way of thinking about the efficiency of high-throughput and high-performance algorithms. For example, it suggests that the memory footprint of a data-intensive algorithm is probably the performance bottleneck on modern machines. In Part II, we use this heuristic for the design of scalable concurrent data structures.

NUMA architectures. Whereas multi-core processors communicate via their shared caches as discussed below, multi-processor systems communicate via the memory bus as Figure 1.4 illustrates. To reduce memory latency and reduce traffic on the shared memory buses, multi-processor systems often opt for processor local memory banks (see Figure 1.4). These so-called non-uniform memory architectures (NUMA) result in less uniformity in the memory access times. Since programmatically all memory is presented as one uniform global range, the slowdown of remotely allocated memory is not transparent to the programmer. Therefore, modern operating systems offer NUMA libraries that allow to control the allocation across the different memory banks.

Throughout the current thesis, little mention is made of these implementation details, in order to make room for other results. We also did not find the need to optimize shared data structures using the NUMA library, as the speedups obtained in our basic algorithms (reachability) were already close to optimal.

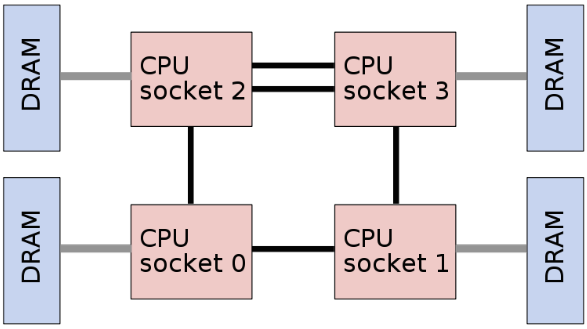


Figure 1.4: Non-Uniform Memory Architectures

Cache coherence. On the practical level of the implementation, cache coherence is another source of problems. Modern processors have taken the design approach to make parallel programming resemble the sequential world as much as possible. To this extent, the available system memory is presented uniformly as one addressable range

Algorithm 1.1 Counting to a billion (sequentially)

```
#define B (1024*1024*1024)

int main (void) {
    int result = 0;
    for (int i = 0; i < B; i++)
        result++;
    return result;
}
```

Algorithm 1.2 Counting to a billion (in parallel)

```
#define P 16

static void count (void *arg) {
    int *counter = (int *) arg;
    for (int i = 0; i < B / P; i++) ( *counter )++;
}

int main (void) {
    pthread_t thread[P];
    int counters[P] = {0};

    for (int i = 0; i < P; i++)
        pthread_create (&thread[i], NULL, count, &counters[i]);

    int result = 0;
    for (int i = 0; i < P; i++) {
        pthread_join (thread[i], NULL);
        result += counters[i];
    }
    return result;
}
```

of bytes, whereas in fact memory is often moved to the core-local or CPU-local L1, L2 or L3 caches. The user cannot control directly what the contents of the cache are. We illustrate the difficulties that this design causes with an example.

Algorithm 1.1 shows a simple program that solves the problem of counting to a billion. The same problem is solved in parallel using PTHREADS in Algorithm 1.2. The main thread launches 16 worker threads at Line 13 and wait for their completion at Line 17. Meanwhile, the worker threads each use their own counter (declared at Line 10) to do a part (1B/16) of the counting at Line 5.

When executing Algorithm 1.1 and Algorithm 1.2 on the same machine with 16 cores, we obtain a runtime of 27 seconds for the first, and 32 seconds for the second implementation. *Our parallelization caused a slowdown!*

This slowdown is caused by the cache coherence protocol, which ensures that all local processor caches reflect the same global state of memory. However, the smallest unit on which is operates is a *cache line*, which consist of 64 bytes or more. It turns out that our array counters lies on the same cache line, which is now sent around to all processor cores for each count operation. This problem is known as *false sharing*.

Weak memory models. In order to improve processor speeds, and adhere to Moore's corollary that clock frequencies double every few years, manufacturers in the past had to employ several tricks. At the basis of their methods is a process called *pipelining*, where individual instructions are pushed on a processing queue inside the processor and are completed in a step-wise fashion. With the increasing clock frequencies, the amount of computation became limited by the depth of the circuit (the distance that the electronic signals need to travel). Therefore, the operation performed at each step of the pipeline is small and many pipeline stages are needed to assemble the final result, sometimes up to twenty-four stages.

This means that the first instruction takes up to twenty-four cycles for completion, but the subsequent instructions 'flow' at the clock frequency. *Unless* an instruction is halted due to a slow reference to memory (a *cache miss*), or a branching instruction whose result cannot be predicted. To remedy this, many transistors on the processor are dedicated to predict branches and govern the cache contents.

However, in the course of time these solutions still turned out to be insufficient to sustain the processor core with useful work. Therefore, processors resort to *out-of-order executions* for decades already. This technique allows the slower instructions to be reordered behind the faster instructions, if they are independent (operate on different data). For example, a memory load may be reordered behind a memory store or another load. The weaker the *memory model*, the more such reorderings are allowed.

Out-of-order executions are invisible to sequential programs, where *sequential con-*

sistency is maintained, i.e. all instructions appear to be executed in the order in which they are issued, informally speaking. However, for parallel programs, these out-of-order executions of remote cores become visible. Thus the methods that were used for a long time to speedup sequential computation, actually hinder parallel computation. The result is an often counter-intuitive programming model, which makes memory contents appear inconsistently across different CPUs, even though cache coherency presents memory as a single continuous range.

Embarrassing parallelism. The inherent difficulties of parallel computing has led many to consider *embarrassingly parallel* algorithms [Var11; HJG08; HJG11; Xu+11]. Embarrassingly parallel solutions use little or no synchronization and thereby are almost trivially correct (often because several instances of the same sequential algorithm are run independently in random fashion).

Vardi [Var11] discusses the difficulty of LTL satisfiability checking. All the different techniques for this problem have their own merits in the sense that they effectively solve their own subset of problems. For this reason, *portfolio-based* approaches, where many different SAT solvers are unleashed on the same problem in parallel, but completely independently, have shown promising results [Xu+11]. He concludes with the suggestion that embarrassing parallelism probably offers the best parallelization method, given the few impressive results from five decades of research on parallelism have delivered.

Embarrassing parallelism in model checking can aid help to locate bugs rapidly, but cannot speedup complete verification as all worker threads would traverse the whole state space independently [HJG08; HJG11]. Recently, techniques have been proposed that aim to remedy this shortcoming by using informed search with limited communication to prevent many redundant computations [Wij11].

1.7 Contributions

The current thesis contains contributions in 3 areas in the field of model checking. These correspond to the different parts of the thesis. After discussing these different contributions, we summarize their impact on the scientific community up until the time of writing of the current thesis (January, 2014).

1.7.1 Scalable Reachability with State Compression

The first main contribution, of *scalable multi-core reachability* described in Chapter 2, lies at the basis of all other work contained in the current thesis. Contrary to the belief of some experts in the field, we showed how reachability can scale almost ideally on modern multi-core hardware. We achieved this by exploiting the hardware's strengths through the use of a carefully designed shared concurrent hash table with low memory footprint (as opposed to using distributed algorithms). This design provides both more flexible load balancing and more flexibility in choosing the search order of the reachability algorithm, thereby aiding on-the-fly model checking.

The second main contribution is the combination of parallel reachability with *efficient and scalable state compression*. The core of this work is the replacement of the concurrent hash table with the concurrent tree data structure that is developed in Chapter 3. Because this structure internally uses the concurrent hash table of Chapter 2, its scalability is equally good. Furthermore, by incrementally updating the tree, we obtain similar, often better, runtimes than with plain (non-compressed) hash table storage.

The obtained compression ratio depends greatly on the structure of the state space, but because it is often highly combinatorial in model checking, it is close to the optimal of 8 bytes per state regardless of the original state size, which could be thousands of bytes. (The fixed quantity depends on hardware characteristics that guide the implementation of the tree.) Extensive experiments indeed demonstrate that in many cases the optimal compression is obtained.

We further reduce the compressed sizes to almost 4 bytes by developing a *concurrent compact hash table* [Cle84] in Chapter 4. The compact table can be used in many other applications, e.g. in BDDs, and therefore represents a contribution in its own right. Moreover, its *dynamic region-based locking* strategy is a novel approach which delivers fine-grained, yet multi-object, mutual exclusion. In Chapter 11, the combination of the tree database with compact hash table is discussed and experiments are presented that confirm the expected compression.

1.7.2 Scalable, LTL Model Checking in Linear Time

Our work on multi-core nested depth-first search algorithms (Mc-ndfs) yielded the first parallel LTL model checking algorithm which can be linear in the size of the graph, but it also introduced a new opportunistic way to parallelize similar algorithms – many important graph algorithms are based on dfs. The impact of this research (see Section 1.7.4) indeed shows that other researchers are beginning to apply similar methods.

In Chapter 5, we introduce the first version of Mc-ndfs: A parallel, dfs-based algorithm that takes as venture point the embarrassingly parallel approach discussed in the previous section, but adds limited communication to improve scalability. Though it only scales for a small set of inputs, it showed the potential of the approach: good on-the-fly performance and little overhead. The contribution of Chapter 6 is a detailed evaluation of the Mc-ndfs algorithm and a comparison against other algorithms: owcty by Barnat et al. [ČP03] and ENdfs by Evangelista et al. [EPY11]. Because the approach of the

ENdfs algorithm is similar but orthogonal [Laa+11; EPY11], the paper also presents a trivial combination of the two, which shows that the algorithms indeed complement each other in practice and together consistently perform better than the owcty algorithm. Finally, Chapter 7 proposes an integrated combination of the algorithms from [EPY11] and [Laa+11]. This algorithm uses less memory than its prequels (also less than owcty), is less complicated, and performs at least equally well.

The combination of partial-order reduction and the parallel model checking of liveness properties forms a difficult problem because an extra condition (to wit: the *ignoring proviso* [EP10]) needs to be implemented that reasons over cycles in the state-space graph. Solutions have been proposed, but all of them severely impede the POR performance by overestimating the proviso: [BBR10a; Hol08; HB07; NG02; Bri+05; BBC05a; Kur+98; BLLL09; LS99]. We solve this completely for an important subset of LTL, namely livelocks, by showing that in this particular case, the proviso can be weakened. In Chapter 8, we propose a parallel version of the dfs_{fifo} algorithm [FS09] for checking livelocks, and provide a proof of correctness. Experiments show excellent scalability and POR for this algorithm on a 48-core machine.

Last, we provide additional experiments in Chapter 11. The implementation of a promela frontend for LTSmin allows the use of all the proposed techniques from Part II and Part III on models created for the popular spin model checker. The results solidify our empirical evaluation of said algorithms by extending the benchmark set with many freely available promela models. Indeed, the new experiments confirm again that our multi-core model checking algorithms are scalable up to 48-cores (previously 16), on-the-fly, and use very little memory even compared to spin's Collapse compression [Hol97b].

1.7.3 Scalable Model Checking of Timed Systems

Chapter 9 presents the first scalable multi-core reachability algorithms for timed automata. Our algorithm supports various abstraction and extrapolation methods in order to obtain a finite state space. This includes the coarsest abstraction, called *inclusion abstraction* or *subsumption*. Experiments show speedups of up to 60 on a 48-core machine, compared to the popular uppaal model checker. The implementation in LTSmin is also compatible with state compression, thus complementing the state-caching technique that is available in the uppaal model checker – both methods show comparable reductions of memory usage, however as of yet, neither implements both techniques simultaneously. We further investigate the influence of search orders on the size of the abstracted state spaces, confirming the observations in [BHV00], but also demonstrating that parallel search orders can reduce the size of the state space.

Chapter 10 presents the first algorithm for LTL model checking of timed automata.

Because the subsumption abstraction introduces a simulation relation on states, cycles representing infinite traces can become *spirals*, as the chapter demonstrates. Therefore, the combination of subsumption with LTL model checking was hitherto an open problem [TYB05]. Moreover, we also give a parallel algorithm based on CNDFS, but extended with subsumption. Experimental results show promising reductions from the abstraction, and reasonable scalability on a 48-core machine. The implementation in LTSMIN is the first available timed model checker that supports full LTL and the latest abstraction and extrapolation techniques.

1.7.4 Impact of the Contributions

Apart from being the basis of the further research presented in the current thesis, the *shared hash table approach* has inspired various other researches into the parallelization of algorithms from diverse fields: Since recently, the model checker SPIN also implements our hash table implementation [Hol12]. Lowe used the hash table implementation to improve the scalability of his concurrent depth-first search algorithms [Low14]. Sulewski [SEK11; Sul12] uses the hash table design for host- based (on the CPU) duplicate detection to solve planning and other problems on GPGPUs. Other GPU-based state-space exploration techniques [WB14] also employ our hash table, using a *warp-the-line* probing sequence instead of the original *walk-the-line* technique (see Chapter 2). Others are still exploring its effectiveness on GPUs [Nee14]. Multi-Core BDDs were realized using an extension on our hash table design [DLP13]. And finally, DIVINE at least planned to implement the same shared hash table approach as indicated in [Bar+10], [BB11, Sec. 2.2.2], and [Bar10, Sec. 2.1.3].

Moreover, *tree compression* was adopted in the DIVINE model checker as of version 3.1 alpha [Hav13]. The compression is not yet turned on by default, likely because its lack of incremental updates does incur a runtime penalty for explicit-state inputs such as DVE models [Sti13]. Since the runtimes with (incremental) tree compression are comparable to those with a plain hash table, the technique has become the default in the LTSMIN model checker.

The intricacies of our *multi-core nested depth-first search algorithms* and their implementation inspired Wan Fokkink, Pieter Hijma and Stefan Vijzelaar to create a student assignment about them [FHV13]. The students are asked to implement the algorithm and encouraged to find improvements to the algorithm and the implementation. Because details of the correctness proof are more intricate than they might seem, this reportedly often leads to incorrect derivative algorithms (and hopefully to equally many learning moments). Their efforts led to the discovery of a bug in our parallel algorithm with extensions. (We did not come up with a correctness proof for this extended algorithm, only for the basic variant.) This bug has been corrected as described in Chapter 5.

Our work on multi-core LTL model checking also inspired others to employ similar techniques, i.e. starting multiple depth-first searches simultaneously with late information sharing. For the PAT model checker, several of such algorithms are in development [Dat13; XLHS13; LSD09b]. Next to similar nested depth-first search algorithms, they also use Tarjan's strongly connected component (SCC) algorithm [Tar72], for finding (fair) accepting cycles [GV04] in parallel. Gavin Lowe [Low14] developed several related concurrent depth-first search based algorithms for identifying SCCs, accepting cycles, and normal cycles in a graph. These algorithms avoid unnecessary work completely at the cost of more synchronization: Searches may block on other searches, but instead of waiting new searches are initiated. As a consequence, more searches can be launched than the available processor cores. Therefore, these are scheduled in a way that is similar to that in fine-grained task-based parallelism [Blu+95; Ayg+09]. Finally, our use of depth-first like search orders was also taken over by the GPU variant of the DiVinE model checker to improve its on-the-fly behavior [Bar+11a, Sec. 5.1].

Our most recent work on *timed systems* has had little time to catch on in the community as of yet. Nonetheless, it has been considered as a means to study the performance of fault-tolerant systems [Fac13] because of its capability to handle larger models than uppaal using multiple cores and tree compression [Dal+12]. At the current author's native Formal Methods and Tools group at the University of Twente, several research plans therefore also include the use of our timed algorithms to study both fault-tolerant systems and biological systems [Sch+12].

1.8 Overview and Reading Guide

The current thesis contains 3 main parts. Because the chapters therein consist of published conference papers that are largely left intact, the parts, as well as the chapters, can be read independently. Additional introductions to the parts and chapters were added to facilitate such random-access patterns by explaining their context within the thesis.

The first main part, *Overview and Reading Guide*, deals mainly with the problem of scaling reachability and combing it with efficient state compression and partial-order reduction. For this reason, it contents focus on concurrent data structures and the properties of modern multi-core machines concerning their scalability. Algorithms are only of secondary importance here.

The second main part, *Discussion and Conclusions*, assumes the scalable data structures as a given and concerns itself mainly with algorithmic solutions. It presumes an understanding of multi-core reachability as presented in Part II, especially Chapter 2.

The third main part, *Conclusions*, focuses on extending both reachability and LTL model checking to the timed domain. For a detailed understanding of the proposed data structures and algorithms, a reading of the preceding parts (or at least Chapter 2 and Chapter 7) is in order.

There is a natural flow between the chapters in each part, as all of them are the result of a single line of research that has been set out in Section 1.5. Preceding chapters often provide more general background information than is presented in the following chapters. This is illustrated in Figure 1.5, where the arrows represent the suggested order for reading the thesis and the dashed arrows represent (weak) dependencies between chapters. We now discuss briefly the independent chapters explaining their relationship and suggesting a reading order.

Chapter 2 is based on the paper *"Boosting Multi-Core Reachability Performance with Shared Hash Tables"*, which was published at FMCAD 2010 [LPW10a]. It describes the main approach we use for scalable multi-core reachability and the underlying lockless data structure. All state-space searches in subsequent chapters are based on the same method, and all data structures use similar lockless approaches. The chapter can be read in isolation.

Chapter 3 is based on the paper *"Parallel Recursive State Compression for Free"*, which was published at SPIN 2011 [LPW11c]. It describes a lockless tree data structure and also discusses its connection to the reachability algorithm. It investigates worst-case and best-case compression ratios analytically and also presents empirical evidence that the average compression is very close to the best-case compression.

Chapter 4 is based on the paper *"A Parallel Compact Hash Table"*, which was published at MEMICS 2011 [VL12]. It presents a concurrent compact hash table, which can store small fixed-size keys in succinct manner. It also presents a correctness proof for the operations on the structure, but does not discuss its use in the context of model checking in detail. The combination of compact hash table and tree compression is discussed in Section 4.4.

Chapter 5 is based on the paper *"Multi-core Nested Depth-First Search"*, which was published at ATVA 2011 [Laa+11]. It presents our first successful attempt at parallel LTL model checking through the use of the NDFS algorithm. This DFS-based algorithm does not lend itself directly for parallelization. We therefore come up with a novel optimistic approach that allows threads to continue searching semi-independently and randomly through the state space. Because it presents a rigorous proof of our parallel algorithm, it could be a useful starting point for readers that are interested in parallelizing other algorithms based on DFS.

Chapter 6 is based on the paper *"Variations on Multi-Core Nested Depth-First Search"*, which was published on invitation at PDMC 2011 [LP11]. It combines the parallel NDFS algorithm, with another algorithm which appeared in the literature. Multiple experiments confirm the scalability of the combined algorithm, but also explore the excellent on-the-fly behavior.

Chapter 7 is based on the paper *"Improved Multi-Core Nested Depth-First Search"*, which was published at ATVA 2012 [Eva+12]. It presents an integrated combination of the algorithms presented in the 2 preceding chapters. CNDFS uses less memory, and is a simpler algorithm, leading to a simpler proof of correctness.

Chapter 8 is based on the paper *"Improved on-the-Fly Livelock Detection"*, which was published at NFM 2013 [LF13]. It proposes to solve the combination of parallel LTL model checking with partial-order reduction, by focusing on an important subset of liveness properties: livelocks. A new parallel algorithm, called $\text{PDFS}_{\text{FIFO}}$, is presented based on the same techniques as presented in the 3 preceding chapters.

Chapter 9 is based on the paper *"Multi-core Reachability for Timed Automata"*, which was published at FORMATS 2012 [Dal+12]. It extends our multi-core reachability to the domain of timed automata. It could be a starting point for those interested in the implementation of timed automata. The discussion of parallel, timed reachability algorithms depends slightly on Chapter 2.

Chapter 10 is based on the paper *"Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction"*, which was published at CAV 2013 [Laa+13b]. It ports CNDFS to the timed setting. It also extends the CNDFS algorithm to use the coarse *subsumption abstraction*, leading to a reduced state-space search. The chapter presents the first realization of parallel LTL model checking for timed automate, but also solves the previously open problem of using coarse abstractions for LTL model checking. The details of the CNDFS algorithm under abstraction are probably only accessible to those who have read Chapter 7.

Part V concludes the current thesis with additional experiments and a reflection on our work.

Chapter 11 details on the experiments done with an implementation of the PROMELA language for the LTSMIN model checker, as described in the paper *"SpinS: Extending LTSmin with Promela through SpinJa"*, which was published at PDMC 2012 [BL13a]. The use of PROMELA provides us with an extensive set of real-world model checker problems, which are used to compare scalability more directly against the state-of-the-art SPIN model checker on a 48-core machine. Moreover, the chapter demonstrates the combination of our reachability algorithm with state compression and partial-order reduction, as presented in the paper *"Guard-Based Partial-Order Reduction"*, which was published at SPIN 2013 [Laa+13a].

Finally, in Chapter 12, we compare the results to related work, evaluate the extent to which our goals have been met, and pose some open questions.

Several appendices add detailed proofs for algorithms in Chapter 5 and Chapter 10.

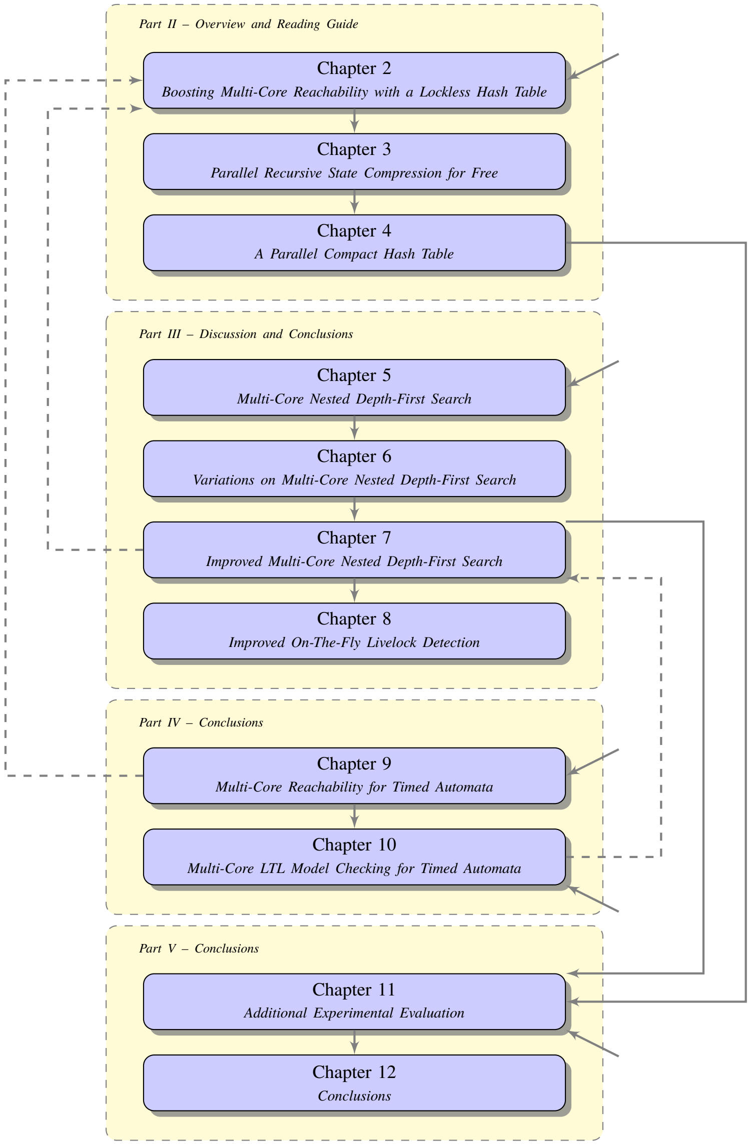

Part II – Overview and Reading Guide
Chapter 2
Boosting Multi-Core Reachability with a Lockless Hash Table
Chapter 3
Parallel Recursive State Compression for Free
Chapter 4
A Parallel Compact Hash Table
Part III – Discussion and Conclusions
Chapter 5
Multi-Core Nested Depth-First Search
Chapter 6
Variations on Multi-Core Nested Depth-First Search
Chapter 7
Improved Multi-Core Nested Depth-First Search
Chapter 8
Improved On-The-Fly Livelock Detection
Part IV – Conclusions
Chapter 9
Multi-Core Reachability for Timed Automata
Chapter 10
Multi-Core LTL Model Checking for Timed Automata
Part V – Conclusions
Chapter 11
Additional Experimental Evaluation
Chapter 12
Conclusions

Figure 1.5: Reading guide for the current thesis

Part II

Data Structures for Multi-Core Reachability

Introduction

At the time that the current work on parallel model checking was initiated, it was widely believed in the community that scalable parallelization of model checking operations was limited to input with large data sizes (state vectors), long transition delays (next-state computation) and high branching factors [HB07; Hol08]. The high-throughput nature of the procedure – many model checkers can generate millions of states per second which can each consists of multiple kilobytes – was considered to be detrimental to scalability. Still others believed that the right algorithms/implementation had yet to be invented [BR08]. In the context of the project Multi-Core Model Checking, we therefore set out to parallelize reachability; the backbone of many more advanced model checking techniques including LTL model checking [VW86; Laa+11; LP11].ex

Previous results [Bar+10; ČP03; Hol08], were based on distributed algorithms which use a hash function to statically assign states to the different worker threads. This so-called *static partitioning* results in high communication overhead, as most states require remote processing. Others used a shared hash table with a locking mechanism [BR08; Hol08]. These approaches led to meager parallel scalability.

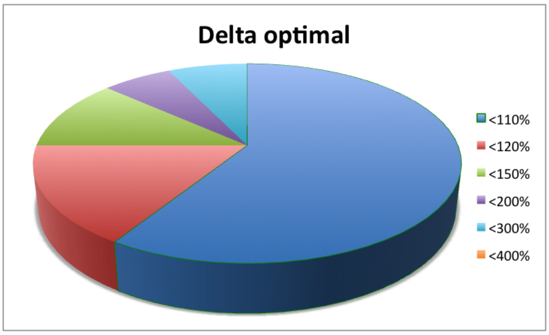
We believed that a shared hash table approach could exploit the strengths of shared-memory multi-processors to a much higher degree, if carefully designed to reduce contention and bandwidth use (the memory footprint). Chapter 2 presents a lockless hash table design which fulfills these requirements. Thus by focusing on the necessary, instead of the possible – the literature is full of *wait-free* hash tables which are useful to guarantee high responsiveness in real-time environments, but likely exhibit limited throughput due to the use of pointers [SS06] and even shared counters [GGH05; GGH04] – we obtain near-ideal scalability for model checking as demonstrated by a large set of experiments on a 16-core machine. (Chapter 2 focuses on the comparison with other model checkers, but our hash table design itself is also evaluated by comparing it against other concurrent hash tables in Chapter 4).

The added benefit of the shared hash table approach, is that the *open set* of still-to-be-explored states remains local, i.e. is not used for communicating states as with static partitioning. A load balancer instead takes care of distributing load, when a worker's open set becomes empty. This reduces communication and allows for more flexibility in the search algorithm, which can now also use a stack as open set implementation to obtain both (pseudo) breadth-first and depth-first search orders.

While strict depth-first search (DFS) is off-limits, as workers influence each other's search order, this is also not needed for the safety properties considered in the current part. And approximate DFS orders are enough to obtain good on-the-fly behavior in this setting. (In theory DFS is likely not parallelizable, as discussed in detail in Section 5.1; nonetheless, we show in Part III, that the depth-first property can still be used to realize parallel LTL model checking in linear-time using a multi-core *nested DFS* algorithm.) Strict BFS order can also be obtained with little extra synchronization as shown in Section 9.6.2, where it is used to reduce state spaces under *subsumption abstraction*.

Great scalability is good to have, but without state compression the model checking procedure will be severely limited by the available main memory. In order to pursue our second research question (Section 1.5), we investigated the parallelization of tree compression, a method that can reduce large state descriptors down to two integers (Section 3.4). Chapter 3 presents a new concurrent algorithm for this tree data structure.

We show with a theoretical model that the optimal compressed state size is 2 integers, or 8 byte. Experiments show that, for more than half of almost 300 benchmarks, the states are indeed reduced to within 110% of this optimal (see the figure to the right, which summarizes Figure 3.13). This includes examples with large state descriptors of around 250 integers.



Again to support a high throughput, our tree reuses the lockless hash table design from Chapter 2 by merging multiple tree tables together. To further reduce the memory footprint, an *incremental tree update algorithm* is proposed. The surprising result is that the runtimes and scalability are sometimes better than those obtained with the hash table approach (see Figure 3.15), hence the title "Parallel Recursive State Compression for Free".

With the realization that the memory consumption of our tree table can be halved using a *compact hash table*, we pursued a parallel version of the latter in Chapter 4. In such a compact table, a technique named *quotienting* is used, which instead of storing full keys, stores only a quotient of the key in the table. Its remainder is used to find a hash location in the table, and the full key can thus be restored from its quotient and the location where it is found using some additional administrative bits to resolve hash collisions [Cle84; GV03]. To parallelize compact hash tables, we propose a new *dynamic region-based locking* scheme and show that the same scheme can also be used for other hash table implementations, such as Knuth's *bidirectional linear* probing [AK74]. Experiments confirm the scalability of this design.

In Section 4.4, we discuss how the compact hash table is to be combined with tree compression. By storing the tree roots inside the compact table, we indeed obtain compressed state sizes that almost reach one integer. With an additional trick of growing this table larger than the leaves table, we can also accommodate more than 4 billion states. In other words, our compact tree compression is not limited by the 2^{32} elements addressable by an integer. Using an information-theoretic model, we show that the obtained compression is indeed close to the lower bound achievable for encoding a stream of states generated by a typical explicit model checker.

Moore's law also held steadily during the execution of this PhD project, and thus at a later stage we acquired access to a 48-core machine. The methods discussed in this part of the thesis have proved to scale with this 3-fold increase in parallelism with little modification: We merely had to adjust a few parameters in our load balancer implementation to obtain near-ideal speedups on the new platform (while preserving the performance on the older platforms). Chapter 11 provides experiments using 48 cores confirming this, but the full benchmark set with hundreds of examples can be inspected online [Laa]. Consequently, our approach has inspired various other researchers to use similar approaches for their parallel algorithms (see Section 1.7.4).

The table below summarizes the goals that the current part meets (c.f. Table 1.1 in Section 1.5.3): It solves multi-core reachability with good on-the-fly behavior by allowing different search orders (depth-first orders often locate 'deeper' bugs faster, especially with multiple parallel worker threads). Excellent compression is supported by means of the parallel tree structure and its compact version. Partial-order reduction for *reachability properties*, i.e. deadlocks, can be computed locally [Laa+13a] and hence

	Explicit state	**+ On-the-fly**	**+ Compression**	**+ POR**
Reachability	✓	✓	✓	✓

can be combined with the parallel methods as demonstrated in Chapter 11. For other *safety properties*, such as *error actions* and *invariants*, partial-order reduction requires global conditions over the state-space, variants of which [BLLL09] can be supported by our parallel algorithms. In Part III, we discuss alternate solutions for the so-called ignoring proviso.

2 Boosting Multi-Core Reachability with a Lockless Hash Table

Alfons Laarman, Jaco van de Pol, Michael Weber

Abstract

The current chapter focuses on data structures for multi-core reachability, which is a key component in model checking algorithms and other verification methods. A cornerstone of an efficient solution is the storage of visited states. In related work, static partitioning of the state space was combined with thread-local storage. This solution leaves room for improvements. The current chapter presents a solution with a shared state storage. It is based on a lockless hash table implementation and scales better. The solution is specifically designed for the cache architecture of modern CPUs. Because model checking algorithms impose loose requirements on the hash table operations, their design can be streamlined substantially compared to related work on lockless hash tables. The resulting speedups are analyzed and compared with related tools. Our implementation outperforms two state-of-the-art multi-core model checkers, SPIN (presented at FMCAD 2006) and DiVinE, by a large margin, while placing fewer constraints on the load balancing and search algorithms.

About this chapter: The current chapter is based on the paper *"Boosting Multi-Core Reachability Performance with Shared Hash Tables"*, which was published at FMCAD 2010 [LPW10a]. An extended report on the work was published at Arxiv [LPW10b] and is integrated in the current chapter.

Compared to the original publication in [LPW10a; LPW10b], a few enhancements have been made to the text presented here. First, we improved the memoized hashes by using a separate hash function (hash_x) for indexing and generating the memoized hash itself in Algorithm 2.3. This independence results in fewer collisions during the filtering on memoized hashes. We found that a fuller table can be rather sensitive to the independence of these hash functions. We also simplified the algorithm's representation by splitting the calculation of cache line indices in a separate algorithm

(Algorithm 2.4). Last, the experimental section was extended with the additional data from the report version of the paper, and we included an additional benchmarks with a static load balancer.

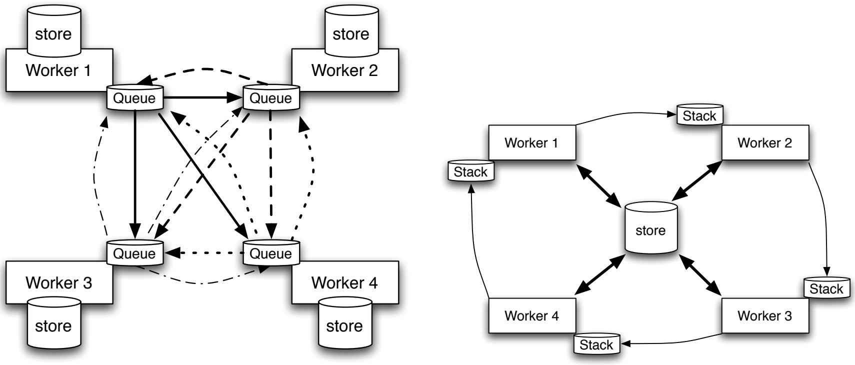
2.1 Introduction

Many verification problems are highly computational intensive tasks that can benefit from extra speedups. Considering the recent hardware trends, these speedups can only be delivered by exploiting the parallelism of the new multi-core CPUs.

Reachability, or full exploration of the state space, is a subtask of many verification problems [CPR06; Bri06]. In model checking, reachability has in the past been parallelized using distributed systems [Bri06]. With shared-memory systems, these algorithms can benefit from the low communication costs as has been demonstrated already [BBR07]. In the current chapter, we show how the performance of state-of-the-art multi-core model checkers, like spin [HB07] and DiVinE [BBR07], can be greatly improved using a carefully designed concurrent hash table as shared state storage.

Motivation. Holzmann and Bošnački used a shared hash table with fine-grained locking in combination with the stack-slicing algorithm in their multi-core extension of the spin model checker [HB07; Hol08]. This shared storage enabled the parallelization of many of the model checking algorithms in spin: safety properties, partial-order reduction and reachability. Barnat et al. implemented the same method in the DiVinE model checker [BBR07]. They chose to implement the classic method of static state-space partitioning, as used in distributed model checking [BR08]. They found the static partitioning method to scale better on the basis of experiments. The authors also mention that they were not able to develop a potentially better solution for shared state storage, namely the use of a lockless hash table. Thus it remains unknown whether reachability, based on shared state storage, can scale.

Using a shared state storage has further benefits. Figure 2.1 shows the different architectures discussed thus far. Their differences are summarized in Table 2.1 and have been extensively discussed by Barnat et al. [BR08]. They also investigate a more general architecture with a shared storage and arbitrary load-balancing strategy (not necessarily stack-slicing). Such a solution is both simpler and more flexible, in the sense that it allows for more freedom in the choice of the exploration algorithm, including (pseudo) DFS, which enables fast searches for deadlocks and error states [RK88]. Holzmann already demonstrates this [Hol08], but could not show desirable scalability of spin (as we will demonstrate). The *stack-slicing algorithm* [Hol08], is a specific case of load balancing that requires DFS. In fact, any well-investigated load-balancing solution [San97a]



(a) Static partitioning

(b) Stack slicing

Figure 2.1: Different architectures for model checkers

Table 2.1: Differences between architectures

Arch.	**Sync. points**	**Pros / Cons**
Figure 2.1(a)	Queue	local (*cache efficient*) storage / *static* load balancing, *high* comm. costs, *limited* to BFS
Figure 2.1(b)	Shared store, stack	low comm. costs / *specific* load balancing, limited to (pseudo) DFS
Shared store	Shared store, (queue)	low comm. costs, *flexible* load balancing, *flexible* exploration algorithm / scalability?

can be used and tuned to the specific environment, for example, to support heterogeneous systems or BFS exploration. Inggs and Barringer use a *lossy* shared hash table [IB02], resulting in reasonable speedups at the cost of precision (states can potentially be revisited), but give little details on the implementation.

Contribution. We present a data structure for efficient concurrent storage of states. This enables scaling parallel implementations of reachability for many desirable exploration algorithms. The precise needs which parallel model checking algorithms impose on shared state storage are evaluated and a fitting solution is proposed given the identified requirements. Experiments show that our implementation of the shared storage scales significantly better than an implementation using static partitioning, but also beats state-of-the-art model checkers. By analysis, we show that our design will scale beyond current state-of-the-art multi-core processors. The experiments also contribute to a better understanding of the performance of the latest versions of SPIN and DiVinE.

Overview. Section 2.2 presents background on reachability, load balancing, hashing, parallel algorithms and multi-core systems. Section 2.3 presents the lockless hash table, which we designed for shared state storage. But only after we evaluated the requirements that fast parallel algorithms impose on such a shared storage. In Section 2.4, the performance is evaluated against that of DiVinE 2 [BBR09b] and SPIN. A fair comparison can be made between the three model checkers on the basis of a set of models from the BEEM database which report the same number of states for both SPIN and DiVinE. We end the current chapter by putting the results we obtained into context, and an outlook on future work (Section 2.5).

2.2 Preliminaries

Reachability in model checking. In model checking, a computational model of the system under verification (hardware or software) is constructed, which is then used to compute all possible states of the system. The exploration of all states can be done symbolically, e.g., using *binary decision diagrams* (BDDs) to represent sets of states, or by enumerating and explicitly storing all states. While symbolic methods are attractive for a certain set of models, they are not a silver bullet: due to *BDD explosion*, sometimes plain enumerative methods are faster. In the current chapter, we focus on enumerative, or explicit-state, model checking.

Enumerative reachability analysis can be used to check for deadlocks and invariants and also to store the whole state space and verify multiple properties of the system at once. Reachability is an exhaustive search through the state space. The algorithm calls for each state the *next-state* function to obtain its successors until no new states are found (Algorithm 2.1). We use an *open set* Q, which can be implemented as a stack or queue, depending on the preferred exploration order: depth- or breadth-first. The initial state s_0 is obtained from the model and added to Q. In the loop starting on Line 4, a state is taken from Q, its successors are computed using the model (Line 7) and each new successor state is put into Q again for later exploration. To determine which state is new, a *closed set* V is used. V can be implemented with a hash table.

Algorithm 2.1 Reachability analysis

```
1:  procedure reachability($s_0$)
2:      $Q := \{s_0\}$
3:      $V := \{s_0\}$
4:      while $Q \neq \emptyset$ do
5:          $s := s \in Q$
6:          $Q := Q \setminus s$
7:          for all $t \in$ next_state($s$) do
8:              if $t \notin V$ then
9:                  $V := V \cup \{t\}$
10:                 $Q := Q \cup \{t\}$
```

Possible ways to parallelize Algorithm 2.1 have been discussed in the introduction. A common denominator of all these approaches is that the strict BFS or DFS order of the sequential algorithm is sacrificed in favor of *thread-local* open sets (fewer contention points). When using a shared state storage (in a general setup or with stack-slicing), a thread-safe set V is required, which will be discussed in the following section.

Load balancing. A naive parallelization of reachability can be realized as follows: perform a depth-limited sequential BFS exploration and hand off the found states to several threads that start executing Algorithm 2.1 ($T = \{$*part of BFS exploration*$\}$ and V is shared). This is called *static load balancing*. For many models this will work due to common 'wide' state spaces. However, models with synchronization points or strict phase structure sometimes exhibit sandglass-shaped state spaces. Hence, threads run out of work when they reach the convergent funnel at the same time. A well-known problem that behaves like this is the *Towers of Hanoi* puzzle; when the smallest disk is on top of the tower only one move is possible.

Sanders [San97a] describes *dynamic load balancing* in terms of a problem P, a *work* operation and a *split* operation. P_{root} is the initial problem. Sequential execution takes $T_{\text{seq}} = T(P_{\text{root}})$ time units. A problem is (partly) solved when calling $work(P,t)$, which takes $min(t, T(P))$ units of time. For parallel reachability, $work(P,t)$ is Algorithm 2.1, where t has to be added as an extra input that limits the number of iterations of the *while* loop on Line 4 (and $P_{\text{root}} = T = \{s_0\}$). When threads become idle, they can poll others for work. The receiver will then split its own problem instance ($split(P) = \{P_1, P_2\}$, $T(P) = T(P_1) + T(P_2)$) and send one of the results to the polling thread.

Parallel architectures. We consider multi-core x86 server and desktop systems. These systems can process a large number of instructions per second, but have a relatively low memory bandwidth. Multiple levels of cache are used to continuously feed the cores with data, forming a steep memory hierarchy. Some of these caches are shared among multiple cores (often L2) and others are local (L1), depending on the architecture of the CPU and number of CPUs. The cache coherence protocol ensures that each core in each CPU has a global view of the memory. It transfers blocks of memory to local caches and synchronizes them if a local block is modified by other cores. Therefore, if independent writes are performed on subsequent memory locations (on the same cache line), a problem known as *cache line sharing* (*false sharing*) occurs, causing gratuitous synchronization and overhead (see Section 1.6.1 for an example).

The cache coherence protocol cannot be preempted. To efficiently program these machines, few options are left. One way is to completely partition the input [BR08], thus ensuring per-core memory locality at the cost of increased inter-die communication. An improvement of this approach is to pipeline the communication using ring buffers, which allows prefetching (explicit or hardwired). This scheme was explored, e.g., by Monagan and Pearce [MP09]. The last alternative is to minimize the *memory working set* of the algorithm [PH05]. We define the memory working set as the number of different memory locations that the algorithm updates in the time window that these usually stay in local cache. A small working set minimizes coherence overhead.

The memory model of the CPU provides vital guarantees about the order in which it commits reads (load instructions) and writes (store instructions) to memory. To improve instruction level parallelism, the individual cores namely reorder and delay (independent) loads and/or stores using *store buffers* (see Section 1.6.1). Sequential programs rely on *sequential consistency* ensuring a total order of both reads and writes (all previous memory operations are committed before the next is executed). In sequential operation this reordering behavior is thus opaque to the programmer, but parallel threads may observe the reordering from other threads. Some version of a memory model is often associated with all CPUs that implement a certain instruction-set architecture. For example, the popular x86 architecture observes *total-store order* (TSO), which guarantees that writes are never reordered (but reads may be). Others provide weaker guarantees (consult [Ber13] for an overview). Here, we are mainly interested in x86 [Int07], which is similar TSO. Intel [Int07] describes that x86 allows reordering of loads after stores, but no reordering of other combinations, roughly summarizing a complex specification. Some programming languages, especially platform-independent ones such as Java [Gos+05], provide their own memory model (c.f. [Gos+05, Sec. 17.4]).

Locks. It is common to ensure mutual exclusion for a critical section of code by locks. However, for resources with high contention, locks become infeasible. *Lock proliferation* improves on this by creating more locks on smaller resources. *Region-based locking* is an example of this, where a data structure is split into separately locked regions based on memory locations. However, this method is still infeasible for computational tasks with very high throughput. This is caused by the fact that the lock itself introduces another synchronization point; and synchronization between processor cores takes time.

Lockless algorithms. For high-throughput systems, *lock-free algorithms* (without mutual exclusion) are preferred. Lock-free algorithms guarantee system-wide progress, i.e., always *some* thread can continue. If an algorithm does not strictly provide progress guarantees (only statistically), but otherwise avoids explicit locks by the same techniques as used in lock-free solutions, it is called *lockless*. Lockless algorithms often have considerably simpler implementations, at no performance penalty. Last, *wait-free algorithms* guarantee per-thread progress, i.e., *all* threads can continue.

Many modern CPUs implement a *compare-and-swap* (CAS) instruction which ensures atomic memory modification while at the same time preserving data consistency if used in the correct manner. This can be done by reading the value from memory, performing the desired computation on it and writing the result back using CAS (Algorithm 2.2). If the latter returns true, the modification succeeded, if not, the computation needs to be redone with the new value, or some other form of collision resolution should be applied.

Algorithm 2.2 "Compare&Swap" specification

```
Pre: $word \neq \textbf{null}$
Post: $(*word_{pre} = testval \Rightarrow *word_{post} = newval) \wedge$
    $(*word_{pre} \neq testval \Rightarrow *word_{post} = *word_{pre}) \wedge$
    returns $(*word_{pre} = testval)$
atomic bool CAS(int *word, int testval, int newval)
```

Lockless algorithms can achieve a high level of concurrency. However, an instruction like CAS easily costs 100–1000 instruction cycles depending on the CPU architecture. Thus, abundant use defies its purpose.

Quantifying parallelism. Parallelism is usually quantified by normalizing the performance gain with regard to a sequential run (*speedup*): $S = T_{seq}/T_{par}$. Linear speedups grow proportional to the number of cores and indicate that an algorithm scales well.

Ideal speedup is achieved when $S \geq N$. For a fair comparison of scalability, it is important to use the fastest tool for T_{seq}, or speedups will not be comparable, since better optimized code is harder to scale (e.g., [HB07]).

Hashing. A well-studied method for storing and retrieving data with amortized time complexity $O(1)$ is *hashing* [Lit80]. A hash function h is applied to the data, yielding an index in an array of *buckets* that contain the data or a pointer to the data. Since the domain of data values is usually unknown and much larger than the image of h, hash collisions occur when $h(D_1) = h(D_2)$, with $D_1 \neq D_2$. Structurally, collisions can be resolved either by inserting lists in the buckets (*chaining*) or by probing subsequent buckets (*open addressing*). Algorithmically, there is a wealth of options to maintain the "chains" and calculate subsequent buckets [Cor+09]. The right choice depends entirely on the requirements dictated by the algorithms that use the hash table.

2.3 A Lockless Hash Table

In principle, Algorithm 2.1 seems easy to parallelize; in practice it is difficult to do this efficiently because of its memory intensive behavior, which becomes more obvious when looking at the implementation of set V. In this section, we present an overview of the options in hash table design. There is no silver bullet design and individual design options should be chosen carefully, considering the requirements stipulated by the use of the hash table. Therefore, we evaluate the demands that the parallel model checking algorithms place on the state storage solution. We also mention additional requirements stemming from the targeted hardware and software systems. Finally, we present a specific hash table design.

2.3.1 Requirements on the State Storage

Our goal is to realize an efficient shared state storage for parallel model checking algorithms. Traditional hash tables associate a piece of data to a unique key in the table. In model checking, we only need to store and retrieve *state vectors*, therefore the key is the state vector itself. Henceforth, we will simply refer to it as *data*. Our specific model checker implementation introduces additional requirements, discussed later. First, we list the definite requirements on the state storage:

- The storage needs only one operation: find-or-put. This operation inserts the state vector if it is not found or yields a positive answer without side effects. We require find-or-put to be concurrently executable to allow sharing the storage among the different threads. Other operations are not necessary for reachability algorithms, since the state space is growing monotonically. By exploiting this feature we can simplify the algorithms, thus lowering the strain on memory, and avoiding cache line sharing. Our choice is in sharp contrast to standard literature on concurrent hash tables, which often favors a complete solution, which is optimized for more general access patterns [PH05; Cli07].

- The storage should not require continual memory allocation, for the obvious reasons that this behavior would increase the *memory working set*.

- The use of pointers on a per-state basis should be avoided. Pointers take a considerable amount of memory when large state spaces are explored (more than 10^8 states are easily reachable with today's model checkers), especially on 64-bit machines. In addition, pointers increase the memory working set.

- The time efficiency of find-or-put should scale with the number of processes executing it in parallel. Ideally, the individual operations should — on average — not be slowed down by other operations executing at the same time, thus ensuring nearly linear speedup. Many hash table algorithms have a large memory working set due to their probing behavior or reordering behavior upon insertions. They suffer performance degradation in high throughput situations as is the case for us.

Specifically, we do not require the state storage to be resizable. The available memory on a system can safely be claimed for the table, because the largest part will be used for it eventually anyway. In sequential operation and especially in the presence of a *delete* operation (shrinking tables), one would consider resizing for the obvious reason that it improves locality and thus cache hits. In a concurrent setting, however, these cache hits have the opposite effect of causing the earlier described cache line sharing among CPUs. We experimented with lockless and concurrent resizing mechanisms and observed large decreases in performance.

Furthermore, the design of the LTSmin tool [BPW10], which we extended with multi-core reachability, also introduces some specific requirements:

- The storage data consists only of integer arrays or vectors of known and fixed length. This is the encoding format for state vectors employed by our language frontends.

- The storage is targeted at common x86 architectures, using only the available (atomic) instructions.

While the compatibility with the x86 architecture allows for concrete analysis, the applicability of our design is not limited to it. Lessons learned here are transferable to other architectures with similar memory hierarchy and atomic operations.

2.3.2 Hash Table Design

We determined that a low memory working set is one of the key factors to achieve maximum scalability. Also, we opt for simplicity whenever the requirements allow for it. From experience we know that complexity of a solution arises automatically when introducing concurrency. These considerations led us to the following design choices:

- **Open addressing**, since the alternative *chaining* hash table design would incur in-operation memory allocation or pre-allocation at different addresses, both leading to a larger memory working set.

- **Walking-the-line** is the name we gave to *linear probing* on a cache line, followed by *double hashing* (also employed elsewhere [Cli07; HST08]). Linear probing allows a core to benefit fully from a loaded cache line, while double hashing realizes better distribution.

- **Separating data** (vectors) in an indexed *data array* (of size $buckets \times |vector|$) ensures that the *bucket array* stays short[2.1] and subsequent probes can be cached.

- **Hash memoization** speeds up probing, by storing the hash (or part of it) in a bucket. This avoids expensive lookups in the data array as much as possible [Cli07].

- **Lockless** operation on the bucket array using a dedicated value to indicate unused buckets. One bit of the hash can be used to indicate whether the vector was already written to the data array or whether writing is still in progress [Cli07].

- **Compare-and-swap** is used as an atomic primitive on the buckets, which are precisely in either of the following distinguishable states: *empty*, *being written* and *complete*.

2.3.3 Hash Table Operations

Algorithm 2.3 shows the find-or-put operation. We assume for now that each line of code can be executed atomically and with sequential consistency. At the end of the current

[2.1] E.g., 1 GB for a 32-bit memoized hash and 2^{28} buckets section, we discuss implementation solutions for different memory models. Buckets are represented by the M array, the separate data by the T array and hash functions used for double hashing by hash_i, where i represents the *hash seed* analogous to the concept of random seeds used to initialize random number sequences. A separate hash function, i.e. a different seed, is used for the memoized hash h and the indexing hash i to reduce collisions. Probing continues (Line 3) until either a free bucket is found for insertion (Line 7–8), or the data is found to be in the hash table (Line 12). Too many probes indicate a full table, which simply causes the application to abort (Line 13).

The *for* loop on Line 5 handles the walking-the-line probing behavior. Algorithm 2.4 captures said behavior. It returns a sequence of indices in the memoized hash array M such that each corresponding bucket lies on the same cache line as the bucket $M[i]$. The cache line can be determined by the pointer to each bucket $M[x]$, as is done here by the function *cl(M[x])*. The point of returning a sequence of indices, is to keep the algorithm

Algorithm 2.3 The find-or-put algorithm

```
Require: $|T| = |M|$
 1: procedure table_find_or_put($\langle T, M \rangle$, $V$)
 2:     $h$ := hash$_0(V)$
 3:     for $count := 1$ to $threshold$ do
 4:         $i$ := hash$_{count}(h)$ mod $|T|$
 5:         for all $i \in$ walk_the_line$(M, i)$ do
 6:             if cas$(M[i], \texttt{empty}, \langle h, \texttt{wait} \rangle)$ then    ▷ Expensive CAS instruction
 7:                 $T[i]$ := $V$                                          ▷ Write data array
 8:                 $M[i]$ := $\langle h, \texttt{done} \rangle$
 9:                 return $\langle$false$, i \rangle$
10:             if $M[i] = \langle h, - \rangle$ then
11:                 await $M[i] = \langle h, \texttt{done} \rangle$
12:                 if $T[i] = V$ then return $\langle$true$, i \rangle$           ▷ Read data array
13:     report table full
```

Algorithm 2.4 Walking the (cache) line

```
Require: $|T| = |M|$
procedure walk_the_line$(M, i)$
    $low$ := $\min(\{x \mid cl(M[x]) = cl(M[i])\})$      ▷ Lowest index on same cache line
    $high$ := $\max(\{x \mid cl(M[x]) = cl(M[i])\})$     ▷ Highest index on same cache line
    return $\langle i, \ldots, high, low, \ldots, i-1 \rangle$
```

deterministic. Note that the simplified code for walk_the_line returns duplicate indices which should be removed to avoid unnecessary probes. The other code inside the loop on Line 5 handles the synchronization among threads. We now explain this part of the algorithm in detail.

Buckets store memoized hashes and the `wait` status bit of the data in the *Data* array. The possible values of the buckets are thus: EMPTY, $\langle h, \texttt{wait}\rangle$ and $\langle h, \texttt{done}\rangle$, where h is the memoized hash. If an empty bucket is encountered on a probe sequence, the algorithm tries to claim it by atomically writing $\langle h, \texttt{wait}\rangle$ to it (Line 6). After finishing the writing of the data, $\langle h, \texttt{done}\rangle$ is written to the bucket (Line 8). Non-empty buckets prompt the algorithm to compare the memoized hashes (Line 10). Only if they match and if any writes to the data array have been completed (Line 11), the value in the data array is compared with the vector (Line 12).

Several aspects of the algorithm guarantee correct lockless operation:

- Whenever a write started for a hash value, the state of the bucket can never become empty again, nor can it be used for any other hash value. This ensures that the probe sequence remains deterministic and cannot be interrupted.

- The CAS operation on Line 6 ensures that only one thread can claim an empty bucket, marking it as non-empty with the memoized hash and with status `wait`.

- The *await* statement at Line 11 waits until the write to the data array has been completed.

Critical synchronization between threads occurs when multiple threads try to write to an empty bucket. The CAS operation ensures that only one will succeed. The others carry on in their probe sequence, possibly waiting until first thread's completion. Eventually, they either find another empty bucket, or the state vector in some bucket (the same bucket or a bucket later in the probe sequence). This design can be seen as a lock on the lowest possible level of granularity (individual buckets), but without a true locking structure and associated additional costs. The algorithm uses a "lock" at Line 11, which can implemented as a spinlock. Although it could be argued that this algorithm is therefore not lock-free, it is possible to ensure local progress in the case that the "blocking" thread dies or hangs (making the algorithm wait-free). Wait-freeness is commonly achieved by making each thread fulfill local invariants, whenever they are not (yet) met by other threads [HS08]. Our measurements show, however, that under normal operation the loop on Line 11 is rarely hit due to the preceding hash memoization check (Line 10). Thus, we took the pragmatic choice of keeping the algorithm, and thus the implementation, as simple as possible.

Implementation. The implementation of Algorithm 2.3 requires exact guarantees from the underlying memory model. Instruction reordering by compilers and processors needs to be avoided across the synchronization points, otherwise the implementation becomes incorrect. It is, for example, a common optimization to execute the body of an *if* statement before the actual branching instruction. Such a speculative execution would keep the processor pipeline busy, but would be a disastrous reordering when applied to Line 6 and Line 7: the actual writing of the data would happen before the bucket is marked as full, allowing other threads to write to the same bucket. Likewise, reordering Line 7 and Line 8 would prematurely indicate that writing the data has completed.

As explained above, our requirements stipulate the support of the x86 architecture. Our language of choice is C, as it allows precise control over memory allocation (and thus implementing the *walking-the-line* probe sequence), and is used by our model checker tool LTSmin [LPW11a]. Our implementation uses the GNU gcc compiler, which provides built-in access to atomic instructions, such as CAS.[2.2] Note that on x86 these atomic instructions imply *memory barriers*, ensuring sequential consistency locally. Furthermore, we may rely on the fact that in the x86 architecture loads and stores are atomic on aligned word-sized data [Int07], such as the memoized hashes in M.[2.3] It is therefore not a problem to implement every line in Algorithm 2.3 atomically.

The next step is dealing with the weak memory model of x86. To this end, gcc's atomic built-ins and/or explicit barriers could be used to prevent reordering all accesses to shared memory locations in Algorithm 2.3 (Line 6 – 8 and Line 10 – 12). However, the memory barriers would then serialize the entire execution resulting in no scalability.

To reason about the minimal number of barriers required, we consider all lines in the code where shared data is read or written to. At Line 6, a memory barrier is unavoidable (CAS is required). At Line 7 – 8, shared data is written, thus not reodered under TSO, hence no memory barrier is required. At Line 10 – 12, shared data is read. These 3 instructions are not reordered by the x86 architecture [Int07]. It should be clear from this explanation that weaker memory models require a (load) barrier on Line 11 (if loads can be reordered), and/or a (store) barrier on Line 8 (if stores can be reordered).

We can also suggest guidelines for an implementation in Java. The Java Memory Model (JMM) makes precise guarantees about the possible commuting of memory reads and writes, by defining a partial order on all statements that effect the concurrent execution model [Gos+05, Sec. 17.4]. A correct implementation in Java should declare the bucket array as *volatile* and use `java.util.concurrent.atomic` package for atomic references and CAS. The volatile bucket array is needed because the JMM does

[2.2] `http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html`.

[2.3] However, one should always avoid volatiles [ER08], and take careful precautions to avoid compiler-time reorderings: `https://www.kernel.org/doc/Documentation/atomic_ops.txt` not entail TSO. We are unsure however whether the JVM will avoid the unnecessary memory barrier at Line 11 (in a loop!), on TSO architectures.

Finally, it is good practice to avoid impotent CAS instructions. This can be done by placing an extra if condition before Line 6 which checks whether indeed $M[i] = \texttt{empty}$ (not shown in Algorithm 2.3). *If CAS is executed blindly, many unnecessary overhead is caused by its implied memory barrier* (which locks the memory bus and waits until all prior memory operations are committed). Agarwal et al. [Aga+10] also mention this optimization.

Validation. Algorithm 2.3 was modeled in PROMELA and checked for deadlocks with SPIN. One bug concerning the combination of write bit and memoized hash was found and corrected.

An enormous amount of experiments with hundreds of different models, further confirms that the number of states and transitions reported always agrees with the results of other model checkers.

2.4 Experiments

2.4.1 Methodology

We implemented the hash table in C for x86 architectures (TSO memory model) as explained in the previous section. We further implemented a load balancer and parallel reachability algorithm from Section 2.2, and integrated everything in our model checking toolset LTSMIN, which we discuss further in the following section. For the following experiments, we reuse not only the input models, but also the `next_state` implementation of DIVINE 2.2. Therefore, a fair comparison with DIVINE 2.2 can be made. Furthermore, we performed experiments with the latest multi-core capable version of the model checker SPIN 5.2.4 [HB07] (DIVINE models were mechanically translated to SPIN's PROMELA input language). For the experiments, we chose full state-space exploration via reachability as load generator for our state storage. Reachability exhibits similar access patterns as more complex verification algorithms, but reduces the code footprint and therefore potential pollution of our measurements with noise.

All model checkers were configured for maximum performance. For all tools, we compiled models to C with high optimization settings (`-O3`) (DIVINE also contains a model interpreter). SPIN's models were compiled with the following flags: `-O3 -DNOCOMP -DNOFAIR -DNOREDUCE -DNOBOUNDCHECK -DNOCOLLAPSE -DNCORE=`N `-DSAFETY -DMEMLIM=100000`; To run the models we used the options: `-m10000000 -c0 -n -w28`. Refer to [Hol11] for details.

We performed our experiments on AMD Opteron 8356 16-core servers with 64 GB RAM, running a patched Linux 2.6.32 kernel.[2.4] All tools were compiled using gcc 4.4 in 64-bit mode with maximal compiler optimizations (`-O3`).

2.4.2 Models

A total of 31 models from the BEEM database [Pel07] have been used in the experiments, which are shown in Table 2.2 (we filtered out models which were too small to be interesting, or too big to fit into the available memory). Every run was repeated at least four times, to exclude any accidental fluctuation in the measurements. Special care has been taken to keep all the parameters across the different model checkers the same. Especially SPIN provides a rich set of options with which models can be tuned to perform optimal. Using these parameters on a per-model basis could give faster results than presented here. It would, however, say little about the scalability of the core algorithms. Therefore, we decided to leave all parameters the same for all the models. We avoid resizing of the state storage in all cases by increasing the initial hash table size to accommodate 2^{28} states (enough for all benchmarked input models).

One parameter that we cannot control is the difference in state vector sizes. DIVINE and SPIN use different vectors because the DVE models have been translated to PROMELA. LTSMIN uses the original DVE models as well. But because LTSMIN is a language-independent model checker, it defines a fixed format for these vectors, which can cause an increase in size of up to a factor 3 (characters are stored as integers). Table 2.2 shows the different state vector sizes in SPIN, DIVINE and LTSMIN. When comparing the different tools therefore, we take great care to compare absolute speedups, i.e. using the runtimes of the fastest tool for T_{seq}.

Because of the translation, the state count of some models is different in SPIN. For this reason, only 19 models could be used for SPIN: only those with similar state counts (less than 20% difference; recall that for SPIN, models are translated from DVE to PROMELA).

2.4.3 Results

Figure 2.2 shows the runtimes of only three models for all model checkers. We observe that DIVINE is the fastest model checker for sequential reachability. Since the last published comparison between DIVINE and SPIN [BBR07], DIVINE has been improved

[2.4] Experiments showed large regressions in scalability on newer 64-bit Linux kernels (degrading runtimes with 10+ cores). Despite being undetected since at least version 2.6.20 (released in 2007!), they were easily exhibited by our model checker. With a repeatable test case, the Linux developers quickly provided a patch: `https://bugzilla.kernel.org/show_bug.cgi?id=15618`

Table 2.2: Model details for DiVinE, LTSmin and spin

Beem Model	Reachable States		State Vector Size [Byte]		
	DiVinE, LTSmin	spin	DiVinE	spin	LTSmin
anderson.6	18,206,917	18,206,919	25	68	76
at.5	31,999,440	31,999,442	20	68	56
at.6	160,589,600	—	20	—	56
bakery.6	11,845,035	11,845,035	24	48	80
bakery.7	29,047,471	27,531,713	24	48	80
blocks.4	104,906,622	88,987,772	23	44	88
brp.5	17,740,267	—	24	—	72
cambridge.7	11,465,015	—	60	—	208
elevator_planning.2	11,428,767	11,428,769	36	52	140
firewire_link.5	18,553,032	—	66	—	200
fischer.6	8,321,728	8,321,730	27	92	72
frogs.4	17,443,219	17,443,221	33	68	120
frogs.5	182,772,126	182,772,130	38	68	140
hanoi.3	14,348,907	14,321,541	63	116	228
iprotocol.6	41,387,484	—	43	—	148
iprotocol.7	59,794,192	—	47	—	164
lamport.8	62,669,317	62,669,317	22	52	68
lann.6	144,151,628	—	28	—	80
lann.7	160,025,986	—	35	—	100
leader_filters.7	26,302,351	26,302,351	36	68	120
loyd.3	239,500,800	214,579,860	18	44	64
mcs.5	60,556,519	53,779,475	26	68	84
needham.4	6,525,019	—	51	—	112
peterson.7	142,471,098	142,471,100	30	56	100
phils.6	14,348,906	13,956,555	45	140	120
phils.8	43,046,720	—	48	—	128
production_cell.6	14,520,700	—	42	—	104
szymanski.5	79,518,740	79,518,740	30	60	100
telephony.4	12,291,552	12,291,554	24	56	80
telephony.7	21,960,308	21,960,310	28	64	96
train-gate.7	50,199,556	—	43	—	128

with a model compiler. In fact, all 3 model checkers use the approach of compiling the model to obtain fast successor generation. SPIN is only slightly slower than DiVinE and shows the same linear curve but with a gentler slope. We suspect that the gradual performance gains are caused by the cost of the inter-thread communication (see Table 2.1).

LTSmin is slower in the sequential cases. We verified that the allocation-less hash table design causes this behavior; with smaller hash table sizes, the sequential runtimes match those of DiVinE. We did not bother optimizing these results, because with two cores, LTSmin is already at least as fast as DiVinE.

Figure 2.3, Figure 2.4 and Figure 2.5 show the speedups measured with LTSmin, DiVinE and SPIN (note that we normalize with T_{seq} of DiVinE, the fastest sequential tool). On 16 cores, LTSmin shows a two-fold improvement over DiVinE and a four-fold improvement over SPIN. We attribute the difference in scalability for DiVinE to the extra synchronization points needed for the inter-process communication by DiVinE. Recall

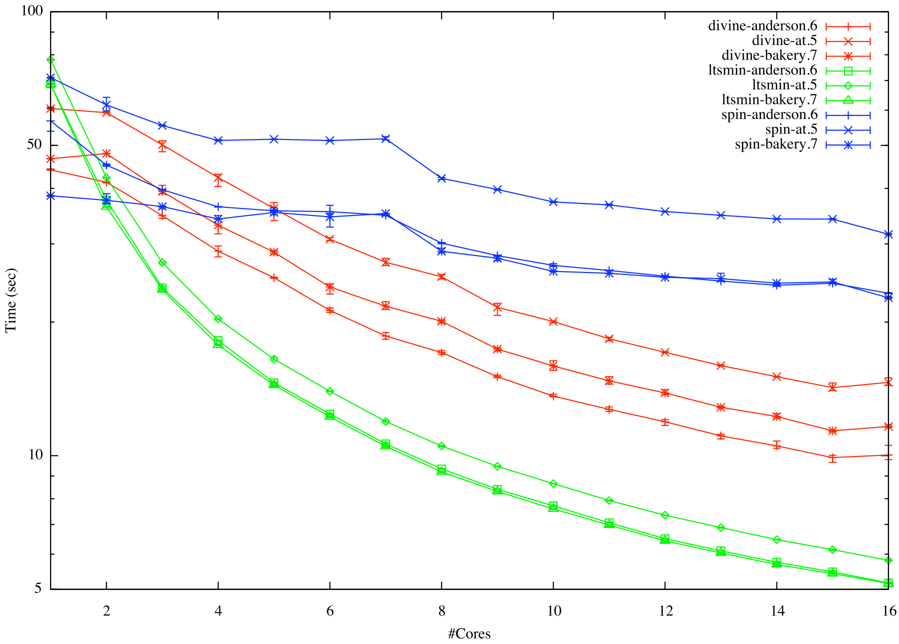


Figure 2.2: (Log-scale) Runtimes in SPIN, LTSmin and DiVinE 2 (3 models)

that the model checker uses static state-space partitioning, hence most successor states are enqueued at other cores than the one which generated them. Another disadvantage of DiVinE is its use of a management thread, which causes the regression at 8 and 16 cores.

The speedups are shown to be linear for LTSmin. Only a speedup of 8 with 16 cores is achieved compared to the sequential case of DiVinE. Figure 2.6 shows the speedup of the individual models with LTSmin as sequential base case (T_{seq}), to illustrate the scalability of the hash table itself. These results demonstrate almost ideal scalability.

spin shows inferior scalability even though it uses (like LTSmin) a shared hash table, while doing load balancing via *stack slicing*. We can only guess that the locking mechanism used in spin's hash table (region locking) are not as efficient as our lockless hash table. However, in LTSmin we obtained far better results even with the slower `pthread` locks. It might also be that stack slicing does not have a consistent granularity, because it uses the (irregular) search depth as a time unit (using the terms from

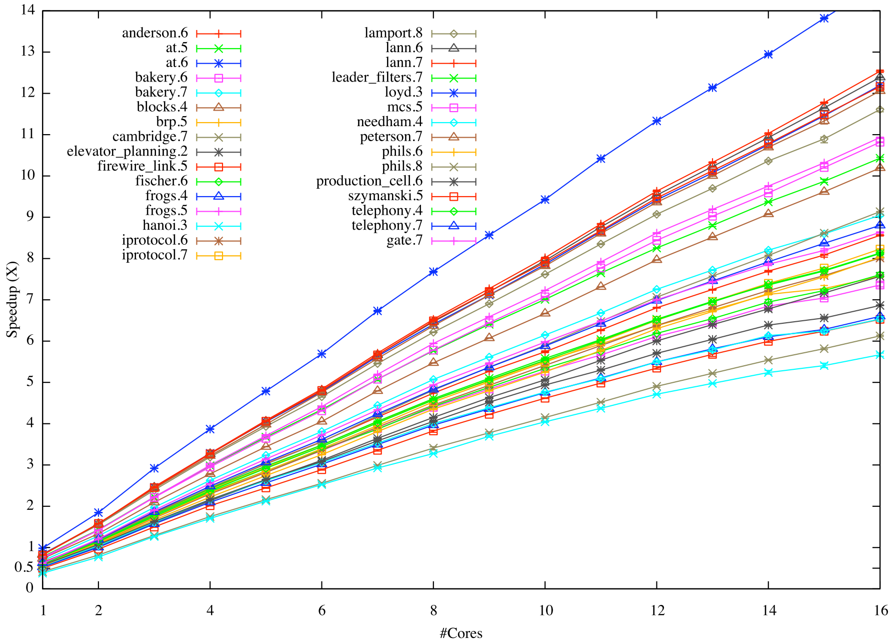


Figure 2.3: Speedup of Beem models with LTSmin (DiVinE as base case).

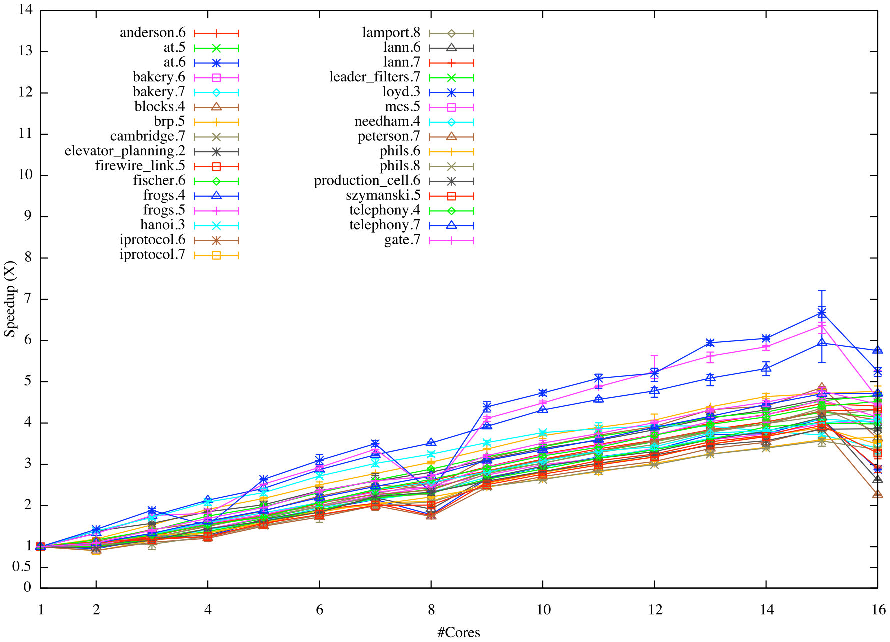


Figure 2.4: Speedup of Beem models with DiVinE 2.2 (DiVinE as base case)

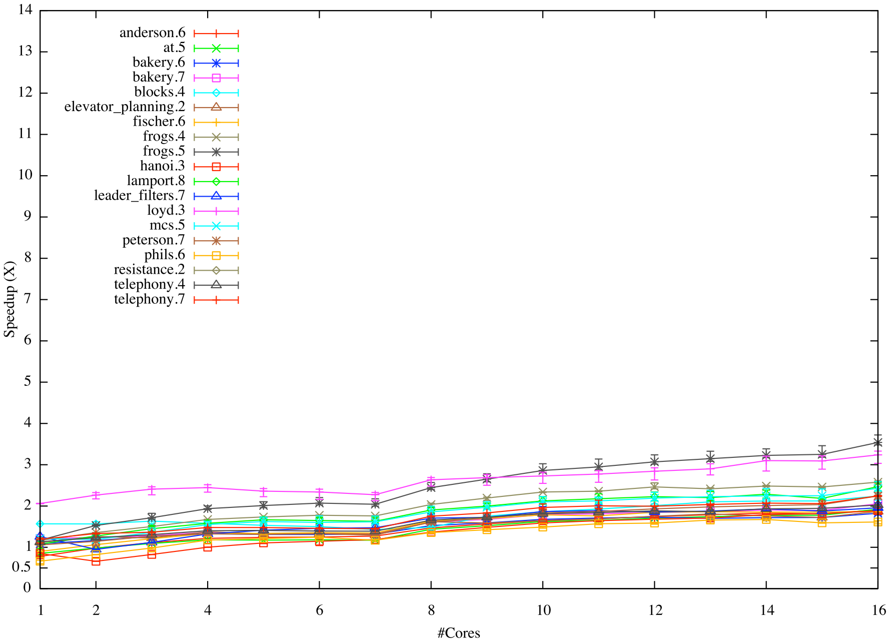


Figure 2.5: Speedup of Beem models with spin (DiVinE as base case)

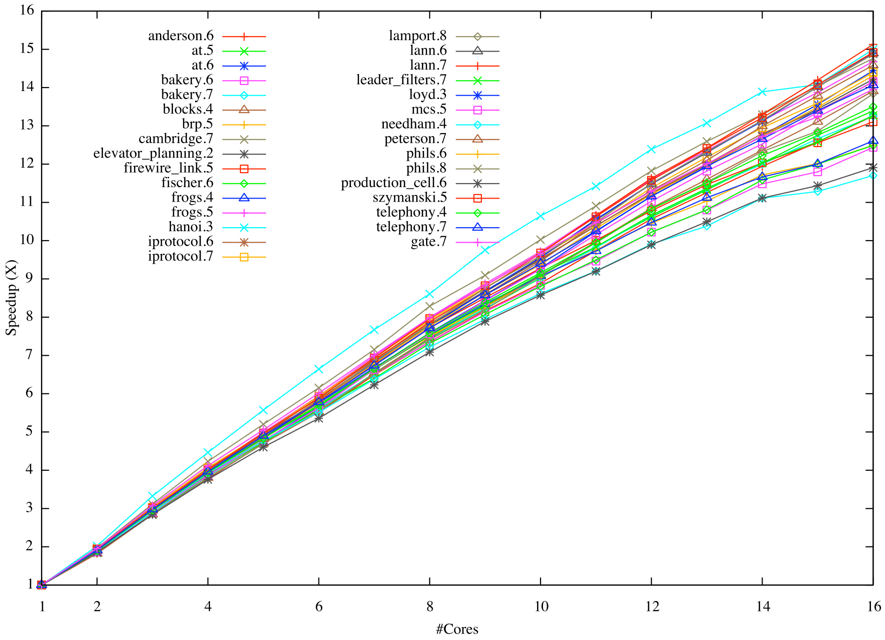


Figure 2.6: Speedup of Beem models with LTSmin (LTSmin itself as base case). This figure shows that the shared hash table implementation attains almost ideal speedups.

Section 2.2: $T(work(P_0, depth)) \gg T(work(P_1, depth)))$.

Remark. A potential reason for the limited scalability of spin could be a memory bandwidth bottleneck. We tested this hypothesis by enabling spin's smaller, *collapsed* state vectors (`-DCOLLAPSE`). We carried out a full spin benchmark run with collapsing enabled (see Figure 3.3) and saw little improvement compared to the speedup results without `COLLAPSE`. These results are consistent with the observation that LTSmin is faster, despite generally producing larger state vectors than both, spin and DiVinE (Table 2.2): in LTSmin, each state variable gets 32-bit aligned (for API reasons, not performance).

Figure 2.7 shows the total times and average speedups over all models, for all tools.

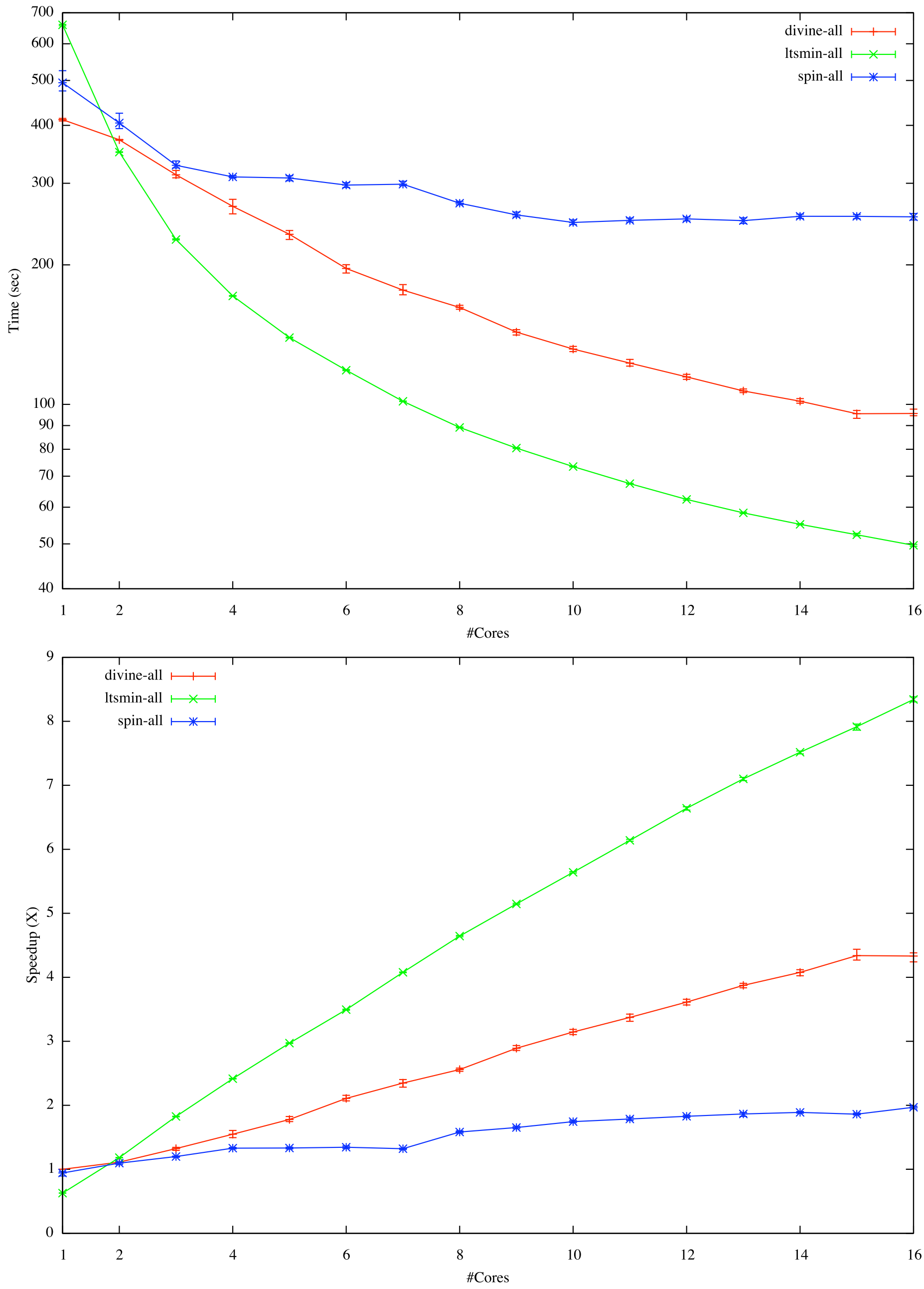


Figure 2.7: Total runtime/speedup of SPIN, DIVINE 2.2 and LTSMIN

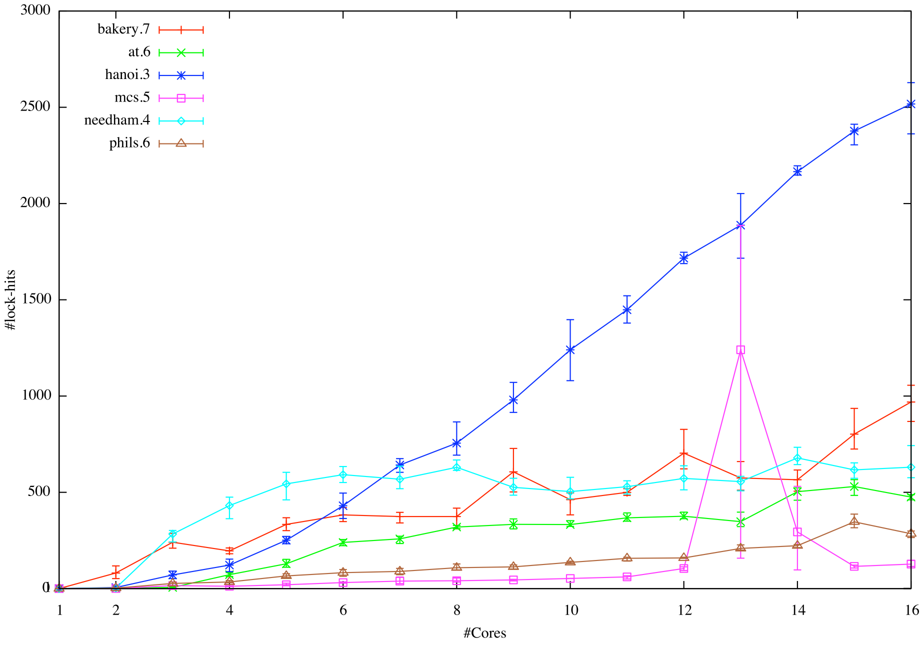


Figure 2.8: Counting how often the algorithm "locks"

2.4.4 Shared-Storage Parameters

To verify our claims about the hash table design, we collected internal measurements and performed synthetic benchmarks for stress testing. First, we measured how often the write "lock" was hit. Figure 2.8 plots the lock hits against the number of cores for several different sized models. For readability, only the worst-performing, and thus most interesting, models were chosen. Even then, the number of lock hits is a very small fraction of the number of table_find_or_put calls (equal to the number of transitions, typically in the hundreds of millions). The Hanoi puzzle performs worst in this respect, probably again due to its sandglass-shaped state space (see Section 2.2).

We measured how the average throughput of Algorithm 2.3 (the number of table_find_or_put calls) is affected by the table *fill rate*, the table size and the read/write ratio. Figure 2.9 illustrates the effects of different read/write ratios on the hash table using synthetic input data. The average throughput remains largely unaffected by a high

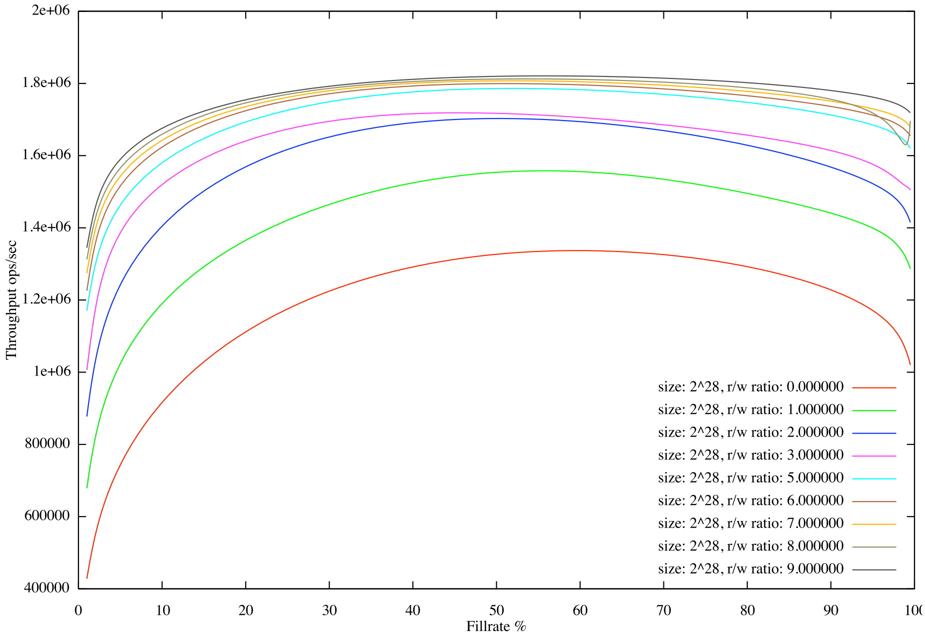


Figure 2.9: Effect of fill rate and r/w-ratio on average throughput

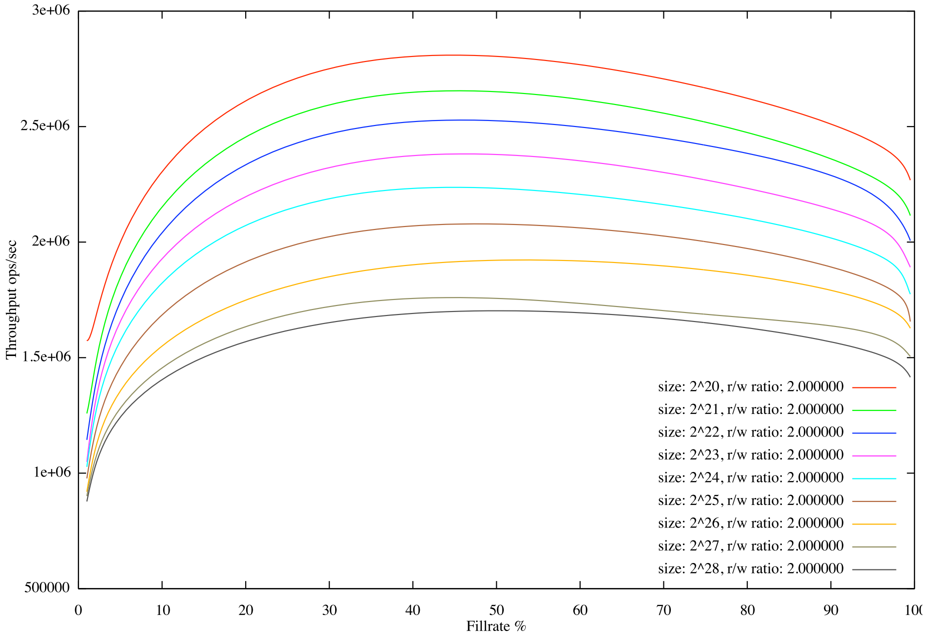


Figure 2.10: Effect of size vs r/w-ratio on average throughput

fill rate, even up to 95 % (as for Figure 2.10, which plots the same lines for different table sizes). We conclude that the asymptotic time complexity of open-addressing hash tables poses little real problems in practice. However, an observable side effect of *oversized hash tables* is lower throughput for low fill rates due to increased cache misses. Our hash table design amplifies this effect because it uses a pre-allocated data array and no pointers. This explains the lower sequential performance of LTSMIN.

We also measured the effect of varying the state vector size and did not find any noticeable change in the speedup behavior (except for the expected lower throughput due to higher data movement). This shows that hash memoization and a separate data array perform well. Walking-the-line probing shows better performance and scalability than double hashing alone, due to cache effects. Although slower on average, walking-the-line followed by double hashing beats simple linear probing at fill-rates above 95 % (in particular, on slower memory subsystems), because it leads to better distribution and thus fewer probes.

2.5 Discussion and Conclusions

We designed a hash table suitable for application in reachability analysis. We implemented it as part of a model checker together with different exploration algorithms (pseudo BFS and pseudo DFS) and explicit load-balancing. We demonstrated the efficiency of the complete solution by comparing the absolute speedups to SPIN 5.2.4 and DIVINE 2.2, both leading tools in this field. We claim two times better scalability than DIVINE and four times better than SPIN *on average* (Figure 2.7), with individual results far exceeding these numbers. We also investigated the behavior of the hash table under different fill rates and found it to live up to the imposed requirements.

Limitations. Without the use of pointers the current design cannot easily cope with variably sized state vectors. In our model checker, this does not pose a problem because states are always represented by a vector of a static length. Our model checker LTSMIN [BPW10; LPW11a] can handle different frontends. It connects to DIVINE-cluster, DIVINE 2.2, PROMELA (via NIPSVM [Web07]), μCRL, MCRL2 and ETF (internal symbolic representation of state spaces). Some of these input languages require variably sized vectors (NIPS). We solve this by an initial exploration which continues until a vector of stable size is found, and aborts when none can be found up to a fixed bound. So far, this limitation did not pose a problem.

For LTSMIN, the results in the sequential case turn out to be around 20% slower than DIVINE 2.2. One of the culprits for this performance loss is the already mentioned suboptimal utilization of cache effects for small models. Indeed, the slowdown is observable in Figure 2.3 mostly for those models with small state spaces according to Table 2.2. Embracing pointers and allocation, like in e.g. [Mic02], could be a potential remedy, however, it is unclear whether such a solution still scales when it actually matters (i.e., for large models).

Further performance is lost in an extra level of indirection (function calls) due to the design of LTSmin, which strictly separates the language frontend from the exploration algorithms. We are willing to pay this price in exchange for the increased modularity of our tool.

Discussion. We make several observations:

- We provide evidence that centralized state storage can be made to scale at least as well as static state-space partitioning, contrary to prior belief [BR08].
- We also show that scalability is not as dependent on long state vectors and transition delays as earlier thought [Hol08]. In fact, we argue that a scaling implementation performs better with smaller state vectors, because the number of operations performed per loaded byte is higher, thus closer to the strengths of modern multi-core systems.
- Shared state storage is also more flexible [BR08], for example allowing pseudo DFS (like the stack-slicing algorithm) and fast deadlock/invariant searches [RK88]. Moreover, it facilitates explicit load balancing algorithms, enabling the exploitation of *heterogeneous systems*. From preliminary experiments with load balancing we conjecture that overhead is negligible compared to static load balancing.
- Performance-critical parallel software needs adaptation to modern architectures (steep memory hierarchies). The performance difference between DiVinE, spin and LTSmin is an indication. DiVinE uses an architecture which is directly derived from distributed model checking and the goal of spin was for “these new algorithms [...] to interfere as little as possible with the existing algorithms for the verification of safety and liveness properties” [Hol08]. With LTSmin, we had the opportunity to tune our design to the architecture of our target machines, with excellent pay-off. We noticed that avoiding cache line sharing and keeping a simple design was instrumental in the outcome.
- Holzmann conjectured that optimized sequential code does not scale well [HB07]. In contrast, our parallel implementation is faster in absolute numbers and also exhibits excellent scalability. We suspect that the (entirely commendable) design choice of spin’s multi-core implementation to support most of spin’s existing features *unchanged* is detrimental to scalability.

Applicability. The components of our reachability can be reused directly for other model checking applications. The hash table and the load balancing algorithms can be reused to realize scalable multi-core (weak) LTL model checking [BBR07; BBR09b], symbolic exploration and space-efficient enumerative exploration. We experimented with the latter using *tree compression* [Blo+08a] based on our hash table. Results are very promising and we follow up on that in Chapter 3 and Chapter 4.

Final note. Figure 2.6 shows that the speedups with LTSmin are almost ideal. We still considered it interesting to investigate the overhead caused by the load balancer, in order to identify the components that cause the most communication. By design, the load balancer should only be called when threads run out of work on their local search stack or queue, thus limiting communication as much as possible. Communication mainly

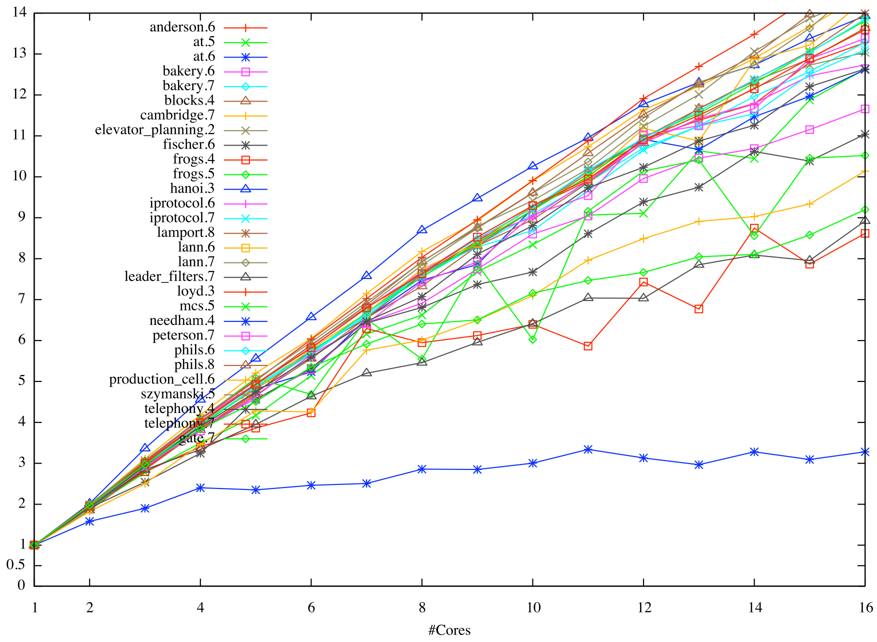


Figure 2.11: Speedup of Beem models with LTSmin using static load balancing (LTSmin itself as base case). Naturally, the sequential runtimes remain unaffected by the different load-balancer, so the figure is comparable to Figure 2.6.

happens via the shared hash table, though sparsely due to its low spatial locality. In an initial version of the exploration algorithm, we employed static load balancing by means of an initial, short BFS exploration, after which the states from the last level were handed off to all threads. The results are shown in Figure 2.11. Several models were insensitive to this static approach (c.f. Figure 2.6), others, like `hanoi` and `frogs`, are very sensitive due to the sandglass shape of their state spaces (see Section 2.2). A look at the runtimes confirmed for us that dynamic load balancing did not come with a noticeable performance penalty for the model that scale with static load balancing, i.e. without any communication except via the hash table. Since `hanoi` and `frogs` also scale well according to Figure 2.6, we can conclude that in all cases the load balancer causes very little overhead (in the sequential base case, load balancing namely does not occur).

Chapter 4 presents some experiments comparing our lockless hash table with other parallel solutions. The results show a clear advantage for our table. We did not yet consider other types of hash tables, like *Cuckoo hashing* [PR04] or *Hopscotch hashing* [HST08]. Cuckoo hashing is an unlikely candidate, since it requires updates on many locations upon inserts, easily resulting in extraneous cache coherence overhead. Hopscotch hashing could be considered because it combines a low memory working set with constant lookup times even under higher load factors. However, Hopscotch hashing increases the memory working set for insertions, potentially sacrificing some speedup. It would still be interesting to investigate its performance relative to our hash table.

3

Parallel Recursive State Compression for Free

Alfons Laarman, Jaco van de Pol, Michael Weber

Abstract

The current chapter focuses on reducing memory usage in enumerative model checking, while maintaining the multi-core scalability obtained in earlier work. We present a multi-core tree-based compression method, which works by leveraging sharing among sub-vectors of state vectors.

An algorithmic analysis of both worst-case and optimal compression ratios shows the potential to compress even large states to a small constant on average (8 bytes). Our experiments demonstrate that this holds up in practice: the median compression ratio of 279 measured experiments is within 17% of the optimum for tree compression, and five times better than the median compression ratio of SPIN's COLLAPSE compression.

Our algorithms are implemented in the LTSMIN tool, and our experiments show that for model checking, multi-core tree compression pays its own way: it comes virtually without overhead compared to the fastest hash table-based methods.

About this chapter: The current chapter is based on the paper *"Parallel Recursive State Compression for Free"*, which was published at SPIN 2011 [LPW11c]. An extended report on the work was published at Arxiv [LPW11b] and is integrated in the current chapter.

Compared to the original version of the paper, we simplified the reachability algorithm and the tree database figure in Section 3.2. We also completely revised Section 3.3, by simplifying the algorithms and adding illustrative examples. Section 3.4 was rewritten, to provide a more detailed understanding of the compression ratios and how they relate to the implementation choices of the tree.

3.1 Introduction

Many verification problems are computationally intensive tasks that can benefit from extra speedups. Considering recent hardware trends, these speedups do not come automatically for sequential exploration algorithms, but require exploitation of the parallelism within multi-core CPUs. In Chapter 2, we have shown how to realize scalable multi-core reachability, a basic task shared by many different approaches to verification.

Reachability searches through all the *states* of the program under verification to find errors or deadlocks. It is bound by the number of states that fit into the main memory. Since states typically consist of large *vectors* with one *slot* for each program variable, only small parts are updated for every step in the program. Hence, storing a state in its entirety results in unnecessary and considerable overhead. State compression solves this problem, as the current chapter will show, at a negligible performance penalty and with better scalability than uncompressed hash tables.

Related work. In the following, we identify compression techniques suitable for (on-the-fly) enumerative model checking. We distinguish between *generic* and *informed* techniques.

Generic compression methods, like Huffman encoding and run length encoding, have been considered for explicit state vectors with meager results [HGP92; GVR99]. These *entropy encoding* methods reduce *information entropy* [CT91] by assuming prevalence of common bit patterns. Such patterns have to be defined statically and cannot be "learned" (as in dynamic Huffman encoding), because it is infeasible to change the encoding during state-space exploration. Otherwise, desirable properties, like fast equivalence checks on states and constant-time state-space inclusion checks, will be lost.

Other work focuses on efficient storage in hash tables [Cle84; GV03] (see also Chapter 4). The assumption is that a uniformly distributed subset of n elements from the universe U is stored in a hash table. If each element in U hashes to a unique location in the table, only one bit is needed to encode the presence of the element. If, however, the hash function is not so perfect or U is larger than the table, then at least a quotient of the key needs to be stored and collisions need to be dealt with. This technique is therefore known as *key quotienting*. While its benefit is that the compression ratio is constant for any input (not just constant on average), compression is only significant for small universes [GV03], smaller than we encounter in model checking (where the universe consists of all possible combinations of the slot values, not to be confused with the set of reachable states, which is typically much smaller).

The information-theoretic lower bound on compression, or the *information entropy*, can be reduced further if the format of the input is known in advance (certain subsets of

U become more likely). This is what constitutes the class of *informed compression* techniques. It includes works that provide specialized storage schemes for certain specific state structures, like Petri nets [EPP05] or timed automata [Lar+97], but also Collapse compression introduced by Holzmann for the model checker Spin [Hol97b]. It takes into account the independent parts of the state vector. Independent parts are identified as the global variables and the local variables belonging to different processes in the Spin-specific language Promela.

Blom et al. [Blo+08a] present a more generic approach, based on a tree. All variables of a state are treated as independent and stored recursively in a binary tree of hash tables. The method was mainly used to decrease network traffic for distributed model checking. Like Collapse, this is a form of informed compression, because it depends on the assumption that subsequent states only differ slightly.

Problem statement. Information theory dictates that the more information we have on the data that is being compressed, the lower the entropy and the higher the achievable compression. Favorable results from informed compression techniques [EPP05; Lar+97; Hol97b; Blo+08a] confirm this. However, the techniques for Petri nets and timed automata employ specific properties of those systems (a deterministic transition relation and symbolic zone encoding respectively), and, therefore, are not applicable to enumerative model checking. Collapse requires local parts of the state vector to be syntactically identifiable and may thus not identify all equivalent parts among state vectors. While tree compression showed more impressive compression ratios by analysis [Blo+08a] and is more generically applicable, it has never been benchmarked thoroughly and compared to other compression techniques, nor has it been parallelized.

Generic compression schemes can be added locally to a parallel reachability algorithm (see Section 3.2). They do not affect any concurrent parts of its implementation and even benefit scalability by lowering memory traffic [HGP92]. While informed compression techniques can deliver better compression, they require additional structures to record uniqueness of state vector parts. With multiple processors constantly accessing these structures, memory usage is again increased and mutual exclusion locks are strained, thereby decreasing performance and scalability. Thus the benefit of informed compression requires considerable design effort on modern multi-core CPUs with steep memory hierarchies.

Therefore, in the current chapter, we address two research questions: (1) does tree compression perform better than other state-of-the-art on-the-fly compression techniques (most importantly Collapse), (2) can parallel tree compression be implemented efficiently on multi-core CPUs.

Contribution. The current chapter explains a tree-based structure that enables high compression rates (higher than any other form of explicit-state compression that we could identify) and excellent performance. A parallel algorithm is presented (Section 3.3) that makes this informed compression technique scalable in spite of the multiple accesses to shared memory that it requires, while also introducing *maximal sharing*. With an incremental algorithm, we further improve the performance, reducing contention and memory footprint.

An analysis of compression ratios is provided (Section 3.4) and the results of extensive and realistic experiments (Section 3.5) match closely to the analytical optima. The results also show that the incremental algorithm delivers excellent performance, even compared to uncompressed verification runs with a normal hash table. Benchmarks on multi-core machines show near-perfect scalability, even for cases which are sequentially already faster than the uncompressed run.

3.2 Background

In Section 3.2.1, we introduce a parallel *reachability* algorithm using a shared hash table similar to the one in Chapter 2. The table's main functionality is the storage of a large set of state vectors of a fixed length k. We call the elements of the vectors *slots* and assume that slots take values from the integers, possibly *references* to complex values stored elsewhere (hash tables or canonization techniques can be used to yield unique values for about any complex value). Subsequently, in Section 3.2.2, we explain two informed compression techniques that exploit similarity between different state vectors. While these techniques can be used to replace the hash table in the reachability algorithm, they are are harder to parallelize as we show in Section 3.2.3.

3.2.1 Parallel Reachability

The parallel reachability algorithm (Algorithm 3.1) launches N threads with unique *id*s and assigns the initial states of the model under verification only to the *open set* S_1 of the first thread (Line 1). The open set can be implemented as a *stack* or a *queue*, depending on the desired search order (note that with $N > 1$, the chosen search order will only be approximated, because the different threads will go through the search space independently). The *closed set* of visited states, *DB*, is shared, allowing threads executing the search algorithm (Line 4-7) to synchronize on the search space and each to explore a (disjoint) part of it (see Chapter 2). The find_or_put function returns *true* when *succ* is found in *DB*, and inserts it when it is not.

Load balancing is needed so that workers that run out of work ($S_{id} = \emptyset$) receive work from others. The function load_balance takes the local open set S_{id} from worker *id*, queries the open sets of the other workers and either detects termination returning *false*, or else transfers remote load to S_{id} and returns *true*. We implemented the function load_balance as a form of Synchronous Random Polling [San97a], which also ensures valid termination detection (see Section 2.2). It returns *false* upon global termination.

```
$S_1$.putall($initial\_states$)
parallel_for($id$ := 1 to $N$)
    while (load_balance($S_{id}$))
        while ($S_{id} \neq \emptyset$)
            $s$ := $S_{id}$.pop()
            if (NEXT-STATE($s$) $= \emptyset$)
                ...report deadlock...
            for ($t \in$ NEXT-STATE($s$))
                if ($\neg$find_or_put($DB$, $t$))
                    $S_{id}$.put($t$)
```

Algorithm 3.1: Parallel reachability algorithm with shared state storage.

DB is generally implemented as a hash table. In Chapter 2, we presented a lockless hash table design, with which we were able to obtain almost perfect scalability. However, with 16 cores, the physical memory, 64GB in our case, is filled in a matter of seconds, making memory the new bottleneck. Informed compression techniques can solve this problem with an alternate implementation of *DB*.

3.2.2 Collapse & Tree Compression

Collapse compression stores logical parts of the state vector in separate hash tables. A logical part is made up of state slots local to a specific process in the model, therefore the hash tables are called *process tables*. References to the parts in those process tables are then stored in a root hash table. Tree compression is similar, but works on the granularity of slots: tuples of slots are stored in hash tables at the fringe of the tree, which return a reference. References are then bundled as tuples and recursively stored in tables forming a binary tree. Figure 3.1 shows the difference between the process tree (depth 2) and tree compression (depth $\log(k)$).

When using a tree to store equal-length state vectors, compression is realized by the sharing of subtrees among entries. Figure 3.2 illustrates this. On the left a set of vectors is represented as stored in a hash table with k-sized buckets (omitting any empty buckets

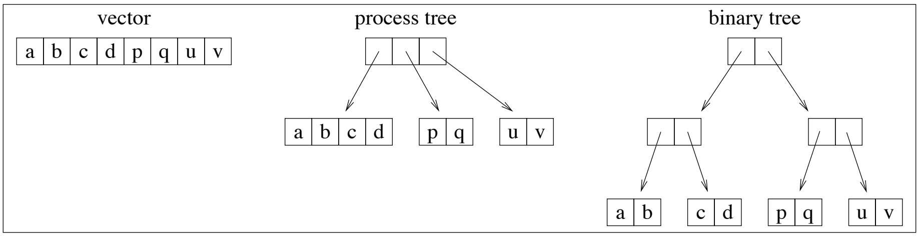
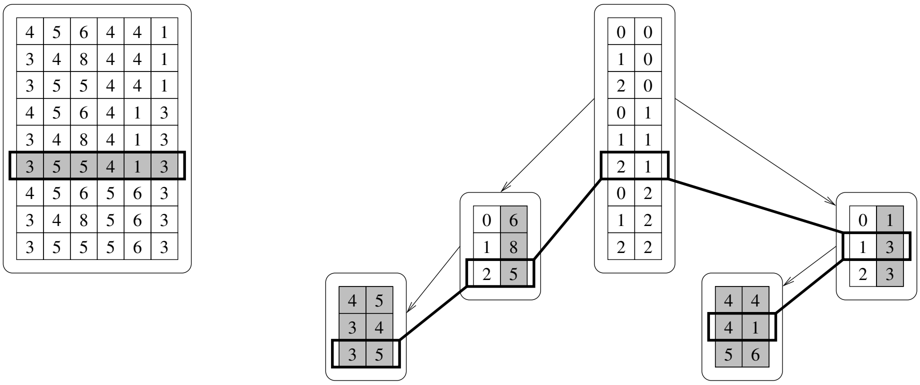


Figure 3.1: Process table and (binary) tree for the system $X(a,b,c,d)\|Y(p,q)\|Z(u,v)$. Taken from [BLL03].



4	5	6	4	4	1
3	4	8	4	4	1
3	5	5	4	4	1
4	5	6	4	1	3
3	4	8	4	1	3
3	5	5	4	1	3
4	5	6	5	6	3
3	4	8	5	6	3
3	5	5	5	6	3

0	0
1	0
2	0
0	1
1	1
2	1
0	2
1	2
2	2

0	6
1	8
2	5

0	1
1	3
2	3

4	5
3	4
3	5

4	4
4	1
5	6

Figure 3.2: Sharing of subtrees in tree compression. Taken from [Blo+08a].

in the table). On the right, we see the same set of vectors, but now stored in a binary tree of tables with buckets of size 2 storing tuples (again omitting any empty buckets). Each tuple in the *root table* represents one state vector. Take the tuple $\boxed{2\,|\,1}$ at location 5 in the root table: We use zero-based, top-to-bottom indexing in the bucket arrays (indexing has benefits over using larger pointers [Jan+06]). The values of the tuple (white) are indices in the hash tables of the root's children in the tree. The indexing continues recursively for the tuple values that are white, until we eventually reach the leaf values (gray), which represent the state reading from left to right.

Assuming that indices have the same size as the slot values (say b bits), we can determine the compression rate in this example. The 9 vectors as stored in the plain hash table on the left, take $9 \times 6b = 54b$. The tree on the other hand takes $9 \times 2b + 4 \times 3 \times 2b = 42b$. The difference between $42b$ and $54b$ may seem minimal, but we only stored 9 relatively short vectors of relatively small size. In model checking, we often deal with millions of vectors with potentially hundreds of slots.

It is easy to see that more vectors improve the compression. For example, we may add a tuple $\boxed{0}\boxed{9}$ to the table on the right. With this we can create 9 new vectors by adding 9 tuples $\boxed{*}\boxed{3}$ to the root table. These new vectors cost only $20b$ more, compared to the $54b$ it would take to store them in a hash table. In fact, with combinatorial state vectors the child tables store only $\sqrt{n}$ tuples, for a parent table storing n tuples, as shown in Section 3.4. In Section 3.5, we show empirically that the combinatorial condition often holds in practice.

3.2.3 Why Parallelization is not Trivial

Adding generic compression techniques to the above algorithm can be done locally by adding a line *compr* := compress(*succ*) after Line 8, and storing *compr* in *DB*. This calculation in compress only depends on the local *succ* and is therefore easy to parallelize. If, however, a form of *informed* compression is used, like Collapse or tree compression, the compressed value comes to depend on previously inserted state parts, and the *compress* function needs (multiple) accesses to the storage.

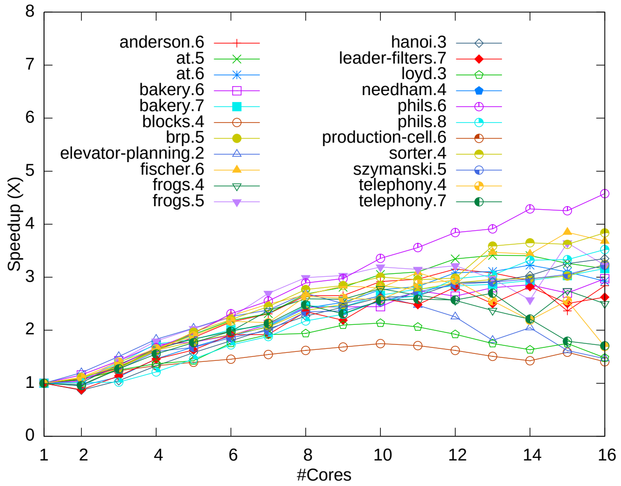


Figure 3.3: Speedups with Collapse.

Global locking or even locking at finer levels of granularity can be devastating for multi-core performance for single hash table lookups (see Chapter 2). Informed compression algorithms, however, need multiple accesses and thus require careful attention when parallelized. Figure 3.3 shows that Spin's Collapse suffers from scalability problems (experimental settings can be found in Section 3.5).

3.3 Tree Database

Section 3.3.1 first describes the original tree compression algorithm from [Blo+08a]. As its implementation has an immediate effect on the obtained compression, we discuss some implementation considerations throughout the section. In Section 3.3.2, we parallelize the structure by merging the multiple hash tables of the tree into a single fixed-size lockless hash table. By simplifying the data structure in this way, we aid scalability.

Furthermore, we prove that it preserves consistency of the database's content. However, as we also show, the new tree will "confuse" tree nodes with leaf nodes and erroneously report some vectors as *found*, while in fact they were not added yet. This is corrected by tagging root tree nodes, completing the parallelization.

Section 3.3.3 shows how tree references can also be used to compact the size of the open set in Algorithm 3.1. Now that the necessary parallelization and space reductions are obtained, the current section is concluded with an algorithm that improves the performance of the tree database by using thread-local incremental information from the reachability search (Section 3.3.4).

3.3.1 Original Sequential Tree Database

The original tree compression algorithm from [Blo+08a] stores the tuples from Figure 3.2 in a *balanced binary tree* of hash tables. Such a tree has $k-1$ tree tables, each of which has a number of siblings of both the left and the right subtree that is equal or off by one. The tree_create function in Algorithm 3.2 generates the Tree structure accordingly, with a TreeTable structure storing *left* and *right* subtrees, a Table *table* and the length of the (sub)tree k. The table_create function allocates a hash table for storing vectors of the length provided by its parameter.

The tree_find_or_put function in Algorithm 3.3 takes as arguments a Tree and a state vector V (both of the same size $k > 1$), and returns a tuple containing a reference to the inserted value and a Boolean indicating whether the value was found (true) or not (false). In the latter case, it is added to the tree as a side effect, just as in the table_find_or_put function described in Chapter 2. The function is recursively called on each half of the state vector (Line 3-4) until the vector length is one. The recursion ends here and a single value of the vector is returned. At Line 5, the returned values of the left (R_l) and right (R_r) subtree are stored as a tuple in the hash table using the table_find_and_put operation. That operation also returns a tuple containing a reference to the value ($\langle R_l, R_r \rangle$) in the hash table and a Boolean indicating whether it was found in the table (true) or had to be added (false).

```
1 type Tree = TreeTable(Tree $left$, Tree $right$, Table $table$, int $k$) | Nil

3 proc Tree tree_create($k$)
4     if ($k$ = 1)
5         return Nil
6     return TreeTable(tree_create($\lceil k/2 \rceil$), tree_create($\lfloor k/2 \rfloor$), table_create(2), $k$)
```

Algorithm 3.2: Tree data structure.

```
1 proc (int, bool) tree_find_or_put(TreeTable($left$,$right$,$table$,$k$), $V$)
2     assert ($|V|$ = $k$)
3     ($R_l$, _) := tree_find_or_put($left$, lhalf($V$))
4     ($R_r$, _) := tree_find_or_put($right$, rhalf($V$))
5     return table_find_or_put($table$, $\langle R_l, R_r \rangle$)

7 proc (int, bool) tree_find_or_put(Nil, $V$)
8     assert ($|V|$ = 1)
9     return ($V[0]$, _)
```

Algorithm 3.3: Tree data structure algorithm for the tree_find_or_put function.

The function lhalf takes a vector V as argument and returns the first half of the vector: $\mathsf{lhalf}(V) = [V_0, \ldots, V_{(\lceil k/2 \rceil - 1)}]$, and symmetrically $\mathsf{rhalf}(V) = [V_{\lceil k/2 \rceil}, \ldots, V_{(k-1)}]$, with $k = |V|$. So, $|\mathsf{lhalf}(V)| = \lceil |V|/2 \rceil$, and $|\mathsf{rhalf}(V)| = \lfloor |V|/2 \rfloor$.

Example 3.1. *Figure 3.4 shows how the vector $\langle 3,5,5,4,1,3 \rangle$ of length 6 is handled by the tree_find_or_put function operating on a tree of length 6 (tree_create (6)). Each recursive call is represented by the vector parameter $V = \langle \ldots \rangle$, and a tuple of 2 squares containing the* return values *of the subsequent recursive call at Line 3–4. The returned values are combined into tuples and stored in the hash table of the respective TreeTable. One such return value is either a slot from the vector (colored gray here), when the corresponding tree (left or right) equals Nil and Line 9 is reached, or a location in* *a hash table when the corresponding tree is a* TreeTable *and Line 5 is reached. In the example, these locations are named a, b, c and d, Their value depends on the hash tables as illustrated in Figure 3.2. Location a for instance points to index 2 in the table of the left-most tree node, storing $\langle 3,5 \rangle$.*

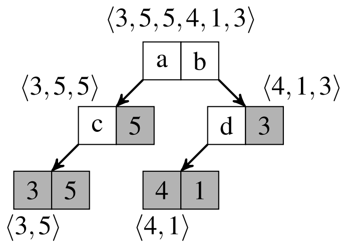


Figure 3.4: Example insertion of a k-sized vector in the tree. Notice the $k-1$ tree nodes.

Implementation considerations. A space-efficient implementation of the hash tables is crucial for good compression ratios. As discussed before, the hash tables in the tree contain free buckets to insert new keys (states). The overhead of these buckets is usually kept low by resizing the hash table when it becomes too full, e.g. at a fill rate of 75%. Resizing is especially necessary in the tree because the different tree tables contain an unpredictable and widely varying number of entries, or tuples (tables may store as few tuples as the maximum number of tuples stored in either of its children, and as many as the product of the number of tuples stored in both its children, as shown in Section 3.4).

However, resizing replaces entries, thereby breaking the *stable indexing* that we used in Example 3.1 (white values in the tuples). These indices cannot simply be modified in the tuples of the parent hash table, because changing the tuple often implies that its hash value changes, thus the tuple needs to be rehashed. In this way, the rehash cascades upwards through the tree, which would be infeasible especially since higher tables can contain quadratically as many entries (see Lemma 3.1).

As a famous saying goes: “all problems in computer science can be solved by another level of indirection” [Spi07]. Indeed, the problem of reifying stable indices was solved by maintaining a second table with references in [Blo+08a]. Figure 3.5 shows how the left-most table in Figure 3.2 would be stored in this resizing table with stable indices. The unoccupied buckets in the table on the left are shown to make clear that

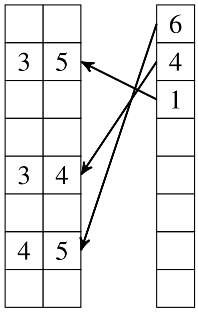


Figure 3.5: Example of a resizing hash table storing tuples with stable indexing.

the buckets are filled without any particular order, as the tuples are hashed to a location. The additional index array on the right is filled in sequential fashion. Upon rehashing of the table, the indices are modified but not replaced in the array, hence it can be used as an index.

David Wheeler, to whom the above quote is attributed, also added the follow-up: “But that usually will create another problem.” Again he is right in this case, as the additional array increases the number of (random) memory references and the storage costs per entry by 50% compared to a non-resizing hash table storing only tuples. While a 50% increase in storage requirements seems modest, this requirement becomes more problematic in the concurrent setting, as we will show in the following section. In Chapter 2, we developed a non-resizing lockless hash table that exhibits excellent scalability because it reduces the *memory footprint* as much as possible. The increased number of memory references in the indexing table of Figure 3.5 retrocedes this benefit.

3.3.2 Concurrent Tree Database

Three conflicting requirements arise when attempting to parallelize Algorithm 3.3:

- Resizing is needed because the load of individual tables is unknown in advance and varies highly.
- Stable indexing is needed, to allow for indexed references to table entries.
- Storing the additional array requires extra memory as explained in the previous section, but it also adds another memory reference. This increases the memory footprint, which in turn reduces scalability, as explained in Chapter 2.

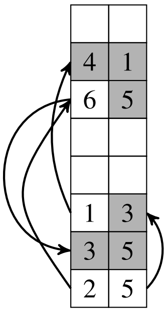


Figure 3.6: Example of the tree structure storing 1 state vector as represented in a merged table. The root of the vector is $\boxed{2\,|\,5}$.

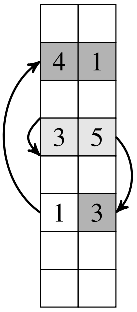


Figure 3.7: The same example with a different hashing function. The (collapsed) root here is $\boxed{3\,|\,5}$.

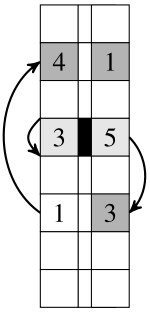


Figure 3.8: Same as Figure 3.7, with an additional root tag bit (middle).

An ideal solution would be to merge all hash tables into a single non-resizable table. This would ensure stable indices without any overhead for administering them, while at the same time allowing the use of a scalable hash table design from Chapter 2. Figure 3.6 shows how the single vector $\langle 3,5,5,4,1,3 \rangle$ is stored in a single table (the hashing function is left implicit, e.g.: the tuple $\langle 6,5 \rangle$ hashes to location 2). The merged-table scheme enables *maximal sharing* of tuples between tree nodes: For example, if $\langle 3,5 \rangle$ would hash to location 3, instead of location 6, then we obtain the tree table shown in Figure 3.7, in which the root, an internal node ($\boxed{6\,|\,5}$ in Figure 3.6) and the leaf all collapsed into that same tuple $\boxed{3\,|\,5}$.

Concerning the correctness of merging the tree tables, we can ask the following questions:

1. *Can all tables safely be merged without corrupting the contents of the database?*
2. *Will the* tree_find_or_put *function return the right Boolean result when the tree tables are merged?*

The maximal sharing as exemplified above already suggests a negative answer to the second question, as the algorithm can no longer distinguish between a tuple that is a root and a tuple that is a leaf (if the leaf tuple is inserted first, as indeed $\boxed{3\,|\,5}$ is, the root tuple $\boxed{3\,|\,5}$ will be seen as already present, hence the corresponding vector will be regarded as *previously inserted*). We will come back to that later; first, we show that the first question can be answered positively.

To argue about the first question, we made a mathematical model of Algorithm 3.3 with one merged hash table. The hash table has stable indices and is concurrent, hence each unique, inserted element will atomically yield one stable, unique index in the table. Therefore, we can describe table_find_or_put as an injective function: $H_2 : \mathbb{N}^2 \to \mathbb{N}$. The tree_find_or_put function can now be expressed as a recurrent relation ($T_k : \mathbb{N}^k \to \mathbb{N}$, for $k > 1$ and $A \in \mathbb{N}^k$):
$T_k(A_0, \ldots, A_{(k-1)}) = H_2(T_{\lceil k/2 \rceil}(A_0, \ldots, A_{(\lceil k/2 \rceil - 1)}), T_{\lfloor k/2 \rfloor}(A_{\lceil k/2 \rceil}, \ldots, A_{(k-1)}))$
$T_1(A_0) = A_0$.
If T provides an injective function (just as H), then the tree with merged tables preserves all inserted vectors.

Theorem 3.1. *For all $k \geq 1$, the function T describing the tree with merged hash tables is injective.*

Proof. To prove (injection): $C \equiv T_k(A) = T_k(B) \implies A = B$, with $A, B \in \mathbb{N}^k$. We use induction over k:

In the base case, $T_1(x) = I(x)$, the identity function satisfies C being injective.

Assume C holds $\forall i < k$ with $k > 1$. We have to prove for all $A, B \in \mathbb{N}^k$, that:
$H_2(T_{\lceil k/2 \rceil}(L(A)), T_{\lfloor k/2 \rfloor}(R(A))) = H_2(T_{\lceil k/2 \rceil}(L(B)), T_{\lfloor k/2 \rfloor}(R(B))) \implies A = B$,
with $L(X) = X_0, \ldots, X_{(\lceil k/2 \rceil - 1)}$ and $R(X) = X_{\lceil k/2 \rceil}, \ldots, X_{(k-1)}$.

Note that:

$$(*) \begin{cases} L(A) = L(B) \land R(A) = R(B)\} \text{ if } A = B \\ L(A) \neq L(B) \lor R(A) \neq R(B)\} \text{ if } A \neq B. \end{cases}$$

Hence,

$$\begin{aligned} & T_k(A) = T_k(B) \\ & \implies H_2(T_{\lceil k/2 \rceil}(L(A)), T_{\lfloor k/2 \rfloor}(R(A))) = H_2(T_{\lceil k/2 \rceil}(L(B)), T_{\lfloor k/2 \rfloor}(R(B))) \\ & \overset{inj.H_2}{\implies} T_{\lceil k/2 \rceil}(L(A)) = T_{\lceil k/2 \rceil}(L(B)) \land T_{\lfloor k/2 \rfloor}(R(A)) = T_{\lfloor k/2 \rfloor}(R(B)) \\ & \overset{ind.hyp.}{\implies} L(A) = L(B) \land R(A) = R(B) \\ & \overset{(*)}{\implies} A = B \end{aligned}$$

Proving that C holds for all A, B and k. □

Now, it follows that an insert of a vector $A \in \mathbb{N}^k$ always yields a unique value for the root of the tree (T_k), thus demonstrating that the contents of the tree database are not corrupted by merging the hash tables of the tree nodes.

However, as we suggested before, Algorithm 3.3 will not always yield the right Boolean answer with merged hash tables. Combining the example of Figure 3.7 with the mathematical model, we have $T_2(3,5) = T_3(3,5) = T_6(3,5) = H_2(3,5)$. Since the leaf is inserted first, the root will find the tuple $\langle 3,5 \rangle$ already inserted.

```
type ConcurrentTree = CTree(Table $table$, int $k$)

proc (int, bool) tree_find_or_put(CTree($table$,$k$), $V$)
    assert ($|V| \leq k \wedge |V| > 0$)
    if ($|V| = 1$)
        return ($V[0]$, _)
    ($R_l$, _) := tree_find_or_put(CTree($table$,$k$), lhalf($V$))
    ($R_r$, _) := tree_find_or_put(CTree($table$,$k$), rhalf($V$))
    ($R$, $B$) := table_find_or_put($table$, $\langle R_l, R_r \rangle$)      ▷ Find/put in tree table
    if ($|V| = k$)                                   ▷ The recursion returned at the root call
        $B$ := if (table_try_tag($R$)) then false else true           ▷ Is the root new?
    return ($R$, $B$)
```

Algorithm 3.4: Concurrent tree data structure algorithm for tree_find_or_put function.

Nonetheless, we can use the fact that T_k is an injection to create a concurrent tree database by adding one bit (a *tag*) to the merged hash table. The tag bit can be seen as having the two values, *non_root* and *is_also_root*, and is always initiated as *non_root*. Algorithm 3.4 defines a new ConcurrentTree structure, only containing a single hash table *table* and the length of the vectors k. Once the recursive calls return at the root node, the table_try_tag function now atomically tries to set the tag on the entry (the tuple) pointed to by R to *is_also_root* in *table*. To this end, it can employ the atomic hardware instruction *compare-and-swap* (CAS) (see Chapter 2). If the function fails to set the tag to *is_also_root* (because the tag was already set to that value), we return true (indicating the state vector was *found* in the tree), and else we return false (indicating that it was not found and has been inserted).

Although we no longer maintain explicit *tree nodes* in Algorithm 3.4, the recursion still follows the same pattern of doing $k-1$ table lookups as explained in Example 3.1. *In the following, we will keep referring to the steps in the recursion as tree nodes, even though they are now virtual nodes.*

Implementation considerations. Crucial for efficient concurrency is *memory layout*. While a bit array or sparse bit vector may be used to implement the tags (using R as index), its parallelization is hardly efficient for high-throughput applications like reachability analysis. Each modified bit will cause an entire cache line (with typically thousands of other bits) to become *dirty*, causing other CPUs accessing the same memory region to be forced to update the line from main memory. The latter operation is multiple orders of magnitude more expensive than normal operations, and also more costly than a simple uncached load. Therefore, we merge the bit array/vector into the hash table *table* as shown in Figure 3.8, for this increases the spatial locality of node accesses with a factor proportional to the width of tree nodes. The small column in the middle represents the bit array with white entries indicating *non_root* and black entries indicating *is_also_root*. Tuple and bit are bit-crammed into a word (e.g. Section 9.6.3).

Furthermore, to implement *table*, we used the lockless hash table presented in Chapter 2, benefiting from its cache-efficient probing behavior. This table normally uses *memoized hashes* in order to speedup probing over larger keys. Since the stored tree nodes can be relatively small (64 or 128 bits), we dropped the memoized hashes. In fact, the buckets in the tree table are so small that we were even able to remove the locking bit from the hash table, because the atomic CAS can operate on entire buckets. The appropriate size of the buckets in the tree table is discussed in Section 3.4.

3.3.3 References in the Open Set

Now that tree compression reduces the space required for state storage, we observe that the open sets of the parallel reachability algorithm can become a memory bottleneck [LPW11a]. A solution is to store references to the root tree node in the open set as illustrated by Algorithm 3.5, which is a modification of Line 4-7 from Algorithm 3.1.

```
while ($ref$ := $S_{id}$.get())
    $state$ := tree_get($DB$, $ref$)
    for ($succ \in$ NEXT-STATE($state$))
        ($newref$, $found$) := tree_find_or_put($DB$, $succ$)
        if ($\neg found$)
            $S_{id}$.put($newref$)
```

Algorithm 3.5: Reachability analysis algorithm with references in the open set.

The tree_get function is shown in Algorithm 3.6. It reconstructs the vector from a reference. References are looked up in *table* using the table_get function, which returns the tuple stored in the table. The algorithm recursively calls itself until $k = 1$; at this point *ref_or_val* is known to be a slot *val*ue – it is not a *ref*erence – and is returned as vector of size 1. Results then propagate back up the tree and are concatenated on Line 7, until the full vector of length k is restored at the root of the tree.

```
proc int[] tree_get(CTree(table,k), val_or_ref)
    if (k = 1)
        return [val_or_ref]
    [R_l, R_r] := table_get(table, val_or_ref)
    V_l := tree_get(CTree(table,⌈k/2⌉), R_l)
    V_r := tree_get(CTree(table,⌊k/2⌋), R_r)
    return concat(V_l, V_r)
```

Algorithm 3.6: Algorithm for tree vector retrieval from a reference

3.3.4 Incremental Tree Database

New states are generated by calling the NEXT-STATE function on found states (starting from the initial states). Often states are very similar due to locality in the model. In the example below, only one slot value has changed, which is not uncommon:

$$\text{NEXT-STATE}(\langle 3,5,5,4,1,3\rangle) = \{\langle 3,5,\mathbf{9},4,1,3\rangle\}$$

The time complexity of the tree compression algorithm, measured in the number of hash table accesses, is linear in the number of state slots ($k-1$ lookups are performed: one at each tree node). However, because of today's steep memory hierarchies these random memory accesses are expensive. Luckily, the same principle that tree compression exploits to deliver good state compression, can also be used to speedup the algorithm: The only tuples that need to be inserted into the tree table are the ancestors of leaves corresponding to slots that actually changed with respect to the previous state. For a state vector of size k, the number of table accesses can be brought down from $k-1$ (the

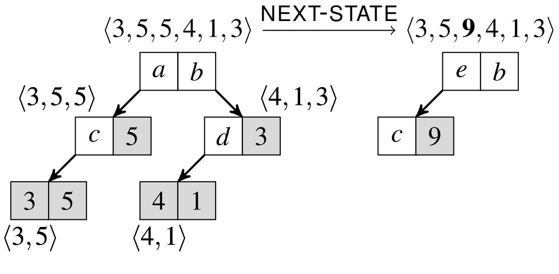


Figure 3.9: Incremental insertion of a vector in the tree database.

total number of nodes in a tree) to $c \times \log_2(k)$ (the height of the tree) assuming $c \leq k$ slots changed. But likely fewer than $c \times \log_2(k)$, because the changed slots can be close to each other in the tree (due to shared paths to the root).

Hence our goal is here to find/put a vector V in the tree database, while reusing the computations used to find/put its predecessor P, i.e. $V \in \textsf{NEXT-STATE}(P)$, as much as possible. So to avoid looking up P again, we need to store all the computed references at the different (virtual) tree nodes. In other words, we must not only keep P in the open set of Algorithm 3.1, but the entire tree associated with it, as shown in Example 3.1. Figure 3.9 presents an incremental tree update for the example considered above.

Algorithm 3.7 presents a reference tree structure that can be used for this purpose: the RefTree. The associated ref_tree_create function creates an empty reference tree with only $\perp$ values in the tree nodes and leaves. The $\perp$ value is meant to be different from the slot values in a vector V, so that the incremental procedure can be initiated (the initial states in Algorithm 3.1 have no predecessors). *While we use the large RefTree structure with additional pointers and length attributes to clarify the algorithms, it is hardly needed to store the full structure in the open set.* In fact, a short vector that concatenates the $k-1$ tuples in the tree suffices to represent the whole RefTree (assuming the fixed length k is known).

```
type RefTree = RTree(RefTree $left$, RefTree $right$, int $k$, int $ref$)|Leaf(int v,bool c)

proc RefTree ref_tree_create($k$)
    if ($k$ = 1)
        return Leaf($\perp$)
    $left$ := ref_tree_create($\lceil k/2 \rceil$)
    $right$ := ref_tree_create($\lfloor k/2 \rfloor$)
    return RTree($left$, $right$, $k$, $\perp$)
```

Algorithm 3.7: RefTree structure to store all tree.

Algorithm 3.8 presents the incremental variant of the tree_find_or_put function. The callee has to supply as additional argument an reference tree of the predecessor P of V ($V \in \textsf{NEXT-STATE}(P)$). Or an empty reference tree created with ref_tree_create, if $V \in$ initial_states. Instead of returning a tuple with a reference (and a Boolean), tree_find_or_put now returns a tuple with a complete reference tree for V, which can in turn be used for incrementally storing any direct successor of V. The Boolean in the returned tuple now indicates whether the part of the vector V that is associated with the current subtree is equivalent to that same part in the predecessor vector P (see Line 3). This Boolean is used on Line 10 as a condition for the hash table access; if the left or

```
1  proc (RefTree, bool) tree_find_or_put(CTree($table$,$k$), $V$, Leaf(val))
2      assert ($|V| = 1$)
3      return (Leaf($V[0]$, $V[0] = val$))

5  proc (RefTree, bool) tree_find_or_put(CTree($table$,$k$), $V$, RTree($left$,$right$,$l$,$ref$))
6      assert ($|V| = l$)
7      assert ($l \leq k \wedge l > 0$)
8      ($R_l$, $B_l$) := tree_find_or_put(CTree($table$,$k$), lhalf($V$), $left$)
9      ($R_r$, $B_r$) := tree_find_or_put(CTree($table$,$k$), rhalf($V$), $right$)
10     if ($B_l \wedge B_r$)                          ▷ Are all descendant leaves unmodified (see Line 3)?
11         return (RTree($R_l$,$R_r$,$l$,$ref$), true)            ▷ Return reference from predecessor
12     else                                                      ▷ If a descendant leaf is modified:
13         ($R$, _) := table_find_or_put($table$, $\langle R_l, R_r \rangle$)          ▷ Find/put tuple in table
14         if ($|V| = k$)                                ▷ The recursion returned at the root call
15             return (RTree($R_l$,$R_r$,$l$,$R$), ¬table_try_tag($R$))              ▷ Is the root new?
16         else return (RTree($R_l$,$R_r$,$l$,$R$), false)           ▷ Return new reference and false
```

Algorithm 3.8: ReferenceTree structure and incremental tree_rec function.

the right subvectors are modified, then the returned reference tree is updated with a new reference that is looked up in *table* at Line 13.

Notice that at the root node, when $k = |V|$, the algorithm returns a different Boolean, indicating whether the tree root is new or not, as deduced from the tag bit discussed above. This difference stems from the fact that the initial callee is the reachability algorithm, which is interested in knowing whether V was already in the database, whereas the recursive calls require information on similarities between V and P.

The incremental tree_find_or_put function changed its interface with respect to Algorithm 3.4. Algorithm 3.9 presents a new search algorithm (Line 4-7 in Algorithm 3.1) that also records the reference tree in the open set. RefTree *refs* has become an input of the tree database, which returns a new RefTree *new_refs* to be stored along *next* in the open set. For simplicity, we store both the vector and the reference tree in the open set, while in fact the first can be reconstructed from the latter. (In fact, when storing the reference tree succinctly in an array, as discussed above, the vector can be read directly from this array.) We measured the speedup of the incremental algorithm compared to the original (for the experimental setup see Section 3.5). Figure 3.10 shows that the speedup is linearly dependent on $\log_2(k)$, as expected.

Because the internal tree node references are stored, Algorithm 3.9 increases the size of the open set by a factor of almost two. This is not of major concern, as the open

```
while ((prev, refs) := $S_{id}$.get())
  for (next ∈ NEXT-STATE(prev))
    (new_refs, found) := tree_find_or_put(DB, next, refs)
    if (¬found)
      $S_{id}$.put((next, new_refs))
```

Algorithm 3.9: Reachability analysis algorithm with incremental tree database.

sets could further be reduced by storing and retrieving it piecemeal to disk [HW07]. For breadth-first search, the access patterns in the open set are regular enough to do this without overhead. To still support other search orders, we can either modify the tree_get function in Algorithm 3.6 to also return the reference trees, or the tree_get function can be integrated into the incremental algorithm (Algorithm 3.8). (We do not present these algorithm algorithms here as they are easy to derive from the above algorithms.) While it seems that the additional lookups required to reconstruct reference trees for predecessor vectors mitigate the benefits of the incremental method, it turns out that often the tuples for the predecessor are still cached: We measured little slowdown when doing this reconstruction before generating all successors of a state at once (about

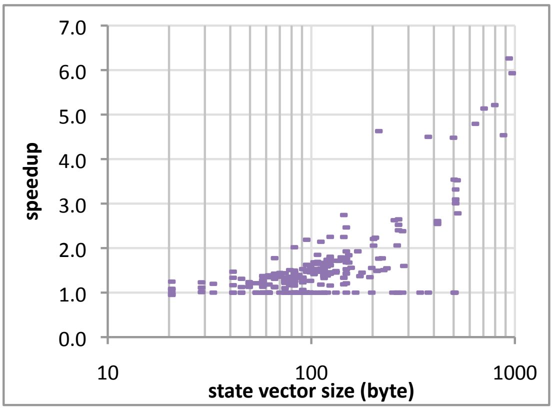


Figure 3.10: Speedup of Algorithm 3.8 wrt. Algorithm 3.4.

10% across a wide spectrum of input models).[3.1] Alternatively, as a trade-off, we could decide to lower the cutoff point in the tree, and store multiple references occurring at a certain tree depth $x < \log_2 k$ in the open set. This latter approach was used in [Blo+08a].

3.4 Analysis of Compression Ratios

In the current section, we establish the worst-case and the best-case compression ratio for hash tables, the tree database and the process table (Collapse compression). We make the following reasonable assumptions about their dimensions:

- The respective database stores a set S of $n = |S|$ state vectors of k slots.[3.2]
- The size of a reference, or index, in the table is w bits.
- The size of the slot is u bits, with $u \leq w$, so we can store a slot in the same space as a reference in the table.[3.3] (Hence, the state-vector universe is: 2^{uk}.)
- The number of processes in Collapse compression is $1 < p \leq k$.
- Keys can be stored without overhead in tables.[3.4]
- We only count occupied table buckets, omitting the space reserved for empty buckets.[3.5]
- k is a power of 2.[3.6]

As explained in the footnotes, most of these assumptions are introduced to simplify the model, while others reflect the requirements imposed by implementation details, such as the way that references are implemented.

[3.1] For this reason, tree compression with references on the stack has become the default state storage method in the LTSmin model checker [LPW11a; BPW10].

[3.2] The fixed length of state vectors does not prevent us from model checking more dynamic systems, as we can safely over-approximate this length. The good compressions and performance of incremental tree compression ensure that the overhead does not matter much, as Section 3.5 demonstrates.

[3.3] As noted above, complex slots values can easily be hashed in separate tables to obtain unique values.

[3.4] This assumption holds for tree tables proposed here, as we explain at the end of the current section, but is less realistic for hash tables storing large state vectors as explained in Chapter 2. Since we compare compressions with respect to hash table storage, this is a safe approximation.

[3.5] Especially in the case of model checking, this results in a reasonable indication of the compressed sizes, because the size of the checked system depends by and large on the number of entries we can store in the closed set of the reachability algorithm. In other words, our goal is to squeeze as many states as possible in the available memory.

[3.6] Solely assumed to simplify the formulae below.

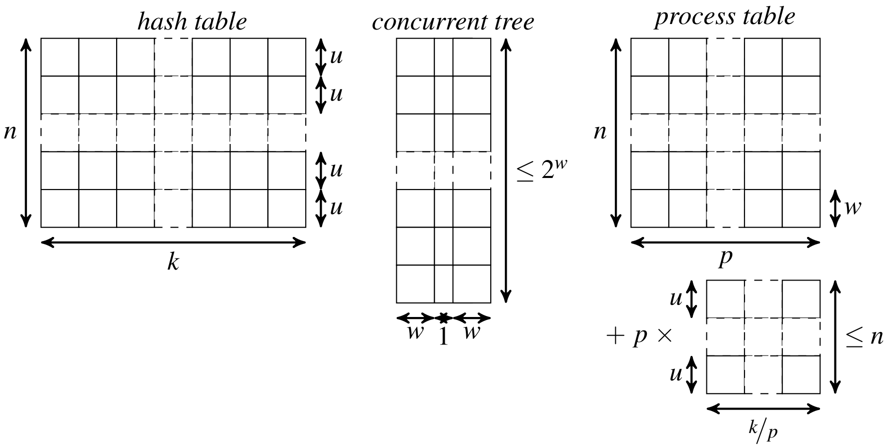


Figure 3.11: From left to right: a hash table, a (concurrent) tree table and a process table with their dimensions.

Figure 3.11 provides an overview of the different data structures and the stated assumptions about their dimensions. Note that buckets are always drawn vertically. The original sequential tree database is not drawn, but can be considered as set of $k-1$ tables which each require 3 w-bit units per bucket as shown in Figure 3.5.

3.4.1 Tree Database

To deduce the average compressed state vector size, we will reason on the number of *tuple entries* stored in a tree database (sequential or concurrent), containing the n state vectors. For simplicity, we disregard the maximal sharing property that was discussed in the previous section. This over-approximates the memory usage of the concurrent tree and thus is a fair assumption. This simplifies the case of the concurrent tree, since its merged table will contain equally many entries as are stored across all tables in the original sequential tree. At the end of the current section, we discuss the possible effects of maximal sharing on the worst- and best-case compression. The number tuple entries stored in the tree database depends on: n, k, and the combinatorial structure of S. The latter is fixed to arrive at the following theorems:

Theorem 3.2. *In the worst case, the tree database requires $k-1$ tuple entries per state vector regardless of the number of vectors [Blo+08a].*

Proof. Consider the case where all states $s \in S$ have k identical slot values: $S = \{\langle v, \ldots \times k \ldots, v\rangle \mid v \in \{1, \ldots, n\}\}$. No sharing can occur between the state vectors in the database, so for each state we store $k-1$ tuples at the tree nodes. □

Corollary 3.1. *In the worst case, the concurrent tree database in Algorithm 3.4 requires less than* $2k(w+1)$ *bits per state vector regardless of the number of vectors in the database.*

Proof. According to Theorem 3.2, the tree table contains $k-1$ tuple entries per state. These entries each require $2w+1$ bits. And: $(k-1) \times (2w+1) = 2kw + k - 2w - 1 < 2kw + 2k = 2k(w+1)$. □

Corollary 3.2. *In the worst case, the sequential tree database in Algorithm 3.3 requires* $3kw$ *bits per state vector regardless of the number of vectors in the database.*

Proof. According to Theorem 3.2, the tree table contains $k-1$ tuple entries per state. These entries each require $3w$ bits. □

The best-case scenario is easy to comprehend from the effects of a good combinatorial structure on the size of the parent tables in the (sequential) tree. If a certain tree table contains d tuple entries, and its sibling contains e entries, than the parent can have up to $d \times e$ entries (all *combinations*, i.e. the Cartesian product). In a tree that is perfectly balanced ($d = e$ for all sibling tables), then the root node has n entries (1 per state), its children have $\sqrt{n}$ entries, its children's children $\sqrt[4]{n}$, etc. Figure 3.12 depicts this scenario.

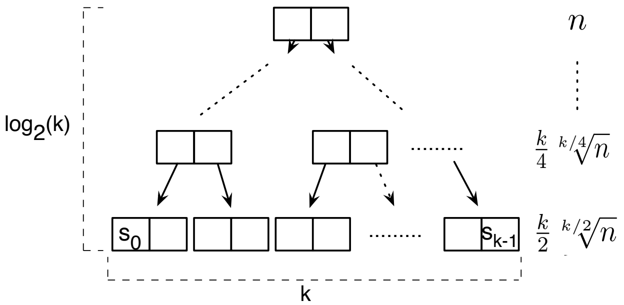


Figure 3.12: Optimal entries per tree node level.

Hence there are a total of $n+2\sqrt{n}+4\sqrt[4]{n}+\ldots(\log_2(k)\text{ times})\ldots+{}^{2}\!/\!{}_{k}\,\sqrt[2/k]{n}$ tuple entries. Dividing this series by n gives a series for the expected number of tuple entries per state: $\sum_{i=0}^{\log_2(k)-1} 2^i \frac{\sqrt[2^i]{n}}{n}$. It hard to see where this series exactly converges, but [Blo+08a] established however a useful upper bound.

Theorem 3.3. *In the best case, the tree database requires a total of* $n+\sqrt{n}(k-2)$ *tuple entries to store n vectors [Blo+08a].*

Proof. In the best case, the root tree table contains n entries and its children contain $\sqrt{n}$ entries. The entries in the root's children represent vectors of size $k/2$. To obtain an upper bound, we assume that the roots children store their $\sqrt{n}$ vectors of size $k/2$ as in the worst case. For one child, according to Theorem 3.2, this requires $k/2-1$ entries per state in all of the child's descendants (including entries in the child's own table). The total number of entries in all tree tables, hence is bounded by: $n+2\sqrt{n}(k/2-1)=n+\sqrt{n}(k-2)$. □

We simplify this upper bound below to obtain a better intuition on the expected experimental results. But first, we provide a lemma that can be used as a guideline for design decisions concerning the implementation of the tree database.

Lemma 3.1. *In the best case, the total number of tuple entries l in all descendants of root table is negligible (*$l \ll n$*), assuming a relatively large number of vectors is stored:* $n \gg k^2 \gg 1$.[3.7]

Proof. If the number of tuple entries in a tree table equals n, then the total number of entries in all its descendants is bounded by $l=\sqrt{n}(k-2)$ (see Theorem 3.3). Replacing k with $\sqrt{n}$, we learn that $l \ll n$. □

This shows that the entries in the root table (or the entries with *is_also_root* tag in the concurrent tree), dominate.

Corollary 3.3. *In the best case, the tree databases requires* $1+\varepsilon$ *tuple entries per state to store n vectors, assuming a relatively large number of vectors is stored:* $n \gg k^2 \gg 1$.[3.7].
(The concurrent tree uses $2w+1$ *bits per entry and the sequential tree* $3w$ *bits per entry.)*

Proof. Follows from Lemma 3.1. □

[3.7] The universe of all possible vectors is 2^{ku} large (vectors are ku bits long). We are interested in storing a very small subset of this universe. In practice, we encounter around 10^9 states with often no more than a few hundreds of slots.

3.4.2 Collapse Process Table

Since the leaves of the process table are directly connected to the root, the compression ratios are easier to calculate. To yield optimal compression for the process table, a more restrictive scenario, than described for the tree above, needs to occur. We require p symmetrical processes with each a local vector of $m = {}^{k}/_{p}$. Related slots may only lay within the bounds of these processes, take $S_i = \{\langle v, \ldots, v\rangle \mid v \in \{1, \ldots, m\}\}$ for $i \in \{1, \ldots, p\}$. Each combination of different local vectors is inserted in the root table (also if $S_i = \{\langle s, 1, \ldots, 1\rangle \mid s \in \{1, \ldots, m\}\}$).

Theorem 3.4. *In the best case, the process table requires a total of $pw + \varepsilon$ bits to store n vectors.*

Proof. In the best case, the root tree table contains n entries and its children contain $\sqrt[p]{n}$ entries. The former requires pw bits per entry, and the latter ${}^{k}/_{pu}$ bits per entry. The total size of the process table becomes $npw + \sqrt[p]{n}\,{}^{k}/_{pu}$ bits. For relatively large $n \gg k^2 \gg 1$ and large $p \gg 1$, this approaches $pw + \varepsilon$ bits per state. □

In the worst case, we can take the same scenario as for the tree.

Theorem 3.5. *In the worst case, the process table requires a total of $pw + ku$ bits per state regardless of the number of vectors it stores.*

Proof. No sharing occurs in the process table if all states $s \in S$ have k identical slot values: $S = \{\langle v, \cdots \times k \ldots, v\rangle \mid v \in \{1, \ldots, n\}\}$. Therefore, the n pw-sized entries are required in the root table, and n ${}^{k}/_{pu}$-sized entries in all p process tables. □

3.4.3 Comparison Against Plain Hash Table Storage

Table 3.1 lists the achieved compressed sizes for states, as stored in a normal hash table, a process table and a tree database. In the hash table, the size of slots might be small ($u \leq w$), which can be used to somewhat condense the states, hence the factor on k. The same holds for the process tables in Collapse compression. Nonetheless, both techniques use memory proportional to the state vector size or the number of processes in the model. For the rest, these results follow directly from the presented theorems.

The worst case of the process table is clearly not as bad as the worst case achieved by the tree. On the other hand, the best-case scenario is not as good as that from the tree, which is the only technique that has the potential compress states to a constant number of references. We also saw that the tree can reach near-optimal cases easily, placing few constraints on related slots (on the same half). Therefore, we can expect the tree to outperform the compression of process table in more cases, because the latter requires

Table 3.1: Theoretical compression bounds for the (sequential and concurrent) tree database, and the process table, compared to plain hash table storage (showing compression as the number of w-bit units used per state).

Structure	Worst case	Best case
Hash table (ideal)	$\frac{u}{w}k$	$\frac{u}{w}k$
Process table (Collapse)	$p+\frac{u}{w}k$	$p+\varepsilon$
Sequential tree database (Algorithm 3.3)	$3k-3$	$3+\varepsilon$
Concurrent tree database (Algorithm 3.4)	$2k-2$	$2+\varepsilon$

more restrictive conditions. Namely, related slots can only be within the fixed bounds of the state vector (local to one process).

Maximal sharing invalidates the worst-case analysis for the concurrent tree database, but other sets of vectors can be thought up to still cause the same worst-case size. In practice, we can also expect little gain from maximal sharing, since the likelihood of similar subvectors decreases rapidly the larger these vectors are. For combinatorial S, we can expect little gain from maximal sharing, since the number of tuples entries in descendant tables are insignificant anyway compared to the root tuples (see Lemma 3.1).

3.4.4 Implementation Details

For the implementation of the concurrent tree database, the following requirements play a role in determining the reference or index size w:

- as illustrated in Figure 3.11, the number of tuples that the tree table can fit is bound by 2^w (as we are using w-bit sized references),
- the parallel algorithm uses 1 extra *tag* bit per tuple entry,
- the atomic CAS instruction required to implement Algorithm 3.4 only works on 8, 16, 32 or 64 bit words (64-bit processors often provide an additional 128-bit word CAS),
- the CAS operands need to be aligned at word-sized boundaries in memory, eliminating the possibility to use bucket sizes that are not a power of two, and
- the tree needs to accommodate as many vectors as possible in the available main memory of modern systems (up and around 64GB).

Table 3.2: The compressed state sizes, max number of states storable and max usable memory for the tree tables, for different w and the best-case compression scenario.

	Best case		Max. states		Usable memory (max)	
	$w = 32$	$w = 64$	$w = 32$	$w = 64$	$w = 32$	$w = 64$
Sequential tree database	> 12 byte	> 24 byte	$\gg 2^{32}$	$\gg 2^{64}$	$\gg 48$GB	$\gg (48\text{GB})^2$
Concurrent tree database	> 8 byte	> 16 byte	$< 2^{31}$	$< 2^{62}$	16GB	$(16\text{GB})^2$

The considerations lead to the logical conclusion that the only feasible reference sizes are $w = 32$ bits and $w = 64$ bits. Table 3.2 summarizes the effects of these choices on both the concurrent and the sequential tree table. From the second columns we see that the concurrent tree can compress states to almost 8 byte, but in that case can only store less than 2^{31} states filling up 16GB of memory, only a part of the available memory on modern machines. Lemma 3.1 ensures that the number non-root tuple entries do not have to be a serious limitation to the number of states stored in the tree. It cannot even store 2^{32} states, because the table needs to accommodate the tag bit (see Figure 3.11). Using 64-bit references these limitations are remedied, but the compressed sizes now surpass the memory use of the sequential database with 32-bit references. The latter also shows the surprising result that it can store more than 2^{32} states. This follows from the fact that the root table is separate and can grow beyond 2^{32} entries because it is not referenced inside any tree table (only on the local open sets in the reachability algorithm, where we can use e.g. 5 byte references).

It would be tempting to also separate at least the root table in the concurrent tree database. However, since we do not know up front how well the compression ratio will be, or in other words whether Lemma 3.1 applies, resizing would be required. In the previous section, we avoided resizing in the concurrent tree as a design decision with the aim of improving scalability. Now that we have merged tables, resizing might be possible again by using the tree_ref function to reconstruct states and reinsert them in a larger table (rehashing the tuples). While indeed technically possible, it still seems infeasible to implement because maximal sharing prevents incremental rehashing of reference trees, as it might be the case that the rehashed tuples are no longer shared. *For these reasons, we decided to implement a concurrent tree with 31-bit references (tree table buckets of 64 bits of which* 2×31 *bits are used for references and 1 bit is used for the tag).* This way, we already can accommodate many more (large) states than in a plain hash table, and use the remaining memory for in-memory open sets and worst-case compression scenarios.

3.5 Experiments

We performed experiments on an AMD Opteron 8356 16-core (4 × 4 cores) server with 64 GB RAM, running a patched Linux 2.6.32 kernel.[3.8] All tools were compiled using GCC 4.4.3 in 64-bit mode with high compiler optimizations (`-O3`).

We measured compression ratios and performance characteristics for the models of the BEEM database [Pel07] with three tools: DIVINE 2.2, SPIN 5.2.5 and our own model checker LTSMIN [BPW10; LPW11a]. LTSMIN implements Algorithm 3.4 using a specialized version of the hash table from Chapter 2 which inlines the *tags* as discussed at the end of Section 3.3.2. Special care was taken to keep all parameters across the different model checkers the same. The size of the hash/node tables was fixed at 2^{28} elements to prevent resizing and model compilation options were optimized on a per tool basis as described in Chapter 2. We verified state and transition counts with the BEEM database and DIVINE 2.2. The complete results with over 1500 benchmarks are available online [Laa11].

3.5.1 Compression Ratios

For a fair comparison of compression ratios between SPIN and LTSMIN, we must take into account the differences between the tools. The BEEM models have been written in DVE format (DIVINE) and translated to PROMELA. The translated BEEM models that SPIN uses may have a different state vector length. LTSMIN reads DVE inputs directly, but uses a standardized internal state representation with one 32-bit integer per *state slot* (state variable) even if a state variable could be represented by a single byte. Such an approach was chosen in order to reuse the model checking algorithms for other model inputs (like mCRL, mCRL2 and DIVINE [BPW09]). Thus, LTSMIN can load BEEM models directly, but blows up the state vector by an average factor of three. Therefore, we compare the average compressed state vector size instead of compression ratios.

Table 3.3 shows the uncompressed and compressed vector sizes for COLLAPSE and tree compression. Tree compression achieves better and almost constant state compression than COLLAPSE for these selected models, even though original state vectors are larger in most cases. This confirms the results of our analysis.

We also measured peak memory usage for full state-space exploration. The benefits with respect to hash tables can be staggering for both COLLAPSE and tree compression: while the hash table column is in the order of gigabytes, the compressed sizes are in the order of hundreds of megabytes. An extreme case is `hanoi.3`, where tree compres-

[3.8] `https://bugzilla.kernel.org/show_bug.cgi?id=15618`, see also Chapter 2

[3.9] The hash table size is calculated on the base of the LTSMIN state sizes

Table 3.3: Original and compressed state sizes and memory usage for LTSmin with hash table (*Table*), Collapse (*Spin*) and tree compression (*Tree*) for a representative selection of all benchmarks.

Model	Orig. State [Byte]		Compr. State [Byte]		Memory [MB]		
	Spin	Tree	Spin	Tree	Table[3.9]	Collapse	Tree
`at.6`	68	56	36.9	8.0	8,576	4,756	1,227
`firewire_tree.5`	68	56	36.9	8.0	6,550	–	94
`iprotocol.6`	164	148	39.8	8.1	5,842	2,511	322
`at.5`	68	56	37.1	8.0	1,709	1,136	245
`bakery.7`	48	80	27.4	8.8	2,216	721	245
`hanoi.3`	116	228	112.1	13.8	3,120	1,533	188
`telephony.7`	64	96	31.1	8.1	2,011	652	170
`anderson.6`	68	76	31.7	8.1	1,329	552	140
`frogs.4`	68	120	73.2	8.2	1,996	1,219	136
`phils.6`	140	120	58.5	9.3	1,642	780	127
`sorter.4`	88	104	39.7	8.3	1,308	501	105
`elev_plan.2`	52	140	67.1	9.2	1,526	732	100
`telephony.4`	54	80	28.7	8.1	938	350	95
`fischer.6`	92	72	43.7	8.4	571	348	66

sion, although not optimal, is still an order of magnitude better than Collapse using only 188 MB compared to 1.5 GB with Collapse and 3 GB with the hash table.

To analyze the influence of the model on the compression ratio, we plotted the inverse of the compression ratio against the state length in Figure 3.13. The line representing optimal compression is derived from the analysis in Section 3.4 and is linearly dependent on the state size (the average compressed state size is close to 8 bytes: two 32-bit integers for the dominating root node entries in the tree).

With tree compression, a total of 279 Beem models could each be fully explored using a tree database of pre-configured size, never occupying more than 4 GB memory. Most models exhibit compression ratios close to optimal; the line representing the median compression ratio is merely 17% below the optimal line. The worst cases, with a ratio of three times the optimal, are likely the result of combinatorial growth concentrated around the center of the tree, resulting in equally sized root, left and right sibling tree nodes. Nevertheless, most sub-optimal cases lie within 200% of the optimal, suggesting only one “full” sibling of the root node. (We verified this.)

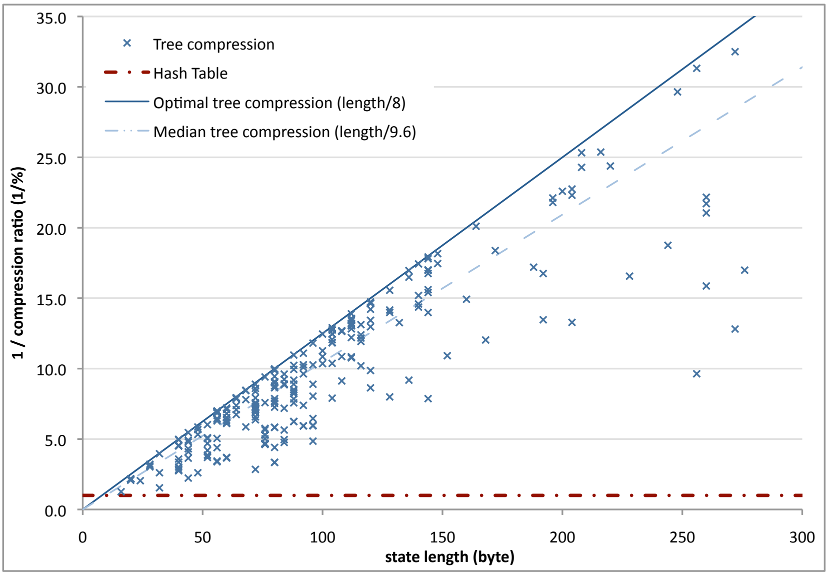
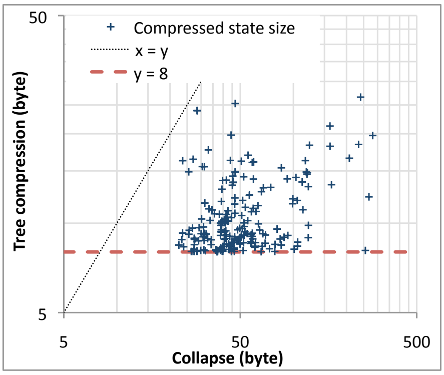


Figure 3.13: Compression ratios for 279 models of the Beem database are close to optimal for tree compression.



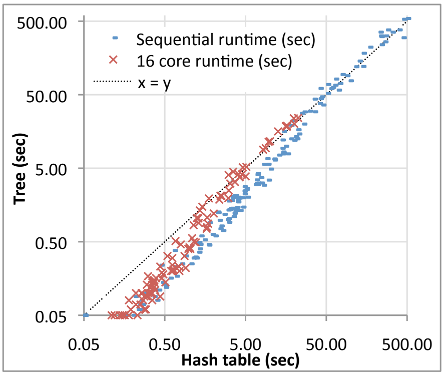


Figure 3.14: Log-log scatter plot of tree-compressed state sizes (smaller is better): for all tested models, tree compression uses less memory.

Figure 3.15: Log-log scatter plot of LTSmin runtimes for state-space exploration with either a hash table or tree compression.

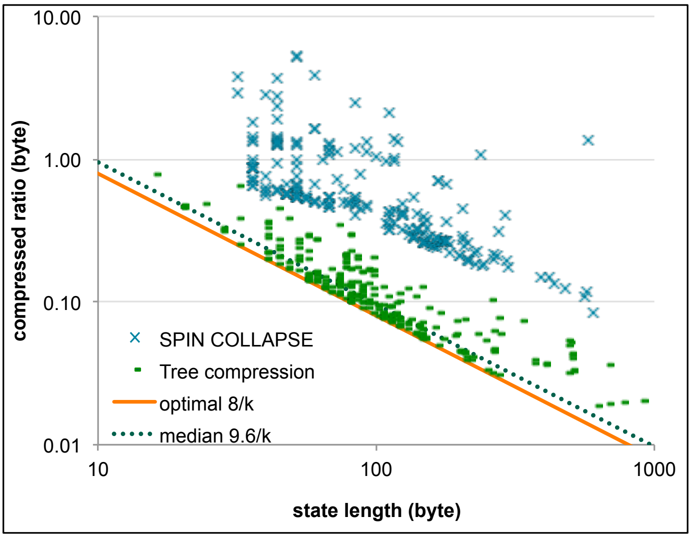


Figure 3.16: Compression ratios for 279 models of the Beem database are close to optimal for tree compression.

Figure 3.14 shows compressed state size of tree compression. The figure shows more clearly how many cases come close to the optimal compression. We also see that cases with bad compression are distributed evenly over the state length axis.

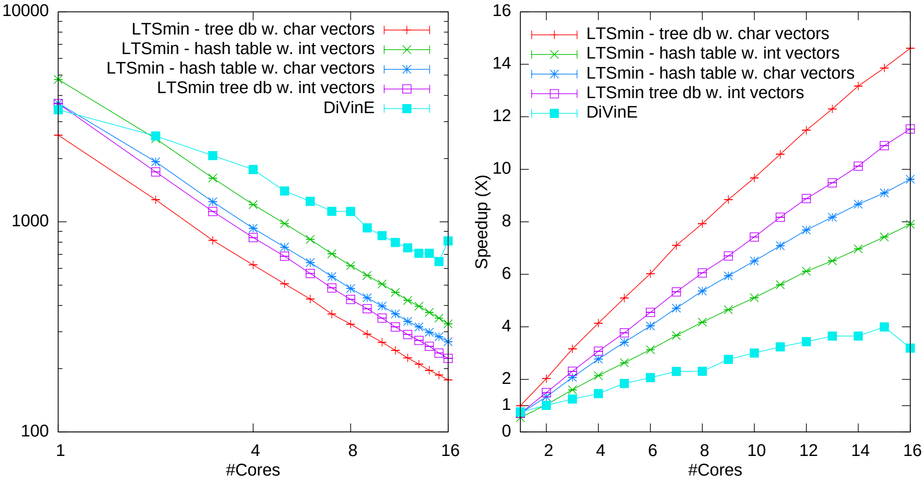
Figure 3.16 compares the compressed sizes of Collapse and tree compression. (We could not easily compare compressed state *space* sizes due to differing number of states for some models). Tree compression performs better for all models in our data set. In many cases, the difference is an order of magnitude. While tree compression has an optimal compression ratio that is four times better than Collapse's (empirically established), the median is even five times better for the models of the Beem database. Finally, as expected (see Section 3.4), we measured insignificant gains from the introduced maximal sharing.

3.5.2 Performance & Scalability

We compared the performance of the tree database with a hash table in DiVinE and LTSmin. A comparison with Spin was already provided in Chapter 2. For a fair comparison, we modified a version of LTSMin[3.10] to use the (three times) shorter state vectors (*char vectors*) of DiVinE directly. Figure 3.17 shows the total runtime of 158 Beem models, which fitted in machine memory using both DiVinE and LTSMin. On average the runtime performance of tree compression is close to a hash table-based search (see Figure 3.17(a)). However, the absolute speedup in Figure 3.17(b) shows that scalability is better with tree compression, due to a lower memory footprint.

Figure 3.15 compares the sequential and multi-core performance of the fastest hash table implementation (LTSMin lockless hash table with char vectors) with the tree database (also with char vectors). The tree matches the performance of the hash table closely.

For both, sequential and multi-core, the performance of the tree database is nearly the same as the fastest hash table implementation, however, with significantly lower memory utilization. For models with fewer states, the tree database outperforms the hash table, undoubtedly due to better cache utilization and lower memory bandwidth.



(a) Total runtime

(b) Average (absolute) speedup

Figure 3.17: Performance benchmarks for 158 models with DiVinE (hash table) and with LTSMin using tree compression and hash table.

[3.10] this experimental version is distributed separately from LTSMin, because it breaks the language-independent interface.

3.6 Conclusions

First, the current chapter presented an analysis and experimental evaluation of the compression ratios of tree compression and Collapse compression, both informed compression techniques that are applicable in on-the-fly model checking. Both analysis and experiments can be considered an implementation-independent comparison of the two techniques. Collapse compression was considered the state-of-the-art compression technique for enumerative model checking. Tree compression was not evaluated as such before. The latter is shown here to perform better than the former, both analytically and in practice. In particular, the median compression ratio of tree compression is five times better than that of Collapse on the Beem benchmark set. We consider this result representative to real-world usage, due to the varied nature of the Beem models: the set includes models drawn from extensive case studies on protocols and control systems, and, implementations of planning, scheduling and mutual exclusion algorithms [Pel11].

Furthermore, we presented a solution for parallel tree compression by merging all tree-node tables into a single large table, thereby realizing maximal sharing between entries in these tables. This single hash table design even saves 50% in memory because it exhibits the required stable indexing without any bookkeeping. We proved that the consistency is maintained and use only one bit per entry to parallelize tree insertions. Lastly, we presented an incremental tree compression algorithm that requires a fraction of the table accesses (typically $\mathcal{O}(\log_2(k))$, i.e., logarithmic in the length of a state vector), compared to the original algorithm.

Our experiments show that the incremental and parallel tree database has the same performance as the hash table solutions in both LTSmin and DiVinE (and by implication Spin, as Chapter 11 confirms). Scalability is also better. All in all, the tree database provides a win-win situation for parallel reachability problems.

Discussion. The absence of resizing could be considered a limitation in certain applications of the tree database. In model checking, however, we may safely dedicate the vast majority of available memory of a system to the state storage.

The current implementation of LTSmin [LPW11a] supports a maximum of 2^{31} tree nodes, yielding about 2×10^9 states with optimal compression. In the future, we aim to create a more flexible solution that can store more states and automatically scales the number of bits needed per entry, depending on the state vector size. Low-level issues have hold us back thus far from implementing this, i.e., the ordering of multiple atomic memory accesses across cache line boundaries behave erratically on certain processors.

While the current chapter discusses tree compression mainly in the context of reachability, it is not limited to this context. For example, on-the-fly algorithms for the verification of liveness properties can also benefit from a space-efficient storage of states as demonstrated by Spin with its Collapse compression.

Future work. A few options are still open to improve tree compression. Static analysis of the dependencies between transitions and state slots could be used to reorder state slots and obtain a better balanced tree, and hence better compression (see Section 3.4). Much like the variable ordering problem of BDDs [Bry86], finding the optimal reordering is an exponential problem (a search through all permutations). While, we are able to improve most of the worse cases by automatic variable reordering, we did not yet find a good heuristic for at least all Beem models.

Finally, it would also be interesting to generalize the tree database by accommodating for the storage of vectors of different sizes.

Final remark. The small tree node entries cover a limited universe of values: $1+2\times\log_2(n)$. This is an ideal case to employ *key quotienting* using *Cleary table* [Cle84] or *very tight hash table*s [GV03]. A solution for this is presented in the subsequent chapter. The result is a parallel tree database implementation that in the (common) optimal case uses only one integer (4 bytes per state), halve of the memory requirements of the current tree database.

4 A Parallel Compact Hash Table

Alfons Laarman, Steven van der Vegt

Abstract

We present the first parallel compact hash table algorithm. It delivers high performance and scalability due to its dynamic region-based locking scheme with only a fraction of the memory requirements of a regular hash table.

About this chapter: The current chapter is based on the paper *"A Parallel Compact Hash Table"*, which was published at MEMICS 2011 [VL12].

The original paper [VL12] remains largely the same, modulo some small textual improvements. We removed the general introduction. We added an extra section detailing the use of the *concurrent Cleary table* in the concurrent tree database from Chapter 3. This section requires no new experiments, as we can theoretically show that the compression ratio of the new *Cleary tree* is directly related to the compressions obtained with the tree (Section 3.5 demonstrates that most results lie close to the optimum).

We also added a section that establishes an information-theoretic lower bound on the storage requirements for typical model checking problems. We show that the Cleary compact tree can actually reach this bound in theory when compression is optimal. The experiments from Section 3.5 already demonstrated that this is more often the case than not in practice.

4.1 Introduction

Data structures, like *hash tables*, are crucial building blocks for these systems and many have been parallelized [Cli07; HS08] to prevent a multi-core crisis. A hash table stores a subset of a large *universe* U of keys and provides the means to lookup individual keys in constant time. It uses a *hash function* to calculate an address h from the unique *key*.

The entire key is then stored at its hash or home location in a table (an array of *buckets*): $T[h] \leftarrow key$. Because often $|U| \gg |T|$, multiple keys may have the same hash location. We can handle these so-called *collisions* by calculating alternate hash locations and searching for a key in the list of alternate locations, a process known as *probing*.

In the case that $|U| \leq |T|$, a hash table can be replaced with a *perfect hash function* and a bit array, saving considerable memory. The former ensures that no collisions can occur, hence we can simply turn “on” the bit at the home location of a key, to add it to the set. *Compact hashing* [Cle84; DM09] generalizes this concept for the case $|U| > |T|$ using a technique called *key quotienting*. Hashing in T is done using the key’s *quotient*, while T stores only the part of the key that was not used for hashing: the *remainder*. The complete key can now be reconstructed from the value in T and the home location of the key. If, due to collisions, the key is not stored at its home location, additional information is needed. Cleary [Cle84] solved this problem with very little overhead by imposing an order on the keys in T and introducing three administration bits per bucket.

The bucket size b of Cleary compact hash tables is thus dependent on U and T as follows: $b = w - m + 3$, with the key size $w = \lceil log_2(|U|)\rceil$ and $m = \lceil log_2(|T|)\rceil$. Assuming that all the buckets in the table can be utilized, the compression ratio obtained is thus close to the information-theoretic lower bound of storing a subset of U in a list T, where $b_{optimal} = w - m + 1$ [GV03]. Note that good compression ratios ($\frac{b}{w}$) are only obtained when m is significant with respect to w.

Problem description. Compact hashing has never been parallelized, even though it is ideally suited to be used inside more complex data structures, like *tree tables* (see Chapter 3) and *binary decision diagrams* (BDDs) [DLP13]. Such structures maintain large tables with small pieces of constant-sized data, like pointers, yielding an ideal m and w for compact hashing. But even more interesting than obtaining some (constant-factor) memory reductions, is the ability to store more information in machine-sized words, for efficient parallelization depends crucially on memory alignment and low-level operations on word-sized memory locations [Cli07; LPW10a].

Contributions. We present an efficient scheme to parallelize both the Cleary table and the order-preserving *bidirectional linear probing* (BLP) algorithm that it depends upon. The method is *lockless*, meaning that it does not use operating system locks, thereby providing the performance required for use in high-throughput environments, like in BDDs, and avoiding memory overhead.

Our algorithm guarantees read/write exclusion, but not on the lowest level of buckets, as in [Cli07; LPW10a], nor on fixed-size regions in the table as in *region-based locking*, aka *striped locking*, but instead on the logical level of a *cluster*: a maximal subarray $T[i \dots j]$ such that $\forall x : i \leq x \leq j \implies T[x].occ$,where $T[x].occ$ denotes a filled bucket. We call this novel method: *dynamic region-based locking* (DRL).

Finally, we show how the Cleary table can be used in *tree compression* to almost halve the compressed sizes for state vectors in model checking. An information-theoretic model for the state entropy demonstrates that this compression is close to the optimum.

4.2 Background

In the current section, we explain the *Cleary table* and the BLP algorithm it uses. Finally, we discuss some parallelization approaches that have been used before for hash tables and the issues that arise when applying them to the Cleary table.

For this discussion, the distinction between *open-addressing* and *chained* hash tables is an important one. With open addressing, the probing for alternate locations is done inside the existing table as is done in BLP and hence also in Cleary tables. While *chained* or *closed-addressing* hash tables resolve collisions by maintaining (concurrent) linked lists at each location in the table.

4.2.1 Bidirectional Linear Probing

The simplest form of open addressing is *linear probing* (LP): alternate hash locations in the table are calculated increasing by one to the current location. While this probing technique provides good spatial locality, it is known for producing larger clusters, i.e., increasing the average probing distance [Cli07].

BLP [AK74] mitigates the downside of LP, by enforcing a global order on the keys in the buckets using a *monotonic hash function*: if $k_1 < k_2$ then $hash(k_1) \leq hash(k_2)$. Therefore, the lookup of a key k boils down to: compare the k to the bucket at the home location h, if $T[h] > k$, probe left linearly ($h' \leftarrow h - 1$), until $T[h'] = k$. If k is not present in the table, the probe sequence stops at either an *empty* bucket, denoted by $\neg T[h'].occ$, or when $T[h'] < k$. If $T[h] < k$, do the reverse.

To maintain order during an insert of a key, the BLP algorithm needs to move part of a cluster to the left or the right in the table, thereby making space for the new key at the correct (in-order) location. This move is usually done with pair-wise swaps, starting from the empty bucket at one end of the cluster. Therefore, this is referred to as the *swapping* operation. For algorithms and a more detailed explanation, please refer to [AK74; Veg11].

4.2.2 A Compact Hash Table Using the Cleary Algorithm

As explained in Section 4.1, Cleary's compact hash table [Cle84] stores only the remainder of a key in T. With the use of the sorting property of the BLP algorithm and 3 additional *administration bits* per bucket, the home location h of the remainder can be reconstructed, even for colliding entries that are not stored at their home location. For this to work, the hash function needs to be *reversible* in addition to being *monotonic*. [Cle84] describes some solutions for this. We will use $hash^{-1}$ for the reversed version.

The *rem* function is the complement of the monotonic hashing function and calculates the remainder, e.g., $rem(x) = x\%10$ and $hash(x) = x/10$.[4.1] A *group h* is a sequence of successive remainders in T with the same home location h. All adjacent groups in T form a cluster, which by definition is enclosed by empty buckets (see Section 4.1).

The first administration bit *occ* is used to indicate occupied buckets. The *virgin* bit is set on a bucket h to indicate the existence of the related group h in T. And finally, the *change* bit marks the last (right-most) remainder of a group, such that the next bucket is empty or the start of another group.

Figure 4.1 shows the Cleary table with $|T| = 10$ that uses the example *hash* and *rem* functions from above. A group h is indicated with g_h. Statically, keys can be reconstructed by multiplying the group number by 10, and adding the remainder: $key(j) = group(T[j]) \times 10 + T[j] = hash^{-1}(group(T[j])) + T[j]$. For example, bucket 6 stores remainder 8 and $group(6) = 4$, therefore $key(6) = 4 \times 10 + 8 = 48$.

The algorithms maintain the following invariants [Cle84]: the amount of *change* and *virgin* bits within a cluster is always equal, and, when a virgin bit is set on a bucket, this bucket is always occupied.

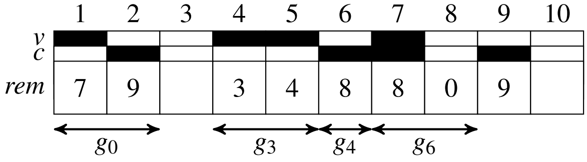


Figure 4.1: Example Cleary table with 10 buckets containing 8 remainders, 2 clusters and 4 groups, representing the keys: 7, 9, 33, 34, 38, 48, 60, 69. The upper two rows of the buckets represent the *v*irgin and the *c*hange bits. The occupied bit is not shown (buckets without values are unoccupied).

[4.1] To increase the performance of the hash function, it is common practice to apply an invertible randomization function to the key before hashing it [AK74; Cle84; GV03]. Throughout the current chapter, we assume keys to be randomized.

The FIND function in Algorithm 4.1 makes use of these invariants as follows: it counts the number of *virgin* bits between the home location *h* and the left end of the cluster in *c* (see VCOUNT-LEFT). Since the last encountered *virgin* bit corresponds to the left-most group, the group *h* can now be located by counting *c change* bits to the right (Line 13-17). The first iteration where $c = 1$ marks that start of group *h*. Hence, the algorithm starts comparing the remainders in $T[j]$ with $rem(k)$ at Line 14, and returns *FOUND* when they are equal. Once *c* becomes 0 again, the group *h* did not contain the key, and *NOT_FOUND* is returned at Line 18.

Algorithm 4.1 Functions for finding (**a**) and inserting (**b**) a key in a Cleary table.

```
 1: procedure VCOUNT-LEFT($j$)
 2:     $c \leftarrow 0$                          ▷ count variable
 3:     while $T[j].occ$ do
 4:         $c \leftarrow c + T[j].virgin$
 5:         $j \leftarrow j - 1$
 6:     return $j, c$
 7: procedure FIND($k$)
 8:     $j \leftarrow hash(k)$
 9:     if $\neg T[j].virgin$ then
10:         return NOT_FOUND                 ▷ false
11:     $(j, c) \leftarrow$ VCOUNT-LEFT($j$)
12:     $j \leftarrow j + 1$
13:     while $c \neq 0 \wedge T[j].occ$ do
14:         if $c = 1 \wedge T[j] = rem(k)$ then
15:             return FOUND                 ▷ true
16:         $c \leftarrow c - T[j].change$
17:         $j \leftarrow j + 1$
18:     return NOT_FOUND                     ▷ false
```

```
Require: $(\exists i : \neg T[i].occ) \wedge \neg$FIND$(k)$
 1: procedure PUT($k$)
 2:     $h \leftarrow hash(k)$
 3:     $(j, c) \leftarrow$ VCOUNT-LEFT($h$)
 4:     $T[j] \leftarrow rem(k)$
 5:     $T[j].occ \leftarrow 1$
 6:     $T[j].change \leftarrow 0$
 7:     while $c \neq 0$ do
 8:         if $T[h].virgin \wedge c = 1 \wedge$
 9:         $T[j+1] > rem(k)$ then
10:             return
11:         $c \leftarrow c - T[j+1].change$
12:         SWAP($T[j+1], T[j]$)
13:         $j \leftarrow j + 1$
14:     if $T[h].virgin$ then
15:         $T[j-1].change \leftarrow 0$
16:     $T[j].change \leftarrow 1$
17:     $T[h].virgin \leftarrow 1$
```

The PUT function in Algorithm 4.1b inserts the remainder of *k* in the empty bucket left of the cluster around *h* at Line 4-6 and swaps it in place at Line 7-13 (SWAP only swaps the remainder and the *change* bit). In this case, *in place* means two things: within group *h* as guaranteed by Line 7 and Line 8, and sorted by remainder value as guaranteed by Line 9. Furthermore, PUT guarantees the correct setting of the administration bits. First, the *occ* bit is always set for every inserted element at Line 5. Also, before return, the *virgin* bit is always set for $T[h]$ (see Line 8 and Line 17).

To understand the correct setting of the *change* bits, we introduce an invariant: at

Line 8, $group(T[j+1]) \leq h$. Consequently, a return at Line 10, means that the remainder is not swapped to the end of group h, therefore the *change* bits do not require updating. On the other hand, if the **while** loop terminates normally, the remainder is swapped to the end of group h, therefore the *change* bit needs to be set (Line 16). If group h already existed ($T[h].virgin =$ true), the previous last remainder of the group needs to have its *change* bit unset (Line 15).

We illustrate PUT with an example. Inserting the key 43 into the table of Figure 4.1 gives a $h = hash(43) = 4$ and $rem(43) = 3$. Searching for the empty bucket left of the cluster at Line 3, results in $j = 2$ and $c = 2$, since there are two *virgin* bits in buckets 3 and 4. The remainder is initially inserted in $T[2]$ (Line 4-6). At Line 12 the remainder in bucket 2 is swapped with bucket 3 (the *virgin* bit remains unchanged). These steps are repeated until j points to bucket 5. Then, at Line 11 c becomes 1, indicating $group(T[j+1]) = h$. In the next iteration ($j' = j - 1$), the condition at Line 8-9 holds, meaning that the remainder is at its correct location: at the start of g_4.

If instead, we were inserting the key 49, c would have become 0, ending the **while** loop with $j = 6$ (Line 7), after swapping the remainder 9 to bucket 6. Because g_4 already existed, the previous *change* bit (now on $T[5]$) is unset by Line 14-15. Finally, the *change* bit at bucket 6 is set by Line 16.

To make groups grow symmetrically around their home locations and keep probing sequence shorter, it is important that the PUT function periodically also starts inserting remainders from the right of the cluster (not shown in the algorithm). Our experimental results confirm that a random choice between the two insert directions yields the same probe distances as reportedly obtained by the optimal replacement algorithms in [AK74].

4.2.3 Related Work on Parallel Hash Tables

In the current subsection, we recapitulate some relevant, existing approaches to parallelize hash tables. With relevant, we mean parallel hash tables that can efficiently store smaller pieces of data (remember, from the introduction, that the key size w should be significant with respect to m for compact hashing to be effective). Furthermore, the scalability should be good for high-throughput systems like inside BDDs.

We use the following abbreviations:

Many parallel hash table implementations are based on chaining. More advanced approaches even introduce more pointers per bucket, for example: *split-ordered lists* [HS08, Sec. 13.3], which: “move[s] the buckets among the [keys], instead of moving the [keys] among the buckets”. While these kind of hash tables lend themselves well for maintaining small sets in parallel settings like graphical user interfaces, they are less suited for our goals for two reasons: (1) the pointers require relatively much additional memory

Abbr.	*Meaning*
LP	Linear Probing
BLP	Bidirectional Linear Probing
LHT	Lockless Hash Table
RBL	Region-Based Locking
DRL	Dynamic Region-based Locking
PCT	Parallel Cleary Table

compared to the small bucket sizes that are so typical for compact hashing and (2) the pointers increase the *memory working set*, which is disastrous for scalability on modern computer systems with steep memory hierarchies [LPW10a; Cli07].

Slightly more relevant to our cause is the use of operating system locks to make access to a hash table (chained or open addressing) concurrent. One lock can be used for the entire table, but this is hardly scalable. Alternatively, one lock can be used per bucket, but this uses too much memory (we measured 56 bytes for POSIX locking structures, this excludes any memory allocated by the constructor). A decent middle way is to use one lock for a group of buckets. The well-known *striped* hash table [HS08, Sec. 13.2.2], does this for chained tables. To employ the same idea for an open-addressing table, it does not make sense to 'stripe' the locks over the table buckets. Preferably, we group subsequent buckets into one region, so that only one lock needs to be taken for multiple probes. We dub this method *region-based locking* (RBL).

Lockless hash tables avoid the use of operating system locks entirely. Instead, atomic instructions are used to change the status of buckets ("locking" in parentheses). A *lockless hash table* (LHT) is presented in Chapter 2, based on ideas from [Cli07]. It uses open addressing with LP and even modifies the probe sequence to loop over cache lines ("walking the line") to lower the memory working set and achieve higher scalability. For maximum scalability, only individual buckets are "locked" using one additional bit; the only memory overhead that is required.

None of the above-mentioned methods are suitable for *ordered hash tables*, like BLP and Cleary tables. First the regions in RBL are fixed, while the clusters in ordered tables can be at the boundary of a region. While this could be solved with more complicated locking mechanism, it would negatively affect the performance of RBL, which is already meager compared to the lockless approaches (see Sec. 4.6). The lockless approach, in turn, also fails for ordered hash tables since it is much harder to "lock" pairs of buckets that are swapped atomically. And even if it would be technically possible to efficiently perform an atomic pairwise swap, it would severely increase the amount of (expensive) atomic operations per insert (Section 4.3.2 discusses the complexity of the swapping operations).

[Veg11] introduced a lockless algorithm for BLP that "locks" only the cluster during swapping operation. FIND operations do not require this exclusive access, for an ongoing PUT operation can only cause false negatives that can be mitigated by another *exclusive* FIND operation. However, this method is not suitable for the Cleary table, since its FIND function is *probe sensitive*, because it counts the *virgin* and *change* bits during probing. Therefore, it can cause false positives in case of ongoing swapping operations. The current chapter is an answer to the future work of [Veg11].

4.3 Dynamic Region-Based Locking

In the current section, we first present *dynamic region-based locking* (DRL): a locking strategy that is compatible with the access patterns of both the BLP algorithm with its swapping property and the Cleary table with its probe-sensitive lookup strategy. We limit our scope to a procedure that combines the FIND and PUT functions, described in the previous section, into the FIND-OR-PUT function, which searches the table for a key *k* and inserts *k* if not found. The reason for this choice is twofold: first, it covers all issues of parallelizing the individual operations, and second, the FIND-OR-PUT operation is sufficient to implement advanced tasks like *model checking* (see Chapter 1).

Additionally, in Section 4.3.2, we show that DRL only slightly increases the number of memory accesses for both BLP and PCT. From this and the limited number of atomic operations that it requires, we conclude that its scalability is likely as good as LHT's, which we can indeed confirm in Section 4.6. We end with a correctness proof of DRL in Section 4.3.3.

4.3.1 Parallel FIND-OR-PUT Algorithm

In the previous section, we have seen that the lockless method presented in [Cli07; LPW10a], is not suitable for Cleary tables, since it would require atomic operations on multiple pair-wise swaps. Region-based locking is neither appropriate, since clusters grow "organically" and may span multiple fixed-size regions. Here, we introduce a *dynamic region-based locking* (DRL) scheme that can be used in combination with both the BLP algorithm and the Cleary compact hash table with its *probe-sensitive* FIND operation. It uses one extra bit field per bucket (*lock*) to provide light-weight mutual exclusion. This method has limited memory overhead and does not require a context switch and additional synchronization points like operating system locks.

The atomic functions TRY-LOCK and UNLOCK control this bit field and have the following specifications: TRY-LOCK requires an empty and unlocked bucket and guarantees an empty, locked bucket or otherwise fails. UNLOCK accepts multiple buckets and ensures all are unlocked upon return (each atomically, the multiple arguments are merely syntactic sugar). These functions can be implemented using the processor's CAS(a, b, c) operation, which updates a word-sized memory location at a with c atomically, if and only if the condition b holds for location a [HS08, Ch. 5.8]. CAS returns the initial value at location a, used to evaluate the condition.

Algorithm 4.2 shows the dynamic locking scheme for the FIND-OR-PUT algorithm. First, at Line 3, the algorithm tries to write k to $T[h]$, only if the home location h is empty and unlocked ($\neg T[h].lock \wedge \neg T[h].occ$). The function GRAB-UNOCC-EMPTY does this and returns the previous value of the bucket as *old*. The success of the operation can be determined from *old* (see Line 4). If a lock or full bucket was detected, the algorithm is restarted at Line 7.

Algorithm 4.2 Concurrent bidirectional linear find-or-put algorithm

```
 1: procedure FIND-OR-PUT($k$)
 2:     $h \leftarrow hash(k)$    ▷ non-excl. write:
 3:     $old \leftarrow$ GRAB-UNOCC-EMPTY($T[i], k$)
 4:     if $\neg old.occ \wedge \neg old.lock$ then
 5:         return INSERTED
 6:     else if $old.lock$ then
 7:         return FIND-OR-PUT($k$)▷ retry
 8:     if FIND($k$) then    ▷ non-excl. read
 9:         return FOUND
10:     $left \leftarrow$ CL-LEFT($h$)
11:     $right \leftarrow$ CL-RIGHT($h$)
12:     if $\neg$TRY-LOCK($T[left]$) then
13:         return FIND-OR-PUT($k$)▷ retry
14:     if $\neg$TRY-LOCK($T[right]$) then
15:         UNLOCK($T[left]$)
16:         return FIND-OR-PUT($k$)▷ retry
17:     if FIND($k$) then    ▷ exclusive read
18:         UNLOCK($T[left]$, $T[right]$)
19:         return FOUND
20:     PUT($k$)    ▷ exclusive write
21:     UNLOCK($T[left]$, $T[right]$)
22:     return INSERTED
```

From Line 10 onwards, the algorithm tries to acquire exclusive access to the cluster around $T[h]$. Note that $T[h]$ is occupied. At Line 10 and Line 11, the first empty location left of and right of h are found in T. If both can be locked, the algorithm enters a local *critical section* (*CS*) after Line 16, else it restarts at Line 13 or Line 16 (after releasing all taken locks). In the *CS*, the algorithm can now safely perform exclusive reads and exclusive writes on the cluster (Line 17 and Line 20).

DRL is suitable in combination with the FIND and PUT operations of both BLP and the Cleary table. If we are implementing the BLP algorithm using this locking scheme, then FIND at Line 8 can perform a non-exclusive read (concurrent to any ongoing write operations). The possibility of a false negative is mitigated by an upcoming exclusive read at Line 17. For the Cleary algorithm, however, the non-exclusive read needs to be dropped because the probe-sensitive lookup mechanism might yield a false positive due to ongoing swapping operations.

4.3.2 Complexity and Scalability

Two questions come to mind when studying the DRL: (1) What is the added complexity compared to the sequential BLP or Cleary algorithm? (2) What scalability can we expect from such an algorithm. Below, we discuss these matters.

For *ordered hash tables*, like BLP and Cleary tables, the cluster size L depends on the load factor α, as follows: $L = (\alpha - 1)^{-2} - 1$ [AK74], where $\alpha = n/|T|$ and n the number of inserted keys. Since DRL probes to the empty buckets at both ends of the cluster, it requires $(\alpha - 1)^{-2} + 1$ bucket accesses. When implementing the Cleary table using DRL, this is the complexity for the FIND-OR-PUT operation independent whether an insert occurred or not, because in both cases it “locks” the entire cluster. Note that we do not count the bucket accesses of the called FIND and the PUT operations, since, in theory, these could be done simultaneously by the CL-LEFT and CL-RIGHT operations. In practice, this seems unnecessary, because the cluster will be cache hot after locking it.

The sequential Cleary FIND and PUT algorithms have to probe to one end of the cluster to count the virgin and change bits, hence require more bucket accesses: $1/2(\alpha - 1)^{-2} + 1/2$ (again assuming that we can count both in one pass or that the second pass is cached and therefore insignificant). We conclude that Cleary+DRL (with one worker thread) is only twice as slow as the original Cleary algorithm.

For BLP+DRL the story changes, but the outcome is the same. The sequential BLP algorithm does not have to probe to the end of the cluster and is empirically shown to be much faster than LP [AK74]. However, DRL+BLP is correct with non-exclusive reads as long as an unsuccessful FIND operation is followed by an exclusive FIND to mitigate false negatives, as is done in Algorithm 4.2. But false negatives are rare, so again the parallel FIND operation is not much slower than the sequential one. The same holds for the PUT operation, since the sequential version on average needs to swap half of an entire cluster and the parallel version “locks” the whole cluster.

Scalability of DRL can be argued to come from three causes: first, the I/O complexity (in memory access) of the parallel algorithm is the same the sequential versions, as shown above, second, the number of (expensive) atomic operations used is low, DRL uses zero, one or two (with the very rare possibility of several retries), and third, the memory accesses are all consecutive. We analyze the third cause in some more detail.

To mitigate the effect of slow memories, caching is important for modern multi-core systems. Each memory access causes a fixed region of memory, known as a cache line, to be loaded into the CPU's cache. If it is written to, the entire line is invalidated and has to be reload on all cores that use it; an operation which is several orders of magnitude more expensive than other operations using in-cache data. We have shown before that highly scalable hashing algorithms can be obtained by lowering the number of cache lines that are accessed: the *memory working set* (see Section 2.2).

The open-addressing tables discussed in the current chapter exhibit only consecutive memory accesses. And while it seems that the amount of buckets probed in the Cleary algorithm is high, typically few cache lines are accessed. For example, there are 26 bucket accesses on average for $\alpha = 0.8$, while on average only $\lceil 26/64 \rceil + 26/64 = 1.41$ cache lines are accessed, assuming a bucket size of 1 byte and a cache line size of 64 byte. When α grows to 0.85, we get 1.71 cache line accesses on average, and when $\alpha = .9$, 3.59 accesses. Note finally that with buckets of 1 byte, the cleary algorithm can store keys of more than 32 bit for large tables, e.g, if $m = 28$, then $w = b + m - 3 = 8 + 28 - 3 = 33$, while the non-compacting hash table requires five bytes per bucket to store as much data. In conclusion, we can expect Cleary+DRL to perform and scale good at least up to load factors of 0.8 and exhibit competitive performance to that of Chapter 2.

4.3.3 Proof of Correctness

To prove correctness, we show that Algorithm 4.2 is *linearizable*, i.e., its effects appear instantaneously to the rest of the system [HS08, Ch. 3.6]. Here, we do this in a constructive way: first, we construct all possible local schedules that Algorithm 4.2 allows, then we show by contradiction that any interleaving of the schedules of two workers always respects a certain critical section (*CS*) of the algorithm, and finally, we generalize this for more workers. From the fact that *CS* is the only place where writes occur, we can conclude linearizability.[4.2] We assume that all lines in the code can be executed as atomic steps.

If the home location of a key k is empty, correctness follows from the properties of the atomic CAS operation at Line 3. For every other table accesses (Line 17 and Line 20), we prove that never two workers can be in their *CS* for the same cluster.

The '$\rightarrow$' operator is used to denote the *happens-before relation* between those steps [HS08]. For example, '$\text{CL-RIGHT}_i(\bar{x}) \rightarrow \text{TRY-LOCK}_i(x)$' means that a Worker i always first executes CL-RIGHT writing to the variable x (Line 11), and subsequently calls TRY-LOCK using (reading) the variable x. We omit the subscript i, if it is clear from the context which worker we are talking about. We concern ourselves with the following local happens-before order: $\text{CAS}(h) \rightsquigarrow \text{CL-LEFT}(\bar{l}) \rightarrow \text{CL-RIGHT}(\bar{r}) \rightarrow \text{TRY-LOCK}(l) \rightsquigarrow$

[4.2] For completeness sake, we should also mention that we only allow for false positives to occur in non-exclusive reads and that unsuccessful non-exclusive reads are always followed by a read operation in the *CS*, i.e., an exclusive read.

$\textsc{try-lock}(r) \rightsquigarrow (occ(l) \oplus occ(r))$, where $occ(x)$ signifies a fill of a bucket ($T[x].occ \leftarrow 1$) and $\rightsquigarrow$ indicates a happens-before relation dependent on a condition. Depending on the replacement end (left or right), $\textsc{put}$ fills one of the buckets at the end of the cluster, hence the exclusive-or: $\oplus$. Furthermore, we write l_i, r_i and h_i for: the *left* variable, the *right* variable and the home-location $h_i = hash(k)$, all local to a Worker i.

Lemma 4.1. *Algorithm 4.2 ensures that when two workers try to enter their CS for the same cluster, then:* $l_i = l_j \vee r_i = l_j \vee l_i = r_j \vee r_i = r_j$.

Proof. Assume Worker W_i is in its *CS*, and Worker W_j is about to enter the *CS* for the same cluster. Since W_i is in its *CS*, we have $T[l_i].lock$ and $T[r_i].lock$. W_i is going to perform the step $occ(l_i)$ or $occ(r_i)$. Note that these operations might influence the clusters, as two clusters separated by only one empty bucket, may become one upon filling the bucket.

Worker W_j has yet to enter its *CS*, executing the steps: $\textsc{cas}(h_j) \rightarrow \textsc{cl-left}(\bar{l}_j) \rightarrow \textsc{cl-right}(\bar{r}_j)$. With a generalizable example, Figure 4.2 illustrates five non-trivial cases that we consider, where W_j starts with a h_j respective to the cluster l_i, r_i. Clusters in T are colored gray and we assume that they are separated by one empty bucket (white), because more empty buckets makes the resulting cases only more trivial. There are several representative home-locations marked with h^a to h^e (e.g., choosing a different location within the same cluster leaves the results of the $\textsc{cl-left}$ and $\textsc{cl-right}$ operations unaffected). Locations on the right of r_i follow from symmetry. Below, we consider the outcome of all the cases for h_j. We use the fact that there are no empty buckets between l_j and r_j.

$h_j = h^a$: Because $T[h_j].occ$, $\textsc{cas}(h_j)$ fails. W_j performs the steps $\textsc{cl-left}(\bar{l}_j) \rightarrow \textsc{cl-right}(\bar{r}_j)$. Since $l_j = 1 < r_j = 3 < l_i$, Lemma 4.1 is vacuously true.

$h_j = h^b$: This location is unoccupied and not locked, so the $\textsc{cas}(h_j)$ succeeds and the algorithm returns never reaching *CS*, making Lemma 4.1 vacuously true.

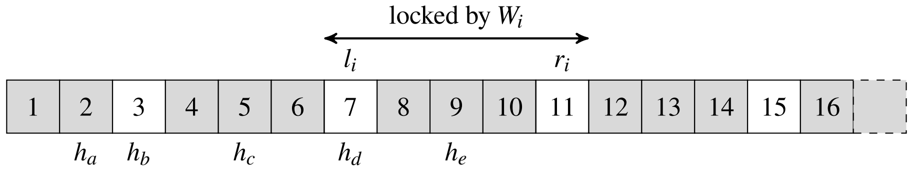


Figure 4.2: Several clusters and empty positions. The cluster 8-10 is locked by worker W_i. Location marked with h^a to h^e potential home locations for worker W_j.

$h_j = h^c$: This location is occupied so $\textsc{cas}(h_j)$ fails. Next, the step $\textsc{cl-left}(\bar{l}_j)$ results in $l_j = 3$. The result r_j of $\textsc{cl-right}$ is dependent on the state of W_i. If W_i has not already performed any occ or did perform $occ(11)$, then $r_j = 7$. If W_i has executed $occ(7)$, then $r_j = 11$. So, $r_j = 7 = l_i \vee r_j = 11 = r_i$.

$h_j = h^d$: The success of the $\textsc{cas}(h_j)$ depends on the state of W_i. If W_i has not performed any steps, then $\textsc{cas}(h_j)$ restarts the algorithm at Line 7. If W_i has performed $occ(7)$, then W_j continues with $\textsc{cl-left}(\bar{l}_j)$ and $\textsc{cl-right}(\bar{r}_j)$, resulting in $l_j = 3, r_j = 11 = r_i$. If W_i has performed step $occ(11)$, then $l_j = 7 = l_i, r_j = 15$.

$h_j = h^e$: Since h^e is occupied, $\textsc{cas}(h_j)$ fails again. W_j continues with the $\textsc{cl-left}(\bar{l}_j)$ and $\textsc{cl-right}(\bar{r}_j)$. The result depends on if W_i has executed $occ(7)$ or $occ(11)$. We distinguish five interleavings:
1: $\textsc{cl-left}(\bar{l}_j) \rightarrow \textsc{cl-right}(\bar{r}_j) \rightarrow (occ_i(7) \oplus occ_i(11)) \Rightarrow l_j = 7, r_j = 11 = r_i$
2: $\textsc{cl-left}(l_j) \rightarrow occ_i(7) \rightarrow \textsc{cl-right}(r_j) \Rightarrow l_j = 7 = l_i, r_j = 11 = r_i$
3: $\textsc{cl-left}(l_j) \rightarrow occ_i(11) \rightarrow \textsc{cl-right}(r_j) \Rightarrow l_j = 7 = l_i, r_j = 15$
4: $occ_i(7) \rightarrow \textsc{cl-left}(l_j) \rightarrow \textsc{cl-right}(r_j) \Rightarrow l_j = 3, r_j = 11 = r_i$
5: $occ_i(11) \rightarrow \textsc{cl-left}(l_j) \rightarrow \textsc{cl-right}(r_j) \Rightarrow l_j = 7 = l_i, r_j = 15$

Thus, under the above assumption: $l_i = l_j \vee r_i = l_j \vee l_i = r_j \vee r_i = r_j$. □

Theorem 4.1. *No two workers can be in their CS at the same time and work on the same cluster such that* $l_i \leq l_j \leq r_i \vee l_i \leq r_j \leq r_i \vee (l_j \leq l_i \wedge r_j \geq r_i)$.

Proof. By contradiction, assume the opposite: both W_i and W_j reach their *CS* and $l_i \leq l_j \leq r_i \vee l_i \leq r_j \leq r_i \vee (l_j \leq l_i \wedge r_j \geq r_i)$. Without loss of generality because of symmetry, we assume again W_i to have entered its *CS* first. The steps for W_j to arrive in its *CS* are: $\textsc{cas}(h_j) \rightarrow \textsc{cl-left}(l_j) \rightarrow \textsc{cl-right}(r_j) \rightarrow \textsc{try-lock}(l_j) \rightarrow \textsc{try-lock}(r_j)$.
The remaining step for W_i is: $occ(l_i) \oplus occ(r_i)$
W_i hash performed $\textsc{try-lock}(l_i) \rightarrow \textsc{try-lock}(r_i)$, thus we have $T[l_i].lock \wedge T[r_i].lock$. According to Lemma 4.1 that at least one of the locations l_j and r_j equals either l_i or r_i. Therefore, W_j will always fail with either $\textsc{try-lock}(l_j)$ or $\textsc{try-lock}(r_i)$. This conclusively proves mutual exclusion for two workers. Since additional workers cannot influence W_j in such a way that Lemma 4.1 is invalidated, Theorem 4.1 also holds for $N > 2$ workers. □

Absence of deadlocks (infinite restarts at Line 7, Line 13 and Line 16), follows from the fact that all "locks" are always released before a restart or a return. Furthermore, we have absence of livelocks, because workers first "lock" the left side of a cluster. The one which locks the right side first, wins. With a fair scheduler the algorithm is also starvation-free, because each worker eventually finished its *CS* in a finite number of steps. From this, we conclude that Algorithm 4.2 is linearizable.

4.4 Concurrent Cleary Tree Compression

In the current section, we show how the Cleary table can be used in the tree database, to almost halve its memory usage in the ideal case. *We will call the keys inserted in the tree from now on 'states' to distinguish them from the keys stored in hash tables.* Like in Section 3.4, we assume that n states of length $k > 1$ are stored in the tree, that references can be encodes as z-bit indices in the tree table, and that the u-bit parts of the states can also be stored in the place of a reference, i.e. $u < z$. (We use z for the reference size instead of w used in Section 3.4 to avoid confusion with the size of the universe $w = |U|$.) For simplicity, we also assume that hash tables in the tree have a size (number of buckets) that is a power of 2. and that the same holds for k.

Tree compression was introduced in the Chapter 3. The techniques recursively indexes keys of fixed lengths k in a binary tree of tuples. With the understanding that tree vectors have a certain structure, i.e. in the context of model checking they consist of an array of u-bit values for the variables in the model, we can see that the tree introduces sharing between similar sub-vectors of different keys inserted into the tree store.

Section 3.4 explains how the concurrent tree uses a single table with buckets of size $2z-1$ bits, to store tuples with z bit values and one root tag. This single-table solution avoids resizing and additional indexing tables, to increase scalability. We chose $z = 31$ in the implementation, to keep compressed sizes low (approaching 64 bit) and store close to 2^{31} states in the optimal case (which experiments showed to be very common).

The dimensions of the concurrent tree table are ideal for the Cleary table: $w = 2z = 62$ and $m = z = 31$ results in compressed bucket sizes of $b = w - m + 3 = 35$ bits. Almost half of the original size. However, the Cleary table does not provide stable indices, as values are moved across buckets to maintain an order. Stable indices are required in the tree table, because the tuples stored in the table refer to other tuples in the same table.

To still use the Cleary table in the tree, we now drop the no-resizing requirement, reasoning that instead of resizing, the reachability search explained in Section 3.3 can also be reinitiated completely with a larger table size. Moreover, as we will see, several other benefits can be attained when dropping this requirement, such as a greater storage capacity in the tree.

To use the Cleary table in the tree, we first split the single table into a table of roots and a table of internal tree nodes. We no longer need the tag bit to distinguish roots from other tuples in the tree, raising z from 31 to 32. Figure 4.3 shows the new configuration, with a single vector stored in the tree. The arrows represent the references in the tree. Since only the internal table has incoming references, it is the only table that needs to have less than 2^z buckets. As a consequence, the roots table can grow larger than 2^z and thus the tree can store more than 2^z states (the size of the internal store does not have to

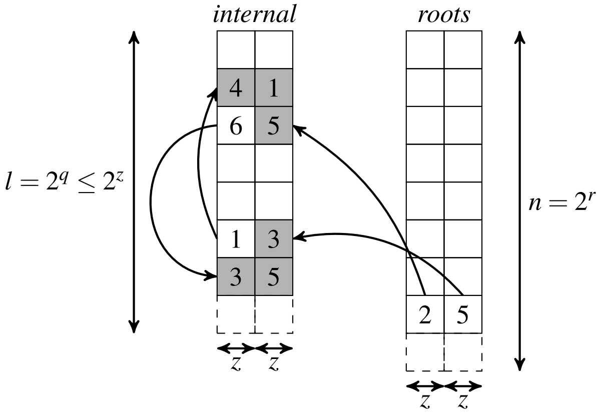


Figure 4.3: New tree table storing a single vector $\langle 3,5,5,3,4,1 \rangle$. To obtain the Cleary tree, the roots table can be implemented with a Cleary table.

be a limitation here according to Lemma 3.1).

It is easy to establish that the new configuration does not use more memory compared to concurrent tree with a single table presented in Section 3.3, although it is harder to fill available memory precisely now that two tables have to compete for space. To analyze this, we disregard maximal sharing, which over-approximates the compression. In practice, maximal sharing is also unlikely to have large effect in practice Section 3.4.

In the new tree table, we can use the Cleary table as root table. This results into a reduction of memory use:

Lemma 4.2. *Disregarding maximal sharing, the Cleary tree uses at least $r-3$ less memory per state than the concurrent tree, where r is the* $\log_2$ *size of the root table in Figure 4.3.*

Proof. The function $T_k \colon \mathbb{N}^k \to \mathbb{N}$ describes the index of root tuple entries for all $s \in S \subseteq \mathbb{N}^k$ vectors in the *concurrent tree*, with $|S| = n$. By Theorem 3.1, this function is injective, therefore all (different) n vectors have a unique root: There are n root tuples. If $k > 2$, each root points to 2 internal tuple entries. Take the left child, it is described by $T_{\lceil k/2 \rceil}$, representing the tree table indices of tuples for all $n_1 \leq n$ different left halves $S' = \{s \in \mathbb{N}^{\lceil k/2 \rceil} \mid \exists s' \in \mathbb{N}^{\lfloor k/2 \rfloor} \wedge s + s' \in S\}$. Since also this subtree is injective, $n_1 = |S'|$ tuple entries are stored to represent these vector halves. We can repeat the argument at each internal tree node, and end up with $n + \sum_{i=1}^{k-2} n_i$ tuple entries, as there is no sharing between entries.

The Cleary tree, can be described by a function $TC_k = C(T_{\lceil k/2 \rceil}, T_{\lfloor k/2 \rfloor})$, where C represents the index in the Cleary table where the root tuple is stored (the fact that this index is not stable does not matter, because they are not referenced in the tree). Following the same reasoning as for the concurrent tree, we see that the Cleary tree also stores $n + \sum_{i=1}^{k-2} n_i$ tuple entries for the same vector set S. But now the n root entries are stored in the Cleary table.

The $\sum_{i=1}^{k-2} n_i$ tuple entries of internal entries are stored using using $2z$ bits in the Cleary tree; less than the $2z+1$ bits in the concurrent tree. For each state in the tree, one tuple is stored in the root table (see Section 3.4). In the concurrent Cleary table, the size of the tuples stored (the universe) depends on $l = 2^q$, thus we have $w = 2q$ and $m = r$. Hence, each tuple requires $b = w - m + 4 = 2q - r + 4$ bits. In the concurrent tree, each tuple (internal and root) requires $2z + 1$ bits. Therefore, the Cleary tree uses at least $(2z+1) - (2q - r + 4) = 2z - 2q + r - 3$ bits less per state.

If the size of the internal table is chosen to just fit all tuple entries, then $z = q$. Therefore, the Cleary tree uses $r - 3$ bits less per state. □

Since the root table may be larger than the internal table, the Cleary tree can halve the memory compared to the concurrent tree. For example, taking $z = q = 31$ bits and $r = 35$ bits, we have an optimal case of $2z + 1 = 63$ bits per state, whereas the previous lemma tells us that the Cleary tree requires 31 bits ($r - 3 = 32$ bits less in this case).

Corollary 4.1. *In the optimal case, the Cleary tree approaches z bits per state, assuming a relatively large number of vectors is stored $n \gg k^2 \gg 1$.*

Proof. In optimal case, the n root entries dominate the tree: $l \ll n$ (see Corollary 3.3). For this reason, the root table can be greater than the internal table: $r > z$, while $q = z$. As the root table is a Cleary table, a larger number of buckets increases the hash quotient, in turn reducing the remainder and the bucket size. Using Lemma 4.2, we see that indeed z bits per root entry can be surpassed.

Even with ideal sharing, the internal table still can contain up to $l = k^2 - 2k$ entries of size $2z$ (see Corollary 3.3). Averaging their total size over all n states in the Cleary tree, a few bits per state account for the memory use of the internal table.

In total, the Cleary tree thus uses around z bits per state. □

Corollary 4.2. *Using only around z bits per state, the Cleary tree can still store more than 2^z states.*

Proof. Follows from the proof of Corollary 4.1. □

Corollary 4.2 is perhaps a surprising and counter-intuitive result. The following section confirms it.

The worst case compression ratios can be derived as well. But obviously, in case tree compression is far from optimal, there is little merit in using the Cleary tree instead of the concurrent tree. Table 4.1 compares the compressed sizes just derived, to those with the concurrent tree from Section 3.3. The choice of the size of z depends on the implementation as discussed below.

Table 4.1: The compressed state sizes, maximum number of states storable and implementation choice for z for the Concurrent tree and the Cleary tree.

	Best case	Max. states	z in impl.
Concurrent tree	$> 2z+1$ bits	$< 2^z$	$z = 31$
Cleary tree	$\approx 2z-r+4$ bits $(q = z)$	$2^r > 2^z$	$z = 32$

Implementation considerations. As explained in Section 3.4.4, hardware constraints dictate the implementation to a large degree. *We implemented the internal table using* $z = 32$ *bits* (64bit buckets in the internal table), and variable size $l = 2^q \leq 2^z = 2^{32}$. References to the internal are bit-packed to $2q \leq 64$ bits, before storing them as root tuples in the Cleary root table. We further fixed the Cleary table's bucket size to $b = 32$ bits, making r variable under the constraint $b = 32 \geq w - m + 4 = 2q - r + 4$. Thus the user should ensure that $r \geq 2q - 28$.

4.5 An Information-Theoretic Lower Bound

This section establishes an information-theoretic lower bound for the storage space required per state. The fact that the tree yields good compression comes from the fact that they contain structure and have combinatorial values as we saw in Section 3.4. States generated by a, e.g. a model checker, have these properties.

The *Cleary tree* stores far larger vectors than the *Cleary table*. The universe of vectors in the tree is $U_{CT} = \{0,1\}^{uk}$ of which only a small subset $S \subseteq U_{CT}$ can ever be stored in memory: $n = |S| \lll |U_{CT}|$. The universe of the Cleary table U_{CL}, on the other hand, is only slightly bigger than the stored subset $S' \subset U_{CL}$, otherwise the compression ratio obtained is not interesting: $\frac{w-m-3}{w}$ with $m \approx |S'|$ and $w = U_{CL}$.

We can still consider the *information entropy* contained in the *larger* states, but a different approach is needed than in Section 4.1. *Information theory* abstracts away from the computational nature of a program by considering sender and receiver as black boxes that communicate data (signals) via a channel. The goal for the sender is to encode the data is small as possible, such that the receiver is still able to decode it back to the original. The encoded size depends on the amount of *entropy* in the data. In the most basic case, no statistical information is known about the data: each of X possible messages has an equal probability of taking one of its values and the entropy H is maximal: $H(X) = log_2(|X|)bit$. The entropy thus corresponds to the number of bits needed for the encoded message.

If more is known about the statistical nature of the information coming from the sender, the entropy is lower and encoding can be applied to reduce the number of bits needed per piece of information (bytes in the previous example). A simple example is when we take into account the character frequency of the English language for encoding sentences. Assuming that certain characters are much more common, a code of fewer bits can be used for them, while longer codes can be reserved for other characters. To calculate the entropy in this example, we need the probability of occurrence $p(x)$ for each character $x \in X$ in the English language. We can deduce this from analyzing a dictionary, or better a large corpus of texts. The entropy than becomes: $H(X) = \sum_{x \in X} -p(x)\log_2(p(x))$

We apply the same principle now to structured data. As example, we use state vectors as processed in a model checker. In the previous section, we were reminded that states consist of k slots of each u bits. In the previous chapter, we also saw that states are generated by a next-state function, and locality ensures similar successors, e.g.:

$$\textsc{next-state}(\langle 3,5,5,4,1,3\rangle) = \{\langle 3,5,\mathbf{9},4,1,3\rangle, \langle 3,5,5,4,\mathbf{2},3\rangle, \dots\}$$

As the predecessor is thus always known in the model checker's reachability procedure, we can abstract away from this one-to-many relation and view the states arriving at the tree as a k-periodic stream of u-bit slots, as illustrated in Figure 4.4. The stream

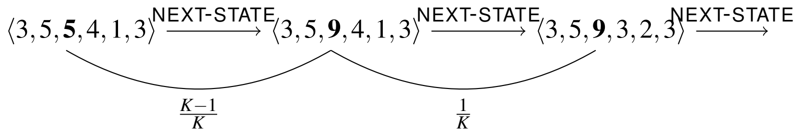


Figure 4.4: The states generated with the NEXT-STATE function seen as a stream. Assumed probabilities are shown for the bold slot values.

can also be described as: $\langle v_0^0, \dots v_{k-1}^1\rangle, \langle v_0^1, \dots v_{k-1}^1\rangle, \dots \langle v_0^{n-1}, \dots v_{k-1}^{n-1}\rangle$. Given that the predecessor state is always known, it makes sense to describe the probability of different slot values of a state with respect to its predecessor: To encode a slot v_j^i with $i \geq 0$ and $0 \leq j < k-1$, the encoder can then always look at the predecessors' value of the corresponding slot v_j^{i-1} to derive the absolute probabilities over all values.

Since we are interested in establishing a *lower bound* we may safely under-approximate the number of slot values changed with respect to a state's predecessor. It make sense to assume that only 1 slot changes, since with lower values, the same state is generated to often (and we do not require space to store equal states in the tree). Thus we take the following *relative* probabilities:

$$p(v_j^i \neq v_j^{i-1}) = \frac{1}{k}$$

$$p(v_j^i = v_j^{i-1}) = \frac{k-1}{k}$$

With $y = 2^u$, notice that the there are $y-1$ possible values for which a state slot can differ from its predecessor: $p(x) = \frac{1}{k(y-1)}$ for $x \neq v_j^{i-1}$. This results in the following definition of entropy per slot:

$$H_{slot}(s_j^i) = -\frac{k-1}{k}\log_2(\frac{k-1}{k}) + \sum_{i=1}^{y-1} -\frac{1}{k(y-1)}\log_2(\frac{1}{k(y-1)})$$

For reasonably large y and k, i.e. $1 \ll k \ll y$, we arrive at:

$$H_{slot} \approx \frac{1}{k}(\log_2(y) + \log_2(k) + k\log(\frac{k}{k-1}))$$

From this we can derive the entropy of a state:

$$H_{state} = k \times H_{slot} = \log_2(y) + \log_2(k) + k\log(\frac{k}{k-1})$$

Given that u is often an integer in model checking software, it is common to have $u = z$, hence the entropy per state can approach $\log_2(y) = \log_2(2^z) = z$ for $1 \ll k \ll y \ll n$. This provides some intuition behind Corollary 4.2, as the information entropy is indeed is not dependent on the number of states n. From Corollary 4.1, we conclude that the Cleary tree can approach the information-theoretic optimum.

4.6 Experiments

In the current section, we show an empirical evaluation of the *parallel Cleary table* (PCT), i.e. Cleary+DRL, by comparing its absolute performance and scalability with that of BLP+DRL, LHT and RBL. In our experiments, several parameters have been fixed as follows: $m = 28$, $b = 16$ for PCT, while for the non-compacting tables $b = 64$, and finally $\alpha = 0.9$. These parameters reflect best the goals we had in mind for this work, since all tables can store pointers larger than 32 bits. Furthermore, the load factor and bucket size for PCT is higher than the values discussed in Section 4.3.2, creating a healthy bias against this algorithm. Additionally, we investigated the influence of different load factors on all tables.

We used the following benchmark setup. All tables were implemented in the C language using pthreads.[4.3] For RBL, we determined the optimal size of the regions by finding the size that yielded the lowest parallel runtime, as the scalability depends largely on this parameter [DJK13]. For table of 2^{28} buckets, this turned out to be 2^{13}. The benchmarks were run on Linux servers with 4 AMD Opteron(tm) 8356 CPUs (16 cores total) and 64GB memory. The maximum key size w that all tables can store in our configuration is 40: for PCT we have $w = b + m - 4 = 16 + 28 - 4 = 40$, and for BLP, LHT and RBL we have $w = 64 - 2 = 62$ (2 for the *lock* and *occ* bit). Therefore, we fed the tables with 40 bit keys, generated with a pseudo random number generator.

Table 4.6 gives the runtimes of all hash tables for different read/write ratios and load factor of 90%. Beside the runtimes with 1, 2, 4, 8, and 16 cores (T_N for $N \in \{1, 2, 4, 8, 16\}$), we included the runtimes of the sequential versions of the algorithms T_{seq}, i.e., the algorithm run without any locks and atomic instructions. From this, we

Table 4.2: Runtimes of BLP, RBL, LHT and PCT with r/w ratios 0:1, 3:1 and 9:1.

Alg.	LHT			RBL			BLP			PCT		
r/w ratio	0:1	3:1	9:1	0:1	3:1	9:1	0:1	3:1	9:1	0:1	3:1	9:1
T_{seq}	77.5	242.4	569.2	76.7	239.9	563.2	71.8	279.1	676.0	54.5	368.9	1050.
T_1	81.6	255.2	599.2	145.9	565.4	1404.	97.5	302.0	726.3	77.3	565.9	1543.
T_2	51.6	157.6	371.0	85.0	327.6	813.4	60.8	188.8	443.9	44.4	317.7	863.9
T_4	26.5	77.9	184.0	46.2	170.2	424.9	31.3	94.0	219.1	23.4	159.7	431.9
T_8	13.9	39.6	92.9	24.0	89.4	219.2	16.5	47.8	110.3	11.5	79.7	216.0
T_{16}	7.7	21.1	48.8	13.5	48.6	120.5	9.4	25.5	57.2	6.0	41.6	112.9

[4.3] Available at: `http://fmt.cs.utwente.nl/tools/ltsmin/memics-2011`

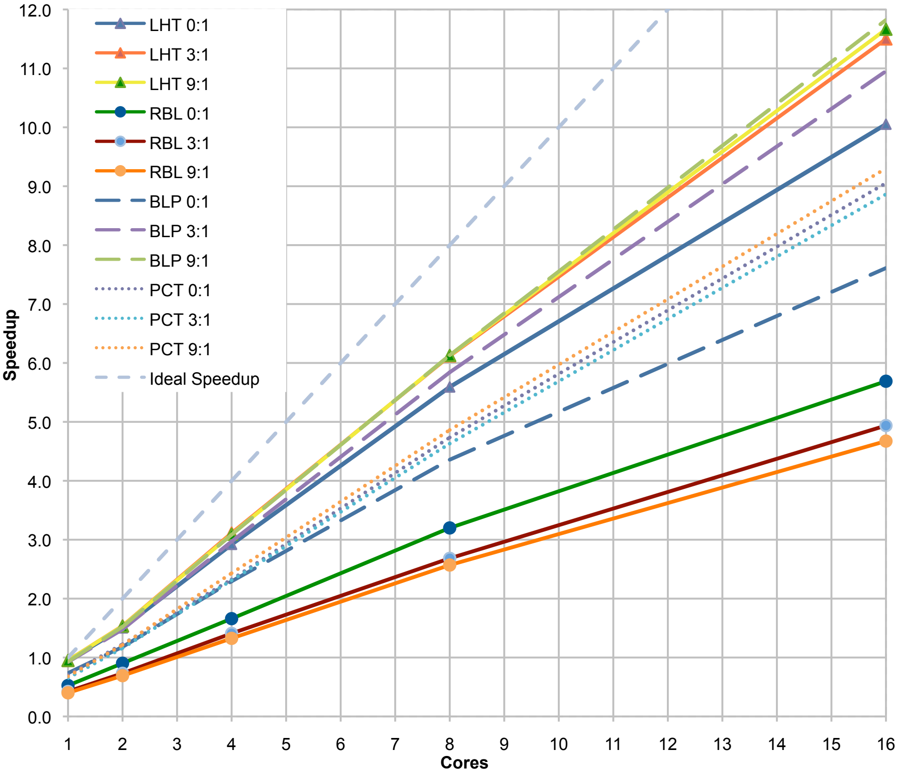


Figure 4.5: Speedups of BLP, RBL, LHT and PCT with r/w ratios 0:1, 3:1 and 9:1.

can deduce the overhead from the parallelization. Comparing the runs with a r/w ratio of 0:1, we see that the sequential variants have more or less the same runtime (PCT is slightly faster, due to its compacter table). Only the lockless algorithms show little overhead when we compare T_{seq} to T_1, while DRL shows that the POSIX mutexes slow the algorithm down by a factor of two. The same trend is reflected in the values for T_N with $N > 1$.

If we now focus our attention to the higher r/w ratios, we see that reads are much more expensive for PCT. This was expected, since non-exclusive reads in DRL are not allowed for PCT as explained in the previous section. To investigate the influence of the r/w ratio, we plotted the absolute speedups ($S_N = T_{seq}/T_N$) of the presented runs in

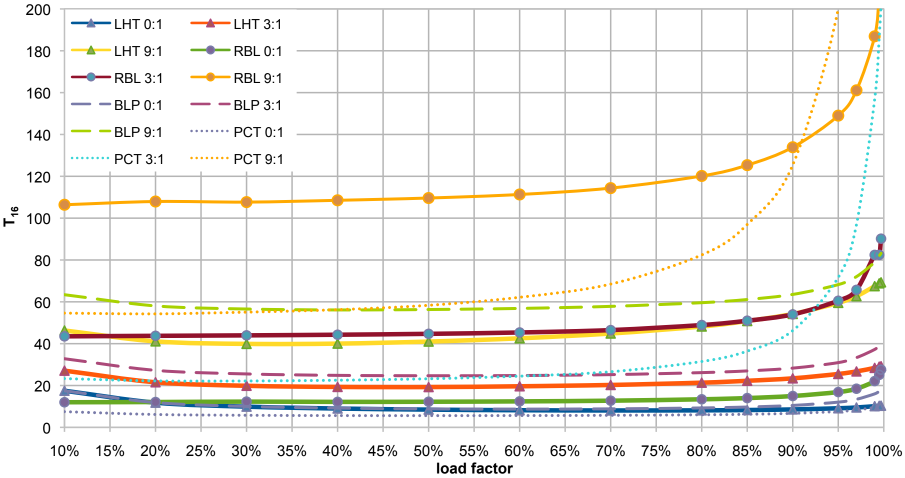


Figure 4.6: 16-core runtimes of BLP, RBL, LHT and PCT.

Figure 4.5. The lightweight locking mechanism of DRL delivers good scalability for PCT and BLP, almost matching those of LHT. While PCT speedups are insensitive to the r/w ratio, since the algorithm always performs the same locking steps for both read and write operations, BLP shows much better speedups for higher r/w ratios. Finally, we see that RBL is no competition to the lockless algorithms.

To investigate the effects of the load factor, we measured the 16-core runtimes of all algorithms for different load factors. To obtain different load factors we modified the number of keys inserted and not the hash table size, therefore we plotted the normalized runtimes T^{norm} in Figure 4.6 ($T^{norm} = T/\alpha$, where $\alpha = n/|T|$ is the load factor and n the number of keys inserted). Due to the open-addressing nature of the hash tables presented here, the asymptotic behavior is expected for α close to 100% (the probe sequences grow larger as the table fills up). However, this effect is more pronounced for PCT, again because of the read-write exclusion, and for RBL, because more locks have to be taken once the probe distance grows.

The concurrent Cleary tree implementation is available in the multi-core LTSmin backend [LPW11a]. Benchmarks and compression ratios can be found in Chapter 11.

4.7 Discussion and Conclusions

We have introduced DRL: a lockless mechanism to parallelize BLP and Cleary compact hash tables efficiently. We have shown, analytically and empirically, that these Parallel Cleary Tables (PCT) scale well up to load factors of at least 80%. This is acceptable, since the compression ratio, obtained by compact hashing, can be far below this value.

With experiments, we also compared both parallel ordered hash tables (PCT and BLP) with a state-of-art lockless hash table (LHT) and a region-based locking table that uses operating system locks (RBL). We found that PCT and BLP can compete with LHT in terms of scalability, but adds a factor 2 of performance overhead. On the other hand, RBL scales worse than the other lockless tables. We finally showed that PCT comes with higher costs for FIND operations and higher load factors. However, this also holds for the sequential algorithm because it has to probe to the end of the cluster as the analysis showed and as is reflected by the good speedups that PCT still exhibits.

While we concentrated in this work on a parallel FIND-OR-PUT algorithm, we think that other operations, like individual FIND, p and DELETE operation, can be implemented with minor modifications.

In future work, we would like to answer the following questions: Could DRL be implemented with locking only one side of the cluster and the home location? Could PCT be implemented with non-exclusive reads? The former could further improve the scalability of DRL, while the latter could transfer the performance figures of parallel BLP to those of PCT. We would also like to eliminate the superfluous occupied bit [DM09, page 5] and see if DRL could be used on similar hashing schemes such as Robin Hood hashing [CLM85].

We further showed the use of the Cleary table in tree compression. The resulting compression comes close to the information-theoretic optimum as our model for a lower bound on state entropy shows.

Part III

Algorithms for Multi-Core LTL Model Checking

Introduction

With the realization of scalable multi-core reachability and compatible state compression techniques in Part II, many safety properties can be checked efficiently (see Section 1.4). To support all safety properties and also liveness properties, we need parallel algorithms for checking temporal logics, e.g. LTL. Several parallel LTL model checking algorithms already exist (Section 1.5.1), but these are all based on distributed algorithms which loose the optimal time complexity and on-the-fly property of the automata-theoretic approach to model checking. In the current part, we pursue a linear-time algorithm for LTL checking, an important open problem according to many researchers:

> "It is as yet an open problem how a liveness verification algorithm could be generalized to the use of more than two processing cores while retaining a low search complexity."
>
> [HB07]

> "One of the most important open problems of parallel LTL model checking is to design an on-the-fly scalable parallel algorithm with linear time complexity."
>
> [BBR10b]

In the current part, we exploit the strengths of the parallel reachability from Part II – its flexibility with respect to search orders and its on-the-fly capability – to create a new parallel version of the traditional linear-time NDFS algorithm for finding *accepting cycle*s in a graph. The accepting cycles constitute (all) counterexamples in the automata-theoretic approach to model checking, thus solving the problem of LTL checking. A parallel NDFS algorithm is proposed in Chapter 5 and gradually improved in Chapter 6 and Chapter 7. Our experimental results show that the resulting CNDFS algorithm (Chapter 7), delivers scalable parallel LTL checking with improved on-the-fly behavior (see Section 7.4.4, but also Section 6.4) and which is linear in the size of the graph.

Because the sequential NDFS relies on the DFS order [Kru05], of which the parallelization is theoretically infeasible as explained in Section 5.1, we rely instead on eager independent, or embarrassingly, parallel computation, with late *global* propagation of results. In the worst case, this approach could result in a speedup of 1, with all processors performing the same computations (the *work complexity* becomes $P \times N$, with P the number of processors and N the size of the state space, while the *time* complexity remains equal to that of a sequential algorithm). Experiments however show that for practical problems this does not occur (Section 6.3.4 and Section 7.4).

CNDFS supports the excellent state compression by means of tree compression as introduced in Chapter 3. The goal posed by Subquestion 2 (Section 1.5.3) of supporting other reduction techniques, such as partial-order reduction, is however not completely met, as we do not present a way to implement the necessary ignoring proviso [EP10]. While we have indications that CNDFS can support it at least to some extend, we opted instead to focus on an important subset of LTL: livelocks. Chapter 8 presents a new parallel DFS$_{\text{FIFO}}$ algorithm [LF13] for solving livelocks, which delivers optimal scalability and at the same time excellent partial-order reductions.

The table below describes the contributions that the current part makes towards solving the goals of the thesis (c.f. Table 1.1 in Section 1.5.3). Scalable and on-the-fly multi-core LTL checking of explicit-state formalisms is now added to the table. State compression is still supported by exchanging the hash table with the lockless tree table of Chapter 3. Only partial-order reduction is not supported because of the difficulty of implementing the ignoring proviso in parallel. The parallel DFS$_{\text{FIFO}}$ algorithm solves this problem for livelocks.

Property	**Explicit state**	**+ On-the-fly**	**+ Compression**	**+ POR**
Reachability	✓	✓	✓	✓
LTL	✓	✓	✓	X
. . Livelocks	✓	✓	✓	✓

Many of the experiments presented in the current part are done on 16-core machines (except in Chapter 7 and Chapter 8). In Chapter 11, we present strong evidence that these methods also scale on 48-core machines, and in absolute terms when compared to SPIN, a leading model checker implementation.

5
Multi-Core Nested Depth-First Search

Alfons Laarman, Rom Langerak, Jaco van de Pol, Michael Weber,
Anton Wijs

Abstract

The LTL Model Checking problem is reducible to finding accepting cycles in a graph. The Nested Depth-First Search (NDFS) algorithm detects accepting cycles efficiently: on-the-fly, with linear-time complexity and negligible memory overhead. The only downside of the algorithm is that it relies on an inherently-sequential, depth-first search. It has not been parallelized beyond running the independent nested search in a separate thread (dual core).

In the current chapter, we introduce, for the first time, a multi-core NDFS algorithm that can scale beyond two threads, while maintaining exactly the same worst-case time complexity. We prove this algorithm correct, and present experimental results obtained with an implementation in the LTSMIN toolset on the entire BEEM benchmark database. We measured considerable speedups compared to the current state of the art in parallel cycle detection algorithms.

About this chapter:

The current chapter is based on the paper *"Multi-core Nested Depth-First Search"*, which was published at ATVA 2011 [Laa+11].

The original text from [Laa+11] has been improved by correcting an error in the multi-core nested depth-first search algorithm with extensions (Algorithm 5.4). In the original version it was not taken into account that the coloring introduced by the *all-red* extension could cause the same early backtracking problem that is discussed in Section 5.4.2. This problem was remedied by an additional wait statement. We thank Wan Fokkink and Stefan Vijzelaar for pointing out this problem. We further extended the discussion on the automata-theoretic approach to model checking, to better support the natural reading order of the current thesis. The general introduction was removed. Finally, we also updated experiments with new benchmarks, as we discovered a bug in the implementation that affected the speedups.

Note that Chapter 7 presents a superior algorithm.

5.1 Introduction

Typically, in order to fully verify whether a system specification adheres to a given temporal property, a model checking algorithm needs to store the entire so-called *state space* in memory. A state space is a directed graph which explicitly describes all potential behavior of the system specification (see Chapter 1). Recent observations [BBR10b] support that research should be focused on achieving faster model checking (MC); currently, memory capacity of the latest hardware allows the analysis of very large state spaces, but the required time to do so is often impractically long.

One advanced MC task is the verification of full Linear Temporal Logic (LTL) properties [BK08]. LTL can be subdivided into two classes of properties: safety properties, e.g. "nothing bad ever happens", and liveness properties, e.g. "eventually something good happens". While safety properties can be handled with so-called *reachability*, which entails visiting all states in the state space reachable from the initial state, liveness properties require a more complicated analysis.

An algorithm introduced by Courcoubetis et al. [Cou+92], often referred to as *Nested Depth-First Search* (NDFS), is particularly useful for checking liveness properties. It has a linear time-complexity and runs *on-the-fly*, i.e. without the need to generate the whole state space, and requires only two bits per state [SE05].

While reachability has been parallelized efficiently in Part II, a linear-time multicore LTL MC algorithm was still unknown. NDFS cannot trivially be adapted to a multicore setting, since it relies on depth-first search (DFS), which is often considered inherently sequential. In particular, the problem of establishing lexicographic DFS postorder (with fixed successor ordering) in a digraph has been shown to be P-complete [Rei85]. As it is generally believed that $\mathsf{P} \neq \mathsf{NC}$, where NC or "Nick's Class" [Coo79; Pip81] represents efficiently parallelizable problems, it is also likely that P-complete problems are not parallelizable.

But even though many other parallel LTL MC algorithms have been introduced over the course of years, none of them exhibits a worst-case linear-time complexity (or even $\mathcal{O}(n \times \log(n))$, with n the number of states) and the complete on-the-fly property [BBR10b; BBR09a; Bar+10].

Recent developments, which we group here under the term *swarm verification* (SV) [HJG08; HJG11], have introduced new DFS-based techniques [Dwy+07; SG03] to perform MC tasks in parallel. Although mainly targeted at distributed-memory settings, in which multiple machines are employed, SV can trivially be used on a multi-core, i.e., shared-memory, machine as well. However, when doing so, the fact that the memory is shared is obviously not exploited.

In the current chapter, we first propose SV-based multi-core NDFS with shared state storage. While this speeds up cycle detection significantly, in the absence of accepting cycles each core still has to traverse the complete state space. Next, we introduce a fine-grained and basic sharing mechanism between threads. Even though parallel search may endanger the correctness of a multi-core NDFS by breaking the postorder, we prove that our algorithm is in fact correct. We subsequently add several known NDFS optimizations [SE05] to the new parallel setting. Finally, we demonstrate its usefulness in practice by comparing many experimental results obtained with an implementation of our algorithm with results obtained with existing parallel LTL MC algorithms.

Contributions. We present the first multi-core on-the-fly LTL model checking algorithm which is linear-time in the size of the input graph, and has a potential speedup greater than two. We provide a rigorous proof of its correctness and many benchmarks. Though the new algorithm does not scale perfectly for all inputs yet, we still believe to have come one step closer to solving the open question, put forth by Holzmann et al. and Barnat et al. [HB07; BBR09a], of finding a time-optimal, scalable, parallel algorithm for accepting cycle detection.

Next, in Section 5.2, the preliminaries behind LTL MC are explained. Related work is discussed in Section 5.3. We propose a multi-core NDFS algorithm, prove its correctness and provide optimizations in Section 5.4. Section 5.5 contains a discussion on the experiments we conducted. Finally, in Section 5.6, considerations are addressed, conclusions are drawn and possibilities for future work are given.

5.2 Background (LTL Model Checking)

LTL MC entails checking that a system under verification $\mathcal{P}$ satisfies an LTL property ϕ, which may be a liveness property that reasons over infinite traces of the system ("eventually something good happens"). We first explain the automata-theoretic approach to this problem, and then discuss an existing algorithm to solve it.

5.2.1 The Automata-Theoretic Approach to LTL Model Checking

LTL model checking is usually performed following the automata-based approach originating from [VW86] that proceeds in several steps. In the current chapter, we focus only on the last step of the process that can be reduced to a graph problem: given a graph representing the synchronized product of the Büchi property automaton and the state space of the system, find a cycle containing an accepting state. Any such identified cycle determines an infinite execution of the system violating the LTL formula. In the current chapter, we will only reason about *Büchi automata* that result from the *synchronous product* of a Büchi property automaton and a system graph describing the dynamic behavior of the modeled system.

Definition 5.1. *A* Büchi automaton *(BA) is a quadruple* $\mathcal{B} = (\mathcal{S}, s_I, \textsc{next-state}(), \mathcal{F})$*, with* $\mathcal{S}$ *a finite set of states,* s_I *the initial state,* post: $\mathcal{S} \to 2^{\mathcal{S}}$ *the successor function, and* $\mathcal{F} \subseteq \mathcal{S}$ *a set of accepting states.*

The use of the $\textsc{next-state}()$ function, instead of a *transition relation*, reflects the fact that this cross product can be generated on-the-fly [VW86].

Notations. Let $\mathcal{B} = (\mathcal{S}, s_I, \textsc{next-state}(), \mathcal{F})$ be a BA. If for $s, t \in \mathcal{S}$, we have $t \in \textsc{next-state}(s)$, then we can also write $s \to t$. The reflexive transitive closure of $\to$ is denoted by $\to^*$, and the transitive closure by $\to^+$. We call $s \to^* t$ and $s \to^+ t$ *paths* through $\mathcal{B}$, i.e. sequences of states connected by the successor function. Sometimes we interpret a path π as a set of states, and write $s \in \pi$, meaning that $s \in \mathcal{S}$ is included in the sequence of states of π. A *run* through $\mathcal{B}$ is an infinite path starting at s_I. Finally, we call a run π *accepting* if and only if for infinitely many $s \in \pi$, we have $s \in \mathcal{F}$. Checking the existence of such a run is called the *emptiness problem*.

To check an LTL property ϕ on $\mathcal{P}$, it suffices to solve the emptiness problem for the product of the state graph $\mathcal{G}_{\mathcal{P}}$ and the Büchi automaton $\mathcal{B}_{\neg\phi}$ (e.g. [VW86]). Here, $\mathcal{G}_{\mathcal{P}}$ is an explicit representation of all possible behavior of $\mathcal{P}$ in the form of a graph, and $\mathcal{B}_{\neg\phi}$ is the Büchi automaton accepting all infinite paths described by the negation of ϕ. A counterexample for ϕ in $\mathcal{B} = \mathcal{G}_{\mathcal{P}} \times \mathcal{B}_{\neg\phi}$ exists iff there exists some $a \in \mathcal{F}$ such that $s_I \to^* a$ and $a \to^+ a$ (i.e. there is an accepting run), where the latter is called an "accepting cycle". Hence, solving the emptiness problem corresponds with determining the reachability of an accepting cycle.

The fact that the cross product can be generated *on-the-fly* avoids that we have to generate (and store) $\mathcal{G}_{\mathcal{P}}$ in its entirety, before calculating the cross product. Moreover, the LTL MC procedure – we discuss one in the next section – can terminate early when a counterexample is found, often ensuring that only a small part of $\mathcal{B}$ needs to be explored and stored.

5.2.2 Sequential LTL Model Checking Algorithms

The first linear-time algorithm to detect accepting runs was proposed by Courcoubetis et al. [Cou+92] and, today, is often referred to as *nested depth-first search* (NDFS). In the current chapter, we propose a *multi-core* NDFS (MC-NDFS).

Following [Boš02], we first discuss a non-linear algorithm (Algorithm 5.1) to illustrate the principle. It performs an outer search, called dfs_blue, to find accepting states (see Line 16). This *blue search* marks states on the stack cyan (Line 12) and visited states blue (Line 19), hence its name (note that initially, all states are white). The *nested search* (dfs_nested) then searches for a cycle over the accepting state, which we refer to as the *seed* of the search. It may search the entire state space as it always starts with an empty visited set R (Line 17). When it encounters a cyan state (Line 7), it found an accepting cycle, as the cyan stack of the blue search leads to the seed [HPY96]. Conversely, it is easy to see that if a cycle exists, it is reported, as an *independent* nested search is launched for all accepting states.

The obvious problem with Algorithm 5.1 is that it is quadratic in the size of the state space: For each (accepting) state, the entire state space may be visited in the nested search. Realizing that dfs_blue sorts the accepting states in *DFS postorder*, i.e. dfs_nestedis called in the backtrack of dfs_blue, we may see that this is entirely not necessary. For if a nested search from a seed a encounters a state s from a previous nested search from a', and a lies on a cycle with s, then so must a' lie on a cycle with s. If this is not the case, it would contradict the fact that a' was processed before a in the postorder (see [Tar72] for a detailed explanation). Since the search from a' did not encounter a

Algorithm 5.1 A nested accepting cycle detection algorithm (non-linear)

```
 1 proc ncd($s_I$)
 2   dfs_blue($s_I$)
 3   report no cycle
 4 proc dfs_nested($s$)
 5   $R := R \cup \{s\}$
 6   for all $t$ in NEXT-STATE($s$) do
 7     if $t.color$=$cyan$
 8       report cycle & exit
 9     else if $t \notin R$
10       dfs_nested($t$)
11 proc dfs_blue($s$)
12   $s.color$ := $cyan$
13   for all $t$ in NEXT-STATE($s$) do
14     if $t.color$=$white$
15       dfs_blue($t$)
16   if $s \in \mathcal{F}$
17     $R := \emptyset$
18     dfs_nested($s$)
19   $s.color$ := $blue$
```

Algorithm 5.2 An adapted New NDFS algorithm

```
 1 proc nndfs($s_I$)
 2   dfs_blue($s_I$)
 3   report no cycle
 4 proc dfs_red($s$)
 5   for all $t$ in NEXT-STATE($s$) do
 6     if $t.color$=$cyan$
 7       report cycle & exit
 8     else if $t.color$=$blue$
 9       $t.color$ := $red$
10       dfs_red($t$)
11 proc dfs_blue($s$)
12   $s.color$ := $cyan$
13   for all $t$ in NEXT-STATE($s$) do
14     if $t.color$=$white$
15       dfs_blue($t$)
16   if $s \in \mathcal{F}$
17     dfs_red($s$)
18     $s.color$ := $red$
19   else
20     $s.color$ := $blue$
```

cycle through s (if it did, it would have terminated, contradicting a subsequent search from a). In the nested search for a, the algorithm therefore does not have to explore s, or any other s' visited in a previous nested search from some seed a' (processed earlier in the postorder). In other words, R does not have to be emptied at Line 17.

The basic NDFS algorithm is thus equivalent to Algorithm 5.1 without Line 17.[5.1] Over the years, extensions to NDFS have been proposed in, e.g., [HPY96; SE05; GS09]. We build on the *New NDFS* (NNDFS) algorithm from [SE05] (Algorithm 5.2). It improves NDFS by combining the blue, cyan and red color in a single 2-bit variable (here: *color*). The algorithm now delays the red coloring of the seed until after the red search (Line 18), so it stays cyan during the search for cycle detection at Line 6. Algorithm 5.2 does not include the *early cycle detection* in dfs_blue from the original NNDFS, for this extension does not contribute to the understanding of MC-NDFS. In Section 5.4.4, we retrofit MC-NDFS with this and other extensions.

NNDFS thus runs in linear time, since each reachable state is at most visited twice, once in the blue DFS and once in a red DFS. An intuitive proof of correctness is given in [Cou+92]. In [GS09], a standalone correctness proof is given for NNDFS with early cycle detection and an extension called *allred* (both are explained in Section 5.3). Section B.1 gives a detailed correctness proof for a roughly equivalent algorithm, which may serve as an introduction to the proof of MC-NDFS later in the current chapter.

[5.1] Although Algorithm 5.1 already contains the extension to detect cycles via the cyan stack [HPY96].

5.3 Related Work

Two prominent classes of linear-time algorithms to detect accepting runs are formed by the *NDFS-based* and the *strongly connected component* (SCC)-*based* algorithms (explained below). The performance of both classes of algorithms is known to be similar, up to some exceptions: Algorithms in the NDFS class use less memory, while algorithms in the SCC class tend to find counterexamples faster [GV04; SE05; GS09]. Since we propose an NDFS-based algorithm, the emphasis here is on related work in the NDFS class. Finally, we also discuss breadth-first search (BFS)-*based* algorithms.

SCC-based algorithms. Strongly connected components (SCCs) are the subgraphs of a graph in which each state can reach all other states [Tar72], informally speaking. An *non-trivial SCC* contains at least one transition. Non-trivial SCCs with accepting state therefore contain an accepting cycle. Hence, several researchers suggested the use of Tarjan's algorithm [Tar72] to find accepting cycles. To make the algorithm more on-the-fly several extensions have been introduced [Cou99; GV04].

NDFS. As mentioned in Section 5.2, NDFS was introduced in [Cou+92]. There, a correctness proof is given based on the fact that red DFSs are initiated for accepting states based on the postorder enforced by the blue DFS. Holzmann et al. [HPY96] observe that it suffices in a red DFS to check the reachability of a state currently on the stack of the blue DFS, i.e. a state colored cyan in NNDFS, since such a state can reach the accepting state which initiated the current red DFS, closing an accepting cycle.

Schwoon and Esparza [SE05] combine all of the above extensions and observe that some combinations of colors can never occur. This allows them to introduce a *two-bit color encoding*, also encoding a cyan color for states on the stack of the blue DFS. Finally, Gaiser and Schwoon [GS09] introduce the *allred* extension and give a standalone proof for their NNDFS. The *allred* extension incorporates an additional check in the blue DFS: if all successors of a state *s* are red, then *s* can be colored red as well. This avoids some calls of dfs_red. We will show later that for our MC-NDFS, this extension is very useful.

Parallel NDFS. Holzmann and Bošnački [HB07] proposed a dual-core NDFS (NDFS-2) based on the observation that a transition initiating a red DFS is an "irreversible state transition", i.e. it splits the state graph. A new thread is launched to handle the red DFS. Since both DFSs are still inherently sequential, the number of threads cannot exceed two, and both potentially have to search the entire state graph. Courcoubetis et al. already mentioned that the two DFSs could be interleaved.

Prominent model checking approaches primarily aimed at settings with distributed memory, e.g., when using a cluster or grid, are swarm verification (SV) [HJG08; HJG11] and *Parallel Randomized* DFS [Dwy+07; SG03] (PRDFS). These are so-called *embarrassingly parallel* [Fos95] techniques, since the individual workers operate fully independently, i.e. without communication with the other workers. From here on, when mentioning SV, we refer to existing SV and PRDFS techniques. Note that the search direction of a DFS is determined by the order in which states are selected for exploration from NEXT-STATE(s) (for any $s \in \mathcal{S}$), e.g. on Line 13 of Algorithm 5.2. In SV, basically each worker performs a DFS with a unique ordering of the successor states. In this way, workers explore different parts of the reachable state graph first. This method has proven to be very successful for bug-hunting. In the absence of bugs, though, the graph will be explored N times, with N the number of workers, since the workers are unaware of each other's results. Although not explicitly mentioned before, SV can be performed in a multi-core setting as well with each worker performing the NDFS algorithm.

BFS-based methods. Several other LTL MC methods exists which are not DFS-based. Instead these algorithms rely on BFS techniques [BBC03a] and are therefore easier to parallelize, even in a distributed setting. On the down side, the linear-time complexity and on-the-fly property is often lost. All of these algorithms have been designed for the distributed setting and some were ported to multi-core machines [BBR10b; Bar+10] (namely: OWCTY, MAP and OTF_OWCTY).

Negative Cycle (NEGC) [Bri+01] uses a similar fixed-point approach, but instead propagates a negative index from accepting states. The accepting cycle detection problem is thus reduced to finding negative cycles. The algorithm is not on-the-fly and the performance has been found to be inferior to other solutions [Bri+04].

Every accepting cycle contains a back-level transition, which jumps back from a state that is l levels from the initial state, to a state that is $\leq l$ levels from the initial state. *Back-Level Edge* (BLEDGE) [BBC03b] uses this information to find the cycles with a fixed point computation. It is not on-the-fly and its performance has been found to be meager in practice [Bri+04]. An on-the-fly version of the algorithm has however been developed in [BBC05b].

One-Way-Catch-Them-Young (OWCTY) [ČP03], repeatedly removes states that cannot be part of an accepting cycle. It is sufficient to only remove states without (not yet removed) successors and states that have no accepting predecessors. To compute these, the algorithm propagates the number of preceding accepting states. The algorithm is also not on-the-fly at all.

Maximal Accepting Predecessor (MAP) [BBR10b] performs multiple forward reachability computations to propagate the preceding accepting state with a maximal index

Table 5.1: Sequential and multi-core LTL MC algorithms and their worst-case time complexity, scalability, and on-the-fly property. ($\mathcal{T}$ is the set of reachable transitions, $|\mathcal{S}|$ the number of states, $|\mathcal{F}|$ the number of accepting states, and h the height of the Scc quotient graph [ČP03].)

Algorithm	Source	Time complexity	Scalability	On-the-fly
Ndfs	[Cou+92]	$\mathcal{O}(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert)$	1 core	Yes
Couvreur-Tarjan	[Cou99]	$\mathcal{O}(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert)$	1 core	Yes
GV-Tarjan	[GV04]	$\mathcal{O}(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert)$	1 core	Yes
Ndfs-2	[HB07]	$\mathcal{O}(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert)$	2 cores	Yes
Negc	[Bri+01]	$\mathcal{O}(\vert\mathcal{S}\vert\cdot\vert\mathcal{T}\vert)$	N cores	No
Map	[BBR10b]	$\mathcal{O}(\vert\mathcal{F}\vert^2\cdot\vert\mathcal{T}\vert)$	N cores	Heuristic
Bledge	[BBC03b]	$\mathcal{O}(\vert\mathcal{T}\vert\cdot(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert))$	N cores	No
Otf_Bledge	[BBC05b]	$\mathcal{O}(\vert\mathcal{T}\vert\cdot(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert))$	N cores	Heuristic
Owcty	[ČP03]	$\mathcal{O}(h\cdot\vert\mathcal{T}\vert)$	N cores	No
Otf_Owcty	[BBR09a]	$\mathcal{O}(h\cdot(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert))$	N cores	Heuristic
Mc-ndfs	Chapters 5, 6 7[5.2]	$\mathcal{O}(N\cdot(\vert\mathcal{S}\vert+\vert\mathcal{T}\vert))$	N cores	Yes

for every state. This computation reaches a fixed point after a while, at which moment the maximal accepting predecessor of at least one accepting state on a cycle, has to be itself. Map preserves the on-the-fly property to the extent that it is heuristic: cycles can be detected early (when an accepting state finds itself as maximal predecessor), but this is not guaranteed.

By combining Map with Owcty, the same property is transferred to the new *On-The-Fly One-Way-Catch-Them-Young* (Otf_Owcty) algorithm. For the important class of weak LTL, the algorithm has been shown to be time-optimal [BBR09a], therefore it is the current state of the art in multi-core LTL MC.

Table 5.1 gives a brief overview of all sequential and parallel LTL MC algorithms discussed so far with their worst-case complexities and on-the-fly behavior. In the subsequent sections (and later chapters), we develop a multi-core version of Ndfs (Mc-ndfs) that scales to more than 2 cores and inherits the on-the-fly property of the original algorithm. The scheduling approach of this algorithm is optimistic, based on on SV, but with communication, thus the complexity ranges from 1 time the complexity of the original Ndfs, to N times its complexity.

5.4 Multi-Core Ndfs

5.4.1 A Basic Multi-Core Swarmed Ndfs

As already mentioned, SV is compatible with a shared-memory setting. However, the independence of workers in SV may result in duplicated states on the different machines, hence, when mapped naively to a multi-core machine, the shared memory is not exploited. Therefore, we store all states in a shared lockless hash table or tree table, which have been shown to scale well for this purpose in Part II.

A basic SV NDFS algorithm executes an instance of Algorithm 5.2 for each worker i with thread-local color variables. The two bits needed per state per worker are small compared to the state itself and for a dozen or so workers, memory usage is still lower than for SCC-based algorithms [SE05]. Local permutations of the NEXT-STATE function direct workers to different regions of the state graph, resulting in fast bug-finding typical for SV. With NEXT-STATE$_i^b$ (NEXT-STATE$_i^r$), we denote the permutation of successors used in the blue (red) DFS by worker i. For inputs without accepting cycles this solution does not scale. In the next section, we attack this problem.

5.4.2 Multi-Core Ndfs with Global Coloring

A naive sharing of colors between multi-core workers is prone to influence the independent postorders on which the correctness of the NDFS algorithm relies [Cou+92]. In the current section, we present a color-sharing approach which preserves correctness. The next section provides a correctness proof of this MC-NDFS algorithm.

The basic idea behind MC-NDFS in Algorithm 5.3 is to share information in the backtrack of the red DFSS (dfs_red). A new (local) color *pink* is introduced to signify states on the stack of a red DFS, analogous to cyan for a blue DFS. When a red DFS backtracks, the states are globally colored red. These red states are now ignored by both *all* blue and red DFSS, thus pruning the search spaces for all workers i.

Additionally, we count the number of workers that initiate dfs_red in $s.count$ (Line 10) and wait with backtracking until this counter is 0 (Line 21,22). This enforces that if multiple workers call dfs_red from the same accepting state, they will finish simultaneously. Figure 5.1 illustrates the necessity of this synchronization by a simple counterexample that could occur in absence of this synchronization.

A worker 1 could explore a, b, u, v, w, backtrack from w, explore t and backtrack all the way to the accepting state b where it will call a dfs_red at Line 11. Then this dfs_red$(b, 1)$ could explore u, v, w and halt for a while. Now, a worker 2 could start

[5.2] And Evangelista et al. (see Section 5.6).

Algorithm 5.3 A Multi-core NDFS algorithm, coloring globally red in the backtrack

```
proc mc_ndfs($s$,$N$)
  dfs_blue($s, 1$)$\|$..$\|$dfs_blue($s, N$)
  report no cycle

proc dfs_blue($s$,$i$)
  $s.color[i]$ := $cyan$
  for all $t$ in NEXT-STATE$_i^b(s)$ do
    if $t.color[i]$=$white \land \neg t.red$
      dfs_blue($t$,$i$)
  if $s \in \mathcal{F}$
    $s.count$ := $s.count$ + 1
    dfs_red($s$,$i$)
  $s.color[i]$ := $blue$

proc dfs_red($s$,$i$)
  $s.pink[i]$ := true
  for all $t$ in NEXT-STATE$_i^r(s)$ do
    if $t.color[i]$=$cyan$
      report cycle & exit all
    if $\neg t.pink[i] \land \neg t.red$
      dfs_red($t$,$i$)
  if $s \in \mathcal{F}$
    $s.count$ := $s.count$ − 1
    await $s.count$ = 0
  $s.red$ := true
  $s.pink[i]$ := false
```

dfs_red$(b, 2)$ in a similar fashion. Next, it could explore w, v, u, backtrack, mark u red and halt for a while. Then worker 1 continues to mark w red.

Note that the two accepting cycles contain red states, but both workers can still detect a cycle by continuing to explore v and t (b is cyan in the local coloring of both workers). However, a third worker can endanger this potential, while the first two workers halt for a while. After worker 3 searches a and subsequently t and b in a blue DFS, it will start a dfs_red at b, but because its successors are now red, worker 3 will backtrack and mark b red. Note that exactly this step is prevented by adding the **await** statement. Continuing with dfs_red$(a, 3)$, states t and a will also become red, obstructing workers 1 and 2 from finding a cycle.

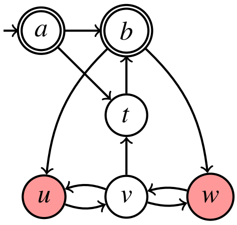


Figure 5.1: Counterexample to correctness of MC-NDFS without **await** statement.

No worker finds a cycle in this way, which thus constitutes a counterexample for correctness. However, because worker 3 is forced to wait for the completion of the red Dfss of workers 1 and 2 before it can backtrack from state b in dfs_red$(b, 3)$, this counterexample is invalid for Mc-ndfs.

Finally, we note that Mc-ndfs in Algorithm 5.3 is presented in a form that eases analysis of correctness: without superfluous details. For example, the *pink* variable of states is separate from the *color* variable, which stores only the colors white, blue and cyan. The two-bit color encoding of [SE05] is thus dropped for a while. In the following section, we prove correctness of Mc-ndfs, after which we amend the algorithm in Section 5.4.4 with the extensions discussed in Section 5.3. The *allred* extension is shown to improve sharing between workers significantly.

5.4.3 Correctness Proof

In the current section, we provide a correctness proof for Mc-ndfs. For brevity and understandability this proof is kept brief: Some lemmas are given at upfront, and the reasoning in the proofs is kept coarse. A detailed proof can be found in Appendix A.1. We assume that each line of the code above is executed atomically. The global state of the algorithm is the coloring of the input graph $\mathcal{B}$ and the program counter of each worker.

We use the following notations: The sets $White_i$, $Cyan_i$, $Blue_i$ and $Pink_i$ contain all the states colored white, cyan, blue, and pink by worker i, and Red contains all the red states. E.g., if $s.color[i] = blue$, we write $s \in Blue_i$. It follows from the assignments of the respective colors to the *color* variable that $White_i$, $Cyan_i$ and $Blue_i$ are disjoint. Also, we denote the state of one worker as dfs_red$(s, i)@X$, meaning that worker i is executing l.X in dfs_red for a state s. Finally, we use the modal operator $s \in \Box X$ to express that $\forall t \in$ next-state$(s) : t \in X$.

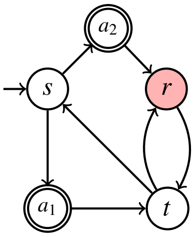


Figure 5.2: An obstructed accepting cycle.

Correctness of Mc-ndfs hinges on the fact that it will never miss all reachable accepting cycles, i.e. it will always find one if one exists. Recall from Section 5.2 that Ndfs ensures that all reachable states are visited only once by both dfs_blue and dfs_red. Mc-ndfs ensures that each reachable state is visited *at least* once by both some dfs_blue and dfs_red, therefore for a reachable $a \in \mathcal{F}$, there is at least one dfs_red(a,i)@11 for some i, that initiates the recursion of the dfs_red.

This recursion continues at Line 19, where it tries to find a $t \in Cyan_i$ at Line 16 that would close the cycle. Now, if the cycle $a \rightarrow^+ a$ exists, worker i will either find a $t \in Cyan_i$, or is obstructed because it encounters a $t \in Red$ at Line 18. Figure 5.2 illustrates that workers can obstruct each other from finding cycles. For example, it is possible that a worker 1 initiates a dfs_red for a_1, marking r red. Then, a worker 2, with a different next-state_i^b, could start a dfs_red for a_2 and be obstructed from finding cycle $\{a_2, r, t, s\}$.

We first state invariants that express basic relations between the colors in Mc-ndfs. These invariants are proven in the full proof which is presented in Appendix A.1. Then, after Lemma 5.1, we prove the crucial insight (Theorem 5.1), the algoritm's termination (Theorem 5.2) and soundness and completeness (Theorem 5.3), i.e. when a counterexample is reported, an accepting cycle indeed exists, and when an accepting cycle exists in the graph, the algorithm will report a counterexample.

L1. $\forall i\colon Blue_i \cup Pink_i \subseteq \Box(Blue_i \cup Cyan_i \cup Red)$ (see Lemma A.6 and A.12)

L2. $Red \subseteq \Box(Red \cup \bigcup_i (Pink_i \setminus Cyan_i))$ (see Lemma A.10)

L3. $\forall i, a \in \mathcal{F}\colon a \in Blue_i \implies a \in Red$ (see Lemma A.13)

L4. $\forall i, a \in \mathcal{F}\colon a \in (Pink_i \setminus Red) \implies a \in Cyan_i$ (see Lemma A.15)

L5. $\forall i\colon Pink_i \subseteq (Blue_i \cup Cyan_i)$ (see Lemma A.11)

Lemma 5.1. *The following invariant holds for Mc-ndfs:* $\forall s \in Red, a \in \mathcal{F} \setminus Red\colon s \rightarrow^* a \implies (\exists i, p \in Pink_i, c \in Cyan_i\colon s \rightarrow^+ p \xrightarrow{\neg Red}{}^{+} c \rightarrow^* a)$

Proof. We show that the property follows from the previous invariants L1–4. Assume $s \rightarrow^* a$ for some $s \in Red$ and $a \in \mathcal{F}$ with $a \notin Red$. Let $s' \in Red$ be the *last* red state on the path $s \rightarrow^* a$. Then, since $s' \neq a$, it has a successor $t \notin Red$ in this path. By L2 we obtain $t \in Pink_i$ for some worker i, so let $p := t$.

Note that $t \neq a$, otherwise by L4 $t \in Cyan_i$ and by L2 $t \notin Cyan_i$. So we find another successor t' such that $s \rightarrow^* s' \rightarrow t \rightarrow t' \rightarrow^* a$. Assume towards a contradiction that no state on the path $t' \rightarrow^* a$ is in $Cyan_i$; recall that $t' \rightarrow^* a$ contains no Red states either. Then by L1, all states on $t' \rightarrow^* a$ are in $Blue_i$. But then also $a \in Blue_i$ and by L3, $a \in Red$, a contradiction. So there exists a $c \in Cyan_i$ with $s \rightarrow^* p \rightarrow^+ c \rightarrow^* a$. $\Box$

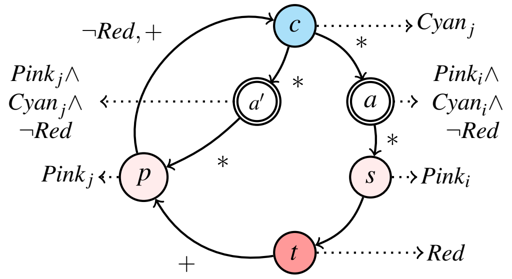


Figure 5.3: Snapshot of the cycle in the *last* “obstructed cycle search”. Edges with $*, +$ indicate paths of length ≥ 0 and > 0. Dotted arrows denote node colors and $\neg Red, +$ a path without red.

Theorem 5.1. *Mc-ndfs cannot miss all accepting cycles.*

Proof. Assume an Mc-ndfs run would miss all accepting cycles. Since there are only finitely many cycles, we can investigate the *last* “obstructed cycle” in this run, i.e., the last time that a dfs_red (which originated from some accepting state a on a cycle) encounters *Red*. That is, we are in dfs_red(s,i)@18 but we see $t \in Red$, although $s \to t \to^* a$.

Note that $a \notin Red$: Just before dfs_red(a,i)@11, $a.count$ was increased by Line 10. Therefore, no other worker can make a red, because they are all forced to wait at Line 22.[5.3]

Hence we can apply Lemma 5.1, to obtain a path $p \xrightarrow{\neg Red}{}^{+} c$ for some $p \in Pink_j$ and $c \in Cyan_j$. It follows that there is an $a' \in \mathcal{F}$ with $c \to^* a' \to^* p$ (property of dfs stacks). Figure 5.3 provides an overview of the shape of the subgraph that we just discussed with the deduced colorings.

But now we have constructed a cycle for worker j which has not yet been obstructed. This contradicts the fact that we were considering the *last* obstructed cycle. We conclude that there is no last obstructed cycle, hence there exists no run that misses all cycles. $\square$

This proves partial correctness of Mc-ndfs. In order to prove that an accepting cycle will eventually be reported, the algorithm is required to terminate.

[5.3] A race condition can occur here, because worker i could increase $a.count$ right after some worker j passed the check at Line 22 in dfs_red(a,j). Next, worker i would start its dfs_red(a,i), and find that $a \in \Box(Red)$. So i will also make a red and return from dfs_red. It does not matter whether i or j makes a red first. Therefore, we can safely ignore such race conditions.

Theorem 5.2. *Mc-ndfs always terminates with some report at Line 3 or Line 17.*

Proof. Assuming dfs_red terminates, we can conclude termination of dfs_blue from the fact that for each worker i the set $Blue_i \cup Cyan_i$ grows monotonically (blue is never removed). Eventually, all the states are in the set and the blue search ends. Termination of the **await** statement at Line 22 state follows from the basic observation that every worker i can have at most one counter increment on some accepting state, which is decremented at Line 21 before waiting. Hence, when worker i is waiting, there can be no other worker waiting for i. Finally, all red dfss terminate because also the set $Red \cup Pink_i$ grows monotonically. □

Theorem 5.3. *Mc-ndfs reports* **cycle** *if there exists a reachable accepting cycle in the input graph $\mathcal{B}$ and it reports* **no cycle** *otherwise.*

Proof. By Theorem 5.2, the algorithm terminates with some report. If a cycle is reported at Line 17 by worker i, we find an $s \in Pink_i$ and $t \in Cyan_i$ with $s \to t$. In that case there is a state $a \in \mathcal{F}$ on the stack such that $t \to^* a \to^* s \to t$, so there is indeed an accepting cycle.

Otherwise, if no cycle is reported at Line 3, all workers have terminated without reporting a cycle. By Theorem 5.1 there is no accepting cycle in the graph. □

5.4.4 Extensions

We can improve Mc-ndfs further. Algorithm 5.4 presents Mc-nndfs, which is Mc-ndfs with the extensions discussed in Section 5.3. First, we opted to extend Mc-ndfs with *allred* [GS09] (Line 16 and Line 24–28). Since the parallel workload of the Mc-ndfs algorithm depends entirely on the proportion of the state graph that can be marked red (see Section 5.5.2), *allred* can improve the scalability. Second, early cycle detection in dfs_blue (Line 19–21) is needed to compete with Scc-based algorithms. Finally, the introduction of the two-bit color-encoding from [SE05] for each worker will eliminate the extra bit per worker used for the pink color.

Sketch of correctness. The *allred* extension in dfs_blue introduces a new red coloring of a state s at Line 28, affecting the proof of Lemma 5.1. But, since $s \in \Box(Red)$, the induction hypothesis can be applied for the successor t of s. Furthermore, the proof of Theorem 5.1 also depends on the seed a not becoming red, i.e. $a \notin Red$, while other workers are still performing a dfs_red for it. The *allred* coloring introduces a new opportunity where this might happen. Therefore, *allred* coloring at Line 27 should be delayed until $a.count = 0$, just like at Line 13.

Algorithm 5.4 Mc-ndfs with extensions (Mc-nndfs)

```
1  proc mc_ndfs($s$,$N$)
2    dfs_blue($s,1$)||..||dfs_blue($s,N$)
3    report no cycle

4  proc dfs_red($s$,$i$)
5    $s.color[i]$ := $pink$
6    for all $t$ in NEXT-STATE$^r_i(s)$ do
7      if $t.color[i]$=$cyan$
8        report cycle & exit all
9      if $t.color[i] \neq pink \wedge \neg t.red$
10       dfs_red($t$,$i$)
11   if $s \in \mathcal{F}$
12     $s.count$ := $s.count - 1$
13     await $s.count$=0
14   $s.red$ := true

15 proc dfs_blue($s$,$i$)
16   allred := true
17   $s.color[i]$ := $cyan$
18   for all $t$ in NEXT-STATE$^b_i(s)$ do
19     if $t.color[i]$=$cyan$ $\wedge$
20         $(s \in \mathcal{F} \vee t \in \mathcal{F})$
21       report cycle & exit all
22     if $t.color[i]$=$white \wedge \neg t.red$
23       dfs_blue($t$,$i$)
24     if $\neg t.red$
25       allred := false
26   if allred
27     await $s.count$ = 0
28     $s.red$ := true
29   else if $s \in \mathcal{F}$
30     $s.count$ := $s.count$ + 1
31     dfs_red($s$,$i$)
32   $s.color[i]$ := $blue$
```

Due to the early cycle detection at Line 19–21, some accepting cycles can be detected already in the blue search: At Line 20, via the properties of the blue dfs stack, we have: $s_I \rightarrow^* t \rightarrow^* s \rightarrow t$ with $t \in \mathcal{F} \vee s \in \mathcal{F}$.

The two-bit color encoding overwrites the value of the $s.color[i]$ at Line 5. However, L5 shows that only $Cyan_i$ and $Blue_i$ are affected (not $White_i$). The removal of s from $Blue_i$ does not affect dfs_red, since it is insensitive to $Blue_i$. The removal of s from $Cyan_i$ seems more problematic, since cycle detection on Line 7 depends on it. However, we also know that the only case where s is removed from $Cyan_i$, is in the initial dfs_red call from Line 11 (recursive dfs_red calls are never made on $Cyan_i$ states, since a cycle would be detected at Line 16 and Line 19 would not have been reached). Hence, $s \in \mathcal{F}$. It turns out that if there exists a path $\pi \equiv s \rightarrow^* s$ with $(\pi \setminus s) \cap Cyan_i = \emptyset$, this accepting cycle would have been detected by early cycle detection in dfs_blue ($s_I \rightarrow^* s \rightarrow^* s' \rightarrow s$ with $s \in \mathcal{F}$). Hence, we do not need any provisions to *fix* the removal of s from $Cyan_i$. This fact was overlooked by Schwoon et al.[SE05; GS09], leading them to complicate their Nndfs algorithm (Algorithm 5.2) with delayed red coloring of accepting states.

5.5 Experiments

We implemented NNDFS, multi-core SV NNDFS and MC-NNDFS in the multi-core backend of the LTSMIN model checking tool suite [LPW11a]. This enabled us to use the same input models (without translation) and the same language frontend (compiler). We also implemented randomized NEXT-STATE$_i$ functions to direct threads to different regions of the state space, as discussed in Section 5.4.1.

We performed experiments on an AMD Opteron 8356 16-core (4 × 4 cores) server with 64 GB RAM, running a patched Linux 2.6.32 kernel. All tools were compiled using gcc 4.4.3 in 64-bit mode with high compiler optimizations (`-O3`). For comparison purposes, we used all 453 models with properties of the BEEM database [Pel07]. To mitigate random effects in the benchmarks, runtimes are always averaged over 6 benchmark runs. We compared MC-NNDFS against multi-core SV NNDFS to answer the question whether a more integrated multi-core approach can win against an *embarrassingly parallel* algorithm. Furthermore, we compared with the best existing parallel LTL MC algorithm OTF_OWCTY, as implemented in DIVINE 2.5.1 [Bar+10].

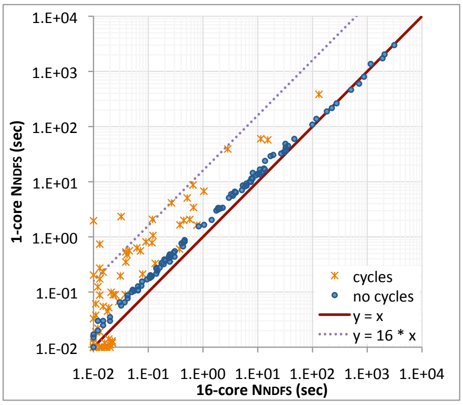
Due to the *on-the-fly* nature of LTL algorithms, we distinguish models containing accepting cycles from models that do not contain them. On the former set, algorithms that build the state space *on-the-fly* and terminate early when a counterexample can be found, are expected to perform very well.

5.5.1 Models with Accepting Cycles

We demonstrate the merits of multi-core SV NNDFS by comparing the runtimes with the sequential NNDFS. As expected, SV speeds up the detection of accepting cycles (crosses in Figure 5.4) significantly compared to sequential NNDFS runs. We do not expect to see perfect speedups (16× on 16 cores) across all benchmarks, since the search is undirected and some threads traverse parts of the state space which do not contribute to finding a cycle. However, for some models, multi-core SV NNDFS does exhibit perfect speedups, or even superlinear speedups. Due to randomization, multiple workers are more likely to find counter examples[Dwy+07; SG03].

Both multi-core SV NNDFS and MC-NNDFS find accepting cycles roughly within the same time (Figure 5.5), there is only a small edge for MC-NNDFS (most crosses are in the upper half of the figure), due to work sharing effects. Apparently, the global red coloring does not cause much “obstruction” (see Section 5.4.3).

We isolated those runs of MC-NNDFS on models with cycles, that have a runtime longer than 0.1 sec, because only those yield meaningful scalability figures. Figure 5.6 on the next page shows that these models scale very well (the figure is cut off after a



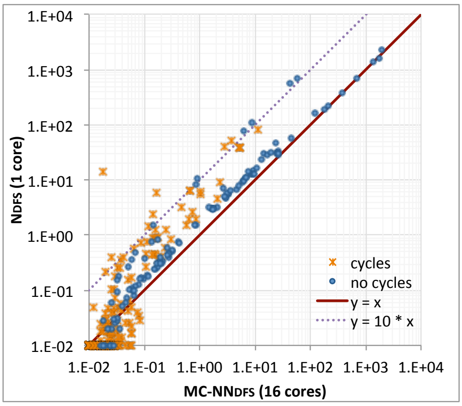


Figure 5.4: Log-log scatter plot of multi-core SV NNDFS/ sequential NNDFS runtimes.

Figure 5.5: Log-log scatter plot of MC-NNDFS/ multi-core SV NNDFS runtimes.

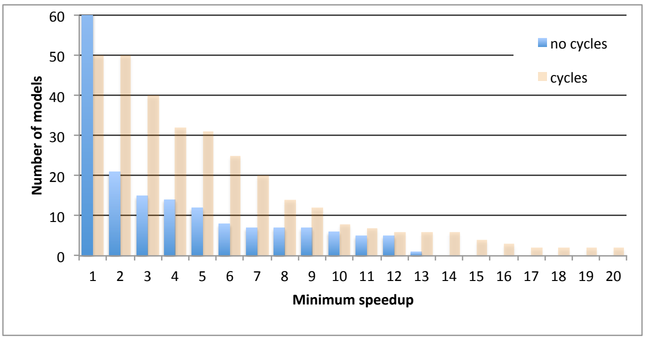


Figure 5.6: Model counts of speedups with MC-NNDFS (base case: sequential NNDFS)

speedup of 20, but it extends well beyond speedups of 100). Out of 50 models with cycles (and runtimes ≥ 0.1 sec), ≈ 50 % exhibit at least six-fold speedups and a few exhibit superlinear speedups (factor > 16).

Finally, a comparison with OTF_OWCTY unsurprisingly shows that MC-NNDFS finds counterexamples much faster (crosses in Figure 5.7), due to its depth-first on-the-fly nature, while OTF_OWCTY is only heuristically on-the-fly.

5.5.2 Models without Accepting Cycles

For models without accepting cycles, on-the-fly algorithms lose their edge over other algorithms, as the state space has to be traversed fully. We demonstrate this with our multi-core SV NNDFS benchmark runs, which degrade timewise to sequential NNDFS (dots in Figure 5.4). We note that multi-core SV NNDFS causes little overhead compared to the sequential NNDFS version, hence it would be safe to run multi-core SV if the presence of a counterexample is uncertain.

However, when comparing multi-core SV NNDFS against MC-NNDFS (Figure 5.5), we observe significant speedups, in some cases more than ten-fold (dotted line) on 16 cores. Again, we isolated the runs of MC-NNDFS on models without cycles that run more than 0.1 sec (Figure 5.6). We observed at least ten-fold speedups for 5 models out of 91 such models (the y axis of the figure is cut off). In the BEEM database, we verified the nature of the 20 models that exhibit speedup greater than factor two. These include: *leader election* and other *communication protocols*, *hardware models*, *controllers*, *cache coherence protocols* and *mutual exclusion algorithms* (see Table 5.2).

Figure 5.7 reveals that MC-NNDFS can mostly keep up with the performance of

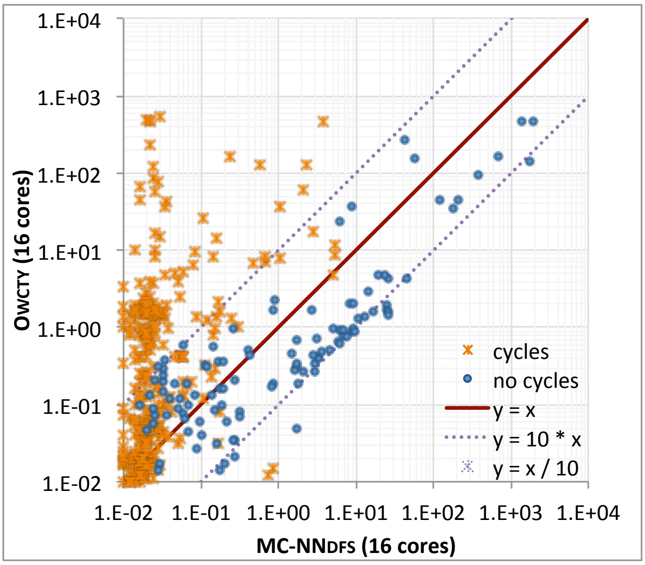


Figure 5.7: Log-log scatter plot of MC-NNDFS/ OTF_OWCTY runtimes. Above the diagonal MC-NNDFS wins, below OTF_OWCTY wins.

Table 5.2: Mc-nndfs runs with speedup ≥ 2 for models without cycles

Model	Speedup (16)
leader_filters.6.prop2	13,17
leader_election.5.prop2	12,63
leader_election.4.prop2	12,26
leader_election.6.prop2	12,16
leader_filters.7.prop2	12,05
leader_election.3.prop2	10,87
leader_filters.5.prop2	9,92
leader_filters.3.prop2	6,74
leader_election.2.prop2	5,90
protocols.4.prop2	5,51
leader_election.1.prop2	5,24
leader_filters.4.prop2	5,00
protocols.5.prop4	4,84
leader_filters.2.prop2	4,69
protocols.4.prop4	3,82
lifts.6.prop2	2,68
lifts.3.prop2	2,33
rether.3.prop5	2,14
szymanski.2.prop4	2,08
rether.5.prop5	2,08
rether.7.prop5	2,01

Otf_Owcty. However, on some models without accepting cycles DiVinE is faster by a factor of 10 on 16 cores. Which algorithm performs best in these cases likely depends on model characteristics, which we have yet to investigate.

However, we did investigate the lack of Mc-nndfs scalability for some models without cycles in Figure 5.6. All these cases lack states colored red by dfs_red. However, this does not hold the other way around: many models with few of these red states still exhibit speedups. This can be attributed to the red coloring by the *allred* extension (which we counted separately). In fact, for all models without cycles, the proportion of states colored red by dfs_red turned out to be negligible, while *allred* accounts for the vast majority of the red colorings.

We found that the number of red colorings is strongly dependent on the exploration

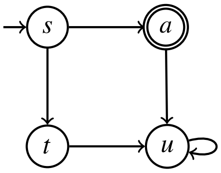


Figure 5.8: Exploration order can influence r_N

order (NEXT-STATE$_i$). Figure 5.8 illustrates that this is indeed possible. If a search advances first from s through t, then t (and s) cannot be colored red. However, if a is visited first, then u becomes red, hence later also t and s. It would be interesting to find a heuristic that maximizes red colorings.

We also observed that the speedup S_N is dependent on the fraction of red states r_N, as can be expected from the fact that r_N is the fraction of work that can be parallelized: $S_N \approx \frac{T_{seq}}{T_{seq} \times (1-r_N) + T_{seq} \times r_N / N} = \frac{1}{1-(1-1/N)r_N}$, where $T_{seq} \times (1 - r_N)$ is duplicated work. This shows us that the algorithm barely waits for a long time at Line 22, which is also confirmed by direct measurements.

5.6 Conclusions

In the current chapter, we introduced a multi-core NDFS algorithm, starting from a multi-core SV version, and proved its correctness. Its time complexity is linear in the size of the input graph, and it acts on-the-fly, addressing an open question put forward by Holzmann et al. and Barnat et al. [HB07; BBR09a]. However, in the worst case, each worker might still traverse the whole graph. We showed empirically that the algorithm scales well on many inputs. The on-the-fly property of MC-NNDFS, combined with the speedups on cycle-free models, makes MC-NNDFS highly competitive to OTF_OWCTY.

The experiments were needed because MC-NNDFS is a heuristic algorithm: in the worst case (no accepting states, hence no red states) no work is shared between workers and the performance reduces to the SV version. However, in these cases no other known linear-time parallel algorithm obtains any speedup (including dual-core NDFS [HB07]).

The space complexity of MC-NNDFS remains decent: per state $2 \times N$ local color bits, $\log_2(N)$ bits for the *count* variable, and one global red color bit, with N workers. The *count* variable could be omitted, at the expense of inspecting the pink flags of all other workers. However, this would lead to a significant contention. The overhead of $\log_2(N)$ bits per state is insignificant next to the space required by the local colors.

Final remark. We have strong indications that Mc-nndfs can be improved.

First, the previous section showed that a heuristic for exploration order might be of great benefit for the scalability. In Chapter 7, we investigate the influence of the exploration order better using different models.

Second, at the time of the publication of the work presented in the current chapter, a paper by Evangelista et al. was published describing a similar algorithm. This rival algorithm uses however a very different approach, and therefore in Chapter 6 we show how these algorithms can be combined. Chapter 7 presents again an improvement on this combined algorithm that is better integrated and reduces memory usage (cndfs).

6

Variations on Multi-Core Nested Depth-First Search

Alfons Laarman, Jaco van de Pol

Abstract

Recently, two new parallel algorithms for on-the-fly model checking of LTL properties were presented at the same conference: *Automated Technology for Verification and Analysis, 2011*. Both approaches extend Swarmed NDFS, which runs several sequential NDFS instances in parallel. While parallel random search already speeds up detection of bugs, the workers must share some global information to speedup full verification of correct models. The two algorithms differ considerably in the global information shared between workers, and how they synchronize.

Here, we provide a thorough experimental comparison between the two algorithms on a multi-core machine. Both algorithms were implemented in the same framework of the model checker LTSMIN, using similar optimizations, and have been subjected to the full BEEM model database.

Because both algorithms have complementary advantages, we constructed an algorithm that combines both ideas. This combination clearly has an improved speedup. We also compare with the alternative parallel algorithm for accepting cycle detection OTF_OWCTY. Finally, we study a simple statistical model for input models that do contain accepting cycles. The goal is to distinguish the speedup due to parallel random search from the speedup attributable to work sharing.

About this chapter: The current chapter is based on the paper *"Variations on Multi-Core Nested Depth-First Search"*, which was published at PDMC 2011 [LP11].

The original text was not modified, except for the fix of the multi-core NDFS algorithm that was also applied in the previous chapter, as explained in the 'about' section of that chapter. Also, the general introduction was removed.

Note that Chapter 7 presents a superior algorithm.

6.1 Introduction

During the last decades, processor speeds have been greatly increased, making model checkers much more powerful. Where early papers on model checking discussed the verification of models with a few thousand states, currently we can easily handle billions of states (see Part II). Recently, however, these advances are grinding to a halt, because of physical limits inside the CPU cores. Instead, the number of logical computing cores increases. Nonetheless, model checking can still benefit from the progress made by CPU manufacturers, if the algorithms are parallelized.

A complication is that DFS (and thus NDFS) is inherently sequential [Rei85]. Barnat et al. have therefore introduced breadth-first search (BFS) based algorithms, such as *Maximal-Accepting-Predecessors* (MAP [Bri+04]) and *One-Way-Catch-Them-Young* (OWCTY [ČP03]). These algorithms deliver excellent speedups, but sacrifice linear-time complexity. However, their latest combined OTF_OWCTY algorithm [BBR09a], is linear-time for the class of weak LTL properties and also useful for bug hunting. It is therefore the current state of the art in multi-core LTL model checking.

Recently, also two parallel NDFS-based algorithms were introduced [EPY11; Laa+11] (the latter is described in Chapter 5). Both take as starting point a randomized parallel search by a swarm of NDFS workers. While this is useful for bug-hunting, it does not really help in the absence of bugs, in which case all workers traverse the full state space. To improve speedup, both algorithms share some global information between workers, in order to reduce the amount of work even in the absence of accepting cycles. ENDFS from Evangelista et al. [EPY11] shares a lot of information, but this may break the required DFS order. A sequential repair procedure steps in when a potentially dangerous situation is detected. On the other hand, LNDFS from Chapter 5 shares less global information and adds extra synchronization. This avoids dangerous situations and the need for a repair strategy. However, this leads to a reduced amount of work sharing in some cases.

Contributions. The main goal of the current chapter is to experimentally compare both multi- core NDFS algorithms. In order to enable a fair comparison, we extended ENDFS with the same optimizations as used in LNDFS. We implemented both algorithms in the same framework of LTSMIN. Finally, we subjected both implementations to the full BEEM benchmark database [Pel07], running them on shared memory machines with up to 16 cores. Note that actual runtimes had not yet been reported for ENDFS, although workload distributions were shown in [EPY11]. Also, for LNDFS, we have rerun the experiments from Chapter 5.

Another contribution is a simple combination of the ENDFS and LNDFS algorithms, improving the speedup compared to both of them. We also compare all mentioned algorithms with the Otf_Owcty algorithm, both for bug hunting and for full verification. Finally, based on a simple statistical model [HJN08], we investigate how much of the speedup in the parallel Ndfs algorithms should be contributed to the effects of parallel random search and what is the contribution of the more clever work sharing schemes.

The algorithms are explained in Section 6.2. The experimental results are presented in Section 6.3. Section 6.4 contains the discussion on parallel random search. Our conclusions are summarized in Section 6.5.

6.2 Parallel Algorithms to Detect Accepting Cycles

Model checking properties from Linear Temporal Logic (LTL) entails verifying that all runs of a given system satisfy some safety or liveness property. In the automata-theoretic approach [VW86; BK08], a Büchi automaton is constructed that accepts all infinite words corresponding to those runs of the original system that violate the property. So the problem is reduced to the emptiness check of ω-regular languages. A Büchi automaton accepts a word if it visits some accepting state infinitely often. For finite automata, this implies that there is a cycle through some accepting state.

Definition 6.1. *A* Büchi automaton *is a quadruple $\mathcal{B} = (\mathcal{S}, s_I, \textsc{next-state}, \mathcal{F})$, where $\mathcal{S}$ is the finite set of states, $s_I \in \mathcal{S}$ is the initial state,* $\textsc{next-state} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ *the successor function, and $\mathcal{F} \subseteq \mathcal{S}$ the set of accepting states.*

Note, that the use of the next-state function reflects the way in which the Büchi automaton is computed *on-the-fly* from the input model. When appropriate, we refer to the complete automaton as graph or state space.

The purpose of all algorithms in the current chapter is to detect an accepting cycle in this graph. For states $s,t \in \mathcal{S}$, we write $s \rightarrow t$ if $t \in \textsc{next-state}(s)$, and $\rightarrow^+$ ($\rightarrow^*$), for its (reflexive) transitive closure. An accepting cycle is some state $a \in \mathcal{F}$, which is reachable from the initial state ($s_I \rightarrow^* a$) and lies on a non-trivial cycle ($a \rightarrow^+ a$).

6.2.1 Nested Depth-First Search

The first linear-time algorithm to detect accepting cycles was proposed by Courcoubetis et al. [Cou+92] and is referred to as Nested Depth-First Search (Ndfs). Ndfs also enjoys the on-the-fly property. This means that the algorithm can terminate as soon as a cycle is detected, without the need to visit (or even construct) the whole graph. This makes Ndfs very suitable for bug hunting, besides its use for full verification. Various extensions

Algorithm 6.1 The (sequential) New Ndfs algorithm adapted from [SE05]

```
 1 proc nndfs($s$)
 2   dfs_blue($s$)
 3   report no cycle

 4 proc dfs_red($s$)
 5   $s.color$ := $red$
 6   for all $t$ in NEXT-STATE($s$) do
 7     if $t.color$ = $cyan$
 8       report cycle & exit
 9     else if $t.color$ = $blue$
10       dfs_red($t$)

11 proc dfs_blue($s$)
12   $s.color$ := $cyan$
13   for all $t$ in NEXT-STATE($s$) do
14     if $t.color$ = $cyan$ and ($s \in \mathcal{F} \vee t \in \mathcal{F}$)
15       report cycle & exit
16     if $t.color$ = $white$
17       dfs_blue($t$)
18   if $s \in \mathcal{F}$
19     dfs_red($s$)
20   else
21     $s.color$ := $blue$
```

and optimizations to Ndfs have been proposed [HPY96; SE05; GS09]. Algorithm 6.1 most closely resembles *New Ndfs* [SE05].

In Algorithm 6.1, nndfs (s_I) initiates a *blue dfs* from the initial state, so called since explored states are colored blue (we assume that initially all states are white). A newly visited state is first colored *cyan* ("it is on the DFS-stack"), and during backtracking after exploration, it is colored full *blue*. However, if at Line 18 the blue dfs backtracks over an accepting state $s \in \mathcal{F}$, then dfs_red(s) is called, which is the nested *red* dfs to determine whether there exists a cycle containing s. As soon as a cyan state is found on Line 7, an accepting cycle is reported [HPY96; SE05]. In the blue dfs, *early cycle detection* is possible, at Line 14,15. Due to early cycle detection, it does not matter that the cyan color of s is overwritten by red at Line 5 (see Section 5.4.4).

Ndfs runs in linear time, since each reachable state is visited at most twice, once in the blue dfs and once in a red dfs. The correctness of Ndfs essentially depends on the fact that the red Dfss are initiated on accepting states in the postorder imposed by the blue dfs. So the red search will never hit another accepting state that is not already red.

6.2.2 Embarrassing Parallelization: Swarmed Ndfs

The inherently DFS nature of the blue search makes Ndfs hard to parallelize, since computing the postorder is a P-complete problem [Rei85]. One response has been to develop entirely different algorithms based on Breadth-First Search, cf. Sec. 6.2.6.

Another approach would be to simply run N isolated instances of Ndfs (Algorithm 6.1) in parallel, in the hope that this *swarm* of Ndfs workers will detect accepting cycles earlier [HJG08; Laa+11]. Local permutations of the NEXT-STATE function direct the workers to different regions of the state space, so their search becomes independent. With NEXT-STATE_i^b (NEXT-STATE_i^r) we denote the permutation of successors used in the blue (red) DFS by worker *i*. Section 6.4 analyses the expected and actual improvements due to parallel randomized search.

Although Swarmed NDFS is expected to be profitable for bug hunting, it does not show a speedup in the absence of accepting cycles, in which case all workers have to go through the complete state space. Indeed, the worst-case complexity of all parallel NDFS variations in the current chapter is $\mathcal{O}(|\rightarrow| \cdot |N|)$, i.e. linear both in the size of the Büchi automaton and in the number of workers.

To improve average speedup, some more synchronization between the workers is needed. Note that a naive global sharing of colors between multiple workers would be incorrect, because it would destroy the postorder properties on which NDFS relies. Next, we discuss two recent proposals for sharing information between the NDFS workers.

6.2.3 LNDFS: Sharing the Red Color Globally

The basic idea behind LNDFS in Algorithm 6.2 is to share information in the backtrack of the red DFSs. A new *pink* color is introduced at Line 5 to signify states on the stack of a red DFS, analogous to cyan for a blue DFS. The cyan, blue and pink colors are all local to worker *i*, but the red color is shared *globally*. On backtracking from the red DFS, states are colored red at Line 14. These red states are ignored by *all blue and red* DFSs (Line 21,9), thus pruning the search space for all workers *i*. To improve pruning during the blue search, the amount of red states is even increased by the *all-red* extension from [GS09] (Line 16 and Line 23-27).

To ensure correctness, it is necessary to synchronize the red coloring of accepting states (see Line 13). Otherwise, the algorithm is incorrect for more than two workers (see Section 5.4.2, which provides a correctness proof for $N > 0$ workers). Scalability of the LNDFS algorithm could be hampered by the need for synchronization, but waiting is only needed when multiple workers start a red search from the same accepting state; this is rare in practice. Another reason for limited scalability is that work is only pruned when states can be marked red. Despite the all-red extension, for input graphs with no (or very few) accepting states, all workers still have to traverse the whole graph.

6.2.4 ENDFS: an Optimistic Approach with Repair Strategy

The basic idea of ENDFS in Algorithm 6.3 [EPY11] is to share both the blue and the red colors globally; only the cyan and pink colors are local per worker. We deviate from the description in [EPY11] by adding a cyan stack and early cycle detection as optimizations, because this enables a fair comparison with LNDFS. Consequently, we also renamed the local colors.

Sharing the blue color can lead to problems, as the postorder is not preserved by the algorithm. ENDFS optimistically proceeds, but if it encounters accepting states that are not yet red during the red search, they are marked dangerous at Line 11. Eventually, dangerous states are double-checked in a repair stage, by a separate sequential NDFS using worker-local colors only, at Line 29-30. Note that for technical reasons, states are not colored red during backtracking, but just collected in the thread-local set R_i at Line 6. Only after termination of the red DFS they are made red (provided they are not dangerous) at Line 26-28.

Scalability of the ENDFS algorithm could be hampered by the repair stage, because this proceeds sequentially. Also, marking states red occurs relatively late, potentially leading to more duplicate work within the red DFS.

Algorithm 6.2 The LNDFS algorithm, pruning blue and red DFS by a global red color, adapted from Chapter 5.

```
 1 proc lndfs($s,N$)
 2   dfs_blue($s,1$)||..||dfs_blue($s,N$)
 3   report no cycle

 4 proc dfs_red($s,i$)
 5   $s.color[i]$ := $pink$
 6   for all t in NEXT-STATE$_i^r(s)$ do
 7     if $t.color[i] = cyan$
 8       report cycle & exit all
 9     if $t.color[i] \neq pink \wedge \neg t.red$
10       dfs_red($t,i$)
11   if $s \in \mathcal{F}$
12     $s.count$ := $s.count - 1$
13     await $s.count = 0$
14   $s.red$ := true

15 proc dfs_blue($s,i$)
16   $allred$ := true
17   $s.color[i]$ := $cyan$
18   for all t in NEXT-STATE$_i^b(s)$ do
19     if $t.color[i] = cyan$ and $(s \in \mathcal{F} \vee t \in \mathcal{F})$
20       report cycle & exit all
21     if $t.color[i] = white \wedge \neg t.red$
22       dfs_blue($t,i$)
23     if $\neg t.red$
24       $allred$ := false
25   if $allred$
26     await s.count = 0
27     $s.red$ := true
28   else if $s \in \mathcal{F}$
29     $s.count$ := $s.count + 1$
30     dfs_red($s,i$)
31   $s.color[i]$ := $blue$
```

6.2.5 A Combined Version: New Mc-ndfs

We have recapitulated two very recent Mc-ndfs algorithms, which both seem to have their merits and pitfalls. ENdfs, in the end, resorts to a sequential repair strategy, but it avoids some work duplication due to the global blue color. LNdfs does not need a repair strategy, but the blue dfs is only pruned when there are sufficiently many red states, and the algorithm may have to wait for synchronization. A simple idea suggests itself here: we could combine the two algorithms and try to reconcile their strong points. The idea is simply to run the optimistic algorithm, i.e. Algorithm 6.3, but when dangerous states are encountered at Line 30, we call the parallel algorithm LNdfs (rather than Ndfs).

We expect an improved speedup, because using ENdfs ensures good work sharing, even in the absence of accepting states. And using LNdfs parallelizes the repair strategy, avoiding the important sequential bottleneck of ENdfs. In the actual implementation, we also used a simple load balancing strategy: when a worker finishes ENdfs, it starts helping other workers still in their LNdfs repair phase.

Algorithm 6.3 The optimistic ENdfs algorithm, marking dangerous states, adapted from [EPY11].

```
 1 proc endfs($s,N$)
 2    dfs_blue($s,1$)$\|..\|$dfs_blue($s,N$)
 3    report no cycle

 4 proc dfs_red($s,i$)
 5    $s.pink[i]$ := true
 6    $R_i := R_i \cup \{s\}$
 7    for all $t$ in NEXT-STATE$_i^r(s)$ do
 8       if $t.cyan[i]$
 9          report cycle & exit all
10       if $t \in \mathcal{F} \wedge \neg t.red$
11          t.$dangerous$ := true
12       if $\neg t.red \wedge \neg t.pink[i]$
13          dfs_red($s,i$)

14 proc dfs_blue($s,i$)
15    $s.cyan[i]$ := true
16    for all $t$ in NEXT-STATE$_i^b(s)$ do
17       if $t.cyan[i]$ and $(s \in \mathcal{F} \vee t \in \mathcal{F})$
18          report cycle & exit all
19       if $\neg t.cyan[i] \wedge \neg t.blue$
20          dfs_blue($t,i$)
21    $s.cyan[i]$ := false
22    $s.blue$ := true
23    if s $\in \mathcal{F}$
24       $R_i := \emptyset$
25       dfs_red($s,i$)
26       for all $r \in R_i$ do
27          if $\neg r.dangerous \vee s = r$
28             $r.red$ := true
29       if $s.dangerous$
30          nndfs($s,i$)
```

6.2.6 One-Way-Catch-Them-Young with Maximal Accepting Predecessors

In the next section, we will compare the performance of the various NDFS implementations in terms of their absolute timing and speedup behavior. We will also compare them with the current state-of-the-art algorithm in parallel symbolic model checking, OTF_OWCTY [BBR09a] by Barnat et al., which is a member of the branch of BFS-based algorithms (other algorithms in this class are discussed in Section 5.3).

Basically, it extends the *One-Way-Catch-Them-Young* algorithm (OWCTY [ČP03]), with an initialization phase incorporated from the *Maximal-Accepting-Predecessor* algorithm (MAP [Bri+04]). In a nutshell, MAP iteratively propagates unique node identifiers to successors. As soon as an accepting state receives its own identifier, a cycle is detected. OWCTY is based on topological sort and iteratively eliminates states that cannot lie on an accepting cycle, because they have no predecessors.

These algorithms are generally based on BFS, which is more easy to parallelize than DFS. However, these algorithms sacrifice linear-time behavior and the on-the-fly property. The resulting combination is linear-time for Büchi automata generated from the class of weak LTL properties, and shows on-the-fly behavior for several cases.

6.3 Experiments

We implemented multi-core Swarmed NDFS and Algorithm 6.2 and Algorithm 6.3 in the multi-core backend of the LTSMIN model checking tool suite [LPW11a; BPW10; BPW09].[6.1] We performed experiments on an AMD Opteron 8356 16-core (4 × 4 cores) server with 64 GB RAM, running a patched Linux 2.6.32 kernel. All tools were compiled using gcc 4.4.3 in 64-bit mode with high compiler optimizations (`-O3`).

We measured performance characteristics for all 453 models with properties of the BEEM database [Pel07] and compared the runs with the best known parallel LTL model checking algorithm OTF_OWCTY as implemented in DIVINE 2.5 [Bar+10]. In fact, we used the latest release available from the development repository on 23 March 2011, which was close to the 2.5 version, except for a few relevant bug fixes.

Note that OTF_OWCTY has been implemented in DIVINE, whereas all NDFS-based algorithms have been implemented in LTSMIN. This should be taken into account when comparing absolute runtimes. LTSMIN implements a generic interface around the fast implementation of the NEXT-STATE() function of DIVINE, resulting in sequential runtimes that can be twice as slow. On the other hand, LTSMIN internally uses shared hash tables, which are shown to scale better, at least for reachability (see Chapter 2).

[6.1] Available on the LTSMIN website: `http://fmt.cs.utwente.nl/tools/ltsmin/`.

To account for the random nature of the algorithms, all experiments were executed a total of 5 times. The data presented in the following subsections reflect the average over those 5 experiments.

6.3.1 ENDFS Benchmarks

Evangelista et al. [EPY11] used workload distribution measurements to estimate the scalability of ENDFS. Figure 6.2 reflects their estimated speedups. Figure 6.1 shows the speedups that we obtained by measuring real runtimes of the algorithm.

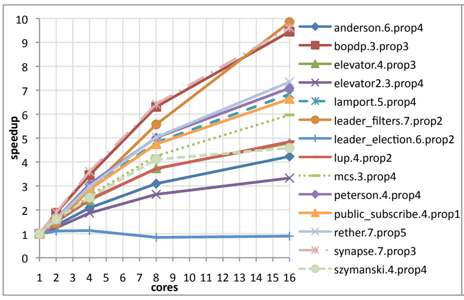


Figure 6.1: Measured speedups ENDFS.

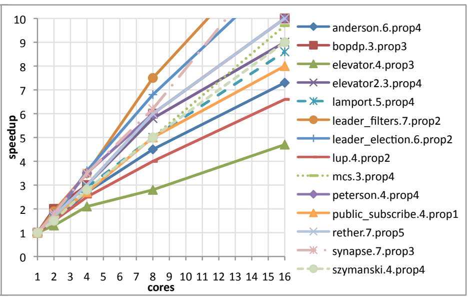


Figure 6.2: Speedups ENDFS in [EPY11].

A comparison with the estimated speedups shows that the trend of the lines has been accurately predicted in most cases. A case by case comparison, however, shows some divergence between the exact numbers: models that scale well in “synthetic” benchmarks of Figure 6.2 as, e.g., `anderson.6.prop4`, `elevator2.3.prop4`, `leader_election.6.prop2` and `szymanski.4.prop4`, do not scale well in practice. We have not investigated the source of these differences, but apparently the amount of dangerous states is quite sensitive to implementation parameters.

Figure 6.3 and Figure 6.4 compress the results from all models of the BEEM database in log-log scatter plots. In both figures, we show models without accepting cycles as dots and models with these cycles as crosses. Comparing ENDFS to NDFS in the first figure, we can distinguish good speedups for the models with cycles, while the other figure shows that ENDFS even improves the results of Swarmed NDFS a little. In Section 6.4, we investigate and compare these effects more thoroughly, using a statistical reference model for random parallel search. As for the models without accepting cycles, we see that most do scale with ENDFS, but hardly beyond a speedup of 10. Even though theoretically possible, we identified no cases where the repair strategy of ENDFS yields speed downs (in the worst case, all workers can traverse the state space 4 times).

We also investigated what caused some inputs to scale poorly. Figure 6.5 shows the percentage of the state space that is covered by the repair procedure. As expected, a high percentage was measured for all models with poor scalability. Figure 6.6 shows the cumulative additional work performed by all workers, by summing up the states visited by all workers in the repair procedure and dividing by the total amount of states ($|\mathcal{S}|$). It is worrisome that the need for repair can increase faster than the number of cores. This suggests that the ENDFS may not scale to many-core systems.

6.3.2 ENDFS versus LNDFS

Figure 6.7 shows the speedups of the LNDFS algorithm. In this set of models, few scale well with this algorithm. The flat lines represent models with relatively few states reachable from accepting states. In these cases, the algorithm can only color few states red, thus limiting work sharing between the workers. As shown in Chapter 5, the fraction of red states is indeed directly related to the speedup that is obtained. The two models `leader_filters.7.prop2` and `leader_election.6.prop2` have state spaces that are colored entirely red, and hence exhibit almost ideal linear speedups. However, Figure 6.9 shows that only few models behave this ideally. Unfortunately, in Chapter 5 we reported better speedups, which we have now tracked down to an im-

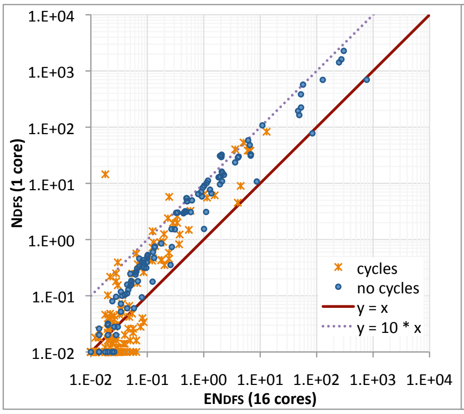


Figure 6.3: NDFS vs ENDFS.

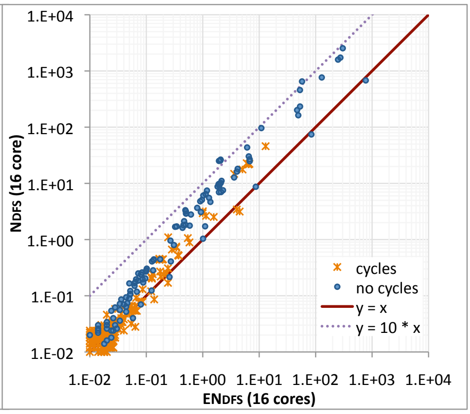
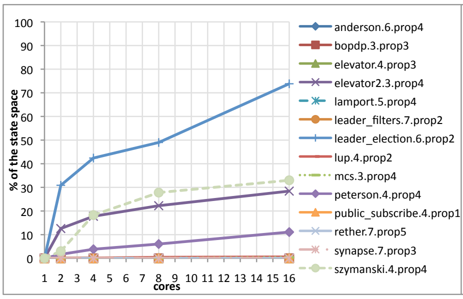


Figure 6.4: Swarmed NDFS vs ENDFS.



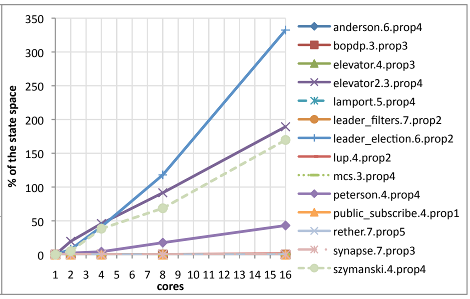
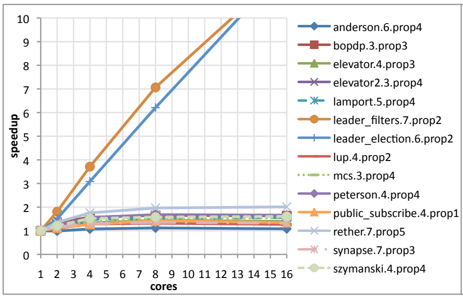


Figure 6.5: % of state space in ENDFS repair.

Figure 6.6: Cumulative extra repair work.

plementation error that led to too many red states.

When comparing ENDFS to LNDFS in Figure 6.10, we witness a few ties (on the thick line), a few winners with LNDFS and by far the most winners with ENDFS. We looked up the models that draw a tie and found that all of them scale with both algorithms. These are therefore not in need of improvements. Most interestingly, the models that scale well with LNDFS correspond to those that do not scale with ENDFS. This indicates that both algorithms are complementary. A fact that is indeed to be expected, because the same accepting states that cause states to be colored red in LNDFS, are potentially marked dangerous in ENDFS. This motivated their combination as described in Section 6.2.5.



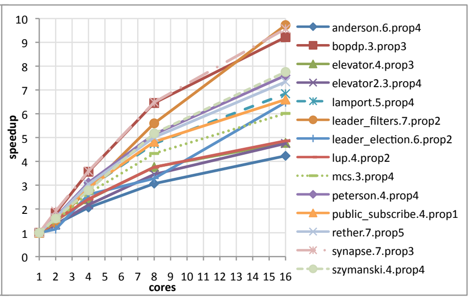
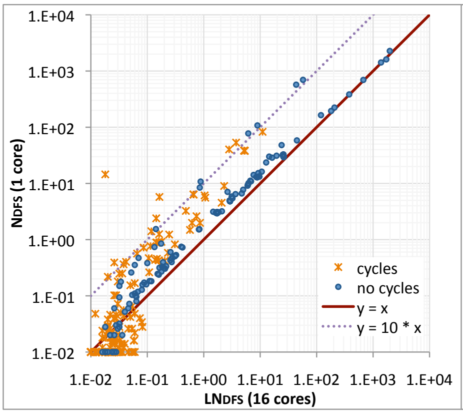


Figure 6.7: Speedups LNDFS.

Figure 6.8: Speedups NMC-NDFS.



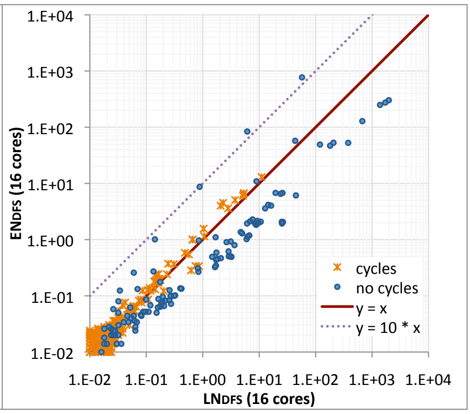


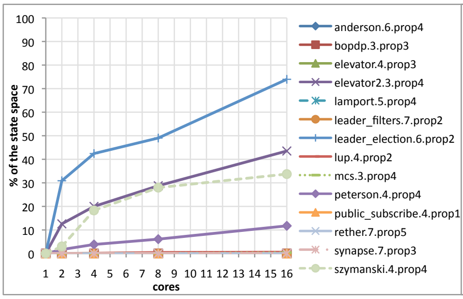
Figure 6.9: Ndfs vs LNdfs.

Figure 6.10: ENdfs vs LNdfs.

6.3.3 NMc-ndfs Benchmarks

In the current subsection, we investigate our ENdfs and LNdfs combination: NMc-ndfs. Figure 6.8 shows that NMc-ndfs improves upon the speedups of ENdfs (see Figure 6.1), and Figure 6.15 confirms that all models scale well with the combined algorithm.

For NMc-ndfs, again, we also calculated the cumulative additional work as a percentage of the state space in Figure 6.11. The state-space coverage by the repair proce-



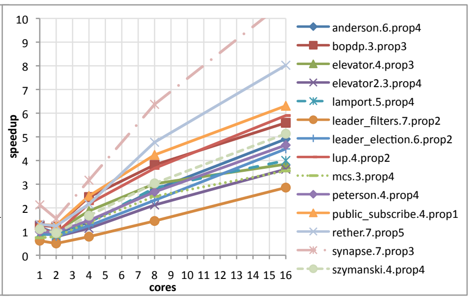


Figure 6.11: Cumulative extra work due to NMc-ndfs repair.

Figure 6.12: Speedups Owcty.

dure is almost equal to that of ENdfs in Figure 6.5. We can then deduce that the repair work is parallelized well by LNdfs, because the cumulative additional work is close to the percentage of state space coverage. This can be explained by the fact that LNdfs is always called on a (dangerous) accepting state in NMc-ndfs, which eventually leads to a red coloring of the entire subgraph reachable from this accepting state. Under these conditions LNdfs can be expected to scale well.

We also checked whether the new combination causes additional overhead, by comparing it directly with its predecessors in Figure 6.13 and Figure 6.14. The first figure shows that no model runs faster with ENdfs than with NMc-ndfs, although in a few examples LNdfs wins, as can be seen in the latter figure. This confirms that LNdfs and ENdfs are complementary and their combination represents the best from both worlds. Indeed, the combination ensures that for all inputs some speedup is obtained.

6.3.4 Parallel Ndfs versus Owcty-Map

Figure 6.16 compares NMc-ndfs with Otf_Owcty. The comparison figures show that the heuristic on-the-fly method of Otf_Owcty is no match for the truly on-the-fly parallel Ndfs algorithms. As for the models without accepting cycles, we can conclude that currently NMc-ndfs provides a good match for Otf_Owcty, in particular for the larger models. For the sake of completeness, we present here Figure 6.17, 6.18, comparing ENdfs/LNdfs and Otf_Owcty. Furthermore, Figure 6.12 shows the absolute

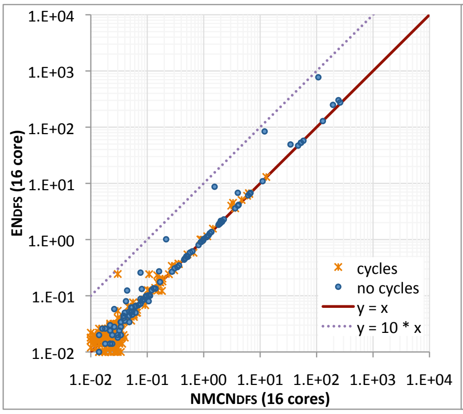


Figure 6.13: ENdfs vs NMc-ndfs.

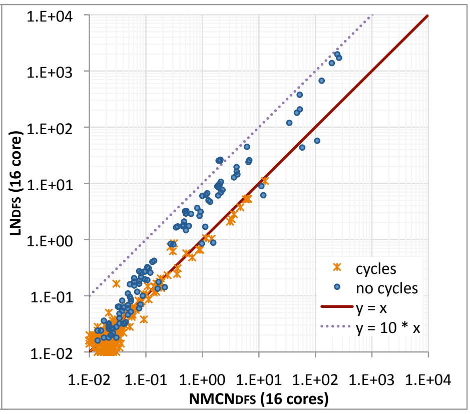
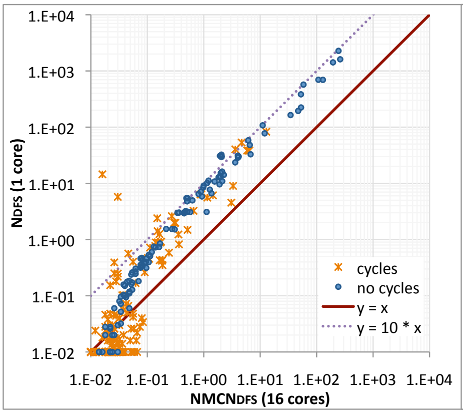


Figure 6.14: LNdfs vs NMc-ndfs.

speedups of Otf_Owcty using the sequential Ndfs runtimes as the base case.



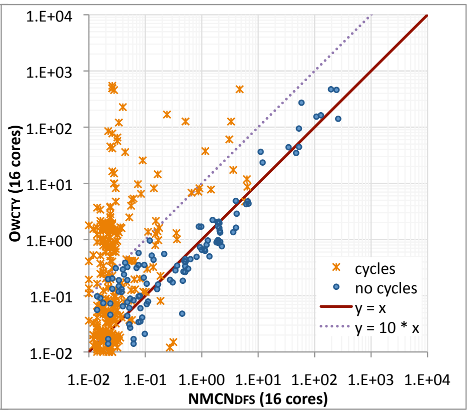
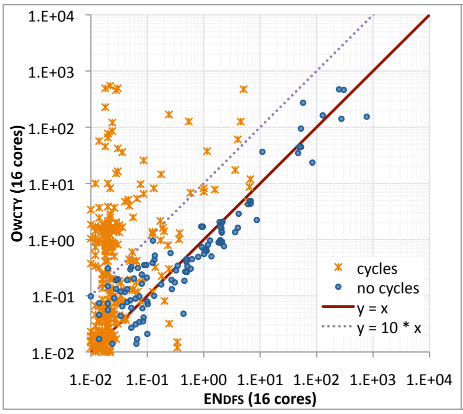


Figure 6.15: Ndfs vs NMc-ndfs.

Figure 6.16: Otf_Owcty vs NMc-ndfs.



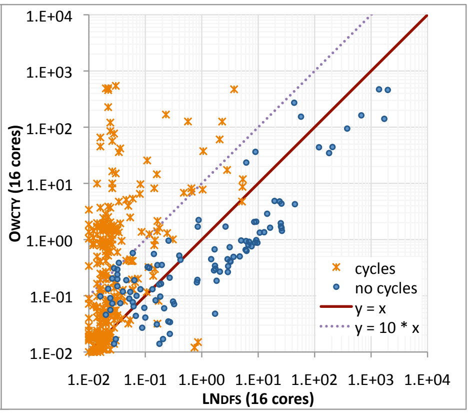


Figure 6.17: Otf_Owcty vs ENdfs.

Figure 6.18: Otf_Owcty vs LNdfs.

6.4 Discussion on Parallel Random Search

As explained in Section 6.2, the multi-core Ndfs algorithms use a randomized next-state( ) function to direct workers to different regions of the state space. In this section, we want to explain the speedup for models with accepting cycles. In particular, we want to distinguish the effect of parallel random search, from the effect of the clever work sharing algorithms.

Our starting point is a simple statistical model as found in [HJN08]. We view NDFS$(\mathcal{B},X)$ as an algorithm that runs on Büchi automaton $\mathcal{B}$ with random seed X, influencing the order of traversing successors. We ran NDFS$(\mathcal{B},X)$ 500 times with random X on a number of Büchi automata $\mathcal{B}$. Each time, we measured $f(\mathcal{B},X)$, the time that it takes for NDFS$(\mathcal{B},X)$ to detect an accepting cycle.

In Figure 6.19, we show the cumulative probability $F(\mathcal{B},t)$ that one NDFS worker will detect an accepting cycle in less than t seconds for some examples from the BEEM database. We can also define $F_N(\mathcal{B},t)$ as the cumulative probability that a swarm of N independent workers will find an accepting cycle within t seconds. Figure 6.20 shows $F_{16}(\mathcal{B},t)$ for the same automata. We also computed the expected time to completion and the standard deviation. The new distribution can be easily computed as:

$$F_N(\mathcal{B},t) = 1-(1-F(\mathcal{B},t))^N$$

From Figure 6.19 and Figure 6.20, we observe that considerable gains can be expected from a simple parallelization as in Swarmed NDFS. It also shows that the actual speedup depends highly on the models: when all runs find an accepting cycle in about the same time (indicated by plateaus connected by a steep curve), the expected gain is much less than when the curve is flatter, as is the case for `anderson.8.prop3`, `bakery.8.prop4` and `peterson.6.prop4`.

Next, we want to compare our actual implementation with these predictions. To this end, we compared the expected completion times with actual completion times, averaged over 5 runs. We collected this information in Table 6.1. In the first two columns (Statistical model), we copied the averages from Figure 6.19, 6.20 for 1 and 16 workers, and computed the expected speedup. Note that this speedup for 16 workers is way below 16. Next, we experimented with four different scenarios described below.

The next column (Distributed), corresponds to Swarmed NDFS as it would run on different machines in a GRID. Here the only synchronization would be to terminate all workers as soon as the first worker has detected a cycle. The runtimes denote the completion time for the earliest run out of 16 independent workers; we again provide the average from 5 experiments. The corresponding speedups match closely to the predicted ones from the statistical model.

Next, we ran the experiments on the multi-core machine with 16 cores described before. Now the workers share the basic infrastructure. This is the same setting as the multi-core Swarmed NDFS from the previous section. For instance, all states will be stored only once in a shared hash table. Also, several workers now share information in the L2 cache. On the other hand, they might now suffer from cache coherence overhead or memory bus contention. The figures under “Shared Memory” show that the

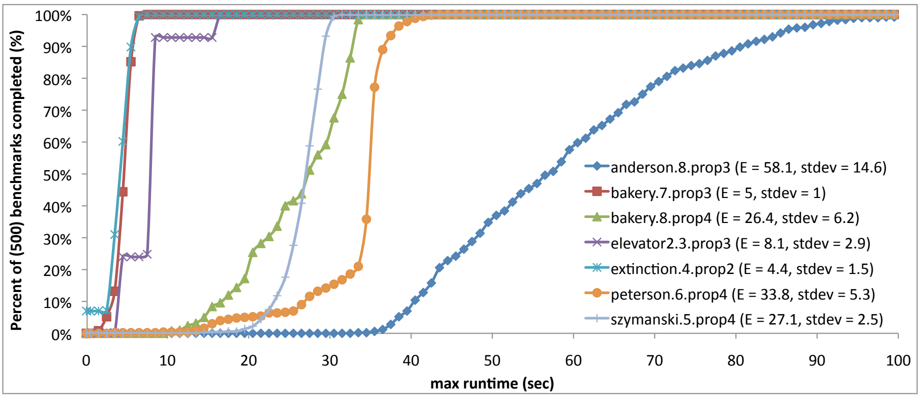


Figure 6.19: Cumulative prob. distribution of finding a bug (measured for 1 worker).

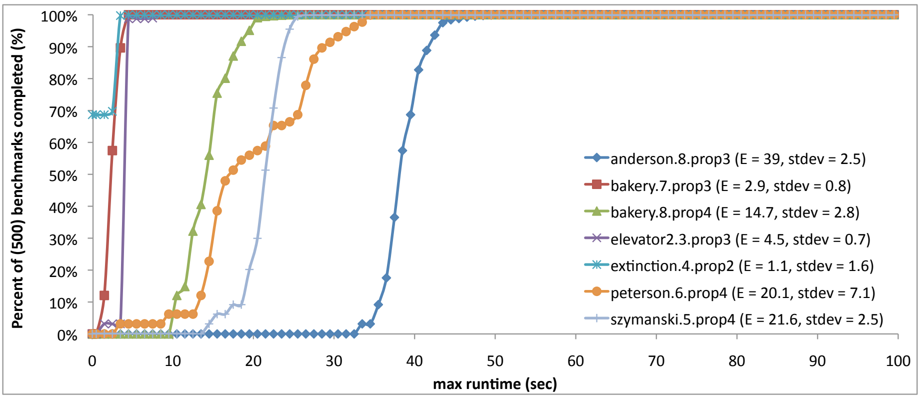


Figure 6.20: Cumulative prob. distribution of finding a bug (calculated for 16 workers).

speedups in a multi-core environment are slightly better than on independent machines (Distributed).

On multi-core machines it becomes easier to share information, in order to guide different workers into different parts of the state space. In that case, one would expect better speedup figures. We did an experiment with what we call the *fresh successor heuristic*. Here a worker will randomly select a globally unvisited successor if that exists, otherwise it randomly selects any successor. As the column Heuristic shows, this can dramatically improve the speedup of 16 workers. In some cases, each time

Table 6.1: Runtimes and speedups of bug hunting using embarrassingly parallel (randomized) Ndfs and LNdfs. The first two columns of the table present the expected completion time derived from 500 sequential experiments for 1 and 16 cores. The other columns give parallel runtimes for, respectively, a distributed implementation, our randomized shared-memory implementation [Laa+11], and another shared-memory implementation using the fresh successor heuristic. The second row gives the speedups.

		Ndfs					LNdfs	
		1 core	16 core				16 core	
	model	Statistical model	Statistical model	Distributed	Shared Memory	Heuristic	Shared Memory	Heuristic
Runtimes (sec)	anderson.8.prop3	58.1	39.0	39.3	39.4	9.6	8.6	3.1
	bakery.7.prop3	5.0	2.9	2.9	2.1	0.6	0.8	0.3
	bakery.8.prop4	26.4	14.7	13.6	12.9	0.6	1.9	1.1
	elevator2.3.prop3	8.1	4.5	4.2	2.6	0.7	2.1	0.2
	extinction.4.prop2	4.4	1.1	0.8	0.5	0.0	0.0	0.0
	peterson.6.prop4	33.8	20.1	24.2	16.7	12.5	2.5	2.2
	szymanski.5.prop4	27.1	21.6	20.9	19.4	0.0	3.3	0.0
Speedups	anderson.8.prop3		1.5	1.5	1.5	6.1	6.7	18.5
	bakery.7.prop3		1.7	1.7	2.4	8.6	6.3	15.2
	bakery.8.prop4		1.8	1.9	2.0	45.7	14.1	23.2
	elevator2.3.prop3		1.8	1.9	3.1	11.7	3.8	41.8
	extinction.4.prop2		4.1	5.9	8.9	??	??	??
	peterson.6.prop4		1.7	1.4	2.0	2.7	13.5	15.6
	szymanski.5.prop4		1.3	1.3	1.4	??	8.3	??

an accepting cycle was found in such a small instant that a meaningful speedup figure could not be computed.

Finally, using LNdfs, the total amount of work is decreased, because workers prune each other's search space. Again, we experimented with two versions, which are shown in the two right-most columns. We computed the average runtime of 5 experiments on 16 cores with the random shared-memory implementation. Note that this is the implementation that was used in all previous experiments in Section 6.3. The figures show again a big improvement over Swarmed NDFS, even on a multi-core machine. Interestingly, the fresh successor heuristic also works very well for the LNDFS-algorithm, speeding up the algorithm several times. Similar findings hold for all other parallel NDFS versions in the current chapter, because they behave similarly on models with accepting cycles (see Figure 6.13 and Figure 6.14).

6.5 Conclusion

In the current chapter, we experimentally compared two recent parallel NDFS-based algorithms, ENDFS [EPY11] and LNDFS [Laa+11]. We also compared them with Swarmed NDFS and with the BFS-based algorithm OTF_OWCTY [BBR09a]. We now summarize the conclusions from our experiments.

For systems with bugs (accepting cycles), both ENDFS and LNDFS outperform OTF_OWCTY by large, so they fully enjoy the on-the-fly property. We have also shown that ENDFS and LNDFS perform much better than parallel random search, as in Swarmed NDFS.

On examples without bugs, it appears that ENDFS beats LNDFS in most of the cases, due to the fact that there are still too few red states to prune the blue search in LNDFS. However, in a number of other cases ENDFS still scales rather badly, due to the fact that the sequential repair strategy traverses large parts of the state space. Interestingly, it is possible to use the parallel LNDFS algorithm as the repair strategy of ENDFS. For this new combined algorithm, all examples of the BEEM database showed a decent speedup.

On examples without bugs, OTF_OWCTY beats both LNDFS and ENDFS in a majority of the cases, but still it is slower on a number of other examples. The combination of ENDFS and LNDFS, however, provided a good match for OTF_OWCTY, especially for the larger inputs. This shows that the new branch of parallel NDFS algorithms is rather promising.

Future work. Although all NDFS-versions have been implemented in the same framework so that we compare the algorithmic differences, OTF_OWCTY was implemented in the DIVINE tool. We note that our computation of the NEXT-STATE() function uses the same code as DIVINE. A reimplementation of OTF_OWCTY using shared hash tables will probably increase its speedup, as indicated by results on pure reachability (see Part II). For complex LTL properties, however, OTF_OWCTY might exhibit non-linear behavior. It would be interesting to compare the bahavior of the multi-core NDFS algorithms for such cases (the properties used in the BEEM database are rather simple).

Final note. In the next chapter, we show a parallel NDFS algorithm that can fully share global information from both the blue and the red search, without the need to resort to a repair strategy.

7 Improved Multi-Core Nested Depth-First Search

Sami Evangelista, Alfons Laarman, Laure Petrucci, Jaco van de Pol

Abstract

The current chapter presents CNDFS, a tight integration of two earlier multi-core nested depth-first search (NDFS) algorithms for LTL model checking. CNDFS combines the different strengths and avoids some weaknesses of its predecessors. We compare CNDFS to an earlier ad-hoc combination of those two algorithms and show several benefits: It has shorter and simpler code and a simpler correctness proof. It exhibits more robust performance with similar scalability, while at the same time reducing memory requirements.

The algorithm has been implemented in the multi-core backend of the LTSMIN model checker, which is now benchmarked for the first time on a 48 core machine (previously 16). The experiments demonstrate better scalability than other parallel LTL model checking algorithms, but we also investigate apparent bottlenecks. Finally, we noticed that the multi-core NDFS algorithms produce shorter counterexamples, surprisingly often shorter than their BFS-based counterparts.

About this chapter: The current chapter is based on the paper *"Improved Multi-Core Nested Depth-First Search"*, which was published at the ATVA 2012 conference [Eva+12].

Building on the combination of of multi-core algorithms in the previous chapter, we present a new integrated algorithm, called CNDFS, with improved performance, reduced memory usage, and shorter correctness proof.

The original text from [Eva+12] has been improved by correcting an error in the basic nested depth-first search algorithm (Algorithm 7.1). A missing check for cyan states in the main DFS loop would cause the previously published algorithm to enter an infinite loop. Also, the general introduction was removed.

7.1 Introduction

Some properties, like *safety properties*, rely on a complete enumeration of system states and can thus be easily parallelized since they do not ask for a specific search order. However, the problem is harder when it comes to the verification of *linear temporal logic* (LTL) properties. LTL model checking can be reduced to a cycle detection problem and state-of-the-art algorithms [Cou+92; Cou99; GV04] proceed depth-first since cycles are more easily discovered using this search order. However, this characteristic also makes them unsuitable for parallel architectures since DFS is inherently sequential [Rei85].

One approach to address this issue is to sacrifice the optimal linear complexity provided by DFS algorithms and switch to BFS-like algorithms, which are highly scalable both theoretically and experimentally. We compare our approach to the best representative of that family. More recently, two algorithms (LNDFS from Chapter 5 and ENDFS from [EPY11]) adapted the well known Nested DFS (NDFS) algorithm [Cou+92] to multi-core architectures. They share the principle of launching multiple instances of NDFS that synchronize themselves to avoid useless state revisits. Although they are heuristic algorithms in the sense that, in the worst case, they reduce to spawn multiple unsynchronized instances of NDFS, the experiments reported in Chapter 5 and Chapter 6 show good practical speedups.

The contribution of the current chapter is an improvement to both the LNDFS and ENDFS algorithms, called CNDFS. This new algorithm is both much simpler and uses less memory, making it more compatible with lossy compression techniques such as tree compression (Chapter 3) that can compress large states down to two integers. We also pursue a thorough experimental evaluation of this algorithm on the models of the BEEM database [Pel07] with an implementation of this algorithm on top of the LTSMIN toolset [BPW10; LPW11a]. The outcome of these experiments is threefold. Firstly, CNDFS exhibits a similar speedup to its predecessors, but achieves this more robustly, with smoother speedup lines, while using less memory. Second, it combines nicely with heuristics limiting the amount of redundant work performed by individual threads. Finally, in the presence of bugs, it reports counterexamples that are usually much shorter than those reported by NDFS and, more importantly, this length tends to decrease as more working threads get involved in the verification. This property is quite appreciable from a user perspective as it eases the task of error correction.

The outline of the current chapter is the following. In Section 7.2 we formally express the LTL model checking problem and review existing (sequential and parallel) algorithms that address it. CNDFS, our new algorithm, is introduced and formally proven in Section 7.3. Our experimental evaluation of this algorithm is summarized in Section 7.4. Finally, Section 7.5 concludes the current chapter and explores some research perspectives to this work.

7.2 Background

We give in this section the few ingredients that are required for the understanding of the current chapter and briefly review existing works in the field of explicit parallel LTL model checking based on the automata theoretic approach.

7.2.1 The Automata-Theoretic Approach to LTL Model Checking

LTL model checking is usually performed following the automata-based approach originating from [VW86] that proceeds in several steps. In the current chapter, we focus only on the last step of the process that can be reduced to a graph problem: given a graph representing the synchronized product of the Büchi property automaton and the state space of the system, find a cycle containing an accepting state. Any such identified cycle determines an infinite execution of the system violating the LTL formula. In the current chapter, we will only reason on *automaton graphs* that result from the product of a Büchi property automaton and a system graph describing the dynamic behavior of the modeled system.

Definition 7.1 (Automaton graph)**.** *An* automaton graph *is a tuple* $\mathcal{G} = (\mathcal{S}, \mathcal{T}, \mathcal{F}, s_0)$*, where* $\mathcal{S}$ *is a finite set of states;* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ *is a set of transitions;* $\mathcal{F} \subseteq \mathcal{S}$ *is the set of accepting states; and* $s_0 \in \mathcal{S}$ *is an initial state.*

Notations. Let $(\mathcal{S}, \mathcal{T}, \mathcal{F}, s_0)$ be an automaton graph. For $s \in \mathcal{S}$ the set of its *successor states* is denoted by $succ(s) = \{s' \in \mathcal{S} \mid (s, s') \in \mathcal{T}\}$. $(s, s') \in \mathcal{T}$ is also denoted by $s \rightarrow s'$. $s \rightarrow^+ s'$ ($s \rightarrow^* s'$) denotes the (reflexive) transitive closure of $\mathcal{T}$, i.e. the fact that s' is *reachable* from s. A *path* is a state sequence $s_1, \ldots, s_n$ with $s_i \rightarrow s_{i+1}, \forall i \in \{1, \ldots, n-1\}$, a *cycle* is a path $s_1, \ldots, s_n$ with $s_1 = s_n$ and a cycle $C \equiv s_1, \ldots, s_n$ is an *accepting cycle* if $C \cap \mathcal{F} \neq \emptyset$. An *accepting run* is an accepting cycle reachable from the initial state: $s_0, \ldots, s_i, \ldots, s_n$ where $s_i = s_n$. The *LTL model checking problem* consists of finding an accepting run in an automaton graph. An LTL model checking algorithm proceeds on-the-fly if it can report an accepting run without visiting all transitions.

7.2.2 Sequential LTL Model Checking Algorithms

Ndfs [Cou+92] was the first LTL model checking algorithm proposed. It enjoys several nice properties: an optimal linear complexity, the on-the-fly discovery of accepting cycles and a low memory consumption (2 bits per state). Two variations of Tarjan's algorithm for SCC decomposition [Cou99; GV04] have also been proposed with similar characteristics but we focus here on Ndfs as our new algorithm is a direct descendant of this one.

Algorithm 7.1 Ndfs [VW86] as presented in [SE05].

```
 1: procedure NDFS($s_I$)
 2:     $dfsBlue(s_I)$
 3:     report no-cycle
 4: procedure $dfsRed(s)$
 5:     $t.red$ := true
 6:     for all $t$ in NEXT-STATE($s$) do
 7:         if $t.cyan$ then report cycle
 8:         else if $\neg t.red$ then $dfsRed(t)$
 9: procedure $dfsBlue(s)$
10:     $s.cyan$ := true
11:     for all $t$ in NEXT-STATE($s$) do
12:         if $\neg t.cyan \land \neg t.blue$ then
13:             $dfsBlue(t)$
14:     if $s \in \mathcal{F}$ then $dfsRed(s)$
15:     $s.blue$ := true
16:     $s.cyan$ := false
```

The pseudo-code of this algorithm is given by Algorithm 7.1. The algorithm performs a first level DFS (the blue DFS) to discover accepting states. When such a state is backtracked from, a second level DFS (the red DFS) is launched to see whether this accepting state (now called the *seed*) is reachable from itself and is thus part of an accepting cycle. If this is the case, the algorithm reports a cycle *and exits* on Line 7. It is sufficient to find a path back to the stack of the blue DFS [SE05], hence the *cyan* color in Algorithm 7.1. Correctness depends on the fact that different invocations of the red DFS happen in postorder. The algorithm works in linear time: each state is visited at most twice, since the result of a red DFS can be reused in subsequent red DFSs; states retain their red color.

7.2.3 Parallel LTL Model Checking Algorithms for Shared-Memory Architectures

In the field of parallel LTL model checking, the first algorithms designed targeted distributed memory architectures like clusters of machines. This family of algorithms includes Map [Bri+04], Owcty [ČP03] and Bledge [BBC03b; BBC05b]. It is however well known that this kind of message passing algorithm can be easily ported to shared-memory architectures like multi-core computers although the specifics of these architectures must be considered to achieve good scalability [BBR10b]. Their common characteristic is to rely on some form of breadth-first search (BFS) of the graph that has the advantage of being easily parallelized, unlike depth-first search (DFS) [Rei85]. They hence deliver excellent speedups but sacrifice optimality and the ability to report accepting cycles on-the-fly. A combination of Owcty and Map (Otf_Owcty [BBR09a]) restores “on-the-flyness”, is linear-time for the class of weak LTL properties, and maintains scalability (other algorithms are discussed in Section 5.3).

swarm verification [HJG08] consists of spawning multiple unsynchronized instances of Ndfs each exploring the graph in a random way. Accepting cycles are expected to be reported faster thanks to randomized parallel search, but in the absence of such cycles parallelization does not help. This pragmatic strategy however targets graphs that are too large in any case to be explored in reasonable time. The purpose is then to maximize the graph coverage in a given time frame and thereby increase confidence in the model.

Two recent multi-core algorithms follow the principle of the SWARM technique but deviate from it in that working threads executing Ndfs are synchronized through the sharing of some state attributes. In the first one, LNdfs from Chapter 5, workers share the outcome of the red (nested) search which can then also be used to prune the blue search. Since the blue flags are not shared among threads, the red searches are still invoked in the appropriate DFS postorder. The ENdfs algorithm [EPY11] also allows the sharing of blue flags, but a sequential emergency procedure is triggered if the appropriate invocation order of the red DFS is not respected. Moreover, to maintain correctness, information on a red DFS in progress cannot be transmitted in “real time” to other threads: the states visited by a red DFS are only marked globally red after it has returned.

A thorough experimental comparison of ENdfs and LNdfs in Chapter 6 led to the main conclusion that ENdfs and LNdfs complement each other on a variety of models: the larger amount of information shared by ENdfs can potentially yield a better work distribution, but LNdfs is to be preferred when ENdfs threads often launch unfruitful emergency procedures. Since this emergency procedure launches the sequential Ndfs algorithm, large portions of the graph may then be revisited, in the worst case by all workers. Hence, a combination of ENdfs and LNdfs was proposed in Chapter 6 to remedy the downsides of the two algorithms. The principle of that parallel algorithm (called NMCNdfs) is to run ENdfs but replace its sequential emergency procedure by a parallel LNdfs. Experiments show that this combination pays off: NMCNdfs is always at least as fast as ENdfs or LNdfs.

While NMCNdfs combines the strengths of both earlier algorithms in terms of performance, it also conjoins their memory usage. LNdfs requires $2P + \log_2(P) + 1$ bits per state (2 local colors for all P workers, a synchronization counter and a global red bit) and ENdfs $4P + 3$ (2 local colors plus another 2 for the repair procedure and 3 global bits: $\{dangerous, red, blue\}$). Next to more than doubling the memory usage, the conglomerated algorithm is long and complex.

7.3 A New Combination of Multi-Core NDFS

To mitigate the downsides of NMCNdfs, we present a new algorithm, Cndfs, shown in Algorithm 7.2. Like the previous multi-core algorithms, it is based on the principle of swarm worker threads which traverse state space in randomized depth-first order. (indicated by subscript p here), sharing information via colors stored in the visited states, here: *blue* and *red*. With next-state$_p^b$ (next-state$_p^r$), we denote the random permutation of successors used in the blue (red) dfs by worker p. After randomly visiting all successors (Line 13–Line 15), a state is marked blue at Line 16 (meaning "globally visited") and causing the (other) blue DFS workers to lose the strict postorder property.

Algorithm 7.2 Cndfs, a new multi-core algorithm for LTL model checking

```
 1: procedure CNDFS($s_0, P$)
 2:     $dfsBlue_1(s_0) \,||\, .. \,||\, dfsBlue_P(s_0)$
 3:     report no-cycle
 4: procedure $dfsRed_p(s)$
 5:     $\mathcal{R}_p := \mathcal{R}_p \cup \{s\}$
 6:     for all $s'$ do NEXT-STATE$_p^r(s)$
 7:         if $s'.cyan[p]$ then
 8:             report cycle
 9:         if $s' \notin \mathcal{R}_p \wedge \neg s'.red$ then
10:             $dfsRed_p(s')$
11: procedure $dfsBlue_p(s)$
12:     $s.cyan[p]$ := true
13:     for all $s'$ do NEXT-STATE$_p^b(s)$
14:         if $\neg s'.cyan[p] \wedge \neg s'.blue$ then
15:             $dfsBlue_p(s')$
16:     $s.blue$ := true
17:     if $s \in \mathcal{F}$ then
18:         $\mathcal{R}_p := \emptyset$
19:         $dfsRed_p(s)$
20:         await $\forall s' \in \mathcal{R}_p \cap \mathcal{F} : s \neq s' \Rightarrow s'.red$
21:         for all $s'$ do$\mathcal{R}_p$
22:             $s'.red$ := true
23:     $s.cyan[p]$ := false
```

If the state s is accepting, as usual, a red DFS is launched at Line 19 to find a cycle. At this point, state s is called "the seed". All states visited by $dfsRed_p$ are collected in $\mathcal{R}_p$. If no cycle is found in the red DFS, we can prove that none exists for the seed (Proposition 7.1). Still, because the red DFS was not necessarily called in postorder, other (non-seed, non-red) accepting states may be encountered for which we know nothing, except the fact that they are out of order and reachable from the seed. These are handled after completion of the red DFS at Line 20 by simply waiting for them to become red.

Our proof shows that in this scenario there is always another worker which can color such a state red (Proposition 7.3). The intuition behind this is that there has to be another worker to cause the out-of-order red search in the first place (by coloring blue) and, in the second place, this worker can continue its execution because cyclic waiting configurations can only happen for accepting cycles. These accepting cycles would however be encountered first, causing termination and a cycle report (Line 8). After completion of the waiting procedure, CNDFS marks all states in $\mathcal{R}_p$ globally red, pruning other red DFSs.

The crude waiting strategy requires some justification. After reassessing the ingredients of LNDFS and ENDFS, we found that ENDFS is most effective at parallelizing the blue DFS. This is absolutely necessary since the number of blue states (all reachable states) typically exceeds the number of red states (visited by the red DFS). In ENDFS, however, sharing the blue color often led to the expensive (memory and performance wise) sequential repair procedure [EPY11]. We were unable to construct a correct algorithm that colors both blue and red while backtracking from the respective DFS procedures. Therefore, we now want to investigate whether the intermediate solution, using a wait statement as a compromise, leaves enough parallelism to maintain scalability.

CNDFS only uses $P+2$ bits per state plus the sizes of $\mathcal{R}$. In the theoretical worst case (an accepting initial state), each worker p could collect all states in $\mathcal{R}_p$. In our vast set of experiments (cf. Sec. 7.4), however, we found that the set rarely contains more than one state and never more than thousands, which is still negligible compared to $|\mathcal{S}|$. Our experiments also confirmed that memory usage is close to the expected amount.

Correctness. Proving correctness comprises two parts: proving the consistency of the algorithm, i.e. CNDFS reports a cycle *iff* an accepting cycle is reachable from s_0, and *termination*. The former turned out to be easier than for our previous parallel NDFS algorithms. The wait condition in combination with the late red coloring forces the accepting states to be processed in postorder. Stated differently: a worker makes the effects of its $\mathit{dfsRed}_p(s)$ globally visible (via the red coloring), only after all smaller (in postorder) accepting states t have been processed by some $\mathit{dfsRed}_{p'}(t)$. This is expressed by Lemma 7.3. In Theorem 7.1, we finally show that, if the algorithm terminates without reporting a cycle, all accepting states must be red and consequently cannot lie on a cycle. Proof of termination was already discussed briefly and is detailed in Proposition 7.3.

In the following proofs, the graph coloring and the process counter of Algorithm 7.2 are viewed as state properties of the execution. When writing $\mathit{dfsBlue}_p(s)@19$, we refer to the point in the execution at which a worker p is about to call *dfsRed* on a state s at Line 19, within the execution of $\mathit{dfsBlue}_p(s)$. Graph colorings are denoted as follows: $s \in \mathit{Red}$ means that the *red* flag of s is set to true and similarly $s \in \mathit{Blue}$ means that the *blue* flag is set. For local flags we use $s \in \mathit{Cyan}_p$. Also, we use the modal operator $s \in \Box X$, to express $\forall s' \in \textsc{next-state}(s) : s' \in X$. We show that our propositions hold in the initial state ($\forall s \in \mathcal{S} : s \notin \mathit{Red} \wedge s \notin \mathit{Blue} \wedge \forall p \in \{1 \ldots P\} : s \notin \mathit{Cyan}_p$) and inductively that they are maintained by execution of each statement in the algorithm, considering only lines that can influence the truth value of the proposition. Here an important assumption is that all lines of Algorithm 7.2 are executed atomically.

Lemma 7.1. *Red states have red successors: $\mathit{Red} \subseteq \Box \mathit{Red}$.*

Proof. Initially, there are no red states, hence the lemma holds.

States are colored red when $\mathit{dfsBlue}_p$@22 and are never uncolored red. The set of states $\mathcal{R}_p$ that is colored at Line 22 contains all states reachable from the seed s, but not yet red, since $\mathit{dfsRed}_p(s)$ performed a DFS from s over all non-red states. For the red states reachable from s, the induction hypothesis can be applied, hence there are no non-red states reachable from s that are not in $\mathcal{R}_p$. □

Lemma 7.2. *At Line 20, the set $\mathcal{R}_p$ invariably contains (1) the seed s, (2) all non-red states reachable from s and also (3) all states in the set are reachable from the seed s: $\mathit{dfsBlue}_p(s)@20 \Rightarrow (s \in \mathcal{R}_p \wedge (\forall s' \notin \mathit{Red} : s \rightarrow^* s' \Rightarrow s' \in \mathcal{R}_p) \wedge (\forall s'' \in \mathcal{R}_p \Rightarrow s \rightarrow^* s''))$.*

Proof. At Line 5, we have $s \in \mathcal{R}_p$. For the rest, see proof of Lemma 7.1. □

Lemma 7.3. *The only accepting state that can be colored red at Line 22 (for the first time) is the current seed s itself: $\mathit{dfsBlue}_p(s)@22 \Rightarrow (\mathcal{R}_p \cap \mathcal{F}) \setminus \mathit{Red} \subseteq \{s\}$.*

Proof. Assume $\mathit{dfsBlue}_p(s)$@22 and $\exists a \in (\mathcal{F} \setminus \{s\}) : a \in \mathcal{R}_p$. We show that $a \in \mathit{Red}$.

By Lemma 7.2, $\mathcal{R}_p$ contains at least s and the non-red states reachable from s. After Line 20, all non-seed accepting states in $\mathcal{R}_p$ are red: $(\mathcal{R}_p \cap (\mathcal{F} \setminus \{s\})) \subseteq \mathit{Red}$. Since, $a \in \mathcal{R}_p \cap (\mathcal{F} \setminus \{s\})$, we have: $a \in \mathit{Red}$. □

Proposition 7.1. *The initial invocation of $\mathit{dfsRed}_p(s)$ at Line 19 of Algorithm 7.2 reports a cycle if and only if the seed s belongs to a cycle.*

Proof. $\Leftrightarrow$ is split into two cases: Case $\Rightarrow$: Every state $s' \in \mathit{Cyan}_p$ can reach the seed from $\mathit{dfsBlue}_p(s)$@19 by properties of the DFS stack. Similarly, when $\mathit{dfsRed}_p(s'')$@8, s'' is reachable from the seed s. Therefore, there is a cycle: $s'' \rightarrow s' \rightarrow^* s \rightarrow^* s''$.

Case $\Leftarrow$: assume $\mathit{dfsRed}_p(s)$ at Line 19 finishes normally (without cycle report), while s lies on a cycle C. We show this leads to a contradiction. Since *dfsRed* avoids only red states (Line 9), there would have to be some $r \in C \cap \mathit{Red}$ obstructing the search. The state r can only have been colored red at Line 22 by a worker. W.l.o.g. we investigate the first worker $\mathit{dfsRed}_{p'}$ to have colored r red. p' started for an $s' \in \mathcal{F}$ ($\mathit{dfsBlue}_{p'}(s')$@Line 19).

Since r is not yet red, by Lemma 7.1 $C \cap \mathit{Red} = \emptyset$. Before r is colored red, it is first stored in $\mathcal{R}_{p'}$. By Lemma 7.2, we also have $C \subseteq \mathcal{R}_{p'}$. Either $s' \in C$, then the cycle through s' would have been detected since $s' \in \mathit{Cyan}_{p'}$. Or else $s' \notin C$, and then we have $\{s\} \subseteq (\mathcal{R}_{p'} \setminus \mathit{Red})$ when $\mathit{dfsBlue}_{p'}(s')$@22, contradicting Lemma 7.3. □

Proposition 7.2. *Red states never lie on an accepting cycle.*

Proof. Initially, there are no red states, hence the proposition holds.

When $\mathit{dfsBlue}_p(s)@22$, the set of states $\mathcal{R}_p$ is colored red. The only accepting state to be colored red is the seed s (Lemma 7.3). By Proposition 7.1, this state s does not lie on an accepting cycle. Hence, Proposition 7.2 is preserved. □

Lemma 7.4. *Blue states have blue or cyan successors: $\mathit{Blue} \subseteq \bigcup_p \Box(\mathit{Blue} \cup \mathit{Cyan}_p)$.*

Proof. Initially there are no blue states, hence the lemma holds.

Only at Line 16, states are colored blue, after each successor t has been skipped at Line 14 ($t \in \mathit{Cyan}_p \cup \mathit{Blue}$), or processed by $\mathit{dfsBlue}_p$ at Line 15 (leading to $t \in \mathit{Blue}$). States can be uncolored cyan (Line 23), but only after they have been colored blue (Line 16). □

Lemma 7.5. *A blue accepting state, that is not also Cyan_p for some worker p, must be red: $\forall a \in (\mathit{Blue} \cap \mathcal{F}) : (\forall p \in \{1 \dots P\} : a \notin \mathit{Cyan}_p) \Rightarrow a \in \mathit{Red}$.*

Proof. Assume $s \in (\mathcal{F} \cap \mathit{Blue})$ and $\forall p \in \{1 \dots P\} : s \notin \mathit{Cyan}_p$. We show that $s \in \mathit{Red}$.

State s can only be colored blue when $\mathit{dfsBlue}_p(s)@16$. There, it still retains its cyan coloring from Line 12, it only loses this color at Line 23. But, since $s \in \mathcal{F}$, Line 22 was reached and there $a \in \mathcal{R}_p$ by Lemma 7.2. Hence, $s \in \mathit{Red}$ at Line 23. □

7

Proposition 7.3. *Algorithm 1 always terminates with a report.*

Proof. The individual DFSs cannot proceed indefinitely due to a growing set of red and blue states. So eventually a cycle (Line 8) or no cycle is reported (Line 3). However, progress may also halt due to the wait statement at Line 20. We now assume towards a contradiction that a worker p is waiting indefinitely for a state $a \in \mathcal{F}$ to become red: $\mathit{dfsBlue}_p(s)@20$, $s \neq a$ and $a \in \mathcal{R}_p$. We will show that either a will be colored red eventually, or a cycle would have been detected, contradicting the assumption that p keeps waiting.

By Lemma 7.2, a is reachable from s: $s \rightarrow^+ a$. And by Line 16, $s \in \mathit{Blue}$. Induction on the path $s \rightarrow^* a$, using Lemma 7.4, tells us that: either all states are blue (1) or there is a cyan state on this path (2):

1. $a \in \mathit{Blue} \wedge \forall p \in \{1 \dots P\} : a \notin \mathit{Cyan}_p$: by Lemma 7.5, $a \in \mathit{Red}$, which contradicts the assumption that p is waiting for a to become red. (Note that $\exists p' \in \{1 \dots P\}$: $a \in \mathit{Cyan}_{p'}$ is handled in Case 2.)
2. $\exists c \in \mathit{Cyan}_{p'} : s \rightarrow^+ c \rightarrow^* a$, then depending on the identity of worker p', we have:

A) $p = p'$: but then $\mathit{dfsRed}_p(s)$ would have terminated on cycle detection ($C \equiv s \rightarrow^+ c \rightarrow^+ s$), except when dfsRed_p did not reach c in presence of a red state lying on C. However, this would contradict Proposition 7.2.

B) $p \neq p'$: we show that either p' is executing or going to execute $\mathit{dfsRed}_{p'}(a)$. To eventually color state a red, worker p' must not end up itself in a waiting state: $\mathit{dfsBlue}_{p'}(a')@20$. First, consider the case $a' \neq a$. We also have $s \rightarrow^+ c \rightarrow^* a'$ (stack $\mathit{Cyan}_{p'}$). Hence, by Lemma 7.2, also $a' \in \mathcal{R}_p$. Therefore, we can assume w.l.o.g. that $a = a'$ and only consider $\mathit{dfsBlue}_{p'}(a)@20$. We can repeat the reasoning process of this proof, with $p \equiv p'$ and $s \equiv a$. But since there are finitely many workers, the chain of processes waiting for each other eventually terminates, except the hypothetical configuration of a cyclic waiting dependency, which we consider finally.

To exclude cyclic dependencies, assume $n \geq 2$ workers are simultaneously waiting for each other's seed to be colored red at Line 20. We have: $\mathit{dfsBlue}_1(s_1)@20 \wedge \cdots \wedge \mathit{dfsBlue}_n(s_n)@20 \wedge s_2 \in \mathcal{R}_1 \wedge \cdots \wedge s_1 \in \mathcal{R}_n$. This is only possible if $s_1 \rightarrow^+ s_n \wedge \cdots \wedge s_n \rightarrow^+ s_1$, hence there is a cycle: $s_1 \rightarrow^+ \cdots \rightarrow^+ s_n \rightarrow^+ s_1$. However, this contradicts that the red DFSs (which terminate anyway) would have detected this cycle (Proposition 7.1). □

Theorem 7.1. *Algorithm 7.2 reports an accepting cycle if and only if one is reachable from s_0.*

Proof. By Proposition 7.3, the algorithm is guaranteed to terminate with some report, forming the basis for two cases: Case ⇒: $\mathit{dfsRed}_p(s)@8$ implies a cycle (Proposition 7.1).

Case ⇐: At Line 3, we have $s_0 \in \mathit{Blue}$ and $\forall p \in \{1,\ldots,P\}: \mathit{Cyan}_p = \emptyset$ by properties of DFS. Now, by Lemma 7.4, we have: $\forall s \in \mathcal{G} : s_0 \rightarrow^* s \Rightarrow s \in \mathit{Blue}$. Hence, all reachable accepting states must be red by Lemma 7.5 and do not lie on cycles by Proposition 7.2. □

7.4 Experimental Evaluation

The experiments in Chapter 2, Chapter 6 and Chapter 5 were performed earlier on 16-core machines. Meanwhile, in accordance with Moore's law applied to parallelism, we obtained access to a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 6168). The added parallelism puts extra stress on the scalability of our algorithms and therefore also forces a repetition of some of our previous experiments. We investigated the cause for the performance difference between various algorithms: NMCNDFS from Chapter 6, CNDFS (the current chapter), OTF_OWCTY [BBR09a] and reachability from Chapter 2. Work duplication due to overlapping stacks can cause slowdowns for all multi-core NDFS variants, as can long await cycles in CNDFS. We introduced counters to measure and study these effects. Initially, we focus on models without cycles, the hardest case for these algorithms. Later, we move on to show that CNDFS exhibits the same on-the-fly performance as the multi-core NDFS variants of the previous two chapters.

7.4.1 Experimental Setup

We have used models from the BEEM database [Pel07].[7.1] From each type of model, we selected the variants with more than 9 million states. Our CNDFS algorithm is implemented in the multi-core backend of the LTSMIN model checking toolset [LPW11a], based on a dedicated scalable lock-free hash table. For a fair comparison with previous algorithms, we also implemented some NDFS optimizations (see Section 5.4.4), *all-red* and *early cycle detection*. All-red colors a state s red, if all its successors are red after Line 15 of Algorithm 7.2; correctness follows from Proposition 7.2. Early cycle detection detects certain accepting cycles already in the blue search.

LTSMIN 1.9[7.2] was compiled with GCC 4.4.2 (with optimization -O2) and ran with: `dve2lts-mc --threads=N -s28 --state=table --strategy=name`, where `name` can be `cndfs` or `endfs,lndfs`, representing the different algorithms (see Chapter 6). We used DIVINE 2.5.2 [Bar+10] as OTF_OWCTY implementation, compiled and run with equivalent parameters. Since LTSMIN reuses its next-state function, both tools are comparable (see Chapter 2).

7.4.2 Models without Accepting Cycles

In Chapter 6, we showed that NMCNDFS was the best scaling LTL model checking algorithm on 16 core machines. Hence, we started comparing plain CNDFS and NMCNDFS. Table 7.1 shows the average runtime of both algorithms over five runs on all benchmarks, for 1, 8, 16 and 48 cores. The performance of CNDFS is on par with that of NMCNDFS, which is impressive considering the crude waiting strategy of the algorithm.

We confirmed that the time spent at the **await** statement (Line 20 in Algorithm 7.2) is indeed less than 0.01 sec on runs with 48 cores for all BEEM models. This is caused by the all-red extension, which greatly reduces work in the red DFS. Without all-red, we observed high waiting times causing speeddowns with more than 8 cores.

[7.1] All results are available at `http://fmt.cs.utwente.nl/tools/ltsmin/atva-2012/`.

[7.2] `http://fmt.cs.utwente.nl/tools/ltsmin/` next branch, v.1.9 is due Aug. 2012.

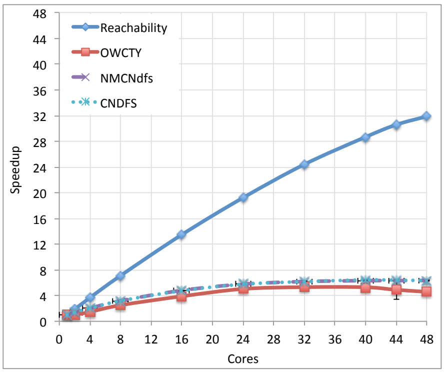


Figure 7.1: Speedup `anderson.6.p4`

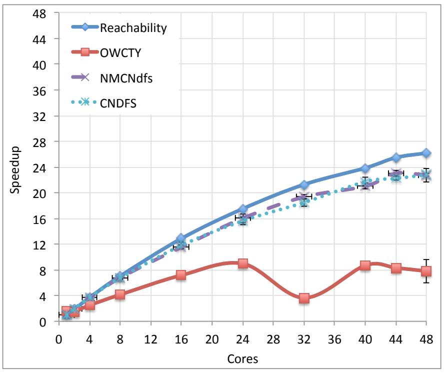


Figure 7.2: Speedup `bobp.4.p3`

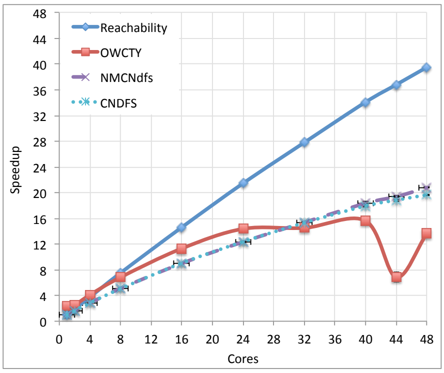


Figure 7.3: Speedup `elevator.5.p3`

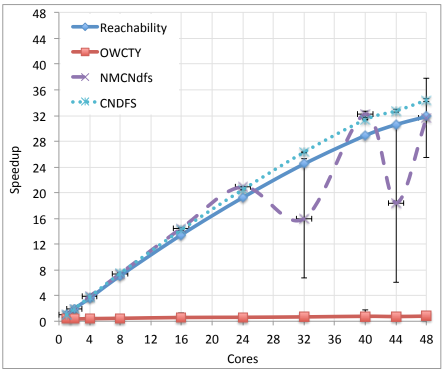


Figure 7.4: Speedup `ldr_flt.6.p2`

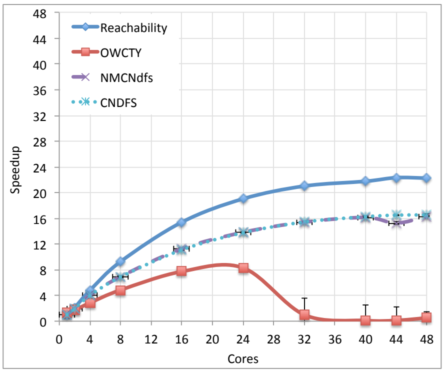


Figure 7.5: Speedup `rether.7.p5`

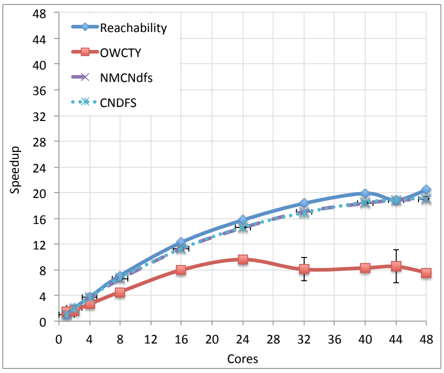


Figure 7.6: Speedup `synapse.7.p3`

Table 7.1: Runtimes (sec) with NMCNDFS and CNDFS for all models.

	States	NMCNDFS				CNDFS			
		1	8	16	48	1	8	16	48
anderson.6.prop2	2.9E+7	144.0	46.5	31.3	23.7	146.6	47.2	31.7	23.6
anderson.6.prop4	3.6E+7	172.9	54.1	35.8	27.1	172.9	54.3	36.2	27.3
bakery.9.prop2	1.1E+8	378.9	62.4	35.5	18.9	368.9	64.6	36.9	19.9
bopdp.4.prop3	2.4E+7	74.7	11.1	6.4	3.3	74.9	11.0	6.4	3.3
elevator.5.prop3	2.1E+8	1,387.0	272.7	154.6	67.3	1,390.8	273.3	154.2	71.2
elevator2.3.prop4	1.5E+7	134.6	25.7	15.5	8.7	136.9	25.5	15.8	8.7
lamport.7.prop4	7.4E+7	299.2	61.9	35.5	23.5	297.7	60.8	35.9	22.9
leader_election.6.prop2	3.6E+7	1,495.2	189.5	194.5	31.9	1,501.9	190.1	94.5	32.2
leader_filters.6.prop2	2.1E+8	444.2	59.5	30.4	12.4	439.0	59.5	31.0	12.8
leader_filters.7.prop2	2.6E+7	73.5	9.7	6.4	2.3	73.3	9.4	5.0	2.3
lup.4.prop2	9.1E+6	19.6	4.7	2.9	2.2	19.5	4.7	2.9	2.1
mcs.5.prop4	1.2E+8	538.3	147.0	89.9	58.2	540.3	146.5	90.2	57.1
peterson.5.prop4	2.6E+8	1,186.0	229.4	135.3	84.9	1,146.5	226.2	133.0	83.6
rether.7.prop5	9.5E+6	43.0	6.2	3.8	2.7	43.6	6.3	3.9	2.6
synapse.7.prop3	1.5E+7	37.3	5.6	3.3	2.0	37.1	5.5	3.3	1.9

Additionally, we made a comparison of *absolute speedups* so as to investigate the properties of the different algorithms (Figure 7.1–7.6). For CNDFS and NMCNDFS, we included the standard deviation of the 5 runs as error bars. As the base case for the speedup of the LTL algorithms, we used CNDFS: $S_n = T_1^{\text{CNDFS}}/T_n^{algo}$, for reachability we used its own base case. We included reachability from Chapter 2 to serve as a reference point for CNDFS. We were primarily interested in comparing the scalability of CNDFS with our parallel reachability implementation. After all, sequential NDFS visits each state at most twice; once in the blue DFS and possibly once in the red DFS.

We notice that NMCNDFS and CNDFS are always faster than OTF_OWCTY. The error bars show less robust, fluctuating runtimes for NMCNDFS (e.g. `leader_filters`). Upon investigation it turned out that NMCNDFS sometimes launches a repair search even though we also fitted its ENDFS search with all-red. When only few workers enter this repair search, it cannot be parallelized. In these cases, CNDFS turns to waiting, a

Table 7.2: Expected and actual speedups for Cndfs according to speedup model.

	$\lvert\mathcal{G}\rvert$	B_{48}^{fsh}	R_{48}^{fsh}	S_{48}^{reach}	D_{48}^{fsh}	E_{48}^{fsh}	S_{48}^{fsh}	D_{48}^{cndfs}	E_{48}^{cndfs}	S_{48}^{cndfs}
anderson.6.prop2	3E+7	1E+8	4E+3	30.6	3.6	8.6	6.4	4.7	6.6	4.6
anderson.6.prop4	4E+7	1E+8	3E+3	31.9	3.1	10.2	6.4	4.0	8.0	5.0
bakery.9.prop2	1E+8	2E+8	4E+5	28.0	1.4	20.5	19.2	1.6	17.2	14.3
bopdp.4.prop3	2E+7	3E+7	6E+5	26.2	1.3	20.0	22.8	1.8	14.6	15.5
elevator.5.prop3	2E+8	4E+8	2E+3	39.5	1.9	21.0	19.5	3.2	12.5	9.0
elevator2.3.prop4	1E+7	3E+7	2E+6	33.2	2.0	16.3	15.8	5.3	6.3	8.0
lamport.7.prop4	7E+7	1E+8	6E+4	30.5	1.7	17.6	13.3	1.9	15.8	10.4
leader_el.6.prop2	4E+7	4E+7	4E+4	40.5	1.0	40.4	46.6	1.0	40.3	39.5
leader_filt.6.prop2	2E+8	2E+8	7E+5	31.9	1.0	31.6	34.4	1.0	30.7	29.9
leader_filt.7.prop2	3E+7	3E+7	1E+5	27.6	1.0	27.4	31.9	1.0	26.9	27.8
lup.4.prop2	9E+6	2E+7	4E+3	17.7	2.5	7.1	9.7	4.6	3.8	6.3
mcs.5.prop4	1E+8	3E+8	1E+4	34.4	2.2	15.7	9.5	2.7	12.6	7.3
peterson.5.prop4	3E+8	4E+8	8E+5	34.1	1.6	20.9	13.9	1.9	18.3	11.0
rether.7.prop5	1E+7	2E+7	1E+5	22.3	1.9	11.9	16.5	2.4	9.2	14.3
synapse.7.prop3	2E+7	2E+7	1E+2	20.4	1.1	17.9	19.2	1.2	17.0	18.6

much better strategy, since in total it waits less than 0.01 sec. Also, reachability scales sometimes twice as good as Cndfs; `anderson` even scales 5 times better.

We investigated why the speedup of Cndfs differs from reachability. We measured the total amount of work performed by all workers. In particular, we counted for each benchmark the state count $\lvert\mathcal{G}\rvert$, and the numbers B_n and R_n, the total number of blue and red colorings in a run with n cores. Next, we estimate the duplicate work compared to reachability as $D_n := (R_n + B_n)/\lvert\mathcal{G}\rvert$. We view the reachability speedups S_n^{reach} as ideal (under the plausible assumption that maximal speedup is limited mostly by the memory bandwidth). Hence we can calculate the expected speedup $E_n^{alg} := S_n^{reach}/D_n^{alg}$ for $alg \in \{fsh, cndfs\}$ where fsh is Cndfs with heuristics (see below).

Table 7.2 compares these estimated speedups E_{48} with the actual speedups S_{48}. Note that the estimated speedups for Cndfs E_{48}^{cndfs} correspond nicely with the measured speedups S_{48}^{cndfs} for many benchmarks. Hence, we conclude that the variation in speedup is mainly caused by the degree of work duplication.

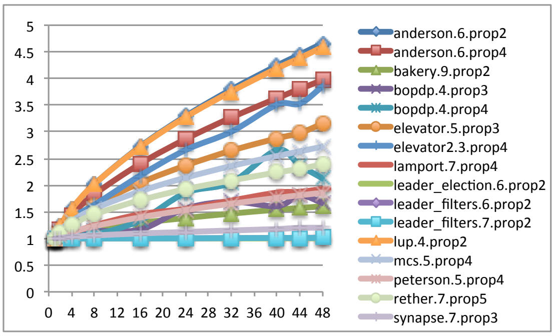


Figure 7.7: Work duplication per core per model

To combat work duplication, we reuse the "fresh successor heuristics" (Chapter 6). If possible, this randomly selects a successor that has not yet been visited before. It is available in the LTSMIN toolset (`--perm=dynamic`). As a consequence, workers tend to be directed towards different regions of the state space, reducing work duplication.

These results are also shown in Table 7.2: D_{48}^{fsh}, E_{48}^{fsh} and S_{48}^{fsh} together with the measured amount of blue and red colorings: B_{48}^{fsh} and R_{48}^{fsh}. The heuristic approach shows quite some improvement, sometimes halving work duplication and doubling speedup (see `elevator`). Still we see duplications as high as 3.6 (see `anderson`). Note that the earlier benchmarks in Figure 7.1–7.6 already use this heuristic.

We expect that in the near future, the number of cores in many-core systems will still grow. Will this increase work duplication and put a limit on speedup of CNDFS? To give an indication, we plotted the increase of work duplication with a growing number of cores with fresh successor heuristics (Figure 7.7). The increase is sublinear, so we expect that speedups will be maintained on larger many-core systems with similar architecture and scaling bandwidth characteristics.

Finally, we note that the size of the input has a small yet significant effect on the amount of work duplication; models with higher state count have less duplication.

7.4.3 Models with Accepting Cycles

In Chapter 6, we experimented thoroughly to investigate the "on-the-flyness" of SWARM NDFS and LNDFS. We noticed that the benefits of independent SWARM verification is limited, on average only yielding a speedup of 2-8 on 16 core machines. LNDFS however yielded speedups from 4 to 14. Combined with the fresh successor heuristic speedups often became superlinear. This is not surprising [RK88], because we verified that in those cases there are many cycles, distributed evenly over the state space.

We performed the same experiments again with CNDFS on a 48 core machine. The results in Table 7.3 show that CNDFS exhibits the same desirable on-the-fly behavior as LNDFS, scaling up to 48 cores. For completeness, we also included the runtimes and speedups with OTF_OWCTY in the table. Its heuristic on-the-fly behavior seems to fail in some cases. It must however be mentioned that the on-the-fly capabilities of this algorithm have recently been improved by changing its exploration order to be more DFS-like [Bar+11a]. In [Bar+11a], performance is reported on par with LNDFS. Unfortunately, we do not have the means (a GPGPU) to reproduce any results here.

Table 7.3: On-the-fly behavior of parallel LTL algorithms

		1 core	48 core				OTF_OWCTY	
		NDFS	LNDFS		CNDFS		1 core	48 core
	model	Rand.	Rand.	Fsh	Rand.	Fsh.	Static	Rand.
Runtimes (sec)	anderson.8.prop3	36.4	4.0	1.2	4.1	0.2	2858.8	1433.2
	bakery.7.prop3	3.2	0.4	0.2	0.3	0.2	2.2	5.2
	bakery.8.prop4	15.7	0.6	0.3	0.6	0.3	73.4	14.3
	elevator2.3.prop3	8.4	1.4	0.2	1.4	0.2	432.3	192.5
	extinction.4.prop2	2.2	0.1	0.1	0.1	0.1	1.8	1.7
	peterson.6.prop4	29.1	0.6	0.5	0.9	0.5	668.4	705.7
	szymanski.5.prop4	1.7	1.4	0.1	1.3	0.2	2.1	376.4
Speedups	anderson.8.prop3		9.1	31.1	8.8	175.0		2.0
	bakery.7.prop3		8.7	18.3	10.9	21.2		0.4
	bakery.8.prop4		28.3	51.1	26.2	48.9		5.1
	elevator2.3.prop3		6.0	51.5	5.9	52.1		2.2
	extinction.4.prop2		30.4	32.1	18.5	28.8		1.0
	peterson.6.prop4		46.1	59.8	33.0	62.4		0.9
	szymanski.5.prop4		1.2	12.0	1.3	10.9		0.0

7.4.4 Counterexample Length

Lengthy counterexamples are hard to study even with good model checking tools. Therefore, finding short counterexamples is quite an important property of model checking algorithms. Strict BFS algorithms deliver minimal counterexamples, while DFS algorithms can yield very long ones. Once the strict BFS/DFS order is loosened, these properties can be expected to fade. This is exactly what both Otf_Owcty and Cndfs do. We studied the length of the counterexamples that these algorithms produce.

For this purpose, 45 models with counterexamples were selected from the Beem database, all algorithms run 5 times, and computed the average counterexample length and standard deviation. The results are summarized in scatter plots with bars representing the standard deviation. Figure 7.8 compares randomized sequential Ndfs (vertical axis) against sequential Otf_Owcty (horizontal axis). Figure 7.9 compares the results of Cndfs with fresh successor heuristic (fsh) against Otf_Owcty on 48 cores.

In the sequential case, most bars are above the equilibrium so, as expected, Ndfs produces longer counterexamples of more variable size compared to Otf_Owcty (which we could not randomize). The parallelism of a 48-core run, however, greatly stabilizes and reduces counterexample lengths for Cndfs, while the randomness added by parallelism introduces variation for Otf_Owcty (horizontal bars). In most cases, Cndfs' counterexamples become shorter than those of Otf_Owcty, a surprising result considering the latter's BFS-like order. An outlier is `plc.4`: All Ndfs algorithms consistently find a counterexample of length 216, while Otf_Owcty finds one of length 2!

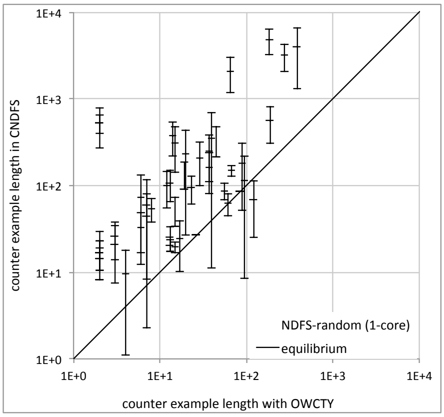


Figure 7.8: Ndfs vs Otf_Owcty (1 core)

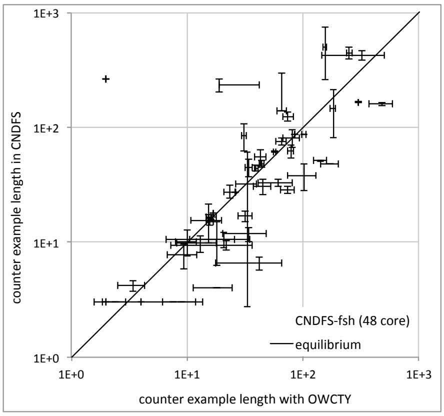


Figure 7.9: Fsh vs Otf_Owcty (48 cores)

7.5 Conclusion

We presented CNDFS, a new multi-core NDFS algorithm. It can detect accepting cycles on-the-fly, and its worst case execution time is linear in the size of the input graph. We showed that CNDFS is considerably simpler than its predecessor NMCNDFS, because of the deep integration of ENDFS and LNDFS. Experiments show that CNDFS delivers performance and scalability similar to its predecessors, but achieves this more robustly. Hence CNDFS is currently the fastest multi-core LTL model checking algorithm in practice. Moreover, CNDFS halves the memory requirements per state per worker thread; an important factor since the total number of cores keeps growing.

Experiments revealed that the main bottleneck for perfect scalability of CNDFS is currently the work duplication due to overlapping stacks. Forcing workers to favor “fresh” successor states already decreases duplication. The same experiments indicate that work duplication grows only linearly in the number of cores, and decreases for larger input sizes. From this we conjecture that CNDFS will scale even beyond 48 cores.

CNDFS shares global information only during or even after backtracking, which leads to potential work duplication. In the worst case, every worker could visit the whole graph, blocking any speedup. During our extensive experiments with the entire BEEM database we have not found such cases. However, we did observe work duplication of factor 3 on 48 cores, so there is room for improvement.

Designing a provably scalable, linear-time algorithm remains an open question. Such an algorithm should cause negligible duplicate work and avoid synchronization by await statements. So far, we have not been able to come up with a correct algorithm without await statements or a repair procedure. An improvement might be to invent a smart work stealing scheme, in which workers can cooperate instead of waiting.

Finally, we demonstrated that counterexamples in CNDFS become shorter with more parallelism, even shorter than counterexamples in parallel BFS-based OTF_OWCTY. This is an interesting and desirable property for a model checking algorithm. It is intriguing that our parallel DFS based algorithm shows good scalability and short counterexamples, usually attributed to BFS algorithms, while still maintaining the linear-time and on-the-fly properties expected from a DFS algorithm.

8 Improved On-The-Fly Livelock Detection

Alfons Laarman, David Faragó

Abstract

Until recently, the preferred method of livelock detection was via LTL model checking, which imposes complex constraints on partial-order reduction (POR), limiting its performance and parallelization. The introduction of the DFS_{FIFO} algorithm by Faragó et al. showed that livelocks can theoretically be detected faster, simpler, and with stronger POR.

For the first time, we implement DFS_{FIFO} and compare it to the LTL approach by experiments on four established case studies. They show the improvements over the LTL approach: DFS_{FIFO} is up to 3.2 times faster, and it makes POR up to 5 times better than with SPIN's NDFS.

Additionally, we propose a parallel version of DFS_{FIFO}, which demonstrates the efficient combination of parallelization and POR. We prove parallel DFS_{FIFO} correct and show why it provides stronger guarantees on parallel scalability and POR compared to LTL-based methods. Experimentally, we establish almost ideal linear parallel scalability and POR close to the POR for safety checks: easily an order of magnitude better than for LTL.

About this chapter: The current chapter is based on the paper *"Improved on-the-Fly Livelock Detection"*, which was published at NFM 2013 [LF13].

8.1 Introduction

Context. In the *automata-theoretic* approach to *model checking* [VW86], the behavior of a system-under-verification is modeled, along with a property that it is expected to adhere to, in some concise specification language. This model $\mathcal{M}$ is then unfolded to yield a *state-space automaton* $\mathcal{A}_{\mathcal{M}}$ (cf. Definition 8.1). All *safety properties*, e.g. *deadlocks* and *invariants*, can be checked directly on the states in $\mathcal{A}_{\mathcal{M}}$ as they represent all configurations of $\mathcal{M}$. This check can be done during the unfolding, *on-the-fly*, saving resources when a property violation is detected early on.

For more complicated properties, like *liveness properties* [BK08], $\mathcal{A}_{\mathcal{M}}$ is interpreted as an ω-automaton whose language $\mathcal{L}(\mathcal{A}_{\mathcal{M}})$ represents all infinite executions of the system. A property φ, expressed in linear temporal logic (LTL), is likewise translated to a Büchi or ω-automaton $\mathcal{A}_{\neg\varphi}$ representing all *undesired* infinite executions. The intersected language $\mathcal{L}(\mathcal{A}_{\mathcal{M}}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$ now consists of all counterexample traces, and is empty if and only if the system is correct with respect to the property. The emptiness check is reduced to the graph problem of finding cycles with designated accepting states in the *cross product* $\mathcal{A}_{\mathcal{M}} \otimes \mathcal{A}_{\neg\varphi}$ (cf. Section 8.2). The nested depth-first search (NDFS) algorithm [Cou+92] solves it in time linear to the size of the product and on-the-fly as well.

Motivation. The model checking approach is limited by the so-called *state-space explosion* problem [BK08], which states that $\mathcal{A}_{\mathcal{M}}$ is exponential in the components of the system, and $\mathcal{A}_{\neg\varphi}$ exponential in the size of φ. Luckily, several remedies exist to this problem: patience, specialization and state-space reduction techniques.

State-space reduction via *partial-order reduction* (POR) prunes $\mathcal{A}_{\mathcal{M}}$ by avoiding irrelevant interleavings of local components in $\mathcal{M}$ [KP88b; Val91b]: only a sufficient subset of successors, the *ample set*, is considered in each state (cf. Section 8.2). For safety properties, the ample set can be computed locally on each state. For liveness properties, however, an additional condition, the *cycle proviso*, is needed to avoid the so-called *ignoring problem* [EP10]. POR can yield exponential reductions.

Patience also pays off exponentially as Moore's law stipulates that the number of transistors available in CPUs and memory doubles every 18 months [Moo65]. Due to this effect, model checking capabilities have increased from handling a few thousand states to covering billions of states recently (the current chapter and [BL13a]). While this trend happily continues to increase memory sizes, it recently stopped benefitting the sequential performance of CPUs because physical limitations were reached. Instead, the available parallelism on the chips is rapidly increasing. So, for runtime to benefit from Moore's law, we must parallelize our algorithms.

Specialization towards certain subclasses of liveness properties, finally, can also help to solve them more efficiently. For instance, a limitation to the *CTL* and the *weak LTL* fragments was shown to be efficiently parallelizable [SZB12; BBR09a]. *In the current chapter, we limit the discourse to livelock properties*, an important subclass (used in about half of the case studies of [Llu][8.1] and a third of [Pel07]) that investigates starvation, occurring if an infinite run does not make *progress* infinitely often. The definition of progress is up to the system designer and could for instance refer to an increase of a counter or access to a shared resource. The SPIN model checker allows the user to specify progress statements inside the specification of the model [Hol11], which are then represented in the model by the state label 'progress' and referenced by the predefined *progress* LTL property [HPY96]. Until 1996, SPIN used a specific livelock verification algorithm based on the original *divergence detection* algorithm proposed by Valmari [Val93] from 1993. Section 6 of [HPY96] states that it was replaced by LTL model checking due to its incompatibility with POR.

Problem. Parallelization of LTL model checking is hard. The current state-of-the-art reveals that parallel cycle detection algorithms either raise the worst-case complexity to L^2 [BBR09a] or to $L \cdot P$ [Eva+12], where L is the size of the LTL cross product and P the number of processors. Moreover, its additional constraints on POR severely limit its reduction capabilities, even if implemented with great care (see models allocation, cs and p2p in Table 1 in the appendix of [EP10]). Last but not least, these constraints also limit the parallelization of POR [BBR10a].

We want to investigate whether better results can be obtained for livelocks, for which recently an efficient algorithm was proposed by Faragó et al. [FS09]: DFS_{FIFO}. In theory, it has additional advantages over the LTL approach:

1. It uses the progress labels in the model directly without the definition of an LTL property; avoiding the calculation of a larger cross product.
2. It requires only one pass over the state space, while the NDFS algorithm, typically used for liveness properties, requires two.
3. It eliminates the need for the expensive cycle proviso with POR.
4. It finds the shortest counterexample with respect to progress.

But DFS_{FIFO} is yet to be implemented and evaluated experimentally, so its practical performance is unknown. Additionally, a few hypotheses stand unproven:

1. The algorithm's strategy to delay progress as much as possible, may also be a good heuristic for finding livelocks early, making it more on-the-fly.
2. Its POR performance might be close to that of safety checks, because the cycle proviso is no longer required [FS09], and the visibility proviso (see Table 8.1) is also positively influenced by the postponing of progress.

[8.1] PROMELA database: `http://www.albertolluch.com/research/promelamodels`.

3. The use of *progress transitions* instead of *progress states* is possible, semantically more accurate, and can yield better partial-order reductions.

Furthermore, no parallelization exists for the DFS$_{\text{FIFO}}$ algorithm.

Contributions. We implemented the DFS$_{\text{FIFO}}$ algorithm in the LTSMIN [BPW10; LPW11a; BL13a], with both progress states and transitions. For the latter, we extended theory, algorithms, proofs, models and implementation. We compare the runtime and POR performance to that of LTL approaches using NDFS. For DFS$_{\text{FIFO}}$, we also investigate the effect of using progress transitions instead of states on POR.

Additionally, we present a parallel livelock algorithm based on DFS$_{\text{FIFO}}$, together with a proof of correctness. While the algorithm builds on previous efficient parallelizations of the NDFS algorithm [Eva+12; Laa+11; LP11], we show that it has stronger guarantees for parallel scalability due to the nature of the underlying DFS$_{\text{FIFO}}$ algorithm. At the same time, it retains all the benefits of the original DFS$_{\text{FIFO}}$ algorithm. This entails the redundancy of the cycle proviso, hence allowing for parallel POR with almost the same reductions as for safety checks.

Our experiments confirm the theoretical expectations: using DFS$_{\text{FIFO}}$ on four case studies, we observed up to 3.2 times faster runtimes than with the use of an LTL property and the NDFS algorithm, even compared to measurements with the SPIN model checker. But we also confirm all hypotheses of Faragó et al.: the algorithm is more on-the-fly, and POR performance is closer to that of safety checks than the LTL approach, making it up to 5 times more effective than POR in SPIN. Our parallel version of the algorithm can work with POR and features the expected linear scalability. Its combination with POR easily outperforms other parallel approaches [BBR10a].

Overview. In Section 8.2, we recapitulate the intricacies of livelock detection via LTL and via non-progress detection, as well as POR. In Section 8.3, we introduce DFS$_{\text{FIFO}}$ for progress transitions with greater detail and formality than in [FS09], as well as its combination with POR. Thereafter, in Section 8.4, we provide a parallel version of DFS$_{\text{FIFO}}$ with a proof of correctness, implementation considerations, and an analysis on its scalability. Section 8.5 presents the experimental evaluation, comparing DFS$_{\text{FIFO}}$'s (POR) performance and scalability against the (parallel) LTL algorithms in SPIN [Hol12; HPY96], DIVINE [BBR10a; BBR09a], and LTSMIN [BL13a; LPW11a]. We conclude in Section 8.6.

8.2 Preliminaries

8.2.1 Model Checking of Safety Properties

Explicit-state model checking algorithms construct $\mathcal{A}_\mathcal{M}$ on-the-fly starting from the initial state s_0, and recursively applying the next-state function NEXT-STATE() to discover all reachable states $\mathcal{R}_\mathcal{M}$. This only requires storing states (no transitions). As soon as a counterexample is discovered, the exploration can terminate early, saving resources. To reason about these algorithms, it is however easier to consider $\mathcal{A}_\mathcal{M}$ structurally as a graph.

Definition 8.1 (State-Space Automaton)**.** *An automaton is a quintuple $\mathcal{A}_\mathcal{M} = (\mathcal{S}_\mathcal{M}, s_0, \Sigma, \mathcal{T}_\mathcal{M}, L)$, with $\mathcal{S}_\mathcal{M}$ a finite set of states, $s_0 \in \mathcal{S}_\mathcal{M}$ an initial state, Σ a finite set of action labels, $\mathcal{T}_\mathcal{M} \colon \mathcal{S}_\mathcal{M} \times \Sigma \to \mathcal{S}_\mathcal{M}$ the transition relation, and $L \colon \mathcal{S}_\mathcal{M} \to 2^{AP}$ a state labeling function, over a set of atomic propositions AP.*

We also use the recursive application of the transition relation $\mathcal{T}$: $s \xrightarrow{\pi}{}^{+} s'$ iff π is a path in $\mathcal{A}_\mathcal{M}$ from s to s', or $s \xrightarrow{\pi}{}^{*} s'$ if possibly $s = s'$. We treat a path π dually as a sequence of states and a sequence of actions, depending on the context. We omit the subscript $\mathcal{M}$ whenever it is clear from the context.

Now, we can define: the reachable states $\mathcal{R}_\mathcal{M} = \{s \in \mathcal{S}_\mathcal{M} \mid s_0 \to^* s\}$, the function NEXT-STATE$() \colon \mathcal{S}_\mathcal{M} \to 2^\Sigma$, such that NEXT-STATE$(s) = \{\alpha \in \Sigma \mid \exists s' \in \mathcal{S}_\mathcal{M} : (s, \alpha, s') \in \mathcal{T}_\mathcal{M}\}$ and $\alpha(s)$ as the unique next-state for s, α if $\alpha \in$ NEXT-STATE(s), i.e. the state t with $(s, \alpha, t) \in \mathcal{T}_\mathcal{M}$. Note that a state $s \in \mathcal{S}$ comprises the variable valuations and process counters in $\mathcal{M}$. Hence, we can use any proposition over these values as an atomic proposition representing a state label. For example, we may write progress $\equiv$ Peterson0 $= CS$ to have progress $\in L(s)$ iff s represents a state where process instance 0 of Peterson is in its critical section CS. Or we can write error $\equiv N > 1$ to express the mutual exclusion property, with N the number of processes in CS. These state labels can then be used to check safety properties using *reachability*, e.g., an *invariant* '$\neg$*error*$'$ to check mutual exclusion in $\mathcal{M}$.

8.2.2 LTL Model Checking

For an LTL property, the property φ is transformed to an ω-automaton $\mathcal{A}_{\neg\varphi}$ as detailed in [VW86]. Structurally, the ω-automaton extends a normal automaton (Definition 8.1) with dedicated accepting states (see Definition 8.2). Semantically, these accepting states mark those cycles that are part of the ω-regular language $\mathcal{L}(\mathcal{A}_{\neg\varphi})$ as defined in Definition 8.3.

To check correctness of $\mathcal{M}$ with respect to a property φ, the *cross product* of $\mathcal{A}_{\neg\varphi}$ with the state-space automaton $\mathcal{A}_{\mathcal{M}}$ is calculated: $\mathcal{A}_{\mathcal{M}\times\varphi} = \mathcal{A}_{\mathcal{M}} \otimes \mathcal{A}_{\neg\varphi}$. The states of $\mathcal{S}_{\mathcal{M}\times\varphi}$ are formed by tuples (s,s') with $s \in \mathcal{S}_{\mathcal{M}}$ and $s' \in \mathcal{S}_{\neg\varphi}$, with $(s,s') \in \mathcal{F}$ iff $s' \in \mathcal{F}_{\neg\varphi}$. Hence, the number of possible states $|\mathcal{S}_{\mathcal{M}\times\varphi}|$ equals $|\mathcal{S}_{\mathcal{M}}| \cdot |\mathcal{S}_{\neg\varphi}|$, whereas the number of reachable states $|\mathcal{R}_{\mathcal{M}\times\varphi}|$ may be smaller. The transitions in $\mathcal{T}_{\mathcal{M}\times\varphi}$ are formed by synchronizing the transition labels of $\mathcal{A}_{\neg\varphi}$ with the state labels in $\mathcal{A}_{\mathcal{M}}$. For an exact definition of $\mathcal{T}_{\mathcal{M}\times\varphi}$, we refer to [BK08].

Definition 8.2 (Accepting states)**.** *The set of accepting states $\mathcal{F}$ corresponds to those states with a label* accept $\in AP$*:* $\mathcal{F} = \{s \in \mathcal{S} \mid \textsf{accept} \in L(s)\}$.

Definition 8.3 (Accepting run)**.** *A lasso-formed path $s_0 \xrightarrow{v}{}^{*} s \xrightarrow{w}{}^{+} s$ in $\mathcal{A}$, with $s \in \mathcal{F}$, constitutes an accepting run, part of the language of $\mathcal{A}$: $vw^{\omega} \in \mathcal{L}(\mathcal{A})$.*

As explained in Section 8.1, the whole procedure of finding counterexamples to φ for $\mathcal{M}$ is now reduced to the graph problem of finding *accepting run*s in $\mathcal{A}_{\mathcal{M}\times\varphi}$. This can be solved by the nested depth-first search (NDFS) algorithm, which does at most two explorations of all states $\mathcal{R}_{\mathcal{M}\times\varphi}$. Since $\mathcal{A}_{\mathcal{M}\times\varphi}$ can be constructed on-the-fly, NDFS saves resources when a counterexample is found early on.

8

8.2.3 Livelock Detection

Livelocks form a specific, but important subset of the liveness properties and can be expressed as the *progress* LTL property: $\Box\Diamond$progress, which states that on each infinite run, *progress* needs to be encountered infinitely often. As the LTL approach synchronizes the state labels of $\mathcal{A}_{\mathcal{M}}$ (see Definition 8.3), it requires that progress is defined on states as in Definition 8.4.

Definition 8.4 (Progress states)**.** *The set of progress states $\mathcal{S}^{\mathcal{P}}$ corresponds to those states with a state label* progress $\in AP$*:* $\mathcal{S}^{\mathcal{P}} = \{s \in \mathcal{S} \mid \textsf{progress} \in L(s)\}$.

Definition 8.5 (Non-progress cycle)**.** *A reachable cycle π in $\mathcal{A}_{\mathcal{M}}$ is a non-progress cycle (**NPcycle**) iff it contains no progress $\mathcal{P}$.*

We define $\mathcal{NP}$ as a set of states: $\mathcal{NP} = \{s \in \mathcal{S}_{\mathcal{M}} \mid \exists\pi : s \xrightarrow{\pi}{}^{+} s \land \pi \cap \mathcal{P} = \emptyset\}$.

Theorem 8.1. *Under $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$, $\mathcal{A}_{\mathcal{M}}$ contains a **NPcycle** iff the cross product with the progress property $\mathcal{A}_{\mathcal{M}\times\Box\Diamond\textsf{progress}}$ contains an accepting cycle.*

Livelocks can however also be detected directly on $\mathcal{A}_{\mathcal{M}}$ if we consider for a moment that a counterexample to a livelock is formed by an infinite run that lacks progress $\mathcal{P}$, with $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$. By proving absence of such *non-progress* cycles (Definition 8.5), we do essentially the same as via the progress LTL property, as Theorem 8.1 shows (see [HPY96] for the proof and details). This insight led to the proposal of dedicates algorithms in [HPY96; FS09] (cf. DFS$_{\text{FIFO}}$ in Section 8.3), requiring $|\mathcal{R}_{\mathcal{M}}|$ time units to prove livelock freedom. The automaton $\mathcal{A}_{\neg\Box\Diamond\mathsf{progress}}$ consists of exactly two states [HPY96], hence $|\mathcal{R}_{\mathcal{M}}| \cdot 2 \leq |\mathcal{R}_{\mathcal{M}\times\varphi}|$. This, combined with the revisits of the NDFS algorithm, *makes the LTL approach up to 4 times as costly as DFS$_{FIFO}$.*

8.2.4 Partial-Order Reduction

To achieve the reduction as discussed in the introduction, POR replaces the NEXT-STATE() with an *ample* function, which computes a sufficient subset of NEXT-STATE() to explore only relevant interleavings w.r.t the property [KP88b].

For deadlock detection, *ample* only needs to fulfill the *emptiness proviso* and *dependency proviso* (Table 8.1). The provisos can be deduced locally from s, NEXT-STATE(s), and dependency relations $D \subseteq \Sigma_{\mathcal{M}} \times \Sigma_{\mathcal{M}}$ that can be statically overestimated from $\mathcal{M}$, e.g. $(\alpha,\beta) \in D$ if α writes to those variables that β uses as guard [Pat11]. For a precise definition of D consult [KP88b; Val91b].

In general, the model checking of an LTL property (or invariant) φ requires two additional provisos to hold: the *visibility proviso* ensures that traces included in $\mathcal{A}_{\neg\varphi}$ are not pruned from $\mathcal{A}_{\mathcal{M}}$, the *cycle proviso* prevents the so-called *ignoring problem* [EP10]. The strong variant **C3** (stronger than A4 in [BK08, Sec. 8.2.2]) is already hard to enforce, so often an even stronger condition, e.g. **C3'**, is implemented. While visibility can still be checked locally, the cycle proviso is a global property, that complicates parallelization [BBR10a]. Moreover, the NDFS algorithm revisits states, which might cause different ample sets for the same states, because the procedure is nondeterministic [HPY96]. To avoid any resulting redundant explorations, additional bookkeeping is needed to ensure a deterministic ample set.

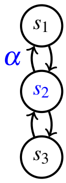
Table 8.1: POR provisos for the LTL model checking of $\mathcal{M}$ with a property φ

C0	*emptiness*	*ample*$(s) = \emptyset \Leftrightarrow$ NEXT-STATE$(s) = \emptyset$
C1	*dependency*	No action $\alpha \notin$ *ample*(s) that is dependent on another $\beta \in$ *ample*(s), i.e. $(\alpha,\beta) \in D$, can be executed in the original $\mathcal{A}_{\mathcal{M}}$ after reaching the state s and before some action in *ample*(s) is executed.
C2	*visibility*	*ample*$(s) \neq$ NEXT-STATE$(s) \implies \forall\alpha \in$ *ample*(s) : α is *invisible*, which means that α does not change a state label referred to by φ.
C3	*cycle*	For a cycle π in $\mathcal{A}_{\mathcal{M}}$, $\exists s \in \pi$: NEXT-STATE$(s) =$ *ample*(s).
C3'	*cycle (impl.)*	*ample*$(s) \neq$ NEXT-STATE$(s) \Rightarrow \nexists\alpha \in$ *ample*(s) s.t. $\alpha(s)$ is on the DFS stack.

8.3 Progress Transitions and dfs$_{\text{fifo}}$ for Non-Progress

In the current section, we refine the definition of progress to include transitions. We then present a new version of DFS$_{\text{FIFO}}$, an efficient algorithm for non-progress detection by Faragó et al. [FS09], which supports this broader definition. We also discuss implementation considerations and the combination with POR.

Progress transitions. As argued in [FS09], progress is more naturally defined on transitions (Definition 8.6) than on states. After all, the action itself, e.g. the increase of a counter in $\mathcal{M}$, constitutes the actual progress. This becomes clear considering the difference in semantics between progress transitions and progress states for livelock detection: The figure on the right shows an automaton with $\mathcal{S}^{\mathcal{P}} = \{s_1\}$ and $\mathcal{T}^{\mathcal{P}} = \{(s_2, \alpha, s_1)\}$. Thus the cycle $s_2 \leftrightarrow s_3$ exhibits only *fake progress* when progress states are used ($\mathcal{P} = \mathcal{S}^{\mathcal{P}}$): the action performing the progress, α, is never taken. With progress transitions ($\mathcal{P} = \mathcal{T}^{\mathcal{P}}$), only $s_2 \leftrightarrow s_3$ can be detected as **NPcycle**. While fake progress cycles could be hidden by enforcing strong (A-)fairness [BK08], SPIN's weak (A-)fairness [Hol11] is insufficient [FS09]. But enforcing any kind of fairness is costly [BK08].



Definition 8.6 (Progress transitions/actions)**.** *We define progress transitions as:* $\mathcal{T}^{\mathcal{P}} = \{(s, \alpha, s') \in \mathcal{T} \mid \alpha \in \Sigma^{\mathcal{P}}\}$*, with* $\Sigma^{\mathcal{P}} \subseteq \Sigma$ *a set of progress actions.*

Theorem 8.2. *DFS$_{\text{FIFO}}$ ensures:* $\mathcal{R} \cap \mathcal{NP} \neq \emptyset \Leftrightarrow$ *dfs-fifo*$(s_0) =$ ***report NPcycle***

Algorithm 8.1 DFS$_{\text{FIFO}}$ for progress transitions and progress states

```
procedure dfs-fifo($s_0$)
    $F := \{s_0\}$        ▷ Frontier queue
    $V := \emptyset$        ▷ Visited set
    $S := \emptyset$        ▷ Stack
    repeat
        $s :=$ some $s \in F$
        if $s \notin V$ then
            DFS($s$)
        $F := F \setminus \{s\}$
    until $F = \emptyset$
    report progress ensured

procedure DFS($s$)
    $S := S \cup \{s\}$
    for all $t := \alpha(s)$ s.t. $\alpha \in$ NEXT-STATE($s$) do
        if $t \in S \wedge \alpha, t \notin \mathcal{P}$ then
            report NPcycle
        if $t \notin V$ then
            if $\alpha, t \notin \mathcal{P}$ then
                DFS($t$)
            else if $t \notin F$ then
                $F := F \cup \{t\}$
    $V := V \cup \{s\}$
    $S := S \setminus \{s\}$
```

DFS$_{\text{FIFO}}$. Algorithm 8.1 shows an adaptation of DFS$_{\text{FIFO}}$ that supports the definition of progress on both states and transitions (actions), so $\mathcal{P} = \mathcal{S}^{\mathcal{P}} \cup \Sigma^{\mathcal{P}}$. Intuitively, the algorithm works by delaying progress as long as possible using a BFS and searching for **NPcycles** in between progress using a DFS. The correctness of this adapted algorithm follows from Theorem 8.2, which is implied by Theorem 8.4 with $P = 1$.

The FIFO queue F holds progress states, or immediate successors of progress transitions (which we will collectively refer to as after-progress states), with the exception of the initial state s_0. The outer *dfs-fifo* loop handles all after-progress states in breadth-first order (similar to 'Frontier' in [LT04]). The DFS procedure, starting from a state in F then explores states up to progress, storing visited states in the set V (Line 22), and after-progress states in F (Line 21). The stack of this search is maintained in a set S (Line 13 and Line 23) to detect cycles at Line 16. All states $t \in S$ and their connecting transitions are non-progress by Line 18, except for possibly the starting state from F. The cycle-closing transition $s \xrightarrow{\alpha} t$ might also be a progress transition. Therefore, Line 15 performs an additional check $\alpha, t \notin \mathcal{P}$. Furthermore, an after-progress state $s \notin \mathcal{S}^{\mathcal{P}}$ added to F, might be reached later via a non-progress path and added to V. Hence, we discard visited states in *dfs-fifo* at Line 7.

Implementation. An efficient implementation of Algorithm 8.1 stores F and V in one hash table (using a bit to distinguish the two) for fast inclusion checks, while F is also maintained as a queue F^q. S can be stored in a separate hash table as $|S| \ll |\mathcal{R}|$. Counterexamples can be reconstructed if for each state a pointer to one of its predecessors is stored [LPW10a]. Faragó et al. showed two alternatives [FS09], which are also compatible with lossy hashing [BHR13].

Combination with POR. While the four-fold performance increase of DFS$_{\text{FIFO}}$ compared to LTL (Section 8.2) is a modest gain, the algorithm provides even more potential as it relaxes conditions on POR, which, after all, might yield exponential gains. In contrast to the LTL method using NDFS, DFS$_{\text{FIFO}}$ does not revisit states, simplifying the *ample* implementation. Moreover, Lemma 8.1 shows that DFS$_{\text{FIFO}}$ does not require the cycle proviso using a visibility proviso from Table 8.2.

Lemma 8.1. *Under $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$, **C2**$^{\mathcal{S}}$ implies **C3**. Under $\mathcal{P} = \Sigma^{\mathcal{P}}$, **C2**$^{\mathcal{T}}$ implies **C3**.*

Table 8.2: POR visibility provisos for DFS$_{\text{FIFO}}$

C2$^{\mathcal{S}}$	$ample(s) \neq \text{NEXT-STATE}(s) \implies s \notin \mathcal{S}^{\mathcal{P}}$
C2$^{\mathcal{T}}$	$ample(s) \neq \text{NEXT-STATE}(s) \implies \forall \alpha \in ample(s) : \alpha \notin \Sigma^{\mathcal{P}}$

Proof. If DFS$_{\text{FIFO}}$ with POR traverses a cycle C which makes progress, i.e. $\exists s \in C\colon s \in \mathcal{S}^{\mathcal{P}} \vee ample(s) \cap C \cap \Sigma^{\mathcal{P}} \neq \emptyset$, **C2**$^{\mathcal{S}}$/ **C2**$^{\mathcal{T}}$ guarantees full expansion of s, thus fulfilling **C3**. If DFS$_{\text{FIFO}}$ traverses a **NPcycle**, it terminates at Line 16. □

Theorem 8.3. *Theorem 8.2 still holds for* DFS$_{\text{FIFO}}$ *with* ***C0, C1, C2***$^{\mathcal{S}}$***/ C2***$^{\mathcal{T}}$ *(Theorem 2 from [FS09]).*

Proof. Lemma 8.1 shows that if the **C0**, **C1** and **C2**$^{\mathcal{S}}$/ **C2**$^{\mathcal{T}}$ hold, so does **C3**. Furthermore, **C0**, **C1** and **C2**$^{\mathcal{S}}$/ **C2**$^{\mathcal{T}}$ are independent of the path leading to s, so $ample(s)$ with DFS$_{\text{FIFO}}$ retains stutter equivalence related to progress [HP94, p.6]. Therefore, the reduced state space has a **NPcycle** iff the original has one. □

8.4 A Parallel Livelock Algorithm based on dfs$_{\text{fifo}}$

Algorithm 8.2 presents a parallel version of DFS$_{\text{FIFO}}$. The algorithm does not differ much from Algorithm 8.1: the DFS procedure remains largely the same, and only *dfs-fifo* is split into parallel *fifo* procedures handling states from the FIFO queue F concurrently. The technique to parallelize the DFS(s,i) calls is based on successful multi-core NDFS algorithms presented in Part III. Each worker thread $i \in 1..P$ uses a local stack S_i, while V and F are shared (below, we show how an efficient implementation can partially localize F). The stacks may overlap (see Line 2 and Line 9), but eventually diverge because we use a randomized next-state function: NEXT-STATE$_i$() (see Line 15).

Algorithm 8.2 Parallel DFS$_{\text{FIFO}}$ (PDFS$_{\text{FIFO}}$)

```
 1: procedure dfs-fifo($s_0, P$)
 2:     $F := \{s_0\}$                       ▷ Frontier queue
 3:     $V := \emptyset$                      ▷ Visited set
 4:     $S_i := \emptyset$ for all $i \in 1..P$       ▷ Stacks
 5:     $fifo(1) \parallel \ldots \parallel fifo(P)$
 6:     report progress ensured
 7: procedure fifo($i$)
 8:     while $F \neq \emptyset$ do
 9:         $s :=$ some $s \in F$
10:         if $s \notin V$ then
11:             DFS($s, i$)
12:         $F := F \setminus \{s\}$
13: procedure DFS($s, i$)
14:     $S_i := S_i \cup \{s\}$
15:     for all $t := \alpha(s)$ s.t. $\alpha \in$ NEXT-STATE$_i(s)$ do
16:         if $t \in S_i \wedge \alpha, t \notin \mathcal{P}$ then
17:             report NPcycle
18:         if $t \notin V$ then
19:             if $\alpha, t \notin \mathcal{P}$ then
20:                 DFS($t, i$)
21:             else if $t \notin F$ then
22:                 $F := F \cup \{t\}$
23:     $V := V \cup \{s\}$
24:     $S_i := S_i \setminus \{s\}$
```

Proof of correctness. Theorem 8.4 proves correctness of Algorithm 8.2. We show that the propositions below hold after initialization of Algorithm 8.2, and inductively that they are maintained by execution of each statement in the algorithm, considering only the lines that influence the proposition. Rather than restricting progress to either transitions or states, we prove the algorithm correct under $\mathcal{P} = \mathcal{S}^{\mathcal{P}} \cup \mathcal{T}^{\mathcal{P}}$. Hence, the dual interpretation of paths (see Definition 8.1) is used now and then. Note that a call to **report** terminates the algorithm and the callee does not return.

Lemma 8.2. *Upon return of* $\textsc{dfs}(s,i)$*, s is visited:* $s \in V$.

Proof. Line 23 of $\textsc{dfs}(s,i)$ adds s to V. □

Lemma 8.3. *Invariantly, all direct successors of a visited state v are visited or in F:* $\forall v \in V, \alpha \in \textsc{next-state}(v) : \alpha(v) \in V \cup F$.

Proof. After initialization, the invariant holds trivially, as V is empty. V is only modified at Line 23, where s is added after all its immediate successors t are considered at Line 16–22: If $t \in V \cup F$, we are done. Otherwise, $\textsc{dfs}(s,i)$ terminates at Line 17 or t is added to V at Line 20 (Lemma 8.2) or to F at Line 22. States are removed from F at Line 12, but only after being added to V at Line 11 (Lemma 8.2). □

Corollary 8.1. *Lemma 8.3 holds also for a state $v \notin V$ in* $\textsc{dfs}(v,i)$ *just before Line 23.*

Lemma 8.4. *Invariantly, all paths from a visited state v to a state $f \in F \setminus V$ contain progress:* $\forall \pi, v \in V, f \in F \setminus V : v \xrightarrow{\pi} f \implies \mathcal{P} \cap \pi \neq \emptyset$.

Proof. After initialization of the sets V and F, the lemma is trivially true. These sets are modified at Line 12, Line 22, and Line 23 (omitting the trivial case):

Line 22 Let i be the first worker thread to add a state t to F in $\textsc{dfs}(s,i)$ at Line 22. If some other worker j adds t to V, the invariant holds trivially, so we consider $t \notin V$. By Line 19, all paths $v \rightarrow^{*} s \rightarrow t$ contain progress. By contradiction, we show that all other paths that do not contain s also contain progress: Assume that there is a $v \in V$ such that $v \xrightarrow{\pi}{}^{+} t$ and $\mathcal{P} \cap \pi = \emptyset$. By induction on the length of the path π and Lemma 8.3, we obtain either $t \in V$, a contradiction, $t \in F \setminus V$, contradicting the assumption that worker i is first, or another $f \neq t$ with $f \in F \setminus V$, for which the induction hypothesis holds.

Line 23 Assume towards a contradiction that i is the first worker thread to add a state s to V at Line 23 of $\textsc{dfs}(s,i)$. So, we have $s \notin V$ before Line 23. By Corollary 8.1, for all immediate successors t of s, i.e. for all $t = \alpha(s)$ such that $\alpha \in \textsc{next-state}(s)$, we have $t \in V$ or $t \in F \setminus V$. In the first case, since $s \neq t$, the induction hypothesis holds for t. In the second case, if $t = s$, the invariant trivially holds after Line 23, and if $t \neq s$, we have $\alpha, t \in \mathcal{P}$, since otherwise $t \in V$ by Line 19 and Line 20 (Lemma 8.2). Thus the invariant holds for all paths $s \rightarrow^+ f$.

□

Remark 8.1. *Note that a state $s \in F$ might at any time be also added to V by some other worker thread in two cases: (1) $s \notin \mathcal{S}^{\mathcal{P}}$, i.e. it was reached via a progress transition (see Line 19), but is reachable via some other non-progress path, or (2) another worker thread j takes s from F at Line 9 and completes* DFS(s, j)*.*

Lemma 8.5. *Invariantly, visited states do not lie on* ***NPcycles****: $V \cap \mathcal{NP} = \emptyset$.*

Proof. Initially, $V = \emptyset$ and the lemma holds trivially. Let i be the first worker thread to add s to V in DFS(s,i) at Line 23. So we have $s \in V$ just after Line 23 of DFS(s,i). Assume towards a contradiction that $s \in \mathcal{NP}$. Then there is a **NPcycle** $s \rightarrow t \rightarrow^+ s$ with $s \neq t$ since otherwise Line 17 would have reported a **NPcycle**. Now by Lemma 8.3, $t \in V \cup F$. By the induction hypothesis, $t \notin V$, so $t \in F \setminus V$. Lemma 8.4 contradicts $s \rightarrow t$ making no progress. □

Lemma 8.6. *Upon return of dfs-fifo, all reachable states are visited: $\mathcal{R} \subseteq V$.*

Proof. After *dfs-fifo*(s_0, P), $F = \emptyset$ by Line 8. By Line 2, Line 11 and Lemma 8.2, $s_0 \in V$. So by Lemma 8.3, $\mathcal{R} \subseteq V$. □

Lemma 8.7. *dfs-fifo terminates and reports an* ***NPcycle*** *or* ***progress ensured****.*

Proof. Upon return of a call DFS(s,i) for some $s \in F$ at Line 11, s has been added to V (Lemma 8.2), removed from F at Line 12, and will never be added to F again. Hence the set V grows monotonically, but is bounded, and eventually $F = \emptyset$. Thus eventually all DFS calls terminate, and *dfs-fifo*(s_0, P) terminates too. □

Lemma 8.8. *Invariantly, the states in S_i form a path without progress except for the first state: $S_i = \emptyset$ or $S_i = \pi \cap \mathcal{S}$ for some $s \xrightarrow{\pi}{}^* s'$ and $\pi \cap \mathcal{P} \subseteq \{s_1\}$.*

Proof. By induction over the recursive DFS(s,i) calls, we obtain π. At Line 20, we have $\alpha, t \notin \mathcal{P}$, but at Line 11 we may have $s \in \mathcal{S}^{\mathcal{P}}$ (by Line 19 and Line 22). □

Theorem 8.4. *PDFS$_{FIFO}$ ensures: $\mathcal{R} \cap \mathcal{NP} \neq \emptyset \Leftrightarrow$ dfs-fifo$(s_0, P) =$* ***report NPcycle***

Proof. We split the equivalence into two cases:
$\Leftarrow$: We have a cycle: $s \xrightarrow{\alpha} t \xrightarrow{\pi} s$ s.t. $(\{\alpha\} \cup \pi) \cap \mathcal{P} = \emptyset$ by Line 16 and Lemma 8.8.
$\Rightarrow$: Assume that *dfs-fifo*$(s_0, P) \neq$ **NPcycle** $\wedge\, \mathcal{R} \cap \mathcal{NP} \neq \emptyset$. However, at Line 6, $\mathcal{R} \subseteq V$ by Lemma 8.6 and Lemma 8.7, hence $\mathcal{R} \cap \mathcal{NP} = \emptyset$ by Lemma 8.5. □

Implementation. For a scaling implementation, the hash table storing F and V (see Section 8.3) is maintained in shared memory using a lockless design (see Part II). Storing also the queue F^q in shared memory, however, would seriously impede scalability due to contention (recall that F is maintained as both hash table and queue F^q). Our more efficient implementation splits F^q into P local queues F_i^q, such that $F \subseteq \bigcup_{i\in 1..P} F_i^q$ (Remark 8.1 explains the $\subseteq$).

To implement load balancing, one could relax the constraint at Line 21 to $s \notin F^q$, so that after-progress states end up on multiple local queues. Provided that $\mathcal{A}_{\mathcal{M}}$ is connected enough, which it usually is in model checking, this would provide good work distribution already. On the other hand, the total size of all queues F_i^q would grow proportional to P, wasting a lot of memory on many cores. Therefore, we instead opted to add explicit load balancing via work stealing. Algorithm 8.3 illustrates this. Iff the local queue F_i^q is empty, the *steal* function grabs states from another random queue F_j^q and adds them to F_i^q, returning false iff it detects termination. Inspection of Lemma 8.3 and Lemma 8.7 shows that removing s from F is not necessary.

The proofs show that correctness of $\mathrm{PDFS}_{\mathrm{FIFO}}$ does not require F to be in strict FIFO order (as Line 9 does not enforce any order). To optimize for scalability, we enforce a strict BFS order via synchronizations[8.2] between the BFS levels only optionally[8.3]. As trade-off, counterexamples are no longer guaranteed to be the shortest with respect to progress, and the size of F may increase (see Remark 8.1).

Algorithm 8.3 An implementation of $\mathrm{PDFS}_{\mathrm{FIFO}}$ with local queues and load balancing

```
procedure fifo(i)
    $F_i^q := \{s_0\}$
    while steal($F_i^q$) do
        $F_i^q := F_i^q \setminus \{s\}$
        if $s \notin V$ then
            DFS($s, i$)
```

Analysis of scalability. Experiments with multi-core NDFS (see Chapter 7) demonstrated that these parallelization techniques make the state-of-the-art for LTL model checking. Because of the BFS nature of $\mathrm{DFS}_{\mathrm{FIFO}}$, we can expect even better speedups. Moreover, in Chapter 5, additional synchronization was needed to prevent workers from *early backtracking*; a situation in which two workers exclude a third from part of the state space. Figure 8.1 illustrates this: Worker 1 can visit s, v, t and u, and then halt.

[8.2] Parallel BFS algorithms, with and without synchronization, are described in Chapter 9.

[8.3] The command line option `--strict` turns on strict $\mathrm{PDFS}_{\mathrm{FIFO}}$ in LTSMIN.

Worker 2 can visit s, u, t and v and backtrack over v. If now Worker 1 resumes and backtracks over u, both v and u are in V. A third worker will be excluded from visiting t, which might lead to a large part of the state space. Lemma 8.3 shows that this is impossible for PDFS$_{\text{FIFO}}$ as the successors of visited states are either visited or in F (treated in efficient parallel BFS), but never do successors lie solely on the stack S_i (as in CNDFS).

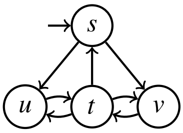


Figure 8.1: Example for early backtracking

8.5 Experimental Evaluation

In the current section, we benchmark the performance of DFS$_{\text{FIFO}}$, and its combination with POR, using both progress states and progress transitions. We compare the results against the LTL approach with progress property using, inter alia, SPIN [Hol11]. We also investigate the scalability of PDFS$_{\text{FIFO}}$, and compare the results against the multi-core NDFS algorithm CNDFS, the state-of-the-art for parallel LTL [BL13a] (see also Chapter 7 and Chapter 11), and the Piggyback algorithm in SPIN (PB). Finally, we investigate the combination of PDFS$_{\text{FIFO}}$ and POR, and compare the results with OWCTY [BBR09a], which uses a topological sort to implement parallel LTL and POR [BBR10a]. Since currently there is no way to combine POR with e.g. CNDFS from Chapter 7, we cannot experiment with this aspect on an MC-NDFS algorithm.

We implemented PDFS$_{\text{FIFO}}$ (*Algorithm 8.2 with work stealing and both strict*[8.3]*/non-strict BFS order*) in LTSMIN [LPW11a; BPW10] 2.0.[8.4] LTSMIN has a frontend for PROMELA, called SPINS [Hol11], and one for the DVE language, allowing fair comparison [BPW10; LPW11a; BL13a] against SPIN 6.2.3 and DIVINE 2.5.2 [BBR09a]. To ensure similar state counts, we turned off control-flow optimizations in SPINS/SPIN, because SPIN has a more powerful optimizer, which can be, but is not yet implemented in SPINS. Only the GIOP model (described below) still yields a larger state count in SPINS/LTSMIN than in SPIN. We still include it as it nicely features the benefits of DFS$_{\text{FIFO}}$ over NDFS.

We benchmarked on a 48-core machine (a four-way AMD Opteron 6168) with 128GB of main memory, and considered 4 publicly available[8.1] PROMELA models with

[8.4] LTSMIN is open source, available at: `http://fmt.cs.utwente.nl/tools/ltsmin`.

progress labels, and adapted SpinS to interpret the labels as either progress states, as in spin, or progress transitions. *leader$_t$* is the efficient leader election protocol $\mathcal{A}_{timing}$ [Far07]. The *Group Address Registration Protocol* (*GARP*) is a datalink-level multicast protocol for a bridged LAN. *General Inter-Orb Protocol* (GIOP) models service oriented architectures. The model *i-Protocol* represents the gnu implementation of this protocol. We use a different leader election protocol (leader$_{DKR}$) from [DKR82] in DVE format [Pel07] for comparison against DiVinE. For all these models, the livelock property holds under $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$ and $\mathcal{P} = \mathcal{T}^{\mathcal{P}}$.[8.5]

8.5.1 Performance

In theory, dfs$_{\text{FIFO}}$ can be up to four times as fast as using the progress LTL formula and ndfs. To verify this, we compare dfs$_{\text{FIFO}}$ to ndfs in LTSmin and spin. In LTSmin, we used the command line: prom2lts-mc --state=tree -s28 --strategy=[dfsfifo/ndfs] [model], which replaces the shared table (for F and V) by a *tree* table for state compression (see Chapter 3). In spin, we used compression as well (*collapse* [Hol11]): cc -O2 -DNP -DNOFAIR -DNOREDUCE -DNOBOUNDCHECK -DCOLLAPSE -o pan pan.c, and pan -m100000 -l -w28, avoiding table resizes and overhead. In both tools, we also ran dfs reachability with similar commands. We write *oom* for runs that overflow the main memory.

Table 8.3 shows the results: As expected, $|\mathcal{R}_{\text{LTL}}|$ is 1.5 to 2 times larger than $|\mathcal{R}|$ for both spin and LTSmin; GIOP fits in memory for dfs$_{\text{FIFO}}$ but the LTL cross product overflows (ndfs). T_{NDFS} is about 1.5 to 4 times larger than T_{DFS} for spin, 2 to 5 times larger for LTSmin (cf. Section 8.2). $T_{\text{DFS}_{\text{FIFO}}}$ is 1.5 to 2 times larger than T_{DFS}, likely caused by its set inclusion tests on S and F. T_{NDFS} is 1.6 to 3.2 times larger than $T_{\text{DFS}_{\text{FIFO}}}$.

Table 8.3: Runtimes (sec) of (sequential) dfs, dfs$_{\text{FIFO}}$, ndfs in spin and LTSmin

	LTSmin					spin			
	$\|\mathcal{R}\|$	$\|\mathcal{R}_{\text{LTL}}\|$	T_{DFS}	$T_{\text{DFS}_{\text{FIFO}}}$	T_{NDFS}	$\|\mathcal{R}\|$	$\|\mathcal{R}_{\text{LTL}}\|$	T_{DFS}	T_{NDFS}
leader$_t$	4.5e7	198%	153.7	233.2	753.6	4.5e7	198%	304.0	1,390.0
garp	1.2e8	150%	377.1	591.2	969.2	1.2e8	146%	1,370.0	2,050.0
giop	2.7e9	oom	21,301.4	43,154.3	oom	8.4e7	181%	1,780.0	4,830.0
i-prot	1.4e7	140%	28.4	41.4	70.6	1.4e7	145%	63.3	103.0

[8.5] Models that we modified are available at http://doiop.com/leader4DFSFIFO.

8.5.2 Parallel Scalability

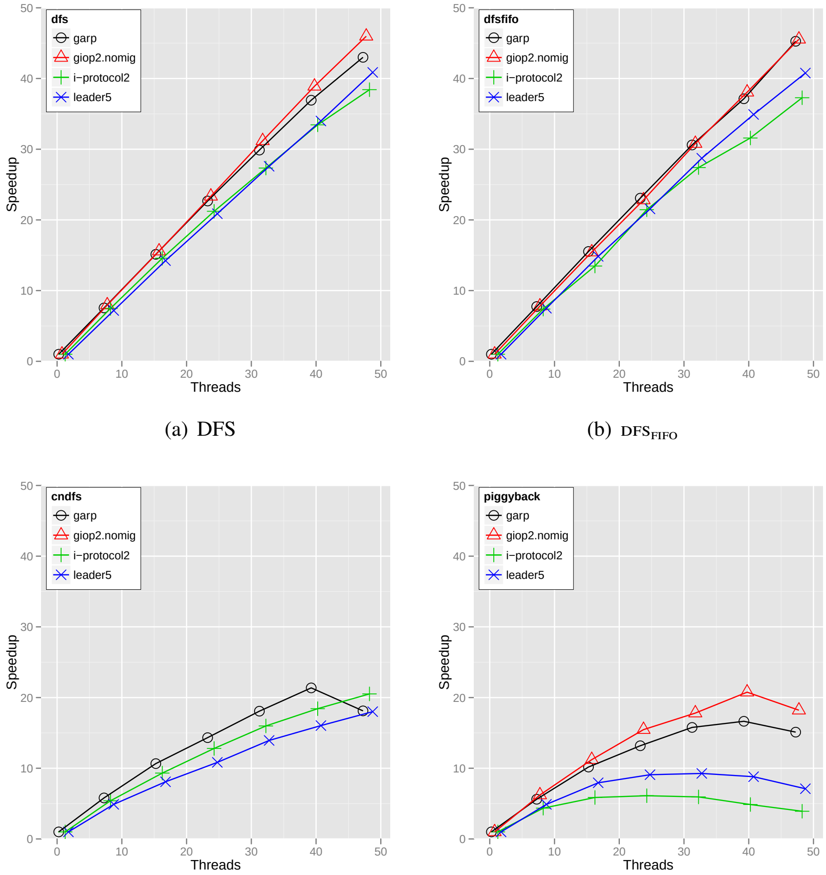
To compare the parallel algorithms in LTSmin, we use the options --threads=P --strategy=[dfsfifo/cndfs], where P is the number of worker threads. In SPIN, we use -DBFS_PAR, which also turns on lossy state hashing [Hol12], and run the pan binary with an option -uP. This turns on a parallel, linear-time, but incomplete, cycle detection algorithm called Piggyback (PB) [Hol12]. It might also be unsound due its combination with lossy hashing [BHR13]. Figure 8.2 shows the obtained speedups: As expected, reachability (see Part II) and PDFS$_{\text{FIFO}}$ scale almost ideally, while CNDFS exhibits sublinear scalability, even though it is the fastest parallel LTL solution (see Chapter 7). PB also scales sublinearly. Since LTSmin sequentially competes with SPIN (Table 8.4, except for GIOP), scalability can be compared.

8.5.3 Parallel Memory Usage

We expected few state duplication on local queues F_i^q (see Remark 8.1). To verify this, we measured the total size of all local queues and hash tables using counters for strict[8.3] and non-strict PDFS$_{\text{FIFO}}$, and CNDFS. Table 8.4 shows $Q_P = \sum_{i \in 1..P} |F_i^q| + |S_i|$ averaged over 5 runs: Non-strict PDFS$_{\text{FIFO}}$ shows little difference from the strict variant, and Q_{48} is at most 20% larger than Q_1 for all PDFS$_{\text{FIFO}}$. Due to the randomness of the parallel runs, we even have $Q_{48} < Q_1$ in many cases. Revisits occurred at most 2.6% using 48 cores. In the case of CNDFS, the combined stacks typically grow because of the larger DFS searches. Accordingly, we found that PDFS$_{\text{FIFO}}$'s *total memory use with 48 cores* was between 87% and 125% compared to sequential DFS. In the worst case, PDFS$_{\text{FIFO}}$ (with tree compression) used 52% of the memory use of PB (collapse compression and lossy hashing) [BL13a] – GIOP excluded as its state counts differ.

Table 8.4: Runtimes (sec) / queue sizes of the parallel algorithms: DFS, PDFS$_{\text{FIFO}}$ and CNDFS in LTSmin, and PB in SPIN

	DFS		PDFS$_{\text{FIFO}}$		CNDFS		PB		PDFS$_{\text{FIFO}}^{strict}$		PDFS$_{\text{FIFO}}^{non\text{-}strict}$		CNDFS	
	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{min}	Q_1	Q_{48}	Q_1	Q_{48}	Q_1	Q_{48}
$leader_t$	153.7	3.8	233.2	5.7	925.7	51.4	228.0	25.9	1.0e6	1.2e6	1.2e6	1.4e6	2.7e6	3.6e7
garp	377.1	8.8	591.2	13.1	1061.0	58.6	1180.0	70.9	1.9e7	2.0e7	1.9e7	5.3e6	5.5e6	6.5e7
giop	2.1e4	463.3	4.3e4	9.7e2	oom	oom	1.2e3	57.8	1.1e9	8.4e8	1.1e9	8.4e8	oom	oom
i-prot	28.4	0.7	41.4	1.1	75.9	3.7	86.2	17.7	1.0e6	1.1e6	1.0e6	1.3e6	8.3e5	1.0e7



(a) DFS

(b) DFS$_{\text{FIFO}}$

(c) CNDFS

(d) SPIN with parallel BFS

Figure 8.2: Speedups of DFS, PDFS$_{\text{FIFO}}$ and CNDFS in LTSMIN, and Piggyback in SPIN

8.5.4 Partial-Order Reduction Performance

LTSMIN's POR implementation (option --por) is based on stubborn sets [Val91b], described in [Pat11], and is competitive to SPIN's [BL13a]. We extended it with the alternative provisos for DFS$_{\text{FIFO}}$: **C2**$^{\mathcal{S}}$ and **C2**$^{\mathcal{T}}$. Table 8.5 shows the relative number of states, using the different algorithms in both tools: For all models, both LTSMIN and SPIN are able to obtain reductions of multiple orders of magnitude using their DFS algorithms. We also observe that much of this benefit disappears when using the NDFS LTL algorithm due to the cycle proviso, although SPIN often performs much better than LTSMIN in this respect. Also DFS$_{\text{FIFO}}$ with progress states (column DFS$^{\mathcal{S}}_{\text{FIFO}}$), performs poorly: apparently, the **C2**$^{\mathcal{S}}$ proviso is so restrictive that many states are fully expanded. But DFS$_{\text{FIFO}}$ with progress transitions (column DFS$^{\mathcal{T}}_{\text{FIFO}}$) retains DFS's impressive POR with at most a factor 2 difference.

Table 8.5: POR (%) for DFS$^{\mathcal{T}}_{\text{FIFO}}$, DFS$^{\mathcal{S}}_{\text{FIFO}}$, DFS and NDFS in SPIN and LTSMIN

	LTSMIN				SPIN	
	DFS	DFS$^{\mathcal{T}}_{\text{FIFO}}$	DFS$^{\mathcal{S}}_{\text{FIFO}}$	NDFS	DFS	NDFS$^{\text{SPIN}}$
leader$_t$	0.32%	0.49%	99.99%	99.99%	0.03%	1.15%
garp	1.90%	2.18%	4.29%	16.92%	10.56%	12.73%
giop	1.86%	1.86%	3.77%	oom	1.60%	2.42%
i-prot	16.14%	31.83%	100.00%	100.00%	24.01%	41.37%

8.5.5 Scalability of Parallelism and Partial-Order Reduction

We created multiple instances of the leader$_{DKR}$ models by varying the number of nodes N and expressed the progress LTL property in DIVINE. We start DIVINE's state-of-the-art parallel LTL-POR algorithm, OWCTY, by: divine owcty [model] -wP -i30 -p. With the options described above, we turned on POR in LTSMIN and ran PDFS$_{\text{FIFO}}$, and CNDFS, for comparison. We limited each run to half an hour (~~*30*~~*'* indicates a timeout). Piggyback reported contradictory memory usage and far fewer states (e.g. $< 1\%$) compared to DFS with POR, although it must meet more provisos. Thus we did not compare against Piggyback and suspect a bug.

Table 8.6 shows that PDFS$_{\text{FIFO}}$ and POR complement each other rather well: Without POR (left half of the table) the almost ideal speedup ($U = \frac{T_1}{T_{48}} = 40.8$) allows to explore one model more: $N \leq 10$ instead of only $N = 9$. When enabling POR (right half of the table), we see again multiple orders of magnitude reductions, while parallel scalability

Table 8.6: POR and speedups for leader$_{DKR}$ using PDFS$_{\text{FIFO}}$, CNDFS and OWCTY

N	Alg.	$\|\mathcal{R}\|$	$\|\mathcal{T}\|$	T_1	T_{48}	U	$\|\mathcal{R}^{\text{POR}}\|$	$\|\mathcal{T}^{\text{POR}}\|$	T_1^{POR}	T_{48}^{POR}	U^{POR}
9	CNDFS	3.6e7	2.3e8	502.6	12.0	41.8	27.9%	0.1%	211.8	n/a	n/a
9	PDFS$_{\text{FIFO}}$	3.6e7	2.3e8	583.6	14.3	40.8	1.5%	0.0%	12.9	3.6	3.5
9	OWCTY	3.6e7	2.3e8	498.7	51.9	9.6	12.6%	0.0%	578.4	35.7	16.2
10	CNDFS	2.4e8	1.7e9	~~30~~'	90.7	~~30~~'	19.3%	5.4%	1102.7	n/a	n/a
10	PDFS$_{\text{FIFO}}$	2.4e8	1.7e9	~~30~~'	109.3	~~30~~'	0.7%	0.1%	35.0	2.5	14.0
10	OWCTY	2.4e8	1.7e9	~~30~~'	663.1	~~30~~'	8.7%	2.2%	~~30~~'	156.3	~~30~~'
11	PDFS$_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	5.1e6	7.1e6	109.8	5.3	20.7
11	OWCTY	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	9.3e7	1.7e8	~~30~~'	1036.5	~~30~~'
12	PDFS$_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	1.6e7	2.2e7	369.1	11.2	33.0
13	PDFS$_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	6.6e7	9.2e7	1640.5	38.1	43.0
14	PDFS$_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	2.0e8	2.9e8	~~30~~'	120.3	~~30~~'
15	PDFS$_{\text{FIFO}}$	~~30~~'	~~30~~'	~~30~~'	~~30~~'	~~30~~'	8.4e8	1.2e9	~~30~~'	527.5	~~30~~'

reduces to $U = 3.5$ for $N = 9$, because of the small size of the reduced state space ($|\mathcal{R}^{\text{POR}}|$). When increasing the model size to $N = 13$ the speedup grows again to an almost ideal level ($U = 43$). With POR, the parallelism allows us to explore two more models within half an hour, i.e., $N \leq 15$. While OWCTY and NDFS also show this effect, it is less pronounced due to their cycle proviso, allowing $N \leq 11$ for OWCTY and $N \leq 9$ for NDFS.

As livelocks are disjoint from the class of weak LTL properties, OWCTY could become non-linear [BBR09a], but it required only one iteration for leader$_{DKR}$.

As PDFS$_{\text{FIFO}}$ revisits states, the random next-state function could theoretically weaken POR (as for NDFS, see Section 8.2). But for all our 5 models, this did not occur.

8.5.6 On-The-Fly Performance

We created a leader election protocol with early (*shallow*) and another with late (*deep*) injected **NPcycles** (see [8.5], [Far07]). Table 8.7 shows the average runtime in seconds (T) and counterexample length (C) over five runs. Since PDFS$_{\text{FIFO}}$ finds shortest counterexamples[8.3], it outperforms CNDFS for the *shallow* version and pays a penalty for the *deep* version. Both algorithms benefit greatly from massive parallelism (see also [LP11]).

Table 8.7: Runtimes and counterexample lengths for CNDFS and PDFS$_{\text{FIFO}}$ on synthetic models containing livelocks at deep and shallow levels in the state space

	CNDFS		PDFS$^{\mathcal{T}}_{\text{FIFO}}$		CNDFS		PDFS$^{\mathcal{T}}_{\text{FIFO}}$	
	T_1	T_{48}	T_1	T_{48}	C_1	C_{48}	C_1	C_{48}
shallow	~~30~~'	7	12	4	~~30~~'	16	16	16
deep	16(once ~~30~~')	2	~~30~~'	451	577	499	~~30~~'	51

8.6 Conclusions

We showed, in theory and in practice, that model checking livelocks, an important subset of liveness properties, can be made more efficient by specializing on it. For our PDFS$_{\text{FIFO}}$ implementation with progress transitions, POR becomes significantly stronger (cf. Table 8.5), parallelization has linear speedup (cf. Figure 8.2), and both can be combined efficiently (cf. Table 8.6).

Our results apply to a broader set of livelock properties that can be expressed more directly using *testing automata* [Val93]. See [HPV02; Han07] for more extensive discussion and comparison to Büchi automata. Similarly, [LT04] presents a tested framework more similar to the original from [Val93], where non-progress is defined on actions, like the progress transitions that we employ. These papers all present algorithms that are equal to Algorithm 8.1, modulo the different contexts (we define progress states/transitions, whereas they define *livelock-monitor states* within the tester framework that 'listen' to divergent traces where these are not to be allowed).

Part IV

Multi-Core Model Checking for Timed Systems

Introduction

The parallel model checking procedures developed in the previous 2 parts are still limited to explicit-state systems. The current part extends these methods for the analysis of timed systems.

Systems with continuous time behavior can be modeled using timed automata [Alu99], timed Petri nets [Ram74] and timed process calculi [BB91]. The transfinite nature of the continuous time variables, or *clocks*, makes explicit-state model checking infeasible, except for when an approximating discretization step is applied beforehand [CHR91; GRU08; CVH04; She+10; BD98b; BD98a]. A complete approach can however be achieved by symbolically abstracting clock valuations as *linear-(in)equality constraints*. To this end, region-based [AD94] and zone-based [Dil89] abstractions have been proposed. Based on these finite abstractions, extensive tool support has been developed over the past decade, of which the state-of-the-art for timed automata is arguably UPPAAL [LPY97; BDL04].

However, existing solutions still exhibit shortcomings of both theoretical and practical nature. First of all, while UPPAAL has been parallelized for distributed systems almost a decade ago [Beh05; BHV00], it is unclear whether, or even unlikely that, it scales on modern multi-core computers. *Unclear* because the distributed version of UPPAAL is unavailable, and *unlikely* because modern multi-core processors nowadays have steeper memory hierarchies, cf. Section 1.6.

Furthermore, extensive support for liveness properties is not available for timed automata. While some tools support the full CTL or LTL, their time abstraction is still limited. The coarsest abstraction methods, combine finite zone-based abstractions with aggressive extrapolation [Beh+06], and on top of that use the partial-order induced by the abstraction to prune parts of the state space during the search. The latter is called *subsumption or inclusion abstraction* and does not preserve Büchi emptiness [Tri09]. In fact, it is an open theoretical problem whether the checking of liveness properties can benefit from the coarsest subsumption abstraction:

> "It would also be interesting to study whether other, coarser, zone-based abstractions, such as the inclusion abstraction proposed in [DT98] or the abstractions proposed in [Beh+06], can be used to check timed Büchi automata emptiness."
>
> [Tri09]

To extend the multi-core reachability methods from Part II for timed automata, the symbolic time abstractions, which are represented algorithmically as *difference bound matrices* (DBMs) [Ben02], need to be stored together with the visited (explicit) states. Because there is a many to one relation between the DBMs and the explicit states, the hash table for the latter should be extended to a *multimap*.

Chapter 9 explains how we implement the timed automata for the language-independent model checker LTSmin. To this end, the opaal model checker [Dal+11] is used as a frontend, which generates a library from a timed automata model that can be loaded by LTSmin. The library implements the next-state generator for states that include a DBM.

Furthermore, a design is proposed for a concurrent multimap to store the explicit and symbolic states, together with a parallel reachability algorithm that implements subsumption abstraction. Mutual exclusion on the multimap is guaranteed by a fine-grained locking mechanism that allows complex access patterns, while still providing enough parallelism. We show that the algorithm is complete and does not revisit states, but we also propose a lockless algorithms that allows for more parallelism, while dropping the latter guarantee.

The parallel algorithms and data structure were implemented in LTSmin. Experiments show that reachability scales reasonably and the non-blocking version almost linearly without revisiting significantly many states. Because the flexible search-order property of the reachability algorithm from Chapter 2 was preserved, the influence of different search orders could also be investigated. Results show the well-known benefit of bfs in this area [BHV00], but surprisingly also reveal a benefit of parallel dfs. Because our concurrent multimap is compatible with the state compression technique from Chapter 3, we also demonstrate that the memory usage is almost as efficient as uppaal's state-caching technique [BLP03] (both techniques are orthogonal and could be combined).

Chapter 10 builds on these results to realize parallel LTL model checking for timed automata for the first time. By using cndfs together with the multimap from the preceding chapter, we obtain scalable timed LTL checking.

We further demonstrate that cndfs can be extended to use the coarse subsumption abstraction in multiple parts of the search for accepting cycles, without losing soundness

Formalism	Property	Explicit state	+ On-the-fly	+ Compression	+ POR
Plain	Reachability	✓	✓	✓	✓
	LTL	✓	✓	✓	X
	. . Livelocks	✓	✓	✓	✓
Timed	Reachability	✓	✓	✓	–
	LTL	✓	✓	✓	–

and completeness. Generally, this is not the case, as subsumption abstraction introduces cycles in the state space, prompting Tripakis to pose it as an open problem (see above). Experimental results demonstrate that this abstraction could yield a 2-fold reduction of the number of states.

The table above describes the contributions that the current part makes towards solving the goals of the thesis (c.f. Table 1.1 in Section 1.5.3). It shows that the scalable multi-core model checking techniques developed for explicit-state formalisms is transferred to the domain of timed systems. Partial-order reduction is not considered in the following chapters, but it will be revisited in the conclusion of the thesis.

9

Multi-Core Reachability for Timed Automata

Andreas Dalsgaard, Alfons Laarman, Kim G. Larsen, Mads Chr. Olesen, Jaco van de Pol

Abstract

Model checking of timed automata is a widely used technique. But in order to take advantage of modern hardware, the algorithms need to be parallelized. We present a multi-core reachability algorithm for the more general class of well-structured transition systems, and an implementation for timed automata.

Our implementation extends the OPAAL tool to generate a timed automaton successor generator in C++, that is efficient enough to compete with the UPPAAL model checker, and can be used by the discrete model checker LTSMIN, whose parallel reachability algorithms are now extended to handle subsumption of semi-symbolic states. The reuse of efficient lockless data structures guarantees high scalability and efficient memory use.

With experiments we show that OPAAL+LTSMIN can outperform the current state-of-the-art, UPPAAL. The added parallelism is shown to reduce verification times from minutes to mere seconds with speedups of up to 40 on a 48-core machine. Finally, strict BFS and (surprisingly) parallel DFS search order are shown to reduce the state count, and improve speedups.

About this chapter: The current chapter is based on the paper *"Multi-core Reachability for Timed Automata"*, which was published at FORMATS 2012 [Dal+12].

The original text remains largely unmodified, except for the removal of the general introduction.

9.1 Introduction

The goal of the current chapter work is to develop scaling multi-core reachability for timed automata [AD94] as a first step towards full multi-core LTL model checking. A review of the history of discrete model checkers shows that indeed multi-core reachability is a crucial ingredient for efficient parallel LTL model checking (see Section 9.2). To attain our goal, we extended and combined several existing software tools:

LTSmin is a language-independent model checking framework, comprising, inter alia, an explicit-state multi-core backend [LPW11a; BPW10; BPW09].

opaal is a model checker designed for rapid prototype implementation of new model checking concepts. It supports a generalized form of timed automata [Dal+11], and uses the uppaal input format.

The UPPAAL DBM library is an efficient library for representing timed automata zones and operations thereon, used in the uppaal model checker [BDL04].

Contributions: We describe a multi-core reachability algorithm for timed automata, which is generalizable to all models where a well-quasi-ordering on the behavior of states exist [FS01]. The algorithm has been implemented for timed automata, and we report on the structure and performance of this prototype.

Before we move on to a description of our solution and its evaluation, we first review related work, and then briefly introduce the modeling formalism.

Overview: Section 9.3 introduces some definitions of modeling formalisms and enumerative model checking for explicit-state systems. In Section 9.4, we describe how opaal and LTSmin are combined and extended to support multi-core model checking of well-structured transitions systems. Section 9.5 and Section 9.6 then explain in detail how opaal and LTSmin were extended. Experiments are shown in Section 9.7. We end with conclusions in Section 9.8.

9.2 Related Work

One efficient model checker for timed automata is the uppaal tool [BDL04; Beh+02]. Our work is closely related to UPPAAL in that we share the same input format and reuse its editor to create input models. In addition, we reused the open source uppaal dbm library for the internal symbolic representation of time zones.

Distributed model checking algorithms for timed automata were introduced in [BHV00; Beh05]. These algorithms exhibited almost linear scalability (50–90% efficiency) on a 14-node cluster of that time. However, analysis also shows that static partitioning used for distribution has some inherent limitations [BOS06]. Furthermore, in the field of explicit-state model checking, the DiVinE tool showed that static partitioning can be reused in a shared-memory setting [BR08]. While the problem of parallelization is considerably simpler in this setting, this tool nonetheless featured suboptimal performance with less than 40% efficiency on 16-core machines (see Chapter 2). It was soon demonstrated that shared-memory systems are exploited better by combining local search stacks with a lockless hash table as shared passed set and an off-the-shelf load balancing algorithm for workload distribution [San97a]. Especially in recent experiments on newer 48-core machines as reported in Section 7.4, the latter solution was clearly shown to have the edge with 50–90% efficiency.

Linear-time, on-the-fly liveness verification algorithms are based on depth-first search (DFS) order (see Part III). Next to the additional scalability, the shared hash table solution also provides more freedom for the search algorithm, which can be pseudo DFS and pseudo breadth-first search (BFS) order [LPW11a], but also strict BFS (see Section 9.6.2). This freedom has already been exploited by parallel NDFS algorithms for LTL model checking (see Chapter 5 and Chapter 7) that are linear in the size of the input graph (unlike their BFS-based counterparts). While these algorithms are heuristic in nature, their scalability has been shown to be superior to their BFS-based counterparts.

9.3 Preliminaries

We will now define the general formalism of well-structured transition systems [FS01; Abd+96], and specifically networks of timed automata under the zone abstraction [CJ99].

Definition 9.1 (Well-quasi-ordering)**.** *A well-quasi-ordering $\sqsubseteq$ is a reflexive and transitive relation over a set X, s.t. for any infinite sequence $x_0, x_1, \dots$ eventually for some $i < j$ it will hold that $x_i \sqsubseteq x_j$.*

In other words, in any infinite sequence eventually an element exists which is "larger" than some earlier element.

Definition 9.2 (Well-structured transition system)**.** *A well-structured transition system is a 3-tuple $(S, \rightarrow, \sqsubseteq)$, where S is the set of states, $\rightarrow : S \times S$ is the (computable) transition relation and $\sqsubseteq$ is a well-quasi-ordering over S, s.t. if $s \rightarrow t$ then $\forall s'. s \sqsubseteq s'$ there $\exists t'. s' \rightarrow t' \wedge t \sqsubseteq t'$.*[9.1]

[9.1] With strong compatibility, see [FS01]

We thus require $\sqsubseteq$ to be a monotonic ordering on the behavior of states, i.e., if $s \sqsubseteq t$ then t has at least the behavior of s (and possibly more), and we say that t *subsumes* or *covers* s.

One instance of a *well-structured transition system* is shaped by the symbolic semantics of *timed automata*. Timed automata are finite state machines with a finite set of real-valued, resettable *clocks*. Transitions between states can be guarded by constraints on clocks, denoted $G(C)$.

Definition 9.3 (Timed automaton)**.** *An extended timed automaton is a 7-tuple* $\mathcal{A} = (L, C, Act, s_0, \rightarrow, I_C)$ *where*

- *L is a finite set of locations, typically denoted by ℓ*
- *C is a finite set of clocks, typically denoted by c*
- *Act is a finite set of actions*
- *$s_0 \in L$ is the initial location*
- *$\rightarrow \subseteq L \times G(C) \times Act \times 2^C \times L$ is the (non-deterministic) transition relation. We normally write $\ell \xrightarrow{g,a,r} \ell'$ for a transition, where ℓ is the source location, g is the guard over the clocks, a is the action, and r is the set of clocks reset.*
- *$I_C : L \rightarrow G(C)$ is a function mapping locations to downwards closed clock invariants.*

Using the definition of extended timed automata we can now define networks of timed automata, as modeled by UPPAAL, see [BDL04] for details. A network of timed automata is a parallel composition of extended timed automata that enables synchronization over a finite set of channel names *Chan*. We let $ch!$ and $ch?$ denote the output and input action on a channel $ch \in Chan$.

Definition 9.4 (Network of timed automata)**.** *Let* $Act = \{ch!, ch? | ch \in Chan\} \cup \{\tau\}$ *be a finite set of actions, and let C be a finite set of clocks. Then the parallel composition of extended timed automata* $\mathcal{A}_i = (L_i, C, Act, s_0^i, \rightarrow_i, I_C^i)$ *for all* $1 \leq i \leq n$*, where* $n \in \mathbb{N}$*, is a network of timed automata, denoted* $\mathcal{A} = \mathcal{A}_1 || \mathcal{A}_2 || \ldots || \mathcal{A}_n$*.*

The concrete semantics of timed automata [BDL04] gives rise to a possibly uncountable state space. To model check it a finite abstraction of the state space is needed; the abstraction used by most model checkers is the *zone abstraction* [Bou04]. Zones are sets of *clock constraints* that can be efficiently represented by *difference bound matrices* (DBMs) [Ben02]. The fundamental operations of DBMs are:

- $D\uparrow$ modifying the constraints such that the DBM represents all the clock valuations that can result from delay from the current constraint set
- $D \cap D'$ adding additional constraints to the DBM, e.g. because a transition is taken that imposes a clock constraint (guard clock constraints can also be represented as a DBM, and we will do so) [9.2]. The additional constraints might also make the DBM empty, meaning that no clock valuations can satisfy the constraints.
- $D[r]$ where $r \subseteq C$ is a clock reset of the clocks in r.
- D/B doing maximal bounds extrapolation, where $B : C \to \mathbb{N}_0$ is the maximal bounds needed to be tracked for each clock. Extrapolation with respect to maximal bounds [Beh+03] is needed to make the number of DBMs finite. Basically, it is a mapping for each clock indicating the maximal possible constant the clock can be compared to in the future. It is used in such a way that if the value of a clock has passed its maximal constant, the clock's value is indistinguishable for the model.
- $D \subseteq D'$ for checking if the constraints of D' imply the constraints of D, i.e. D' is a more relaxed DBM. D' has the behavior of D and possibly more.

Lemma 9.1. *Timed automata under the zone abstraction are well-structured transition systems:* $(S, \Rightarrow_{DBM}, Act, \sqsubseteq)$ *s.t.*

1. *S consists of pairs* (ℓ, D) *where* $\ell \in L$*, and D is a DBM.*
2. $\Rightarrow_{DBM}$ *is the symbolic transition function using DBMs, and Act is as before*
3. $\sqsubseteq: S \to S$ *is defined as* $(\ell, D) \sqsubseteq (\ell', D')$ *iff* $\ell = \ell'$*, and* $D \subseteq D'$*.*

Remark that part of the ordering $\sqsubseteq$ is compared using discrete equality (the location vector), while only a subpart is compared using a well-quasi-ordering. Without loss of generality, and as done in [Dal+11], we can split the state into an explicit part $\mathcal{S}$, and a symbolic part Σ, s.t. the well-structured transition system is defined over $\mathcal{S} \times \Sigma$. We denote the explicit part as $s, t, r \in \mathcal{S}$ and the symbolic part of states by $\sigma, \tau, \rho, \pi, \upsilon \in \Sigma$, and a state as a pair (s, σ).

Model checking of safety properties is done by proving or disproving the reachability of a certain concrete goal location s_g.

[9.2] The DBM might need to be put into normal form after more constraints have been added [Bou04]

Definition 9.5 ((Safety) Model checking of a well-structured transition system)**.** *Given a well-structured transition system* $(\mathcal{S} \times \Sigma, \rightarrow, \sqsubseteq)$*, an initial state* $(s_0, \sigma_0) \in \mathcal{S} \times \Sigma$*, and a goal location* s_g *does a path exist* $(s_0, \sigma_0) \rightarrow \cdots \rightarrow (s_g, \sigma_g')$*.*

In practice, the transition system is constructed *on-the-fly* starting from (s_0, σ_0) and recursively applying $\rightarrow$ to discover new states. To facilitate this, we extend the next-state interface of PINS with subsumption:

Definition 9.6. *A next-state interface with subsumption has three functions:*
INITIAL-STATE$() = (s_0, \sigma_0)$*,*
NEXT-STATE$((s, \sigma)) = \{(s_1, \sigma_1), \dots, (s_n, \sigma_n)\}$ *returning all successors of* (s, σ)*,* $(s, \sigma) \rightarrow (s_i, \sigma_i)$*, and*
COVERS$(\sigma', \sigma) = \sigma \sqsubseteq \sigma'$ *returning whether the symbolic part* σ' *subsumes* σ*.*

9.4 A Multi-Core Timed Reachability Tool

For the construction of our real-time multi-core model checker, we made an effort to reuse and combine existing components, while extending their functionality where necessary. For the specification models, we use the UPPAAL XML format. This enables the use of its extensive real-time modeling language through an excellent user interface. To implement the model's semantics (in the form of a next-state interface) we rely on OPAAL and the UPPAAL DBM library.[9.3] Finally, LTSMIN is used as a model checking backend, because of its language-independent design.

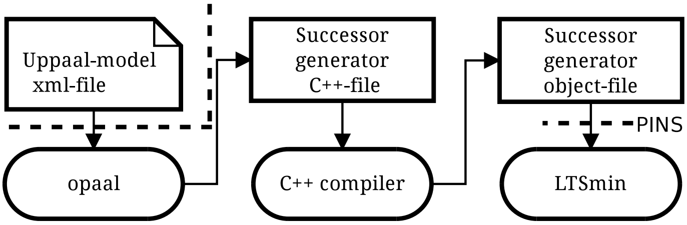


Figure 9.1: Reachability with subsumption [Dal+11]

Figure 9.1 gives an overview of the new toolchain. It shows how the XML input file is read by OPAAL which generates C++ code. The C++ file implements the PINS interface with subsumption specifically for the input model. Hence, after compilation (C++ compiler), LTSMIN can load the object file to perform the model checking.

[9.3] `http://people.cs.aau.dk/~adavid/UDBM/`

Previously, the OPAAL tool was used to generate Python code [Dal+11], but important parts of its infrastructure, e.g., analyzing the model to find max clock constants [Beh+03], can be reused. In Section 9.5, we describe how OPAAL implements the semantics of timed automata, and the structure of the generated C++ code.

The PINS interface of the LTSMIN tool [BPW10] has been shown to enable efficient, yet language-independent, model checking algorithms of different flavors, inter alia: distributed [BPW10], symbolic [BPW10] and multi-core reachability [LPW11a] (see Part II), and LTL model checking [BL13a] (see Part III). We extended the PINS interface to distinguish the new symbolic states of the OPAAL successor generator according to Definition 9.6. In Section 9.6, we describe our new multi-core reachability algorithms with subsumption.

9.5 Successor Generation using opaal

The OPAAL tool was designed to rapidly prototype new model checking features and as such was designed to be extended with other successor generators. It already implements a substantial part of the UPPAAL features. For an explanation of the UPPAAL features see [BDL04, p. 4-7]. The new C++ OPAAL successor generator supports the following features: templates, constants, bounded integer variables, arrays, selects, guards, updates, invariants on both variables and clocks, committed and urgent locations, binary synchronization, broadcast channels, urgent synchronization, selects, and much of the C-like language that UPPAAL uses to express guards and variable updates.

A state in the symbolic transition system using DBMs, is a location vector and a DBM. To represent a state in the C++ code we use a struct with a number of components: one integer for each location, and a pointer to a DBM object from the UPPAAL DBM library. Therefore a state is a tuple: $(\ell_1, \ldots, \ell_n, D)$.

The INITIAL-STATE function is rather straightforward: it returns a state struct initialized to the initial location vector, and a DBM representing the initial zone (delayed, and with invariants applied as necessary). The structure of the NEXT-STATE function is more involved, because it needs to consider the syntactic structure of the model, as can be seen in Algorithm 9.1.

At Line 4, we consider all outgoing transitions for the current location of each process (Line 3). If the transition is internal, we can evaluate it right away, and possibly generate a successor at Line 12. If it is a sending synchronization (`ch!`), we need to find possible synchronization partners (Line 15). So again we iterate over all processes and the transitions of their current locations (Line 14–21).

In the generated C++ code a few optimizations have been made, compared to Algorithm 9.1: The loops on line Line 3 and Line 14 have been unrolled, since the number

Algorithm 9.1 Overall structure of the successor generator

```
 1  proc NEXT-STATE($s_{in} = (\ell_1, \dots, \ell_n, D)$)
 2  out_states := $\emptyset$
 3  for $\ell_i \in \ell_1, \dots, \ell_n$
 4    for all $\ell_i \xrightarrow{g,a,r} \ell'_i$
 5      $D' := D \cap g$
 6      if $D' \neq \emptyset$                                        ▷ is the guard satisfied?
 7        if $a = \tau$                                   ▷ this is not a synchronizing transition
 8          $D' := D'[r]\uparrow$                                              ▷ clock reset, delay
 9          $D' := D' \cap I^i_C(\ell'_i) \cap \bigcap_{k \neq i} I^k_C(\ell_k)$                  ▷ apply clock invariants
10          if $D' \neq \emptyset$
11            $D' := D'/B(\ell_1, \dots, \ell'_i \dots, \ell_n)$
12            out_states := out_states $\cup \{(\ell_1, \dots, \ell'_i, \dots, \ell_n, D')\}$
13        else if $a = ch!$                                             ▷ binary sync. sender
14          for $\ell_j \in \ell_1, \dots, \ell_n, j \neq i$
15            for all $\ell_j \xrightarrow{g_j, ch?, r_j} \ell'_j$                                   ▷ find receivers
16              if $D'' := D' \cap g_j \neq \emptyset$                          ▷ receiver guard satisfied?
17                $D'' := D''[r][r_j]\uparrow$                                       ▷ clock resets, delay
18                $D'' := D'' \cap I^i_C(\ell'_i) \cap I^j_C(\ell'_j) \cap \bigcap_{k \notin \{i,j\}} I^k_C(\ell_k)$        ▷ apply clock invariants
19                if $D'' \neq \emptyset$
20                  $D'' := D''/B(\ell_1, \dots, \ell'_i, \dots, \ell'_j \dots, \ell_n)$
21                  out_states := out_states $\cup \{(l_1, \dots, l'_i, \dots, l'_j, \dots, l_n, D'')\}$
22  return out_states
```

of processes they iterate over is known beforehand. In that manner the transitions to consider can be efficiently found. As an optimization, before starting the code generation, we compute the set of all possible receivers for all channels, for the unrolling of Line 14. In practice there are usually many receivers but few senders for each channel, resulting in the unrolling being an acceptable trade-off.

When doing the max bounds extrapolation (/) in Algorithm 9.1, we obtain the bounds from a location-dependent function $B : L_1 \times \cdots \times L_n \rightarrow (C \rightarrow \mathbb{N}_0)$. This function is precomputed in OPAAL using the method described in [Beh+03].

Some features are not formalized in this work, but have been implemented for ease of modeling. We support integer variables, urgency that can be modeled using urgent/committed locations and urgent channels, but also channel arrays with dynamically computed senders, broadcast channels, and process priorities. These are all implemented as simple extensions of Algorithm 9.1. Other features are supported in the form of a syntactic expansion, namely: selects, and templates.

To make the next-state function thread-safe, we had to make the uppaal DBM library thread-safe. Therefore, we replaced its internal allocator with a concurrent memory allocator (see Section 9.7). We also replaced the internal hash table, used to filter duplicate DBM allocations, with a concurrent hash table.

9.6 Well-Structured Transition Systems in LTSmin

The current section presents the parallel reachability algorithm that was implemented in LTSmin to handle well-structured transition systems. According to Definition 9.6, we can split up states into a discrete part, which is always compared using equality (for timed automata this consists of the locations and variables), and a part that is compared using a well-quasi-ordering (for timed automata this is the DBM).

We recall the sequential algorithm from [Dal+11] (Algorithm 9.2) and adapt it to use the next-state interface with subsumption. At its basis, this algorithm is a search with a waiting set (W), containing the states to be explored, and a passed set (P), containing the states that are already explored.

New successors (t,τ) are added to W (Line 9), but only if they are not subsumed by previous states (Line 8). Additionally, states in the waiting set W that are subsumed by the new state are discarded (Line 9), avoiding redundant explorations.

Algorithm 9.2 Reachability with subsumption [Dal+11]

```
1  proc reachability($s_g$)
2    $W$ := { INITIAL-STATE() }; $P$ := $\emptyset$
3    while $W \neq \emptyset$
4      $W$ := $W \setminus (s,\sigma)$ for some $(s,\sigma) \in W$
5      $P$ := $P \cup \{(s,\sigma)\}$
6      for $(t,\tau) \in$ NEXT-STATE($(s,\sigma)$) do
7        if $t = s_g$ then report & exit
8        if $\nexists \rho : (t,\rho) \in W \cup P \wedge$ COVERS($\rho,\tau$)
9          $W$ := $W \setminus \{(t,\rho) \mid$ COVERS($\tau,\rho$)$\} \cup (t,\tau)$
```

9.6.1 A Parallel Reachability Algorithm with Subsumption

In the parallel setting, we localize all work sets (Q_p, for each worker p) and create a shared data structure L storing both W and P. We attach a status flag passed or waiting to each state in L to create a global view of the passed and waiting set and avoid unnecessary reexplorations. L can be represented as a multimap, saving multiple symbolic state parts with each explicit state part $L : \mathcal{S} \to \Sigma^*$. To make L thread-safe, we protect its operations with a fine-grained locking mechanism that locks only the part of the map associated with an explicit state part s: **lock**$(L(s))$, similar to the spinlocks used in Chapter 2. An off-the-shelf load balancer takes care of distributing work at the startup and when some Q_p runs empty prematurely. This design corresponds to the shared hash table approach discussed in Section 9.2 and avoids a *static partitioning* of the state space.

Algorithm 9.3 presents the discussed design. The algorithm is initialized by calling reachability with the desired number of threads P and a discrete goal location s_g. This method initializes the shared data structure L and gets the initial state using the INITIAL-STATE function from the next-state interface with subsumption. The initial state is then

Algorithm 9.3 Reachability with cover update of the waiting set

```
global $L : \mathcal{S} \to (\Sigma \times \{\text{waiting}, \text{passed}\})^*$

proc reachability$(P, s_g)$
  $L := \mathcal{S} \to \emptyset$
  $(s_0, \sigma_0) := s :=$ INITIAL-STATE()
  $L(s_0) := (\sigma_0, \text{waiting})$
  search$(s, s_g, 1) || \ldots ||$search$(s, s_g, P)$

proc update$(t, \tau)$
  lock$(L(t))$
  for $(\rho, f) \in L(t)$ do
    if COVERS$(\rho, \tau)$
      unlock$(L(t))$
      return true
    else if $f = \text{waiting} \wedge$ COVERS$(\tau, \rho)$
      $L(t) := L(t) \setminus (\rho, \text{waiting})$
  $L(t) := L(t) \cup (\tau, \text{waiting})$
  unlock$(L(t))$
  return false

proc search$((s_0, \sigma_0), s_g, p)$
  $Q_p :=$ if $p = 1$ then $\{(s_0, \sigma_0)\}$ else $\emptyset$
  while $Q_p \neq \emptyset \vee$ balance$(Q_p)$
    $Q_p := Q_p \setminus (s, \sigma)$ for some $(s, \sigma) \in Q_p$
    if $\neg$grab$(s, \sigma)$ then continue
    for $(t, \tau) \in$ NEXT-STATE$((s, \sigma))$ do
      if $t = s_g$ then report & exit
      if $\neg$update$(t, \tau)$
        $Q_p := Q_p \cup (t, \tau)$

proc grab$(s, \sigma)$
  lock$(L(s))$
  if $\sigma \notin L(s) \vee \text{passed} = L(s, \sigma)$
    unlock$(L(s))$
    return false
  $L(s, \sigma) := \text{passed}$
  unlock$(L(s))$
  return true
```

added to L and the worker threads are initialized at Line 7. Worker thread 1 explores the initial state; work load is propagated later.

The **while** loop on Line 23 corresponds closely to the sequential algorithm, in a quick overview: a state (s,σ) is taken from the work set at Line 24, its flag is set to passed by grab if it were not already, and then the successors (t,τ) of (s,σ) are checked against the passed and the waiting set by update. We now discuss the operations on L (update, grab) and the load balancing in more detail.

To implement the subsumption check (line Line 8–9 in Algorithm 9.2) for successors (t,τ) and to update the waiting set concurrently, update is called. It first locks L on t. Now, for all symbolic parts and status flag ρ, f associated with t, the method checks if τ is already covered by ρ. In that case (t,τ) will not be explored. Alternatively, all ρ with status flag waiting that are covered by τ are removed from $L(t)$ and τ is added. The update algorithm maintains the invariant that a state in the waiting set is never subsumed by any other state in L: $\forall s \forall (\rho, f), (\rho', f') \in L(s)\colon f = \text{waiting} \wedge \rho \neq \rho' \Rightarrow \rho \not\sqsubseteq \rho'$ (**Inv. 1**). Hence, similar to Algorithm 9.2 Line 8–9, it can never happen that (t,τ) first discards some (t,ρ) from $L(s)$ (Line 16) and is discarded itself in turn by some (t,ρ') in $L(s)$ (Line 12), since then we would have $\rho \sqsubseteq \tau \sqsubseteq \rho'$; by transitivity of $\sqsubseteq$ and the invariant, ρ and ρ' cannot be both in $L(t)$. Finally, notice that update unlocks $L(t)$ on all paths.

The task of the method grab is to check if a state (s,σ) still needs to be explored, as it might have been explored by another thread in the meantime. It first locks $L(s)$. If σ is no longer in $L(s)$ or it is no longer globally flagged waiting (Line 33), it is discarded (Line 25). Otherwise, it is "grabbed" by setting its status flag to passed. Notice again that on all paths through grab, $L(s)$ is unlocked.

Finally, the method balance handles termination detection and load balancing. It has the side-effect of adding work to Q_p. We use a standard solution [San97b].

9.6.2 Exploration Orders

The shared hash table approach gives us the freedom to allow for a DFS or BFS exploration order depending on the implementation of Q_p. Note, however, that only pseudo-DFS/BFS is obtained, due to randomness introduced by parallelism.

It has been shown for timed automata that the number of generated states is quite sensitive to the exploration order and that in most cases strict BFS shows the best results [BHV00]. Fortunately, we can obtain strict BFS by synchronizing workers between the different BFS levels. To this end, we first split Q_p into two separate sets that hold the current BFS level (C_p) and the next BFS level (N_p) [Aga+10]. The order within these sets does not matter, as long as the current is explored before the next set. Load balancing will only be performed on C_p, hence a level terminates once $C_p = \emptyset$ for all p. At

Algorithm 9.4 Strict parallel BFS

```
1   proc search($s_0, \sigma_0, p$)
2     $C_p$ := if $p = 1$ then $\{(s_0, \sigma_0)\}$ else $\emptyset$
3     do
4       while $C_p \neq \emptyset \lor$ balance($C_p$)
5         $C_p$ := $C_p \setminus (s, \sigma)$ for some $(s, \sigma) \in C_p$
6           ...
7             $N_p$ := $N_p \cup (t, \tau)$
8       $load$ := reduce($sum, |N_p|, P$)
9       $C_p, N_p$ := $N_p, \emptyset$
10    while $load \neq 0$
```

this point, if $N_p = \emptyset$ for all p, the algorithm can terminate because the next BFS level is empty. The synchronizing reduce method counts $\sum_{i=1}^{P} |N_i|$ (similar to `mpi_reduce`).

Algorithm 9.4 shows a parallel strict-BFS implementation. An extra outer loop iterates over the levels, while the inner loop (Line 4–7) is the same as in Algorithm 9.3. Except for the lines that add and remove states to and from the work set, which now operate on N_p and C_p. The (pointers to) the work sets are swapped, after the reduce call at Line 8 calculates the load of the next level.

9.6.3 A Data Structure for Semi-Symbolic States

In Chapter 2, we introduced a lockless hash table, which we reuse here to design a data structure for L that supports the operations used in Algorithm 9.3. To allow for massive parallelism on modern multi-core machines with steep memory hierarchies, it is crucial to keep a low memory footprint. To this end, lookups in the large table of state data are filtered through a separate smaller table of hashes. The table assigns a unique number (the hash location) to each explicit state stored in it: $D\colon \mathcal{S} \to \mathbb{N}$. In finite reality, we have: $D\colon \mathcal{S} \to \{1, \ldots, N\}$.

We now reuse the state numbering of D to create a multimap structure for L. The first component of the new data structure is an array $I[N]$ used for indexing on the explicit state parts. To associate a set of symbolic states (pointers to DBMs) with our explicit state stored in $D[x]$, we are going to attach a linked list structure to $I[x]$. Creating a standard linked list would cause a single *cache line* access per element, increasing the memory footprint, and would introduce costly synchronizations for each modification. Therefore, we allocate multi-buckets, i.e., an array of pointers as one linked list element. To save memory, we store lists of just one element directly in I and completely fill the

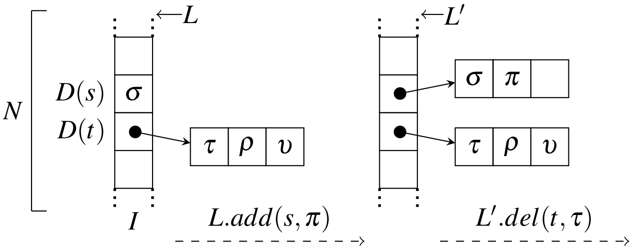


Figure 9.2: Data structure for L, and operations

last multi-bucket.

Figure 9.2 shows three instances of the discussed data structure: L, L' and L''. Each multimap is a pointer (arrow) to an array I shown as a vertical bucket array. L contains $\{(s,\sigma),(t,\tau),(t,\rho),(t,\upsilon)\}$. We see how a multi-bucket with (fixed) length 3 is created for t, while the single symbolic state attached to s is kept directly in I. The figure shows how σ is moved when (s,π) is added by the add operation (dashed arrow), yielding L'. Adding π to t would have moved υ to a new linked multi-bucket together with π.

Removing elements from the waiting list is implemented by marking bucket entries as tombstone, so they can later be reused (see L''). This avoids memory fragmentation and expensive communication to reuse multi-buckets. For highest scalability, we allocate multi-buckets of size 8, equal to a cache line. Other values can reduce memory usage, but we found this sufficiently efficient (see Section 9.7).

We still need to deal with locking of explicit states, and storing of the various flags for symbolic states (waiting/passed). Internally, the algorithms also need to distinguish between the different buckets: empty, tomb stone, linked list pointers and symbolic state pointers. To this end, we can bit-cram additional bits into the pointers in the buckets,

Algorithm 9.5 Bit layout of word-sized bucket

```
struct link_or_dbm {
  bit pointer[60]
  bit flag ∈ {waiting, passed}
  bit lock ∈ {locked, unlocked}
  bit status[2] ∈ {empty, tomb, dbm_ptr, list_ptr}
}
```

as is shown in Algorithm 9.5. Now **lock**$(L(s))$ can be implemented as a spinlock using the atomic *compare-and-swap* (CAS) instruction on $I[s]$ (see Section 2.2). Since all operations on $L(s)$ are done after **lock**$(L(s))$, the corresponding bits of the buckets can be updated and read with normal load and store instructions.

9.6.4 Improving Scalability through a Non-Blocking Implementation

The size of the critical regions in Algorithm 9.3 depends crucially on the $|\Sigma|/|\mathcal{S}|$ ratio; a higher ratio means that more states in $L(t)$ have to be considered in the method update (t, τ), affecting scalability negatively. A similar limitation is reported for distributed reachability [BOS06]. Therefore, we implemented a *non-blocking* version: instead of first deleting all subsumed symbolic states with a waiting flag, we atomically replace them with the larger state using CAS. For a failed CAS, we retry the subsumption check after a reread. L can be atomically extended using the well-known *read-copy-update* technique. However, workers might miss updates by others, as Inv. 1 no longer holds. This could cause $|\Sigma|$ to increase again.

9.7 Experiments

To investigate the performance of the generated code, we compare full reachability in OPAAL+LTSMIN with the current state-of-the-art (UPPAAL).[9.4] To investigate scalability, we benchmarked on a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 6168) with a varying number of threads. Statistics on memory usage were gathered and compared against UPPAAL. We also experimented with different exploration orders and tree compression from Chapter 3. Experiments were repeated 5 times.

We consider three models from the UPPAAL demos: `viking` (one discrete variable, but many synchronizations), `train-gate` (relatively large amount of code, several variables), and `fischer` (very small discrete part). Additionally, we experiment with a generated model, `train-crossing`, which has a different structure from most handmade models. For some models, we created multiple numbered instances, the numbers represent the number of processes in the model.

For UPPAAL, we ran the experiments with BFS and disabled space optimization. The `opaal_ltsmin` script in OPAAL was used to generate and compile models. In LTSMIN we used a fixed hash table (`--state=table`) size of 2^{26} states (`-s26`), waiting set updates as in Algorithm 9.3 (`-u1`) and multi-buckets of size 8 (`-l8`).

[9.4] OPAAL is available at `https://code.launchpad.net/~opaal-developers/opaal/opaal-ltsmin-succgen`, LTSMIN at `http://fmt.cs.utwente.nl/tools/ltsmin/`

9.7.1 Performance & Scalability

Table 9.1 shows the reachability runtimes of the different models in uppaal and opaal+LTSmin with strict BFS (`--strategy=sbfs`). Except for `fischer6`, we see that both tools compete with each other on the sequential runtimes, with 2 threads however opaal+LTSmin is faster than uppaal. With the massive parallelism of 48 cores, we see how verification tasks of minutes are reduced to mere seconds. The outlier, `fischer6`, is likely due to the use of more efficient clock extrapolations in uppaal, and other optimizations, as witnessed by the evolution of the runtime of this model in [Beh+11; Amn+01].

Table 9.1: $\mathcal{S}$, $|\Sigma|$ ($\frac{|\Sigma|}{|\mathcal{S}|}$) and runtimes (sec) in uppaal and opaal+LTSmin (strict BFS)

	$\lvert\mathcal{S}\rvert$	uppaal		opaal+LTSmin (cores)							
		T	$\lvert\Sigma\rvert$	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_2	T_8	T_{16}	T_{32}	T_{48}
train-gate-N10	7e+07	837.4	1.0	1.0	1.0	573.3	297.8	76.7	39.4	21.1	14.4
viking17	1e+07	207.8	1.0	1.5	1.5	331.5	172.5	44.2	22.7	11.9	8.6
train-gate-N9	7e+06	76.8	1.0	1.0	1.0	51.8	27.5	7.2	3.7	2.0	1.4
viking15	3e+06	38.0	1.0	1.5	1.5	67.0	34.8	9.7	5.1	3.0	2.3
train-crossing	3e+04	48.3	20.8	16.1	17.3	23.1	37.8	3.7	2.1	1.4	1.4
fischer6	1e+04	0.1	0.3	50.1	50.1	177.3	112.3	39.4	30.4	27.9	30.3

We noticed that the 48-core runtimes of the smaller models were dominated by the small BFS levels at the beginning and the end of the exploration due to synchronization in the load balancer and the reduce function. This overhead takes consistently 0.5–1 second, while it handles less than thousand states. Hence, to obtain useful scalability measurements we excluded this time for the benchmarks.

To investigate the scalability better, we plotted the speedups in Figure 9.3 using the average runtimes from Table 9.1. The standard deviation of the speedup is plotted as vertical lines (mostly negligible, hence invisible). Most models show almost linear scalability with a speedup of up to 40, e.g. `train-gate-N10`. As expected, we see that a high $|\Sigma|/|\mathcal{S}|$ ratio causes low scalability (see `fischer` and `train-crossing` and Table 9.1). Therefore, we tried the non-blocking variant (Section 9.6.3) of our algorithm (`-n`). As expected, the speedups in Figure 9.4 improve and the runtimes even show a threefold improvement for `fischer.6` (Table 9.2). The efficiency on 48

Table 9.2: $|\Sigma|$ ($\frac{|\Sigma|}{|\mathcal{S}|}$) and runtimes (sec) with non-blocking strict BFS, pseudo DFS and pseudo BFS

	NB Strict BFS				(Pseudo) DFS				(Pseudo) BFS			
	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}	$\lvert\Sigma_1\rvert$	$\lvert\Sigma_{48}\rvert$	T_1	T_{48}
train-gate-N10	1.0	1.0	547.9	14.5	1.0	1.0	647.8	15.6	1.0	1.0	559.3	13.1
viking17	1.5	1.5	320.1	9.2	1.6	1.6	386.5	9.1	1.5	1.5	325.6	7.8
train-gate-N9	1.0	1.0	51.2	1.4	1.0	1.0	61.7	1.7	1.0	1.0	51.9	1.6
viking15	1.5	1.5	64.8	2.5	1.6	1.6	80.2	3.1	1.5	1.5	66.0	2.3
train-crossing	16.1	16.1	23.1	0.9	169.8	179.0	3371.0	297.4	16.1	37.1	24.5	157.5
fischer6	50.1	50.1	196.1	10.7	54.4	39.4	405.1	10.6	50.1	58.1	206.0	32.3

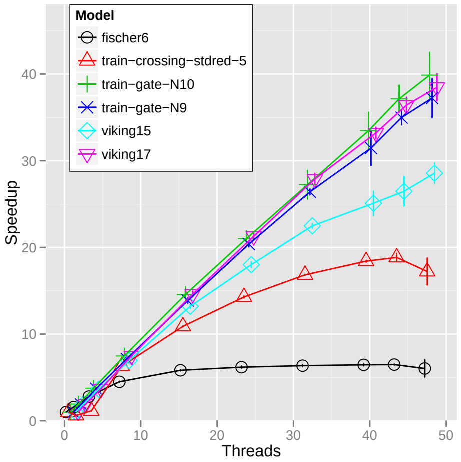
cores remains closely dependent to the $|\Sigma|/|\mathcal{S}|$ ratio of the model (or the average length of the lists in the multimap), but the scalability is now at least sublinear and not stagnant anymore.

We further investigated different search orders. Figure 9.5 shows results with pseudo BFS order (`--strategy=bfs`). While speedups become higher due to the lacking level synchronizations, the loose search order tends to reach "large" states later and therefore generates more states for two of the models ($|\Sigma_1|$ vs $|\Sigma_{48}|$ in Table 9.2). This demonstrates that our strict BFS implementation indeed pays off.

Finally, we also experimented with randomized DFS search order (`--perm=rr --strategy=dfs`). Table 9.2 shows that DFS causes again more states to be generated. But, surprisingly, the number of states actually reduces with the parallelism for the `fischer6` model, even below the state count of strict BFS from Table 9.1! This causes a superlinear speedup in Figure 9.6 and threefold runtime improvement over strict BFS. We do not consider this behavior as an exception (even though `train-crossing` does not show it), since it is compatible with our observation that parallel DFS finds shorter counterexamples than parallel BFS (see Section 7.4.4).

9.7.2 Design Decisions

Some design decisions presented here were motivated by earlier work that has proven successful for multi-core model checking (see Part II and Part III). In particular, we reused the shared hash table and a *synchronous* load balancer [San97b]. Even though



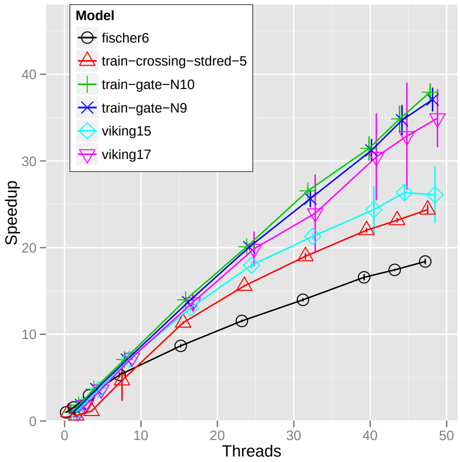
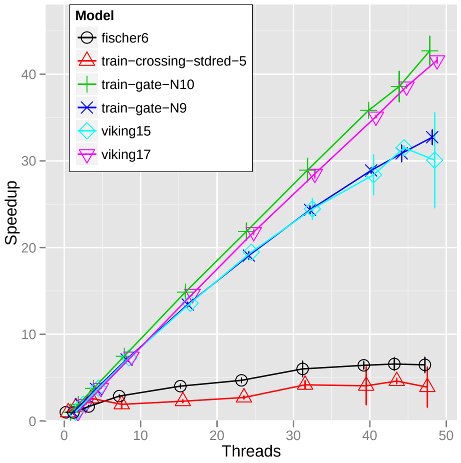


Figure 9.3: Speedup strict BFS

Figure 9.4: Speedup non-blocking strict BFS



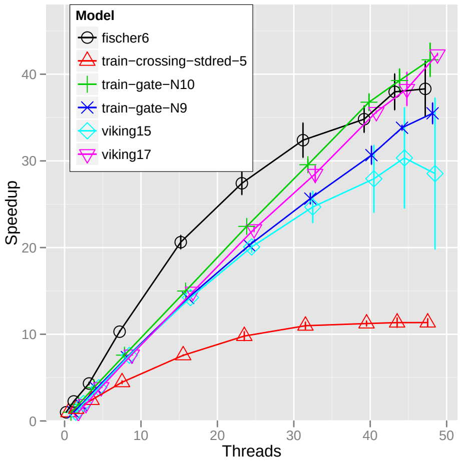


Figure 9.5: Speedup pseudo BFS

Figure 9.6: Speedup randomized pseudo DFS

we observed load distributions close to ideal, a modern work stealing solution might still improve our results, since the work granularity for timed reachability is higher than for untimed reachability. The main bottlenecks, however, have proven to be the increase in state count by parallelism and the cost of the spinlocks due to a high $|\Sigma|/|\mathcal{S}|$ ratio. The latter we partly solved with a non-blocking algorithm. Strict BFS orders have proven to aid the former problem and randomized DFS orders could aid both problems.

9.7.3 Memory Usage

Table 9.3 shows the memory consumption of uppaal (U-S0) and sequential opaal+LTSmin (O+L$_1$) with strict BFS. From it, we conclude that our memory usage is within 25% of uppaal's for the larger models (where these measurements are precise enough). Furthermore, we extensively experimented with different concurrent allocators and found that TBB malloc (used in the current chapter) yields the best performance for our algorithms.[9.5] Its overhead (O+L$_1$ vs O+L$_{48}$ in Table 9.3) appears to be limited to a moderate fixed amount of 250MB more than the sequential runs, for which we used the normal `glibc` allocator.

We also counted the memory usage inside the different data structures: the multimap L (including partly-filled multi-buckets), the hash table D, the combined local work sets (Q), and the DBM duplicate table (dbm). As we expected the overhead of the 8-sized multi-buckets is little compared to the size of D and the DBMs. We may however replace D with the compressed, parallel tree table (T) from Chapter 3. The resulting total memory usage (O+L^T), can now be dominated by L, as is the case for `viking17`. But if we reduce L to a linked list (`-l2`), its size shrinks only by 60% to 214MB for this model (L2). Just a modest gain.

Table 9.3: Memory usage (MB) of both uppaal (U-S0 and U-S2) and opaal+LTSmin

	T	D	L	L2	Q	dbm	O+L$_1$	O+L$_{48}$	O+L$^T_1$	O+L$^T_{48}$	U-S0	U-S2
train-gate-N10	777	5989	499	499	249	1363	8101	8241	2790	3028	6091	3348
viking17	156	1040	536	214	40	87	1704	1931	828	1047	1579	722
train-gate-N9	81	549	50	50	24	61	684	815	214	347	607	332
viking15	32	190	112	44	8	55	364	581	203	423	333	162
train-crossing	0	2	5	7	0	419	426	623	425	622	48	64
fischer6	0	0	5	9	1	176	429	512	290	429	0	4

[9.5] cf. `http://fmt.cs.utwente.nl/tools/ltsmin/formats-2012/` for additional data

For completeness, we included the results of UPPAAL's state space optimization (U-S2). As expected, it also yields great reductions, which is the more interesting since the two techniques are orthogonal and could be combined.

9.8 Conclusions

We presented novel algorithms and data structures for multi-core reachability on well-structured transition systems and an efficient implementation for timed automata in particular. Experiments show good speedups, up to 40 times on a 48-core machine and also identify current bottlenecks. In particular, we see speedups of more than 60 times compared to UPPAAL. Memory usage is limited to an acceptable maximum of 25% more than UPPAAL.

Our experiments demonstrate the flexibility of the search order that our parallel approach allows for. BFS-like order is shown to be occasionally slightly faster than strict BFS but is substantially slower on other models, as previously observed in the distributed setting. A new surprising result is that parallel randomized (pseudo) DFS order sometimes reduces the state count below that of strict BFS, yielding a substantial speedup in those cases.

Previous work has shown that better parallel reachability in Part II crucially enables new and better solutions to parallel model checking of liveness properties (see Part III). Therefore, our natural next step is to port multi-core nested depth-first search solutions to the timed automata setting.

Because of our use of generic toolsets, more possibilities are open to be explored. The OPAAL support for the UPPAAL language can be extended and support for optimizations like symmetry reduction and partial-order reduction could be added, enabling easier modeling and better scalability. Additionally, lattice-based languages [Dal+11] can be included in the C++ code generator. On the backend side, the distributed [BPW10] and symbolic [BPW10] algorithms in LTSMIN can be extended to support subsumption, enabling other powerful means of verification. We also plan to add a join operator to the PINS interface, to enable abstraction/refinement-based approaches [Dal+11].

10 Multi-Core LTL Model Checking for Timed Automata

Alfons Laarman, Mads Chr. Olesen, Andreas Dalsgaard, Kim G. Larsen, Jaco van de Pol

Abstract

The current chapter contributes to the multi-core model checking of timed automata (TA) with respect to liveness properties, by investigating checking of TA Büchi emptiness under the very coarse inclusion abstraction or zone subsumption, an open problem in this field.

We show that in general Büchi emptiness is not preserved under this abstraction, but some other structural properties are preserved. Based on those, we propose a variation of the classical nested depth-first search (NDFS) algorithm that exploits subsumption. In addition, we extend the multi-core CNDFS algorithm with subsumption, providing the first parallel LTL model checking algorithm for timed automata.

The algorithms are implemented in LTSMIN, and experimental evaluations show the effectiveness and scalability of both contributions: subsumption halves the number of states in the real-world FDDI case study, and the multi-core algorithm yields speedups of up to 40 using 48 cores.

About this chapter: The current chapter is based on the paper *"Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction"*, which was published at CAV 2013 [Laa+13b].

Compared to the original text, we extended the discussion on the implementation in Section 10.6.2, relating this content more closely to the prequel work in Chapter 9. This also allows for a more detailed discussion of the scalability results, as reflected by the added conclusion (Section 10.7). We also shortened the title.

10.1 Introduction

Model checking safety properties can be done with reachability, but only guarantees that the system does not enter a dangerous state, not that the system actually serves some useful purpose. To model check such liveness properties is more involved since they state conditions over infinite executions, e.g. that a request must infinitely often produce a result. One of the most well-known logics for describing liveness properties is Linear Temporal Logic (LTL) [BK08].

The automata-theoretic approach for LTL model checking [VW86] solves the problem efficiently by translating it to the Büchi emptiness problem, which has been shown decidable for real-time systems as well [AD94]. However, its complexity is exponential, both in the size of the system specification and of the property. In the current chapter, therefore, we consider two possible ways of alleviating this so-called state-space explosion problem: (1) by utilizing the many cores in modern processors, and (2) by employing coarser abstractions to the state space.

Related work. The verification of timed automata was made possible by Alur and Dill's *region construction* [AD94], which represents clock valuations using constraints, called *regions*. A max-clock constant abstraction, or *k*-extrapolation, bounded the number of regions. Since the region construction is exponential in the number of clocks and constraints in the TA, coarser abstractions such as the symbolic *zone abstraction* have been studied [Dil89], and also implemented in, among others, the state-of-the-art model checker UPPAAL [LPY97]. Later, the *k*-extrapolation for zones was refined to include lower clock constraints in the so-called lower/upper-bound (LU) abstraction proposed in [Beh+06]. Finally, the *inclusion abstraction*, or simply *subsumption*, prunes reachability according to the partial order of the symbolic states [DT98]. All these abstractions preserve reachability properties [DT98; Beh+06].

Model checking LTL properties on timed automata, or equivalently checking timed Büchi automata (TBA) emptiness [HS10], was proven decidable in [AD94], by using the region construction. Bouajjani et al. [BTY97] showed that the region-closed simulation graph preserve TBA emptiness. Tripakis [Tri09] proved that the *k*-extrapolated zone simulation graph also preserves TBA emptiness, while posing the question whether other abstractions such as the LU abstraction and subsumption also preserve this property. Li [Li09] showed that the LU abstraction does in fact preserve TBA emptiness. The status of subsumption in LTL model checking is still open.

One way of establishing TBA emptiness on a finite simulation graph is the nested depth-first (NDFS) algorithm [Cou+92; HPY96]. Recently, some multi-core version of these algorithms were introduced by Evangelista and Laarman et al. [EPY11; Laa+11;

Eva+12] (see Part III for [Laa+11; Eva+12]). These algorithms have the following properties: their runtime is linear in the number of states in the worst case while typically yielding good scalability; they are on-the-fly [LP11] and yield short counterexamples Section 7.4.4. The latest version, called CNDFS, combines all these qualities and decreases memory usage (see Section 7.4).

In previous work, we parallelized reachability for timed automata using the mentioned abstractions (see Chapter 9). It resulted in almost linear scalability, and speedups of up to 60 on a 48-core machine, compared to UPPAAL. The current work extends this previous work to the setting of liveness properties for timed automata. It also shares the UPPAAL input format, and re-uses the UPPAAL DBM library.

Problem statement. Parallel model checking of liveness properties for timed systems has been a challenge for several years. While advances were made with distributed versions of e.g. UPPAAL [Beh05], these were limited to safety properties. Furthermore, it is unknown how subsumption, the coarsest abstraction, can be used for checking TBA emptiness.

Contributions. (1) For the first time, we realize parallel LTL model checking of timed systems using the CNDFS algorithm. (2) We prove that subsumption preserves several structural state-space properties (Section 10.3), and show how these properties can be exploited by NDFS and CNDFS (Section 10.4 and Section 10.5). (3) We implement NDFS and CNDFS with subsumption in the LTSMIN toolset [LPW11a] and OPAAL [Dal+11]. Finally, (4) our experiments show considerable state-space reductions by subsumption and good parallel scalability of CNDFS with speedups of up to 40 using 48 cores.

10.2 Preliminaries: Timed Büchi Automata and Abstractions

In the current section, we first recall the formalism of timed Büchi automata (TBA), that allows modeling of both a real-time system and its liveness requirements. Subsequently, we introduce finite symbolic semantics using zone abstraction with extrapolation and subsumption. Finally, we show which properties are known to be preserved under said abstractions.

10.2.1 Timed Automata and Transition Systems

Definition 10.2 provides a basic definition of a TBA. It can be extended with features such as finitely valued variables, and parallel composition to model networks of timed automata, as done in UPPAAL [BDL04].

Definition 10.1 (Guards)**.** *Let $\mathcal{G}(\mathcal{C})$ be a conjunction of clock constraints over the set of clocks $c \in \mathcal{C}$, generalized by:*

$$g ::= c \bowtie n \mid g \wedge g \mid true$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, =, >, \geq\}$ is a comparison operator. We call a guard downwards closed if all $\bowtie \in \{<, \leq, =\}$.

Definition 10.2 (Timed Büchi Automaton)**.** *A timed Büchi automaton (TBA) is a 6-tuple $\mathcal{B} = (L, \mathcal{C}, \mathcal{F}, l_0, \rightarrow, I_{\mathcal{C}})$, where*

- *L is a finite set of locations, typically denoted by ℓ, where $\ell_0 \in L$ is the initial location, and $\mathcal{F} \subseteq L$, is the set of accepting locations,*
- *$\mathcal{C}$ is a finite set of clocks, typically denoted by c,*
- *$\rightarrow \subseteq L \times \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is the (non-deterministic) transition relation. We write $\ell \xrightarrow{g,R} \ell'$ for a transition, where ℓ is the source and ℓ' the target location, $g \in \mathcal{G}(\mathcal{C})$ is a transition guard, $R \subseteq \mathcal{C}$ is the set of clocks to reset, and*
- *$I_{\mathcal{C}} : L \rightarrow \mathcal{G}(\mathcal{C})$ is an invariant function, mapping locations to a set of guards. To simplify the semantics, we require invariants to be downwards-closed.*

The states of a TBA involve the notion of clock valuations. A clock valuation is a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$. We denote all clock valuations over $\mathcal{C}$ with $\mathcal{V}_{\mathcal{C}}$. We need two operations on clock valuations: $v' = v + \delta$ for a delay of $\delta \in \mathbb{R}_{\geq 0}$ time units s.t. $\forall c \in \mathcal{C}\colon v'(c) = v(c) + \delta$, and reset $v' = v[R]$ of a set of clocks $R \subseteq \mathcal{C}$ s.t. $v'(c) = 0$ if $c \in R$, and $v'(c) = v(c)$ otherwise. We write $v \models g$ to mean that the clock valuation v satisfies the clock constraint g.

Definition 10.3 (Transition system semantics of a TBA)**.** *The semantics of a TBA $\mathcal{B}$ is defined over the transition system $\mathcal{TS}^{\mathcal{B}}_{v} = (\mathcal{S}_v, s_0, \Rightarrow_v)$ s.t.:*

1. *A state $s \in \mathcal{S}_v$ is a pair: (ℓ, v) with a location $\ell \in L$, and a clock valuation v.*
2. *An initial state $s_0 \in \mathcal{S}_v$, s.t. $s_0 = (\ell_0, v_0)$, where $\forall c \in \mathcal{C}\colon v_0(c) = 0$.*

3. $\Rightarrow_v : \mathcal{S}_v \times (\{\varepsilon\} \cup \mathbb{R}_{\geq 0}) \times \mathcal{S}_v$ *is a transition relation with* $(s, a, s') \in \Rightarrow_v$*, denoted* $s \xrightarrow{a} s'$ *s.t. there are two types of transitions:*
   (a) *A discrete (instantaneous) transition:* $(\ell, v) \xrightarrow{\varepsilon} (\ell', v')$ *if an edge* $\ell \xrightarrow{g,R} \ell'$ *exists,* $v \models g$ *and* $v' = v[R]$*, and* $v' \models I_C(\ell')$*.*
   (b) *A delay by* δ *time units:* $(\ell, v) \xrightarrow{\delta} (\ell, v + \delta)$ *for* $\delta \in \mathbb{R}_{\geq 0}$ *if* $v + \delta \models I_C(\ell)$*.*

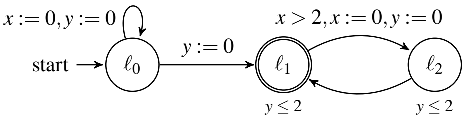


Figure 10.1: A timed Büchi automaton.

We say a state $s \in \mathcal{S}$ is accepting, or $s \in \mathcal{F}$, when $s = (\ell, \ldots)$ and $\ell \in \mathcal{F}$. We write $s \xrightarrow{\delta\ \varepsilon} s'$ if there exists a state s'' such that $s \xrightarrow{\delta} s''$ and $s'' \xrightarrow{\varepsilon} s'$. We denote an infinite run in $\mathcal{TS}_v^{\mathcal{B}} = (\mathcal{S}_v, s_0, \Rightarrow_v)$ as an infinite path $\pi = s_1 \xrightarrow{\delta_1\ \varepsilon} s_2 \xrightarrow{\delta_2\ \varepsilon} s_3 \ldots$ The run is accepting if there exist an infinite number of indices i s.t. $s_i \in \mathcal{F}$. A(n infinite) run's time lapse is $\mathit{Time}(\pi) = \sum_{i \geq 1} \delta_i$. An infinite path π in $\mathcal{TS}_v^{\mathcal{B}}$ is *time convergent*, or Zeno, if $\mathit{Time}(\pi) < \infty$, otherwise it is divergent. For example, the TBA in Figure 10.1 has an infinite run: $(\ell_0, v_0) \xrightarrow{1} (\ell_0, v_0) \xrightarrow{1} \cdots$, that is not accepting, but is non-Zeno. We claim that there is no accepting non-Zeno run, exemplified by the finite run:
$(\ell_0, v_0) \xrightarrow{2\ \varepsilon} (\ell_1, v_1) \xrightarrow{0\ \varepsilon} (\ell_2, v_0) \xrightarrow{0\ \varepsilon} (\ell_1, v_0) \overset{1.9\ \varepsilon}{\not\longrightarrow}$.

Definition 10.4 (A TBA's language and the emptiness problem)**.** *The language accepted by* $\mathcal{B}$*, denoted* $\mathcal{L}(\mathcal{B})$*, is defined as the set of non-Zeno accepting runs. The language emptiness problem for* $\mathcal{B}$ *is to check whether* $\mathcal{L}(\mathcal{B}) = \emptyset$*.*

Remark 10.1 (Zenoness)**.** *Zenoness is considered a modeling artifact as the behavior it models cannot occur in any real system, which after all has finite processing speeds. Therefore, Zeno runs should be excluded from analysis. However, any TBA* $\mathcal{B}$ *can be syntactically transformed to a strongly non-Zeno* $\mathcal{B}$*' [TYB05], s.t.* $\mathcal{L}(\mathcal{B}) = \emptyset$ *iff* $\mathcal{L}(\mathcal{B}') = \emptyset$*. Therefore, in the following, w.l.o.g., we assume that all TBAs are strongly non-Zeno.*

Definition 10.5 (Time-abstracting simulation relation)**.** *A time-abstracting simulation relation R is a binary relation on* $\mathcal{S}_v$ *s.t. if* $s_1 R s_2$ *then:*

- *If* $s_1 \xrightarrow{\varepsilon} s_1'$*, then there exists* s_2' *s.t.* $s_2 \xrightarrow{\varepsilon} s_2'$ *and* $s_1' R s_2'$*.*
- *If* $s_1 \xrightarrow{\delta} s_1'$*, then there exists* s_2' *and* δ' *s.t.* $s_2 \xrightarrow{\delta'} s_2'$ *and* $s_1' R s_2'$*.*

If both R and R^{-1} *are time-abstracting simulation relations, we call R a* time-abstracting bisimulation relation.

10.2.2 Symbolic Abstractions using Zones

A *zone* is a symbolic representation of an infinite set of clock valuations by means of a clock constraint. These constraints are conjuncts (Definition 10.6) of simple linear inequalities on clock values, and thus describe (unbounded) convex polytopes in a $|\mathcal{C}|$-dimensional plane (e.g. Figure 10.2). Therefore, zones can be efficiently represented by Difference Bounded Matrices (DBMs) [Ben02].

Definition 10.6 (Zones)**.** *Similar to the guard definition, let $\mathcal{Z}(\mathcal{C})$ be the set of clock constraints over the set of clocks $c, c_1, c_2 \in \mathcal{C}$ generalized by:*

$$Z ::= c \bowtie n \mid c_1 - c_2 \bowtie n \mid Z \wedge Z \mid \mathit{true} \mid \mathit{false}$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, >, \geq\}$ is a comparison operator. We also use $=$ for equalities, short for the conjunction of $\leq$ and $\geq$.

We write $v \models Z$ if the clock valuation v is included in Z, for the set of clock valuations in a zone $[\![Z]\!] = \{v \mid v \models Z\}$, and for *zone inclusion* $Z \subseteq Z'$ iff $[\![Z]\!] \subseteq [\![Z']\!]$. Notice that $[\![\mathit{false}]\!] = \emptyset$. Using the fundamental operations below, which are detailed in [Ben02], we define the *zone semantics* over *simulation graphs* in Definition 10.7. Most importantly, these operations are implementable in $O(n^3)$ or $O(n^2)$ and closed w.r.t. $\mathcal{Z}$.
clock delay: $[\![Z\uparrow]\!] = \{v + \delta \mid \delta \in \mathbb{R}_{\geq 0}, v \in [\![Z]\!]\}$,
clock reset: $[\![Z[R]]\!] = \{v[R] | v \in [\![Z]\!]\}$, and
constraining: $[\![Z \wedge Z']\!] = [\![Z]\!] \cap [\![Z']\!]$.

Definition 10.7 (Zone semantics)**.** *The semantics of a TBA $\mathcal{B} = (L, \mathcal{C}, \mathcal{F}, \ell_0, \rightarrow, I_{\mathcal{C}})$ under the zone abstraction is a simulation graph: $SG(\mathcal{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \Rightarrow_{\mathcal{Z}})$ s.t.:*

1. *$\mathcal{S}_{\mathcal{Z}}$ consists of pairs (ℓ, Z) where $\ell \in L$, and $Z \in \mathcal{Z}$ is a zone.*

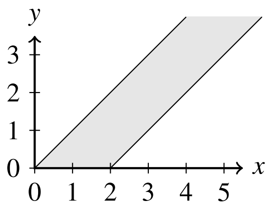


Figure 10.2: A graphical representation of a zone over 2 clocks: $0 \leq x - y \leq 2$.

2. *$s_0 \in \mathcal{S}_\mathcal{Z}$ is an initial state $(\ell_0, Z_0 \uparrow \wedge I_\mathcal{C}(\ell_0))$ with $Z_0 = \bigwedge_{c \in \mathcal{C}} c = 0$.*
3. *$\Rightarrow_\mathcal{Z}$ is the symbolic transition function using zones, s.t. $(s, s') \in \Rightarrow_\mathcal{Z}$, denoted $s \Rightarrow s'$ with $s = (\ell, Z)$ and $s' = (\ell', Z')$, if an edge $\ell \xrightarrow{g,R} \ell'$ exists, and $Z \wedge g \neq false$, $Z' = (((Z \wedge g)[R]) \uparrow) \wedge I_\mathcal{C}(\ell')$ and $Z' \neq false$.*

Any simulation graph is a discrete graph, hence cycles and lassos are defined in the standard way. We write $s \Rightarrow^+ s'$ iff there is a non-empty path in $SG(\mathcal{B})$ from s to s', or $s \Rightarrow^* s'$ if the path can be empty. An infinite run in $SG(\mathcal{B})$ is an infinite sequence of states $\pi = s_1 s_2 \ldots$, s.t. $s_i \Rightarrow s_{i+1}$ for all $i \geq 1$. It is accepting if it contains infinitely many accepting states. If $SG(\mathcal{B})$ is finite, any infinite path from s_0 defines a lasso: $s_0 \Rightarrow^* s \Rightarrow^+ s$.

Definition 10.8 (A TBA's language under Zone Semantics)**.** *The language accepted by a TBA $\mathcal{B}$ under the zone semantics, denoted $\mathcal{L}(SG(\mathcal{B}))$, is the set of infinite runs $\pi = s_0 s_1 s_2 \ldots$ s.t. there exists an infinite set of indices s.t. $s_i \in \mathcal{F}$.*

Because there are infinitely many zones, the state space of $SG(\mathcal{B})$ may also be infinite. To bound the number of zones, *extrapolation* methods combine all zones which a given TBA $\mathcal{B}$ cannot distinguish. For example, k-extrapolation finds the largest upper bound k in the guards and invariants of $\mathcal{B}$, and extrapolates all bounds in the zones $\mathcal{Z}$ that exceed this value, while LU-extrapolation uses both the maximal lower bound l and the maximal upper bound u [Beh+06]. Extrapolation can be refined on a per-clock basis [Beh+06], and on a per-location basis.

Definition 10.9. *An abstraction over a simulation graph $SG(\mathcal{B}) = (\mathcal{S}_\mathcal{Z}, s_0, \Rightarrow_\mathcal{Z})$ is a mapping $\alpha : \mathcal{S}_\mathcal{Z} \to \mathcal{S}_\mathcal{Z}$ s.t. if $\alpha((\ell, Z)) = (\ell', Z')$ then $\ell = \ell'$ and $Z \subseteq Z'$.*
If the image of an abstraction α is finite, we call it a finite abstraction.

Definition 10.10. *Abstraction α over zone transition system $SG(\mathcal{B}) = (\mathcal{S}_\mathcal{Z}, s_0, \Rightarrow_\mathcal{Z})$ induces a zone transition system $SG_\alpha(\mathcal{B}) = (\mathcal{S}_\alpha, \alpha(s_0), \Rightarrow_\alpha)$ where:*

- *$\mathcal{S}_\alpha = \{\alpha(s) \mid s \in \mathcal{S}_\mathcal{Z}\}$ is the set of states, s.t. $\mathcal{S}_\alpha \subseteq \mathcal{S}_\mathcal{Z}$,*
- *$\alpha(s_0)$ is the initial state, and*
- *$(s, s') \in \Rightarrow_\alpha$ iff $(s, s'') \in \Rightarrow_\mathcal{Z}$ and $s' = \alpha(s'')$, is the transition relation.*

We call an abstraction α an *extrapolation* if there exists a simulation relation R s.t. if $\alpha((\ell, Z)) = (\ell, Z')$ then for all $v' \in Z'$ there exist a $v \in Z$ s.t. $v' R v$ [Li09]. This means extrapolations do not introduce behavior that the un-extrapolated system cannot simulate. The abstraction defined by k-extrapolation is denoted by α_k, while the abstraction defined by LU-extrapolation is called α_{lu}. Hence, α_k and α_{lu} induce finite simulation graphs, written $SG_k(\mathcal{B})$ and $SG_{lu}(\mathcal{B})$.

10.2.3 Subsumption Abstraction

While $SG_k(\mathcal{B})$ and $SG_{lu}(\mathcal{B})$ are finite, their size is still exponential in the number of clocks. Therefore, we turn to the coarser inclusion/ subsumption abstraction of [DT98], hereafter denoted *subsumption abstraction*. We extend the notion of subsumption to states: a state $s = (\ell, Z) \in \mathcal{S}_{\mathcal{Z}}$ is *subsumed* by another $s' = (\ell', Z')$, denoted $s \sqsubseteq s'$, when $\ell = \ell'$ and $Z \subseteq Z'$. Let $\mathcal{R}(SG(\mathcal{B})) = \{s \mid s_0 \Rightarrow^* s\}$ denote the set of *reachable states* in $SG(\mathcal{B})$.

Proposition 10.1 ($\sqsubseteq$ is a simulation relation)**.** *If* $(\ell, Z_1) \sqsubseteq (\ell, Z_2)$ *and* $(\ell, Z_1) \Rightarrow (\ell', Z_1')$ *then there exists* Z_2' *s.t.* $(\ell, Z_2) \Rightarrow (\ell', Z_2')$ *and* $(\ell', Z_1') \sqsubseteq (\ell', Z_2')$.

Proof. By the definition of $\sqsubseteq$, and the fact that $\Rightarrow$ is monotone w.r.t $\subseteq$ of zones. □

Definition 10.11 (Subsumption abstraction [DT98])**.** *A subsumption abstraction* $\alpha_{\sqsubseteq}$ *over a zone transition system* $SG(\mathcal{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \Rightarrow_{\mathcal{Z}})$ *is a total function* $\alpha_{\sqsubseteq} : \mathcal{R}(SG(\mathcal{B})) \to \mathcal{R}(SG(\mathcal{B}))$ *s.t.* $s \sqsubseteq \alpha_{\sqsubseteq}(s)$

Note the subsumption abstraction is defined only over the reachable state space, and is *not* an extrapolation, because it might introduce extra transitions that the unabstracted system cannot simulate. Typically α is constructed on-the-fly during analysis, only abstracting to states that are already found to be reachable. This makes its performance depend heavily on the search order, as finding "large" states quickly can make the abstraction coarser [Dal+12].

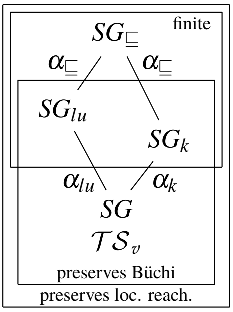


Figure 10.3: Abstractions.

10.2.4 Property Preservation under Abstractions

We now consider the preservation by the abstractions above of the property of *location reachability* (a location ℓ is reachable iff $s_0 \Rightarrow^* (\ell, \ldots)$) and that of Büchi emptiness.

Proposition 10.2. *For any of the abstractions α: α_k [DT98], α_{lu} [Beh+06], $\alpha_k \circ \alpha_{\sqsubseteq}$ [DT98], and $\alpha_{lu} \circ \alpha_{\sqsubseteq}$ [Beh+06], it holds that:*
ℓ is reachable in $\mathcal{TS}_v^{\mathcal{B}} \iff \ell$ is reachable in $SG_\alpha(\mathcal{B})$

Proposition 10.3. *For any finite extrapolation [Li09] α, e.g. the abstractions α_k [Tri09] and α_{lu} [Li09] it holds that:*
$\mathcal{L}(\mathcal{B}) = \emptyset \iff \mathcal{L}(SG_\alpha(\mathcal{B})) = \emptyset$

From hereon we will denote any finite extrapolation as α_{fin}, and the associated simulation graph $SG_{\mathit{fin}}(\mathcal{B})$. To denote that this graph can be generated *on-the-fly* [VW86; BK08; DT98], we use a NEXT-STATE(s) function which returns the set of successor states for s: $\{s' \in \mathcal{S}_{\mathit{fin}} \mid s \Rightarrow s'\}$.

As a result of Proposition 10.3 we can focus on finding accepting runs in $SG_{\mathit{fin}}(\mathcal{B})$. Because it is finite, any such run is represented by a lasso: $s_0 \Rightarrow s \Rightarrow^+ s$. Tripakis [Tri09] poses the question of whether $\alpha_{\sqsubseteq}$ can be used to check Büchi emptiness. We will investigate this further in the next section.

10.3 Preservation of Büchi Emptiness under Subsumption

The current section, investigates what properties are preserved by a subsumption abstraction $\alpha_{\sqsubseteq}$, when applied on a finite simulation graph obtained by an extrapolation, α_{fin}, in the following, denoted as $SG_{\sqsubseteq}(\mathcal{B}) = (SG_{\mathit{fin} \circ \sqsubseteq}(\mathcal{B}))$.

Proposition 10.4. *For all abstractions α, $s \in \mathcal{F} \Leftrightarrow \alpha(s) \in \mathcal{F}$ (by Definition 10.9).*

Proposition 10.5. *An $\alpha_{\sqsubseteq}$ abstraction is safe w.r.t. Büchi emptiness:*

$$\mathcal{L}(\mathcal{B}) \neq \emptyset \implies \mathcal{L}(SG_{\sqsubseteq}(\mathcal{B})) \neq \emptyset$$

Proof. If $\mathcal{L}(\mathcal{B}) \neq \emptyset$, there must be an infinite accepting path π. This path is inscribed [Tri09] in $SG_{\mathit{fin}}(\mathcal{B})$, and because $\sqsubseteq$ is a simulation relation a similar path exists in $SG_{\sqsubseteq}(\mathcal{B})$. □

Proposition 10.5 shows that subsumption abstraction preserves Büchi emptiness in one direction. Unfortunately, an accepting cycle in $SG_{\sqsubseteq}(\mathcal{B})$ is not always reflected in

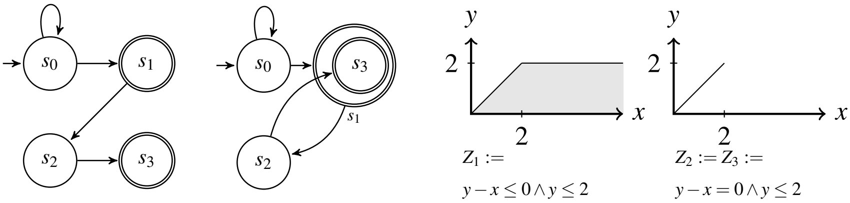


Figure 10.4: The state space $SG_{\sqsubseteq}(\mathcal{B})$ of the model in Figure 10.1 with $\ell_1 \in \mathcal{F}$ contains 4 states (shown on the left): $s_0, s_1 = (\ell_1, Z_1), s_2 = (\ell_2, Z_2)$ and $s_3 = (\ell_1, Z_3)$. The graphical representation of the zones Z_1–Z_3 (right) reveals that $Z_3 \subseteq Z_1$ and hence $s_3 \sqsubseteq s_1$. As $s_3 \sqsubseteq s_1$ and both are reachable, a subsumption abstraction is allowed to map $\alpha_{\sqsubseteq}(s_3) = s_1$, introducing a cycle $s_1 \Rightarrow s_2 \Rightarrow s_1$ in $SG_{\sqsubseteq}(\mathcal{B})$.

$SG_{\mathit{fin}}(\mathcal{B})$, as Figure 10.4 illustrates. The figure visualizes $SG_{\sqsubseteq}(\mathcal{B})$ by drawing subsumed states inside subsuming states (e.g. $s_3 \sqsubseteq s_1$).

However, subsumption introduces strong properties on paths and cycles to which we devote the rest of the current section. In subsequent sections, we exploit these properties to improve algorithms that implement the TBA emptiness check.

Lemma 10.1 (Accepting cycles under $\sqsubseteq$)**.** *If $SG_{\mathit{fin}}(\mathcal{B})$ contains states s, s' s.t. s leads to an accepting cycle and $s \sqsubseteq s'$, then s' leads to an accepting cycle.*

Proof. We have that $s \Rightarrow^* t \Rightarrow^+ t$, and because $\sqsubseteq$ is a simulation relation we have the existence of a state x s.t. $t \sqsubseteq x$:

$$\begin{tikzcd}
s' \arrow[r, Rightarrow, "*"] & t' \arrow[r, Rightarrow] & \cdots \arrow[r, Rightarrow] & x \\
s \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow, "*"] & t \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow] & \cdots \arrow[u, phantom, sloped, "\sqsubseteq"] \arrow[r, Rightarrow] & t \arrow[u, phantom, sloped, "\sqsubseteq"]
\end{tikzcd}$$

From x, we again have a similar path, to some x'. This sequence will eventually repeat some x'', because $SG_{\mathit{fin}}(\mathcal{B})$ is finite. It follows that all states in $x'' \Rightarrow^+ x''$ subsume states in $t \Rightarrow^+ t$, hence the former cycle is also accepting (Proposition 10.4). □

Lemma 10.2 (Paths under $\sqsubseteq$)**.** *If $SG_{\mathit{fin}}(\mathcal{B})$ contains a path $s \Rightarrow^+ s'$ containing an accepting state and $s \sqsubseteq s'$, then s leads to an accepting cycle.*

Proof. Because $\sqsubseteq$ is a simulation relation we have that $s \Rightarrow^+ s'$ and $s \sqsubseteq s'$ implies the existence of some t such that $s' \Rightarrow^+ t$ and $s' \sqsubseteq t$. From t, we again obtain a similar path to some t', s.t. $t \sqsubseteq t'$. Because $SG_{\mathit{fin}}(\mathcal{B})$ is finite, the sequence of t's will eventually repeat some element x, s.t. $x \Rightarrow^+ \cdots \Rightarrow^+ x$.

$$\begin{array}{ccccccccccccc}
s' & \Rightarrow^+ & t & \Rightarrow^+ & t' & \Rightarrow^+ & \cdots & \Rightarrow^+ & t'' & \Rightarrow^+ & x \\
\rotatebox{90}{$\sqsubseteq$} & & \rotatebox{90}{$\sqsubseteq$} & & \rotatebox{90}{$\sqsubseteq$} & & \rotatebox{90}{$\sqsubseteq$} & & \rotatebox{90}{$\sqsubseteq$} & & \rotatebox{90}{$=$} \\
s & \Rightarrow^+ & s' & \Rightarrow^+ & t & \Rightarrow^+ & \cdots & \Rightarrow^+ & x & \Rightarrow^+ & x
\end{array}$$

This gives us the lasso $s \Rightarrow^* x \Rightarrow^+ x$. It also follows that all states in $x \Rightarrow^+ x$ subsume states in $s \Rightarrow^+ s'$, hence the former cycle is accepting (Proposition 10.4). □

10.4 Timed Nested Depth-First Search with Subsumption

In the current section, we extend the classic linear-time Ndfs [Cou+92; SE05] algorithm to exploit subsumption. The algorithm detects accepting cycles, the absence of which implies Büchi emptiness. It is correct for the graph $SG_{\mathit{fin}}(\mathcal{B})$ according to Proposition 10.3. In the following, with *soundness*, we mean that when Ndfs reports a cycle, indeed an accepting cycle exists in the graph, while completeness indicates that Ndfs always reports an accepting cycle if the graph contains one.

The Ndfs algorithm in Algorithm 10.1 consists of an outer DFS (*dfsBlue*) that sorts accepting states s in DFS *postorder*. And an inner DFS (*dfsRed*) that searches for cycles over each s, called the *seed*. States are maintained in 3 color sets:

1. *Blue*, states *explored* by *dfsBlue*,
2. *Cyan*, states on the stack of *dfsBlue* (*visited* but not yet explored), which are used by *dfsRed* to close cycles over s early at Line 8 [SE05], and
3. *Red*, visited by *dfsRed*.

Algorithm 10.1 Ndfs

```
1:  procedure NDFS
2:      $Cyan := Blue := Red := \emptyset$
3:      $dfsBlue(s_0)$
4:      report no cycle
5:  procedure $dfsRed(s)$
6:      $Red := Red \cup \{s\}$
7:      for all $t$ in NEXT-STATE$(s)$ do
8:          if $t \in Cyan$ then report cycle
9:          if $t \notin Red$ then $dfsRed(t)$
10: procedure $dfsBlue(s)$
11:     $Cyan := Cyan \cup \{s\}$
12:     for all $t$ in NEXT-STATE$(s)$ do
13:         if $t \notin Blue \wedge t \notin Cyan$ then
14:             $dfsBlue(t)$
15:     if $s \in \mathcal{F}$ then
16:         $dfsRed(s)$
17:     $Blue := Blue \cup \{s\}$
18:     $Cyan := Cyan \setminus \{s\}$
```

Algorithm 10.1 maintains a few strong invariants, which are already mentioned in [Cou+92; SE05], but for which we include a formal proof in Section B.1:

I0: At Line 13 all red states are blue. (see Corollary B.3).

I1: The only accepting state visited by *dfsRed* is the seed. (see Corollary B.4).

I2: Outside of *dfsRed*, accepting cycles are not reachable from red states. (see Corollary B.5).

I3: A sufficient post-condition for $dfsRed(s)$ is that all reachable states from s are included in *Red* and no cyan state is reachable from it. (see Corollary B.6).

We now try to employ subsumption on the different colors to prune the searches, even though we cannot use it on all colors as $SG_{\sqsubseteq}(\mathcal{B})$ introduces additional cycles as Figure 10.4 showed. To express subsumption checks on sets we write $s \sqsubseteq S$, meaning $\exists s' \in S\colon s \sqsubseteq s'$. And $S \sqsubseteq s$, meaning $\exists s' \in S\colon s' \sqsubseteq s$. At several places in Algorithm 10.1 we might apply subsumption, leading to the following options:

1. On cyan for cycle detection:
   (a) $t \sqsubseteq Cyan$ at Line 8, or
   (b) $Cyan \sqsubseteq t$ at Line 8.
2. On *dfsBlue*, by replacing $t \notin Blue \wedge t \notin Cyan$ at Line 13 with $t \not\sqsubseteq Blue \cup Cyan$.
3. On the blue set (explored states), by replacing $t \notin Blue$ at Line 13 with $t \not\sqsubseteq Blue$.
4. On *dfsRed*, by replacing $t \notin Red$ at Line 9 with $t \not\sqsubseteq Red$.

Subsumption on cyan for cycle detection as in Item 1a makes the algorithm unsound: cycles in $SG_{\sqsubseteq}(\mathcal{B})$ are not always reflected in $SG_{fin}(\mathcal{B})$ (Figure 10.4). There is also no

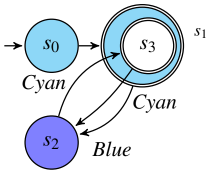
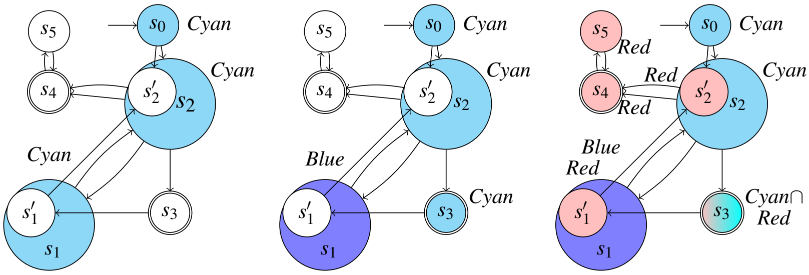


Figure 10.5: Counter example to subsumption on *Blue* and *Cyan* (Item 2).



(a) $dfsBlue(s_1)$ (b) $dfsBlue(s_3)$ (c) $dfsRed$ from s_3

Figure 10.6: Counter example to subsumption on *Blue*

hope of "unwinding" the algorithm upon detecting an accepting cycle that does not exist in the underlying $SG_{fin}(\mathcal{B})$ without losing its linear-time complexity, as the number of cycles can be exponential in the size of $SG_{\sqsubseteq}(\mathcal{B})$.

If, on the other hand, we prune the blue search as in Item 2, the algorithm becomes incomplete. Figure 10.5 shows a run of the modified NDFS on an $SG_{fin}(\mathcal{B})$ with cycle $s_3 \Rightarrow s_2 \Rightarrow s_3$. The *dfsBlue* backtracked over s_2 as $s_3 \sqsubseteq s_1$ and $s_1 \in Cyan$. The *dfsRed* now launched from s_1, will however continue to visit s_3, while missing the cycle as s_2 is not cyan. We also observe that I1 is violated, indicating that the postorder on accepting states (s_3 before s_1) is lost.

It is tempting therefore to use subsumption on blue only, as in Item 3. However, Figure 10.6 shows an "animation" of a run with the modified NDFS which is incomplete. Here state s_1 is first backtracked in the blue search as all successors are cyan (left). Then state s_1 is marked blue; The blue search backtracks to s_2, proceeds to s_3 and backtracks because it finds $s_1' \sqsubseteq s_1 \in Blue$ (middle). Then a red search is started from s_3, which subsumes the cyan stack (s_2) and visits accepting state s_4, violating I1 and missing the accepting cycle $s_4 \Rightarrow s_5 \Rightarrow s_4$.

A viable option however is to use inverse subsumption on cyan as in Item 1b. According to Lemma 10.1, a state that subsumes a state on the cyan stack leads to a cycle. And as the only goal of the red search is to find a cyan state (to close an accepting cycle over the seed), it does not rely on DFS (I3). Thus we may as well use subsumption in the red search as in Item 4. By definition (Definition 10.11), $SG_{\sqsubseteq}(\mathcal{B})$ contains a "larger" state for all reachable states in $SG_{fin}(\mathcal{B})$. So in combination with Item 1b this is sufficient to find all accepting cycles.

The strong invariant (I2) states accepting cycles are not reachable from red states, so

Algorithm 10.2 NDFS with subsumption on red, cycle detection, and red in *dfsBlue*.

```
1:  procedure NDFS()
2:      $Cyan := Blue := Red := \emptyset$
3:      $dfsBlue(s_0)$
4:      report no cycle
5:  procedure $dfsRed(s)$
6:      $Red := Red \cup \{s\}$
7:      for all $t$ in NEXT-STATE$(s)$ do
8:          if $\underline{Cyan \sqsubseteq t}$ then report cycle
9:          if $\underline{t \not\sqsubseteq Red}$ then $dfsRed(t)$
10: procedure $dfsBlue(s)$
11:     $Cyan := Cyan \cup \{s\}$
12:     for all $t$ in NEXT-STATE$(s)$ do
13:         if $(t \notin Blue \cup Cyan \underline{\wedge t \not\sqsubseteq Red})$
14:         then $dfsBlue(t)$
15:     if $s \in \mathcal{F}$ then
16:         $dfsRed(s)$
17:     $Blue := Blue \cup \{s\}$
18:     $Cyan := Cyan \setminus \{s\}$
```

red states can prune the blue search. We can strengthen the condition on Line 13 to $t \notin Blue \cup Cyan \cup Red$. However, this is of no use since by (I0), $Red \subseteq Blue$. Luckily, even states subsumed by red do not lead to accepting cycles (contraposition of Lemma 10.1), so we can use subsumption again: $t \notin Blue \cup Cyan \wedge t \not\sqsubseteq Red$. The benefit of this can be illustrated using Figure 10.4. Once *dfsBlue* backtracks over s_1, we have $s_1, s_2, s_3 \in Red$ by *dfsRed* at Line 16. Any hypothetical other path from s_0 to a state subsumed by these red states can be ignored.

Algorithm 10.2 shows a version of NDFS with all correct improvements. Notice that I2 and I3 are sufficient to conclude correctness of these modifications.

10.5 Multi-Core cndfs with Subsumption

CNDFS, from Chapter 7, is a parallel algorithm for checking Büchi emptiness. By Proposition 10.3, it is correct for SG_{fin}. In the current section, we extend CNDFS with subsumption, in a similar way as we have done for the sequential NDFS in the previous section.

In CNDFS (Algorithm 10.3 without underlined parts), each worker thread i runs a seemingly independent $dfsBlue_i$ and $dfsRed_i$, with a local stack color $Cyan_i$, and its own random successor ordering (indicated by the subscript i of the NEXT-STATE function). However, the workers assist each other by sharing the colors *Blue* and *Red* globally, thus pruning each other's search space.

The main problem that CNDFS has to solve is the loss of postorder on the accepting states due to the shared blue color (similar to the effects of Item 3 as illustrated in Figure 10.6). In the previous section, we have seen that a loss of postorder may cause *dfsRed* to visit non-seed accepting states, i.e. I1 is violated. CNDFS demonstrates that repairing such a *dangerous situation* is sufficient to preserve correctness (see Chapter 7).

To detect a dangerous situation, Cndfs collects states visited by $dfsRed_i$ in a set $\mathcal{R}_i$ at Line 7. After completion of $dfsRed_i$, the algorithm checks $\mathcal{R}_i$ for non-seed accepting states at Line 22. By waiting for these states to become red, the dangerous situation is resolved as the blue state that caused the situation was always placed by some other worker, which will eventually continue as shown by Proposition 7.3. Once the situation is detected to be resolved, all states from the local $\mathcal{R}_i$ are added to *Red* at Line 23.

Cndfs maintains similar invariants as Ndfs (proved in Chapter 7):

I2' Red states do not lead to accepting cycles (see Lemma 7.1 and Proposition 7.1).

I3' After $dfsRed_i(s)$ states reachable from s are red or in $\mathcal{R}_i$ (see Lemma 7.2).

Because these invariants are as strong or stronger than I2 and I3, we can use subsumption in a similar way as for Ndfs. Algorithm 10.3 underlines the changes to algorithm w.r.t. Algorithm 7.2 in Chapter 7. We additionally have to extend the waiting procedure to include subsumption at Line 22, because the use of subsumption in $dfsRed_i$ can cause other workers to find "larger" states.

In the next section, we benchmark Algorithm 10.3 on timed models. The algorithm inherits from Cndfs the property that its *runtime* is linear in the size of the input graph N. However, in the worst case, all workers may visit the same states. Therefore, the complexity of the amount of *work* that the algorithm performs (or the amount of power it consumes) equals $N \times P$, where P is the number of processors used. The randomized successor function NEXT-STATE$_i$ however ensures that this does not happen for most practical inputs. Experiments on over 300 examples confirmed this (see Section 7.4), making Cndfs the current state-of-the-art parallel LTL model checking algorithm.

Algorithm 10.3 Cndfs with subsumption

```
1:  procedure CNDFS($P$)
2:      $Blue := Red := \emptyset$
3:      forall $i$ in $1..P$ do $Cyan_i := \emptyset$
4:      $dfsBlue_1(s_0) \| .. \| dfsBlue_P(s_0)$
5:      report no cycle
6:  procedure $dfsRed_i(s)$
7:      $\mathcal{R}_i := \mathcal{R}_i \cup \{s\}$
8:      for all $t$ in NEXT-STATE$_i(s)$ do
9:          if $\underline{Cyan \sqsubseteq t}$ then
10:             report cycle
11:         if $t \notin \mathcal{R}_i \wedge \underline{t \not\sqsubseteq Red}$ then
12:             $dfsRed_i(t)$
13: procedure $dfsBlue_i(s)$
14:     $Cyan_i := Cyan_i \cup \{s\}$
15:     for all $t$ in NEXT-STATE$_i(s)$ do
16:         if $t \notin Cyan_i \cup Blue \underline{\wedge t \not\sqsubseteq Red}$ then
17:             $dfsBlue_i(t)$
18:     $Blue := Blue \cup \{s\}$
19:     if $s \in \mathcal{F}$ then
20:         $\mathcal{R}_i := \emptyset$
21:         $dfsRed_i(s)$
22:         await $\forall s' \in \mathcal{R}_i \cap \mathcal{F} \setminus \{s\} : \underline{s' \sqsubseteq Red}$
23:         forall $s'$ in $\mathcal{R}_i$ do $Red := Red \cup s'$
24:     $Cyan_i := Cyan_i \setminus \{s\}$
```

10.6 Experimental Evaluation

To evaluate the performance of the proposed algorithms experimentally, we implemented CNDFS without (as in Chapter 7) and with subsumption (Algorithm 10.3) in LTSMIN 2.0[10.1]. The OPAAL [Dal+11] tool[10.2] functions as a frontend for UPPAAL models. Previously, we demonstrated scalable multi-core reachability for timed automata (see Chapter 9).

10.6.1 Experimental Setup

We benchmarked[10.3] on a 48-core machine (a four-way AMD Opteron™ 6168) with a varying number of threads, averaging results over 5 repetitions. We consider the following models and LTL properties:

`csma`[10.4] is a protocol for Carrier Sense, Multiple-Access with Collision Detection with 10 nodes. We verify the property that on collisions, eventually the bus will be active again: $\Box(\texttt{(P0=bus\_collision1)} \implies \Diamond\texttt{(P0=bus\_active)})$.

`fischer-1/2`[10.5] implements a mutual exclusion protocol with 10 nodes; a canonical benchmark for timed automata. As in [Li09], we use the property (1): $\neg((\Box\Diamond\texttt{k=1})\vee(\Box\Diamond\texttt{k=0}))$, where `k` is the number of processes in their critical section. We also add a weak fairness property (2): $\Box((\Box\texttt{P\_1=req}) \implies (\Diamond\texttt{P\_1=cs}))$: processes requesting infinitely often will eventually be served.

`fddi`[10.4] models a token ring system as described in [BTY97], where a network of 10 stations are organised in a ring and can hand back the token in a synchronous or asynchronous fashion. We verify the property from [BTY97] that every station will eventually send asynchronous messages: $\Box(\Diamond\texttt{(ST1=station\_z\_sync)})$.

`train-gate`[10.4] models a railway interlocking, with 10 trains. Trains drive onto the interconnect until detected by sensors. There they wait until receiving a signal for safe crossing. The property prescribes that each approaching train eventually should be serviced: $\Box(\texttt{Train\_1=Appr} \implies (\Diamond\texttt{Train\_1=Cross}))$.

The following command-line was used to start the LTSMIN tool:

`opaal2lts-mc --strategy=[A] --ltl-semantics=textbook --ltl=[f] -s28 --threads=[P] -u[0,1] [m].`

This runs algorithm A on the cross product of the model m with the Büchi automaton of formula f. It uses a fixed hash table of size 2^{28} and P threads, and either subsumption (-u1) or not (-u0). The option ltl-semantics selects textbook LTL semantics as defined

[10.1] Available as open source at: http://fmt.cs.utwente.nl/tools/ltsmin

[10.2] Available as open source at: http://opaal-modelchecker.com

[10.3] All results are available at: http://fmt.cs.utwente.nl/tools/ltsmin/cav-2013

[10.4] From http://www.it.uu.se/research/group/darts/uppaal/benchmarks/

[10.5] As distributed with UPPAAL.

in [BK08, Ch. 4]. To investigate the overhead of CNDFS, we also run the multi-core algorithms for plain reachability on this cross product, even though this does not make sense from a model checking perspective. To compare effects of the search order on subsumption, we use both DFS and BFS.

Note finally, that we are only interested here in full verification, i.e. in LTL properties that are correct w.r.t the system under verification. This is the hardest case as the algorithm has to explore the full simulation graph. To test their on-the-fly nature, we also tried a few incorrect LTL formula for the above models, to which the algorithms all delivered counterexamples within a second. But with parallelism this happens almost instantly (see Section 7.4.4).

10.6.2 Implementation

LTSMIN defines a NEXT-STATE function as part of its PINS interface for language-independent symbolic/parallel model checking [BPW10]. In Chapter 9, we extended PINS with subsumption. OPAAL is used to parse the UPPAAL models and generate C code that implements PINS. The generated code uses the UPPAAL DBM library to implement the simulation graph semantics under *LU-extrapolated zones*. The LTL cross product [BK08] is calculated by LTSMIN.

LTSMIN's multi-core tool [LPW11a] stores states in one lockless hash/tree table in shared memory (see Part II). For timed systems, this table is used to store *explicit state parts*, i.e. the locations and state variables [BDL04]. The DBMs representing zones, here referred to as the *symbolic state parts*, are stored in a separate lockless hash table, while a lockless *multimap* structure efficiently stores full states, by linking multiple symbolic to a single explicit state part (see Chapter 9). Global color sets of CNDFS (*Blue* and *Red*) are encoded with extra bits in the multimap, while local colors are maintained in local tables to reduce contention to a minimum.

Because the proof of the original CNDFS algorithm assumes that each lines in Algorithm 10.3 is executed atomically, we to implement the $\sqsubseteq$ operation as an atomic operation. To this end, we *lock* the multimap using a fine-grained spinlock as discussed in Section 9.6.3. Because this implementation locks individual explicit state parts, it generally allows for enough parallelism, *unless* there are very few explicit state parts compared to symbolic state parts.

10.6.3 Hypothesis

CNDFS for untimed model checking scaled mostly linearly. In the timed automata setting, several parameters could change this picture. In the first place, the *computational intensity* increases, because the DBM operations use many calculations. In modern multi-core computers, this feature improves scalability, because it more closely matches the machine's high frequency/bandwidth ratio (see Chapter 2). On the other hand, the lock granularity increases since a single lock now governs multiple DBMs stored in the multimap as described in the previous section. Nonetheless, for multi-core timed reachability, previous experiments showed almost linear scalability (see Section 9.7), even when using other model checkers (UPPAAL) as a base line. On the other hand, the CNDFS algorithm requires more queries on the multimap structure to distinguish the different color sets.

Subsumption probably improves the absolute performance of CNDFS. We expect that models with many clocks and constraints exhibit a better reduction than others. Moreover, it is known [Beh05] that the reduction due to subsumption depends strongly on the exploration order: BFS typically results in better reductions than DFS, since "large" states are encountered later. CNDFS might share this disadvantage with DFS. However, as shown in Chapter 9, subsumption with random parallel DFS performs much better than sequential DFS, which could be beneficial for the scalability of CNDFS. So it is really hard to predict the relative performance and scalability of these algorithms, and the effects of subsumption.

10.6.4 Experimental Results without Subsumption

We first compare the algorithms BFS, DFS (parallel reachability) and CNDFS (accepting cycles) without subsumption. Table 10.1 shows their sequential ($P = 1$) and parallel ($P = 48$) runtimes (T). Note that sequential CNDFS is just NDFS. We show the number of explicit state parts ($|L|$), full states ($|\mathcal{R}|$), transitions ($|\Rightarrow|$), and also the number of states visited in CNDFS ($|V|$). These numbers confirm the findings reported previously for CNDFS applied to untimed systems: The sequential runtimes ($P = 1$) are very similar, indicating little overhead in CNDFS. For the parallel runs ($P = 48$), however, the number of states visited by CNDFS ($|V|$) increases due to work duplication.

To further investigate the scalability of the timed CNDFS algorithm, we plot the speedups in Figure 10.7. Vertical bars represent the (mostly negligible) standard deviation over the five benchmarks. Three benchmarks exhibit linear scalability, while train-gate and fddi show a sublinear, yet still positive, trend. For train-gate, we suspect that this is caused by the structure of the state space. Because fddi has only 119 explicit state parts, we attribute the poor scalability to lock contention, harming more with a growing number of workers.

Table 10.1: Runtimes (sec) and states counts *without* subsumption.

Model	P	$\|L\|$	$\|\mathcal{R}\|$	$\|V\|_{cndfs}$	$\|\Rightarrow\|_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	135449	438005	438005	1016428	26.1	26.2	27.8
csma	48	135449	438005	453658	1016428	1.0	0.9	0.9
fddi	1	119	179515	179515	314684	26.3	26.6	34.2
fddi	48	119	179515	566093	314684	1.6	0.7	2.7
fischer-1	1	521996	4987796	4987796	19481530	195.9	196.7	212.2
fischer-1	48	521996	4987796	5190490	19481530	4.8	4.6	5.1
fischer-2	1	358901	3345866	3345866	10426444	135.8	136.5	145.5
fischer-2	48	358901	3345866	3541373	10426444	3.4	3.3	3.7
train-gate	1	119989268	119989268	119989268	177201017	1608	1621	1724
train-gate	48	119989268	119989268	319766765	177201017	34.9	45.4	145.8

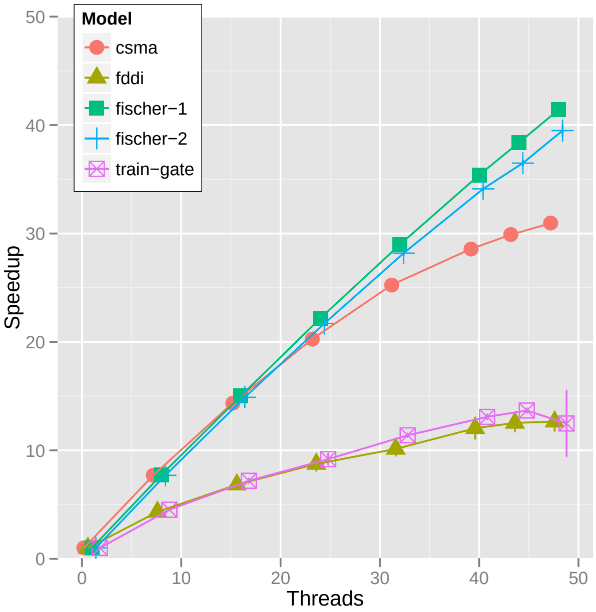


Figure 10.7: Speedups in LTSmin/opaal

10.6.5 Subsumption

Table 10.2 shows the experimental data for Bfs, dfs and Cndfs with subsumption (Algorithm 10.3). The number of explicit state parts $|L|$ is stable, since reachability of locations is preserved under subsumption (Proposition 10.2). However, the achieved reduction of full states depends on the search order, so we now report $|\mathcal{R}|$ per algorithm, as a percentage of the original numbers.

We confirm [Beh05] that subsumption works best for Bfs reachability, with even more than 30-fold reduction for fddi, but none for fischer (cf. column $|\mathcal{R}|_{bfs}$). For these benchmarks, the reduction is correlated to the ratio $X = |\mathcal{R}|/|L|$; e.g. $X \approx 1500$ for fddi and $X \approx 10$ for fischer. Subsumption is much less effective with sequential dfs, but parallel dfs improves it slightly (cf. column $|\mathcal{R}|_{dfs}$).

Cndfs benefits considerably from subsumption, but less so than Bfs: we observe around 2-fold reduction for fddi, fischer and csma (cf. column $|\mathcal{R}|_{cndfs}$). Surprisingly, the reduction for parallel runs of Cndfs is not better than for sequential runs. One disadvantage of Cndfs compared to Bfs is that only red states attribute to subsumption reduction. Probably some “large” states are never colored red. We measured that for all benchmark models, 20%–50% of all reachable states are colored red (except for fischer-2, which has no red states).

Subsumption decreases the runtimes for reachability: a lot for Bfs, and still considerably for dfs, both in the sequential case and the parallel case, up to 48 workers. However, subsumption is less beneficial for the running time of Cndfs (it might even increase), but the speedup remains unaffected.

Table 10.2: Runtimes and states counts *with* subsumption (in % relative to Table 10.1).

Model	P	$\|\mathcal{R}\|_{bfs}$	$\|\mathcal{R}\|_{dfs}$	$\|\mathcal{R}\|_{cndfs}$	$\|V\|_{cndfs}$	$\|\Rightarrow\|_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	48.7	88.9	58.3	94.7	41.2	41.3	90.3	95.2
csma	48	48.7	77.5	58.3	93.6	41.2	64.5	85.3	97.8
fddi	1	3.1	3.4	50.8	53.1	3.4	4.3	4.7	132.3
fddi	48	3.1	2.4	50.8	80.1	3.4	51.0	19.5	121.0
fischer-1	1	17.9	72.4	55.2	91.9	27.0	25.6	78.7	97.3
fischer-1	48	17.9	71.1	55.2	95.9	27.0	33.1	79.6	103.0
fischer-2	1	18.6	68.5	77.5	95.8	28.7	27.0	75.3	98.9
fischer-2	48	18.6	62.7	77.5	95.8	28.7	37.4	72.5	98.3
train-gate	1	100.0	100.0	100.0	100.0	100.0	100.6	100.6	104.3
train-gate	48	100.0	100.0	100.0	100.0	100.0	101.7	83.5	83.1

10.7 Conclusions

We implemented the first parallel model checking algorithm for liveness properties on timed systems. We also contributed to solving the open problem [Tri09] to use inclusion abstraction for liveness properties. Experimentally, we established that these techniques have their own merits: models with sufficiently many discrete states yield great speedups of up to 40 on a 48 core machine. Models with more symbolic states can benefit from abstraction, with 2-fold state-space reductions in several examples.

There is however also room for improvement: As the speedups correlate negatively with the ratio of symbolic states in the state space X, we can conclude that the atomic implementation of the $\sqsubseteq$ (see Section 10.6.2), in fact does constitute a bottleneck for the parallelism. The use of non-blocking algorithms, such as those proposed in Section 9.6.4, could further improve speedups. Finally, subsumption might be exploitable in still different ways for liveness checking, possibly by employing other algorithms such as Owcty.

Part V

Reflections

11 Additional Experimental Evaluation

11.1 Introduction

The current chapter provides stronger experimental evidence for the scalability of the multi-core solutions that we proposed in the current thesis.

The data structures and algorithms in the previous parts have all been rigorously benchmarked. Most of these benchmarks were however performed on a 16-core machine using DVE models from the Beem database [Pel07]. While this provides already good evidence for scalability, in the current chapter, we still improve on it by using real-world models written for the state-of-the-art spin model checker [Hol97a; Hol11]. Moreover, we also transition from the 16-core machine to a 48-core machine. The increase of the parallelism from 16 cores to 48 cores allows us to evaluate our earlier hypothesis that the scalability will be maintained for larger machines (Section 2.5).

To this end, we implemented a frontend for promela models for the language-independent model checker LTSmin [LPW11a; BL13a; BPW10]. This frontend is called SpinS, and partly reuses the parser and interpreter from SpinJa [JR10], which is basically a reimplementation of spin in Java. The SpinS frontend also performs some basic static analysis to provide LTSmin with the dependency information required for partial-order reduction [Laa+13a].

SpinS generates C code from the promela models to enable high-performance model checking with similar performance to spin. This needs to be mentioned, because if a virtual-machine or scripting language would have been used, the base case would experience a slowdown of a factor 4-11 [JR10], as a consequence the obtained speedups reveal little concrete about an algorithm's scalability.

The compact hash table from Chapter 4 is not benchmarked, as the states of many input models are too large to fit in the table cells. Instead, we present benchmarks with the compact tree structure from Section 4.4. These confirm that the compact tree scales well and also delivers the optimal compression of around 4 byte per state for the models considered here.

The following sections will detail experiments that compare our multi-core reachability and multi-core LTL algorithms with similar algorithms in SPIN. We first show that the performance of LTSMIN and SPINS is indeed on par with SPIN, this establishes a valid base-case for comparing the scalability. We then investigate the scalability. Last, we study the state compression obtained for PROMELA models with both the tree and the compact tree.

11.2 Experimental Setup

To compare the performance of PROMELA model checkers, we performed benchmarks with SPIN 6.2.1 [Hol12] and LTSMIN 2.0[11.1] [LPW11a; BL13a; BPW10] on a 48-core machine (a four-way AMD Opteron$^{\text{TM}}$ 6168). We also include some BEEM models [Pel07] to allow comparison with DIVINE 2.5.2 [Bar+10] (these models are written in the DVE language and have been translated to PROMELA). We show here a representative selection.[11.2]

For high performance in SPIN, we compiled models with parallel BFS [Hol12]: `-O3 -DNOBOUNDCHECK -DSAFETY -DNOREDUCE -DBFS_MAXPROCS=48 -DBFS_PAR`. By default, this enables a *lossy hash compaction* (hc) state storage, hence we also compiled using `-DNO_HC`. DIVINE is configured as described in Chapter 2. In LTSMIN, we used a hash table, a tree table and a cleary-tree (*all non-lossy*). All experiments use a fixed table size of 2^{28}. The corresponding command line is: `prom2lts-mc --threads=<N> --state=<cleary-tree/tree/table> -s28 --strategy=<strategy> <model>`. Where the strategy is either a reachability algorithm. e.g. BFS-like search (`bfs`), DFS-like search (`dfs`) or strict BFS (`sbfs`), or a liveness algorithm such as CNDFS (`cndfs`). In the latter case, the additional LTL property needs to be supplied with `--ltl=<formula>`. We report here only on the reachability experiments with BFS-like search: The other strategies result in similar performance [LPW11a]. To accommodate a master thread, SPIN and DIVINE are limited to 47 threads.

[11.1] The LTSMIN website: `http://fmt.cs.utwente.nl/tools/ltsmin`

[11.2] For complete results see `http://fmt.cs.utwente.nl/tools/ltsmin/performance`

11.3 Performance and Scalability of Reachability

Table 11.1 shows the state counts and sequential runtimes of the different tools. Unfortunately, the parallel BFS algorithm of spin generates more states than it should, since its sequential DFS algorithm generates the same amount of states as LTSmin with SpinS does (also included in the table). This is indicative of a bug in the parallel algorithm. A perfect comparison between the parallel tools is thus not always possible. Still we can draw conclusions from the differences of the sequential runtimes. The table namely shows that sequentially the runtimes of LTSmin are competitive to those of spin, which

Table 11.1: Number of states and runtimes of spin's parallel BFS (1 core) and sequential DFS vs LTSmin/SpinS (1 core). Deviating state counts are in **bold**.

	spin ParBFS (hc)		spin ParBFS (nohc)		spin		LTSmin	
	States $\|\mathcal{S}\|$	Runtime	States $\|\mathcal{S}\|$	Runtime	States $\|\mathcal{S}\|$	Runtime	States $\|\mathcal{S}\|$	Runtime
GARP1	**1.6e8**	458.0	**1.6e8**	820.0	4.8e7	377.1	4.8e7	175.8
Peterson4	**9.5e6**	17.5	**9.5e6**	27.1	1.3e7	23.6	1.3e7	22.3
I-Protocol2	**4.0e7**	77.2	**4.0e7**	179.0	1.4e7	28.4	1.4e7	30.0
Anderson.6	1.8e7	73.9	1.8e7	148.0	1.8e7	67.7	1.8e7	47.1
At.5	3.2e7	101.0	3.2e7	205.0	3.2e7	96.4	3.2e7	71.0
Bakery.7	2.8e7	6.3	2.8e7	86.4	2.9e7	55.1	2.9e7	60.0

Table 11.2: Runtimes of sequential and parallel runs in on 48 cores in spin (with and without hash compaction), DiVinE and LTSmin (table/tree/Cleary-tree). The fastest sequential and concurrent runtimes are shown in **bold**.

	spin ParBFS				DiVinE		LTSmin					
	hc		nohc				table		tree		Cleary	
	1	47	1	47	1	47	1	48	1	48	1	48
GARP1	458.0	43.4	820.0	295.0	n/a	n/a	187.9	5.3	**175.8**	**4.6**	196.9	5.1
Peterson4	**17.5**	2.6	27.1	18.3	n/a	n/a	29.6	1.2	22.3	**0.8**	26.9	0.9
I-Protocol2	77.2	30.0	179.0	249.0	n/a	n/a	43.1	1.8	**30.0**	**1.0**	31.9	1.1
Anderson.6	73.9	26.0	148.0	188.0	27.5	8.0	52.8	1.9	**47.1**	**1.5**	57.7	1.7
At.5	101.0	28.0	205.0	239.0	39.8	10.5	**66.0**	2.2	71.0	**2.0**	84.8	2.4
Bakery.7	59.8	6.3	86.4	38.4	32.2	9.0	**52.0**	1.8	60.0	**1.7**	69.4	2.0

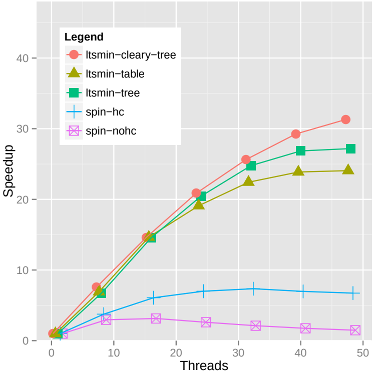


Figure 11.1: Peterson4 (PROMELA)

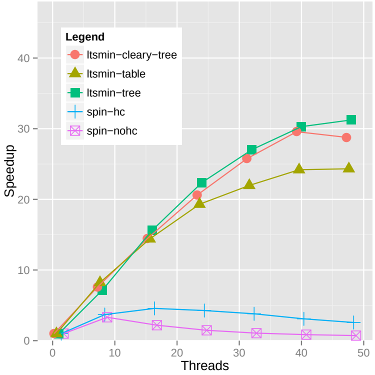


Figure 11.2: I-Protocol2 (PROMELA)

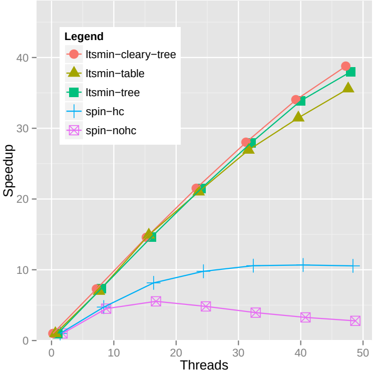


Figure 11.3: GARP1 (PROMELA)

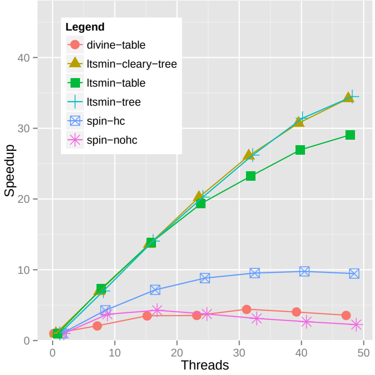


Figure 11.4: Bakery.7 (DVE+PROMELA)

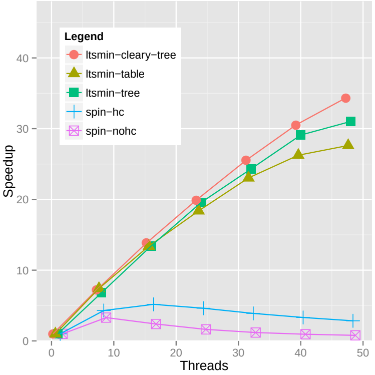


Figure 11.5: Andrsn.6 (DVE+PROMELA)

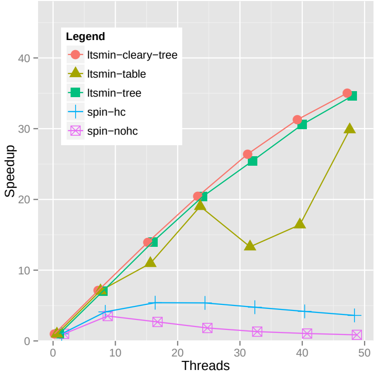


Figure 11.6: At.5 (DVE+PROMELA)

also holds for SPIN's parallel BFS algorithm for those models that still have comparable state counts.

In the following scalability experiments, we will limit our investigation to *relative scalability*, i.e. we use the sequential runtime of the *same* tool (not the fastest tool) to calculate speedups. This allows us to disregard the slight differences in the number if states. In fact, for the scalability this is even an advantage for SPIN, because more states means that more work is available for parallelization and in general results in better speedups. The absolute runtimes are also given in Table 11.2.

Figure 11.1, Figure 11.2 and Figure 11.3 show the obtained speedups with SPIN and LTSMIN for models only available in PROMELA source. Figure 11.4, Figure 11.5 and Figure 11.6 show the obtained speedups with DIVINE, SPIN and LTSMIN for DVE models that were translated to PROMELA [Pel07] (the state count remains the same for these translated versions). The speedups in LTSMIN clearly dominate in the figures. Although not entirely linear, the speedup still increases up to 48 cores. Except for Figure 11.6, where we determined that the load balancer sometimes failed to keep up, as witnessed by an uneven workload distribution. We expect that a modern, asynchronous load balancer implementation, as suggested in Chapter 9, solves this problem. The 48-core runtimes show that LTSMIN's multi-core algorithms are a good addition for PROMELA model checking. Furthermore, we can see that (Cleary-)tree compression introduces little or no overhead.

11.4 Performance and Scalability of LTL Checking

Figure 11.7 and Figure 11.8 show speedups of two models obtained with DIVINE's OWCTY algorithm, SPIN's Piggyback (PB) algorithm [Hol12] (with hash compaction) and LTSMIN's CNDFS (see Chapter 7) algorithm (with hash table). CNDFS shows the best speedups and is sequentially faster than the PB algorithm (by 60%), which comes second in terms of speedup. Three other aspects are of interest when comparing the three algorithms: CNDFS/OWCTY are exact LTL algorithms while the PB may miss counterexamples [Hol12], CNDFS is on-the-fly while the PB explores the whole state space before reporting a counterexample [Hol12] and OWCTY typically explores a large portion of it (see Section 7.4.3), and CNDFS is found to return even shorter counterexamples than a parallel BFS-based algorithm (see Section 7.4.4)! On the other hand, the BFS-based algorithms OWCTY and PB can be distributed on a cluster, as DIVINE demonstrates [Bar+10].

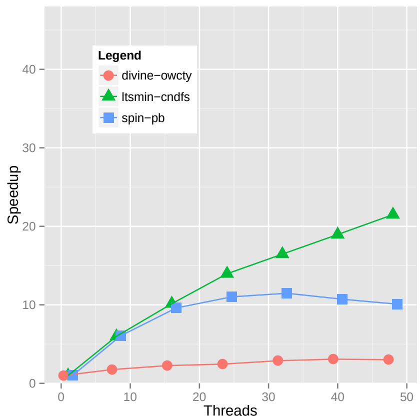


Figure 11.7: Elevator2.3 [Hol12] (DVE+PROMELA)

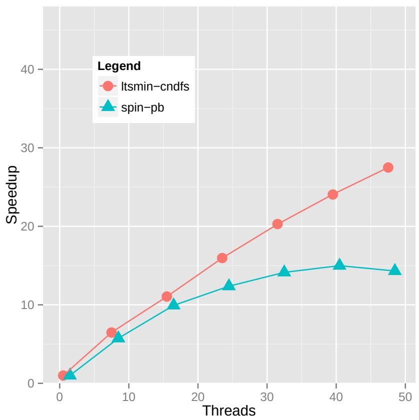


Figure 11.8: Peterson4 ($\Box\Diamond p$) (PROMELA)

11.5 Memory Usage

State compression. We measured the memory usage of DiVinE, LTSmin with and without tree compression and of spin with and without Collapse compression (col) and hash compaction. Table 11.3 shows the memory usage of all these combinations. The first thing we noticed, is that the memory usage is almost independent of the number of threads, showing that the model checkers add little overhead for parallel operation. spin's memory usage is measured by reducing the hash table size to exactly fit the state count, hence overestimated by at most 50%. We can however conclude that tree compression provides great reduction compared to full-state storage in a hash table *making lossy hash compaction redundant*. And the cleary-tree improves upon this by almost a factor of two. In [LPW11c], we compared compression methods in detail.

To further investigate the difference between the compression techniques in LTSmin, we isolated the memory usage of the hash table, the tree table and the Cleary tree. Table 11.4 shows these figures. We deduce that indeed the Cleary tree is able to almost halve the memory usage compared to the normal tree table. To compare the compression ratios better, we also calculated the average memory occupied by a singly state in the state store. These results – now with different benchmark set – comply with the observations made in Chapter 3 and Chapter 4: The tree table has an optimal compression of 8 bytes per state and a median compression of 9.6 bytes per state, and the Cleary tree consistently uses 4 bytes less (see Lemma 4.2). Indeed, the experiments show that this optimum is easily reached in practice, even for a real-world PROMELA model like the GARP protocol.

These are important differences, as the storage of visited states cannot be circumvented as explained in Section 1.4.3. Therefore, the memory occupied by the visited states determines the maximum size of the system that we can model check. (It is true that the size of these state stores can be further reduced by the smart use of caches, as explained in Section 1.4.3, but this can be considered an orthogonal approach to reduce memory consumption). For completeness, we also include the maximum size of the BFS queue. In these case, the queues use a relatively insignificant amount of memory,

Table 11.3: Total memory usage (MB) in SPIN, DiVinE and LTSmin is almost independent of number of threads. The lowest values for sequential and concurrent runs are shown in **bold**.

	SPIN					DiVinE		LTSmin					
	hc		nohc		col			table		tree		Cleary	
	1	47	1	47	1	1	47	1	48	1	48	1	48
GARP1	1.5e4	1.6e4	1.4e5	1.4e5	4.9e4	n/a	n/a	8.7e3	8.8e3	1.1e3	1.3e3	**9.0e2**	**1.1e3**
Peterson4	5.7e3	6.2e3	4.4e4	2.5e4	5.5e3	n/a	n/a	1.3e3	1.3e3	1.5e2	1.6e2	**1.0e2**	**1.0e2**
I-Protocol2	1.2e4	1.2e4	1.3e5	1.3e5	4.8e4	n/a	n/a	2.2e3	2.2e3	1.9e2	2.5e2	**1.4e2**	**1.9e2**
Anderson.6	1.1e4	1.1e4	1.3e5	1.3e5	5.4e4	4.5e3	4.6e3	2.1e3	2.1e3	3.2e2	4.6e2	**2.5e2**	**3.7e2**
At.5	1.2e4	1.2e4	1.3e5	1.3e5	5.4e4	4.6e3	4.9e3	3.1e3	3.1e3	7.3e2	8.0e2	**6.1e2**	**6.6e2**
Bakery.7	1.3e4	1.5e4	1.7e4	1.7e4	6.4e3	4.8e3	4.9e3	2.8e3	2.9e3	4.0e2	4.2e2	**2.5e2**	**2.8e2**

Table 11.4: Memory usage (MB) of state storage (hash table, tree, or Cleary tree) and BFS queues in LTSmin. The lowest values for the state storage are shown in bold.

	Memory	Memory state storage			Bytes per state		
	queues	table	tree	Cleary	table	tree	Cleary
GARP1	14.9	8363	373	**179**	182.0	8.0	**4.0**
Peterson4	0.9	1321	97	**51**	106.0	8.3	**4.3**
I-Protocol2	1.9	2176	109	**56**	158.0	8.1	**4.1**
Anderson.6	7.6	2030	139	**74**	114.0	8.1	**4.1**
At.5	18.2	2912	245	**124**	94.0	8.0	**4.0**
Bakery.7	1.9	2789	243	**138**	102.0	8.8	**4.8**

because we store references in the queue (see Section 3.3.3). Except for some small pieces of static data, the model checker does not use more memory.

These results translate to the setting of LTL checking, as the multi-core Ndfs algorithms presented in part Part III only require a few additional bits per state to record its color. And although local colors require a bit per state *per thread* in the worst case, they are only used for coloring states on the search stack or visited states in the nested search, which we found in practice to be very few (see Section 7.4.2).

Partial-order reduction. LTSmin implements a version of Valmari's [Val91b] *stubborn set* partial-order reduction algorithm. This algorithm was suited for the language-independent interface of LTSmin because it lacks a notion of processes and depends solely on a notion of structural transitions and their guards. On the other hand, the process-oriented ample-set method [KP88b] that is implemented in spin [HP94] is often faster because transitions are far more numerous than processes. To remedy this, we

Table 11.5: POR performance in LTSmin and spin

	No Partial-Order Reduction				Guard-based POR				Ample-set POR			
			LTSmin	spin	LTSmin				spin			
Model	States $\|\mathcal{S}\|$	Trans $\|\mathcal{T}\|$	Time	Time	$\Delta\|\mathcal{S}\|$	$\Delta\|\mathcal{T}\|$	Mem.	Time	$\Delta\|\mathcal{S}\|$	$\Delta\|\mathcal{T}\|$	Mem.	Time
GARP1	48,363,145	247,135,869	166	267	4%	1%	21	68	18%	9%	932	25.2
I-Prot.2	14,309,427	48,024,048	28	30	16%	10%	29	31	24%	16%	240	6.0
Peterson4	12,645,068	47,576,805	23	17	3%	1%	6	3	5%	2%	37	0.5
I-Prot.0	9,798,465	45,932,747	29	38	6%	2%	7	21	44%	29%	362	12.3
BRP	3,280,269	7,058,556	6.0	5.6	29%	15%	15	14	58%	39%	161	2.4
Philo	1,640,881	16,091,905	9.8	10	5%	2%	1.2	4.8	100%	100%	125	10.7
sort	659,683	3,454,988	2.8	3.8	182	181	0.0	0.3	182	182	0.3	0.0
I-Prot.3	388,929	1,161,274	1.0	0.7	14%	7%	0.9	0.9	26%	16%	6.6	0.1
I-Prot.4	95,756	204,405	0.5	0.1	28%	18%	0.5	0.6	38%	28%	2.5	0.0
Snoopy	81,013	273,781	0.6	0.2	12%	4%	0.2	0.7	17%	7%	1.2	0.0
Peterson3	45,915	128,653	0.4	0.0	8%	3%	0.1	0.4	10%	4%	0.5	0.0
SMALL1	36,970	163,058	0.5	0.0	18%	9%	0.1	0.4	48%	45%	0.9	0.0
SMALL2	7,496	32,276	0.4	0.0	19%	10%	0.0	0.4	48%	44%	0.4	0.0
X.509	9,028	35,999	0.4	0.0	10%	4%	0.0	0.4	68%	34%	1.1	0.0
DBM	5,112	20,476	0.4	0.0	100%	100%	0.1	0.5	100%	100%	0.7	0.0
SMCS	5,066	19,470	0.4	0.1	17%	7%	0.0	0.4	25%	11%	0.7	0.0

extended the stubborn set algorithm with efficient heuristics and a *necessary disabling set* [Laa+13a].

We see in Table 11.5 that the reductions obtained by LTSmin's POR are consistently greater than those of spin's ample set. An expected result, as the stubborn set method is more fine-grained than the ample-set method [Val91a]. On the other hand, this also makes the algorithm around 8 times slower than the ample set (see for instance the runtimes of the I-Protocol2 model compensated by the difference in the reductions). We think this is a justifiable trade-off given that the better reductions yield nominally competitive runtimes, e.g. for the Philo, I-Protocol and GARP1 models, and that our method is completely language agnostic (it could easily support for example a process-algebraic language such as mCRL2 [Gro+08], which has a different notion of processes).

Finally, we also used our multi-core algorithms with partial-order reduction to verify a different version of the GARP protocol [KLJ10]. Prior, this model had only been verified using lossy hash compaction in spin. LTSmin with SpinS could explore the smallest instance of the model completely, using a Cleary tree and partial-order reduction, proving that it is indeed free of deadlocks (up to the correctness of the model checker itself). The tool explored upward of 12 billion states.

11.6 Conclusions

We presented additional experiments using a 48-core machine and a new set of promela models. The promela semantics are implemented by SpinS: a new frontend for the LTSmin toolset. We demonstrated how the many capabilities of LTSmin can be exploited and with experiments we showed great enhancements for model checking of promela models: through C code generation its performance is on par with spin's, scalability of reachability is better than spin's latest parallel BFS algorithm, tree compression reduces memory usage with a factor 5 compared to Collapse compression and maintains performance, POR can compete with spin's POR, exact scalable parallel LTL is available for promela for the first time.

We also repeated – on a 48-core machine – the benchmarks of the entire Beem database, the set of models used in Part II and Part III. These results (thousands of benchmarks on hundreds of different models/properties) are available online[11.2] and demonstrate the same scalability and efficient memory usage as found in the current chapter.

We aimed to implement promela's semantics as close as possible to spin's; the state and transition counts for all the models discussed in the current chapter are equal to spin's sequential algorithms (the new parallel algorithms seems to increase the number of states compared to spin's own sequential algorithms).

12 Conclusions

12.1 Summary

We proposed scalable and on-the-fly methods for multi-core model checking of both explicit-state and timed systems.

12.1.1 Multi-Core Reachability

For the first time in model checking, we realized almost ideally scalable multi-core reachability by using an algorithm which exploits shared memory more directly than distributed algorithms do, by means of a shared visited set. This set is implemented by a lockless hash table which we designed specifically for the steep memory hierarchies of modern multi-core systems.

Our method is on-the-fly because of its flexibility with respect to the search order used by the reachability algorithm. It also supports state compression by replacing the hash table with a lockless tree table, which can provide sharing between states. Due to the highly combinatorial nature of most model checking problems, the obtained compression is often close to an optimal that consists of two integers per state in its current implementation that is tailored to the physical hardware constraints of current machines.

Incremental tree compression ensures that the sequential performance of the tree table is similar to that of the hash table. And a parallel compact hash table further reduces the compressed size per state to 1 integer. But our scalable concurrent compact hash table can also be used in isolation for application outside of model checking, such as in BDDs.

All the above methods have been extensively benchmarked using the entire Beem database, which contains hundreds of DVE models, and several real-world promela models. *The results show almost ideal scalability and confirm that compression is close to optimal.* It should be emphasized here that the (sequential) performance of our implementation is on par with that of the state-of-the-art spin model checker, lending extra credibility to the obtained results (a sequential slowdown of a certain factor hides communication costs and results in a similar factor of "free" speedup).

12.1.2 Multi-Core LTL Model Checking

We present the first parallel LTL model checking algorithm which is linear in the size of the graph. Our *multi-core nested depth-first search* (Mc-ndfs) algorithm is based on a novel approach of running multiple parallel depth-first searches in semi-independent fashion. The earlier algorithms we propose, use more independence and seem to preserve enough of the depth-first order to guarantee soundness and completeness, while a later algorithm optimistically continuous searches, repairing out-of-order steps using a waiting strategy. For all of these algorithms we presented rigorous correctness proofs.

Experiments show good scalability of Mc-ndfs, especially cndfs, which performs better than the distributed owcty algorithm.

We show how livelocks can be checked with a parallel dfs_{FIFO} algorithm. This solution supports excellent partial-order reduction and experiments confirm it has better scalability and performance than cndfs.

12.1.3 Multi-Core Model Checking of Timed Systems

We proposed a lockless multimap to store the symbolic abstractions needed for model checking of timed automata. With a precise locked algorithm and a revisiting non-blocking algorithm, we realize scalable multi-core reachability for timed automata.

By porting cndfs to the timed setting, we also realize for the first time parallel LTL model checking of timed systems. We further propose methods to exploit properties of the coarsest time abstraction, called subsumption abstraction, in our LTL model checking technique.

These methods were all implemented and experiments show good scalability, with speedups of up to 60 times with respect to uppaal, a state-of-the-art model checker for timed automata. The subsumption abstraction can reduce the number of states by a factor of 2.

12.1.4 Tool Support

All data structures and algorithms are implemented in the multi-core backend of the LTSmin model checker [LPW11a]. This tool is actively maintained by the Formal Methods and Tools group at the University of Twente.

LTSmin is a language-independent model checker which currently supports DiVinE's DVE language, spin's promela language, uppaal's timed automata, and μCRL and mCRL2 process algebras. For the empirical evaluations of explicit-state systems in the current thesis, we consistently employed DVE and promela models, because these are compiled to binaries and offer fast next-state functions, yielding a realistic measurement of the parallel scalability of our proposed methods. While we have checked that μCRL and mCRL2 models scale equally well, this knowledge offers less insight into the scalability of the methods because of their relatively slower generation of successor states, as explained in Section 12.1.1. Although we were not interested in these languages from our research perspective, they benefit a lot from the parallel methods proposed here, as a user of mCRL2 is more likely to run into runtime bottlenecks.

Next to the multi-core backend, LTSmin also supports efficient BDD-based symbolic model checking techniques [BPW10; BP08], a distributed backend [BPW09], and a sequential backend using the general-state expanding algorithm [BLLL09][Pat11, Sec. 4.6] (the latter is beneficial for tasks that are still not supported by the multi-core backend, such as LTL checking using por). The explicit backends support *incremental hashing* [NR08] via *Zobrist hashing* [Zob69], as explained in [LPW11a]. The symbolic backend also supports multi-core (symbolic) explorations using the parallel BDD packages *Sylvan* [DLP13]. All these backends are language-independent.

Language-independence is achieved through the definition of the generic Partitioned Next-State Interface (pins). Through its pins interface, LTSmin abstracts away language-specific features with a state vector format and semantics. At the same time it exposes internal structure in the form of locality information through dependency matrices as described in Definition 12.1. The locality information provided by the matrix $D_{k\times n}$ can be used for example to learn *the partitioned* transition relation, avoiding many calls to the next-state function or allow symbolic processing [BP08].

Definition 12.1 (pins). *pins [BPW10] defines a state vector format* $S \equiv \langle s_0, s_1, \ldots, s_n\rangle$ *with a fixed number of n slots and fixed domains* $|s_i|$*, an initial-state function:* *initial-state*$:$ *S, and a k-partitioned next-state function:* *next-state*$(\{1,\ldots,k\},S):$ *S, and a dependency matrix* $D_{k\times n}$ *recording read/write dependencies between transitions and slots.*

LTSmin further supports several logics to express (liveness) properties, including LTL, CTL, CTL* ad μ-calculus. Some of these are handled in a generic fashion, trans-forming the state space through so-called PINS2PINS wrappers [DLP12]. For example, LTL is implemented with a wrapper that synchronizes a Büchi automaton on-the-fly, extending: the state vector with the locations of the Büchi automaton, the transition relation with the cross product transitions, and the state-label function with a Büchi acceptance label [Pat11, Sec. 4.6]. μ calculus formulae can be combined to *Parametrized Boolean Equation Systems* for which a *parity game* is generated [KP12], which can then be handled by specific solvers.

To enable POR, PINS was extended with guards in the form of Boolean state label (see Definition 12.2). The status of guards – enabled or disabled – can than be queried per state, and the POR algorithm can then use the additional dependency matrices to decide how to reduce the generated transition system [Val98]. Because the POR algorithm in effect selects a subset of the next-state function, it can be implemented as a PINS2PINS wrapper as well [Laa+13a].

Definition 12.2 (PINS for POR)**.** *The interface defining in Definition 12.1 is extended with a state-label function:* STATE-LABELS(S): $2^{\{1,\ldots,g\}}$*, a guard matrix* $G_{g\times k}$ *recording the labels that function as guards for a specific partitioned transition, and a state-label dependency matrix* $L_{g\times n}$ *recording which state-vector slots are read by each label. Additional matrices can be added to encode commuting transitions and enabling and disabling relations between transitions and state labels.*

12.2 Evaluation

The current section studies to what extent the goals of Section 1.5 have been met.

12.2.1 Scalability

The main research question posed in Section 1.5.3 is: *"Can the model checking procedure scale, linearly or ideally, on modern multi-core machines?"*. We can answer this question *positively*, as the extensive experiments with our reachability and LTL algorithms for both explicit-state and timed systems all show good to ideal speedups.

The experiments with our multi-core reachability algorithms show almost ideal speedups even on a 48-core system, while their sequential performance is on par with state-of-the-art model checkers such as SPIN. Experiments also reveal a significant improvement with respect to the prior state-of-the-art parallel solutions [BR08; HB07]. This can be explained by our more direct use of shared memory, as opposed to the distributed approaches in earlier work [BR08]. Moreover, our lockless data structures take more care to utilize the limited bandwidth and avoid relatively slow random memory accesses of modern multi-core machines.

The CNDFS algorithm also provides good scalability for LTL model checking problems. The algorithm is guaranteed to be linear in the size of the graph, and the first parallel LTL checking algorithm of this kind. The worst-case speedup is 1, but experiments show that this never happens for large real-world inputs. Finally, parallel DFS_{FIFO} for livelock checking has the same properties as CNDFS, but provides better scalability (almost ideal). This can be explained by the breadth-first manner in which it treats progress states.

The scalability of both LTL and reachability algorithms is transferred to the timed setting because the same data structures and algorithms are used. However, for models that exhibit more timing behavior, scalability decreases. This can be explained by the additional locking required on the multimap that manages the symbolic representations of the time abstractions. Unfortunately, we have no way of directly comparing these results with earlier tools that introduced distributed model checking for timed automata.

Section 1.5.4 however defined model checking more broadly to also include systems with hybrid and probabilistic behavior, and properties from branching-time logics such as CTL and the modal μ-calculus: No absolute boundaries for the current research were set upfront, only a logical order of tackling them. In the subsequent section, we will discuss the status of these outstanding open problems.

12.2.2 Correctness

The first subquestion concerns itself with the correctness of the methods proposed here: *"Are our proposed methods for multi-core model checking provably correct?"*.

In the case of the data structures proposed in the current thesis, we consistently offered at least good arguments for their correctness. An abstracted version of the lockless hash table was also implemented in PROMELA and model checked. This process revealed one bug concerning a non-deterministic probe sequence, which we fixed in the algorithm. The fact that the tree table is *consistent* and *durable* in its storage of state vectors follows directly from the fact the tree can be described as an injective function projecting each state to a unique location in its root table. *Atomicity* and *isolation* should be guaranteed by the correctness of the hash table used to implement the tree's root and node tables. For the locking procedure of the compact hash table we provided a proof of correctness that concludes *linearizability*.

In the case of the multi-core nested depth-first search algorithms, we provided in all cases rigorous poofs for their soundness and completeness with respect to determining Büchi emptiness.

In the case of the timed verification methods, we illustrated correctness by showing the correspondences with the sequential algorithm. By contradiction, we showed that our locking strategy of the multimap ensures the absence of revisits in the reachability algorithm. The non-blocking implementations of the reachability algorithms do not exhibit this property and have not been proven correct.

Furthermore, we should note that all our proposed methods preserve the completeness property of model checking. In strong contrast to methods that employ lossy approaches, such as hash compaction, bit-state hashing using *Bloom filter*s, etc.

On the level of the implementation, we consistently validated our work by comparing state counts, transition counts, number of counterexamples and verification outcome with other tools for all input models and all performed benchmarks.

12.2.3 Compatibility

The second subquestion addresses the issue of compatibility with other state-space reduction methods: *“Are our parallel model checking procedures compatible with other existing approaches to tackle the state space explosion problem?”* A multitude of techniques combating state explosion has been presented in Section 1.4.3. Some of these are orthogonal to our approach and have not been considered in the current thesis. For example, we did not study any symbolic verification techniques, nor did we focus on any specific formalisms such as Petri nets.

For the explicit and semi-symbolic (timed) approaches that we did study, several important reduction techniques can be identified, these are: *on-the-fly model checking, state compression and partial-order reduction*. This selection is based on extensive experience in that field with the (explicit-state) model checker SPIN [Hol08; HB07; HJG08; Hol97b; Hol12]. Table 12.1 summarizes to what extent the contributions in the current thesis are compatible with these techniques.

The *on-the-fly* behavior of our contributions is good since the use of a shared state store allows flexible exploration orders. Parallel depth-first orders are indeed known for excellent on-the-fly behavior in many cases [RK88]. Since the parallel LTL algorithms use depth-first strategies, they have the potential to find accepting cycles that are deep in the state space much faster, as our experiments confirm in Section 6.4 and Section 8.5.6. In the case of reachability, our experiments focus on exhaustive exploration in order to benchmark the scalability better: On-the-fly hunting for deadlocks is not tested, as the process often terminates so quickly that it is hardly interesting (which again confirms the usefulness of on-the-fly algorithms). The gains for finding reachability properties can however be deduced from the experiments done with our LTL algorithms in Section 6.4. This behavior is likewise preserved for our solutions for timed systems in Part IV.

State compression is also compatible with all our contributions. Experiments with explicit-state reachability in Part II and timed reachability in Chapter 9 confirm this. Extensive experiments with our multi-core LTL algorithms and tree compression are

Table 12.1: Compatibility of the different contributions with existing state-space reduction techniques

Formalism	Property	Explicit state	+ On-the-fly	+ Compression	+ POR
Explicit	Reachability	✓	✓	✓	✓
	LTL	✓	✓	✓	X
	. . Livelocks	✓	✓	✓	✓
Timed	Reachability	✓	✓	✓	✓
	LTL	✓	✓	✓	X

available online (see footnote[11.2] on page 264), confirming that indeed this combination works equally well.

Tree compression yields an optimal compression of up to 8 bytes per state as Section 3.4 demonstrates. In practice this compression is often achieved due to the combinatorial nature of many inputs, which is indeed confirmed by experiments that show a median compressed size of 9.6 bytes per state (Figure 3.13). With compact hash tables, this median size is further reduced to $9.6-8+4=5.6$ bytes per state: Each state requires at least one root, stored as an 8 byte key in the root table of the normal tree; a table that is replaced by a compact hash table with 4 byte keys in the compact tree (Section 4.4).

Coupled with the fact that incremental tree compression is often equally fast as plain state storage, tree compression makes a good competitor to lossy hashing schemes (see Section 1.4.3) such as hash compaction, which require around 4 byte per state [GVR99]. Though bit state hashing with Bloom filters can offer compressions of a few bits per state, like any hashing shame, their use complicates the checking of liveness properties [BHR13]. We therefore conclude that our proposed compression scheme is even competitive to the lossy hashing schemes.

Partial-order reduction also combines well with multi-core reachability because the computation of ample sets is completely local for reachability properties such as deadlocks [Laa+13a]. These results can be extended to timed systems as the partial-order reduction techniques are similar [Min99; JLX09; Ben+98].

The slowdown of the ample set calculation is even advantageous to the scalability of the multi-core algorithms since Holzmann et al. [HB07] indeed showed that transition delays improve scalability. The smaller state spaces on the other hand cause lower scalability as there is less work to be parallelized. The interesting cases, those whose reduced state space is even very large, should however scale excellently as indeed we could confirm experimentally.

Other safety properties require global analysis and can be handled by our solutions for LTL model checking. Partial-order reduction however has not been achieved in our parallel LTL algorithms as we did not find a way to realize the (global) ignoring proviso in the parallel setting. This is the only case were we still have a negative result as shown by the crosses in Table 12.1. Parallel DFS$_{\text{FIFO}}$ solves this problem partly by providing a solution for livelocks, an important subset of LTL, allowing POR.

12.2.4 Empirical Evaluation

All our experiments are done with the implementation discussed in Section 12.1.4. In all cases the experiments are repeatable because we supplied complete command lines, input models, tool versions and hardware configurations. In some cases, the experimental data and the scripts are available online, e.g. in: Chapter 4, Chapter 9 and Chapter 10

The experiments are extensive as they cover a large set of models in the DVE language obtained from the Beem database, which contains over 30 different types of models drawn from academic studies and games. These models come in different sizes, totaling the benchmark set to over 300 models. Over 400 LTL properties are included. Further experiments with real-world PROMELA models further confirm the results (see Chapter 11).

12.3 Comparison with Recent Related Work

The current section discusses some more recent related work that was done after our researches. Most of this work has already been discussed in Section 1.7.4 and will be revisited with a more technical perspective. Other connections with related work were recently discovered via private communication.

The SPIN model checker was recently refitted with a better multi-core algorithm by Holzmann [Hol12] (see experiments in Chapter 11). The algorithm is similar to the distributed approach presented in [BR08], but improves the communication bottleneck by introducing N to N communication channels: each of the N worker threads has N queues for incoming states and N queues for outgoing states. Holzmann shows for the first time that these algorithms can also scale on modern multi-core machines. Because of the states are statically partitioned over the worker threads, the algorithm can in fact be improved with separate local hash tables, instead of one shared hash table. Furthermore, complications in the implementation limit scalability with exact state storage, so [Hol12] uses lossy hash compaction to remedy this. We implemented this algorithm in LTSmin version 2.0 (command line option `--strategy=pbfs`), and can confirm that the algorithm in fact also scales for exact reachability.

DiVinE recently also implemented compression [Bar+13]. A difference in the DiVinE implementation of tree compression is the choice for n-ary trees with resizing hash tables [Sti13]. The resizing tree table is probably necessary because the distributed algorithms in DiVinE require multiple (sequential) trees to be maintained by the threads running the search algorithm. However, this doubles the size of the optimal compressed size from 8 byte to 16 byte ([Sti13] does not use compact tables yet) because an additional stable index needs to be stored for node entries that are now reindexed upon resizing. The choice for an n-ary tree further increases the optimal compressed sizes, although no analysis of the compression ratios (such as the one in Section 3.4) is provided in [Sti13]. While the n-ary tree configuration may improve the number of tree node lookups, and thereby the runtime, it is unlikely to deliver an exponential gain as incremental technique can do (see Section 3.3.4).

Lately, Evangelista et al. [EKP13] proposed an exact (non-lossy) compression technique based on compact hashing. The algorithm is parallel and was shown to scale on modern machines with an implementation in an interpreted functional language. The technique is based on storing back pointers for states that can be used to reconstruct full states by reexploration. This guarantees a fixed size of the compressed sizes, whereas in tree compression these depend on the combinatorial structure of the states space. The downside is the additional cost for the reexplorations, though this could easily be mitigated with good caching techniques [BLP03].

We furthermore found that the implementation of our tree table resembles techniques often used in BDDs [Jan+06], where pointers are avoided in favor of more compact hash table indices. The same methods were used in our parallel BDD package [DLP13], improving prior results [YO97].

The competitor to cndfs is the distributed owcty algorithm [ČP03]. The algorithm existed a decade before cndfs, and was more recently extended with partial-order reduction [BBR10a] and hash compaction [BHR13]. The worst-case time complexity of owcty that is quadratic in the size of the graph, with the nice feature that it is know to be linear for weak properties, a substantial subset of LTL [BBR09a]. Theoretically the algorithm is however more likely to scale good because it can be implemented with bfs. cndfs on the other hand has a worst-case complexity that is linear in the size of the graph. Its scalability is theoretically limited due to its use of dfs. In practice however, we have shown that cndfs scales better than owcty. The availability of por however

Table 12.2: Comparison between OWCTY and CNDFS

		OWCTY	CNDFS	DFS$_{\text{FIFO}}$
Worst-case complexity	Weak LTL	Linear	Linear	Linear[12.1]
	Full LTL	Quadratic	Linear	No
Scalability	Theoretical	Excellent	Not good	Decent
	Practical	Good	Very good	Excellent
Partial-order reduction	Livelocks	Decent	None	Excellent
	Full LTL	Decent	None	None

tilts the favor towards OWCTY for inputs with much commutative behavior. For livelock properties however parallel DFS$_{\text{FIFO}}$ is the way to go. Table 12.2 summarizes this comparison. Both livelocks and weak LTL properties can be identified statically [BBR09a; Val93], hence we could essentially lump DFS$_{\text{FIFO}}$ and CNDFS together in one column.

SPIN was also extended with an incomplete parallel LTL algorithm called *Piggyback* [Hol12]. Its scalability was discussed in Chapter 11 and Chapter 8, and is good, yet inferior to CNDFS. However, we suspect that its reliance on hash compaction might yield unsound results as explained in [BHR13].

Our parallel timed LTL algorithms also deal with subsumption abstraction. Konnov made us aware that a similar abstraction arises in (dynamic) symmetry reduction techniques [EW05]. This similar abstraction relation was in fact likewise combined with LTL model checking by Konnov et al. [KZ10]. They even used the same NDFS algorithm as a basis and proposed similar (but fewer) points in the algorithms were abstraction could be used. Experiments revealed however that the blowup caused by the depth-first order did not stand up to the gains obtained by the reduction and the effort was abandoned. It is indeed likely that OWCTY would yield smaller state spaces due to its use of BFS [BHV00]. However, it is currently unknown how OWCTY can be combined with subsumption abstraction.

In private communication with Henri Hansen, we learned that the redundancy of the ignoring proviso for DFS$_{\text{FIFO}}$ was already observed by Valmari. In "Stubborn Set Methods for Process Algebras" [Val97], Valmari proved the more general property in Lemma 5.1, showing that actually *all minimal divergence traces* are preserved without the ignoring proviso.

[12.1] Only for livelocks.

12.4 Open Questions

The first way to extend the results in the current thesis is to look at a broader definition of model checking. Avoiding the eternal question whether branching-time or linear temporal logics are more suitable for model checking (see Section 1.4), we could just continue to employ the current results for solving CTL and μ-calculus checking. We could also investigate parallel solutions for symbolic approach based on our parallel hash table, or create a heterogeneous approach by combining distributed and multi-core model checking. Table 12.3 illustrates this by filling in Table 1.1 in Section 1.5.3. We will discuss some of the open questions that the table suggest and discuss related work that may solve it already or could be used to solve it.

Table 12.3: Extending the scope of the current research

Formalism	Property	Explicit state	+ On-the-fly	+ Compression	+ POR	Symbolic	Distributed
Plain	Reachability	✓	✓	✓	✓	✓	?
	LTL	✓	✓	✓	X	?	?
	CTL	?	?	?	?	?	?
	μ-calculus	?	?	?	?	?	?
Timed	Reachability	✓	✓	✓	✓	?	?
	LTL	✓	✓	✓	X	?	?
	CTL	?	?	?	?	?	?
	μ-calculus	?	?	?	?	?	?
Stoch.	Reachability	?	?	?	?	?	?
	LTL	✓	?	?	?	?	?
	CTL	?	?	?	?	?	?
	μ-calculus	?	?	?	?	?	?

Multi-core symbolic BDD-based model checking is less of an open problem as recently van Dijk et al. [DLP13; DLP12] introduced the parallel BDD package *Sylvan*. The proposed technique uses a modified version of the lockless hash table presented in Chapter 2 and is a direct continuation of the current project which already shows promising results.

Important work on the problem of parallel checking of branching-time logics and hybrid/stochastic/probabilistic formalisms was summarized by Luecke and Brim [Cli08], though it mainly focuses on distributed solutions. For CTL, some techniques that scale on multi-cores have been proposed by Saad et al. [SZB12; Saa11]. For probabilistic / stochastic systems some distributed techniques for reachability exist [Blo+08b; BH06]. But also solutions for using multi-cores for LTL checking [Bar+08] and for CTL* checking [IB06] (a superset of both LTL and CTL, but a subset of μ-calculus).

We were unable to find any work on the combination of distributed and shared-memory parallelism, as suggested by the last column in the table. But other work proposes solutions for large-scale distributed model checking using *bulk synchronous parallel* (BSP) computing [GGP12a; GGP12c; GGP12b; MH00]. Such algorithms might be useful to realize the hybrid parallelism required for exploiting massive *cloud computing* environments.

Multi-core reachability. Due to good scalability, we consider reachability mainly solved, though it might still be interesting to investigate:

- Although, we have enough confidence in the correctness of our lockless data structures, it might be interesting to come up a complete machine-checkable proof. Huisman et al. are pursuing this goal in the VerCors project [Ami+12] and are actually using our lockless hash table as a case study [ABH13].

- Reexploration of states is not required for all model checking algorithms. Can an imprecise algorithm deliver better scalable performance? This is hard to imagine as the solutions presented in Part II already perform almost ideally, but there may still be specific inputs for which other approaches are required.

Multi-core nested depth-first search. Our Mc-ndfs algorithms still pose some interesting research directions:

- A main remaining issue is whether por can be combined efficiently with cndfs. The benefit here is that the algorithm already uses dfs, which is traditionally used to implement the required ignoring proviso [Val91a; EP10]. It is easy to prove using Lemma 7.4 that indeed a cycle proviso can be implemented in the outer, blue search by expanding a state fully upon detecting a cyan state, which must be on the local stack. However it remains to see whether the revisiting problem can be solved for cndfs [HPY96], whether the algorithm would still terminate, and whether multiple parallel searches might not cause a large over-estimation of the proviso.

- The most interesting open problem to consider is whether there is a linear-time Büchi emptiness (cycle-detection) algorithm that delivers guaranteed scalability. We did not find a way to combine the sharing of the red color in the LNDFS algorithm with the sharing of the blue color in the CNDFS algorithm (which only does late red coloring after dangerous situations have been repaired). We consider it likely however that there exist ways to improve CNDFS for specific inputs such as weak LTL.

- We conjecture that Algorithm 5.4 (and hence also Algorithm 5.3) is correct for 2 workers without **await** statement. In private communication, Wan Fokkink posed the related conjecture that the algorithms are correct for any number of workers when the **await** condition is modified to $count = 1$ (instead of $count = 0$). As of yet, neither conjecture has been proved.

- The problems for which OWCTY is linear-time can be identified statically by inspecting the LTL property [BBR09a]. In the previous section, we saw how CNDFS is always linear-time, but may scale worse. It is unfortunately still unknown to us which inputs could cause CNDFS to scale bad. In the multitude of benchmarks we presented, none could be identified. It would be interesting to investigate this using artificial input models or even random graphs.

- Additionally, we could try to mechanize our CNDFS proofs. We believe the method is detailed enough to be easily expressible in a theorem prover, which can automatically discharge them. This research would be in line with other attempts to proof correct implementations of formal methods [Esp+13].

- Finally, we could try to invent better *fresh successor heuristics*. For example, we did not try yet to give priority to cyan states, which could speedup the backtracking and hence improve the global sharing in CNDFS.

More useful optimizations. Some other optimization techniques are very useful for specific problem instances:

- State space caching (on disk) could further increase the size of input that can be handled by our algorithms. We suspect that there is little in the way to use current approaches [BLP03; HW07], since our algorithms allow for flexible search orders and load balancing.

- Various works propose methods for obtaining short counterexamples in LTL model checking [GMZ04; HG08]. Normally counterexamples can be rather long due to the depth-first nature of the search. This increases the cost of analyzing them (usually a manual process). The combination of these works might be more challenging. On the other hand, most of these methods are based on (iterative) depth-first searches. It could be interesting to apply our technique of parallelizing such algorithms (see Part III). In [GMZ04], each search iteration is constrained search to the length of the smallest counterexample found thus far. Since we know now that parallel depth-first searches aid in finding shorter counterexamples quicker (see Section 7.4.4), a parallel version of the algorithm in [GMZ04] could yield excellent speedups.
- Fairness can be expressed in LTL, but at great costs. Algorithm-specific solutions solve this much more efficiently [LSD09b; ČP03]. We wonder whether CNDFS can be extended for Büchi automata with additional fairness encoding.
- Bit state hashing can easily be combined with multi-core reachability as the *Bloom filter* data structure can be parallelized directly using atomic instructions. Parallel Bloom filter already exist that optimize towards the caching behavior of modern machines [PSS07]. The combination with LTL model checking seems harder, as the algorithms need to associate data with different states, although Bloom filters do exists for this problem, they are unsound, in addition to being incomplete [Cha+04].
- DBMs could be compressed just like states as is done in [Sti13].

Other directions. Can the ideas behind multi-core NDFS – basically the graph is decorated with information on partial results from local computation to aid the global progress – be reused to parallelize other linear-time algorithms? An interesting candidate is Tarjan's algorithm for detecting strongly connected components (SCCs) [Tar72]. Like NDFS, it is based on DFS and can also be used for cycle detection [Cou99; SE05; GS09], but has far broader applications [BCP08]. More specifically, it can be used for the efficient checking of properties with *strong fairness* [LH00; ČP03]. Currently, the only known parallel solutions for SCC detection are quadratic in the worst case, e.g. [BCP08; Bar+11b; Kre+; LSD09a]

In a more general setting, we could consider various other graph algorithms which rely on depth-first methods.

We did not yet study the effect of the explicit use of the NUMA architectures as we did not find this to be a bottleneck in the current situation. In this light, it is however interesting to note that the switch from a 16-core NUMA to a 48-core NUMA machine resulted in significant contention points in a load balancer implementation which all of the sudden became a bottleneck. This problem was resolved by letting the worker threads each do their own allocation of thread-local memory (in the previous situation, an main thread was assigned with this task). Apparently, the operating system thus allocates memory automatically on the local memory bank. We could further optimize the implementation by using the NUMA library explicitly for e.g. the allocation of the lockless hash table or tree table: The library allows the bucket array to be allocated in distributed fashion over the different memory banks.

As identified in the previous section, it is currently unknown how OWCTY can be combined with subsumption abstraction (including the abstraction used in dynamic symmetry reduction). It could yield better state-space reductions due to its use of BFS.

12.5 Predicting the Future

Heterogeneous systems will become more common already with AMD's latest Kaveri microprocessors which include a completely integrated and autonomous GPU. The advance of NUMA architectures is also unstoppable. Eventually, we will arrive at Network on Chip processor designs. And alternative approaches to parallel reachability, such as proposed in Saad et al. [SZB10; SZB11; Saa11], might be necessary. They use a dynamic way to distribute states over the available processing cores. The distribution is controlled by a dedicated shared data structure, which could be adapted to the more heterogeneous architecture.

Finally, a consequence of Moore's law is that memory hierarchies grow ever steeper. This causes random memory accesses to become more expensive, and in turn non-linear algorithms will become more expensive. We hope that use of parallel DFS searches might fill a gap here, although a precise evaluation of their suitability depends on a more rigorous analysis of their scalability.

Part VI

Appendices

A Proofs for Chapter 5

A.1 Correctness Proof for Mc-ndfs

This appendix presents the full proof of Mc-ndfs, Algorithm 5.3 in Chapter 5. We assume that each line of the code above is executed atomically. The global state of the algorithm is the coloring of the input graph $\mathcal{B}$ and the program counter of each worker. The approach presented here is different from the one in Section 5.4.3, which is based

Algorithm A.1 A Multi-core Ndfs algorithm, coloring globally red in the backtrack

```
proc mc_ndfs($s$,$N$)
  dfs_blue($s, 1$)$\|..\|$dfs_blue($s$,$N$)
  report no cycle
proc dfs_blue($s$,$i$)
  $s.color[i]$ := $cyan$
  for all $t$ in NEXT-STATE$_i^b(s)$ do
    if $t.color[i]$=$white \wedge \neg t.red$
      dfs_blue($t$,$i$)
  if $s \in \mathcal{F}$
    $s.count$ := $s.count$ + 1
    dfs_red($s$,$i$)
  $s.color[i]$ := $blue$
proc dfs_red($s$,$i$)
  $s.pink[i]$ := true
  for all $t$ in NEXT-STATE$_i^r(s)$ do
    if $t.color[i]$=$cyan$
      report cycle & exit all
    if $\neg t.pink[i] \wedge \neg t.red$
      dfs_red($t$,$i$)
  if $s \in \mathcal{F}$
    $s.count$ := $s.count - 1$
    await $s.count$ = 0
  $s.red$ := true
  $s.pink[i]$ := false
```

on reductio ad absurdum. Instead, the following proof solely relies on invariants that always hold, independent of the program counters or at certain lines (pre- and post-conditions). The main correctness result in Theorem A.1 follows directly from these invariants.

We use the following notations: The sets $White_i$, $Cyan_i$, $Blue_i$ and $Pink_i$ contain all the states colored white, cyan, blue, and pink by worker i, and Red contains all the red states. E.g., $s.color[i] = blue$, we write $s \in Blue_i$. It follows from the assignments of the respective colors to the *color* variable that $White_i$, $Cyan_i$ and $Blue_i$ are disjoint. We distinguish between normal *return* and *termination* (**exit all**).

The *Hoare triple* $\{P\}\ C\ \{Q\}$ [Hoa69] style notation expresses two facts at once: (1) If *pre-condition* P holds before calling C, then *post-condition* Q holds upon return of C, and (2) always when C is called, the pre-condition P holds. Whenever a function reaches a **report** statement, it terminates after reporting, i.e. there is no normal function return, making any post-condition vacuously true. Finally, we use the modal operator $s \in \Box X$ to express that $\forall t \in \textsc{next-state}(s) : t \in X$.

Definition A.1 (Pre-condition mc_ndfs). *mc_ndfs adheres to the specification:* $\{\bigcup_i Blue_i = \bigcup_i Cyan_i = \bigcup_i Pink_i = Red = \emptyset \wedge \bigcup_i White_i = \mathcal{S}\}$ mc_ndfs(s_I) $\{\ldots\}$

First, we present a few basic lemmas that allow us to reason precisely on the behavior of the DFS.

Lemma A.1 (Pre-/post-conditions dfs_blue (1)). *dfs_blue adheres to:*
$\{\exists C \subseteq \mathcal{S} \colon Cyan_i = C \wedge s \notin Cyan_i\}$ dfs_blue(s,i) $\{Cyan_i = C\}$

Proof. Only at Line 2 and Line 8, dfs_blue(s,i) can be called. At Line 2, by Definition A.1, $Cyan_i = \emptyset$, hence $s \notin Cyan_i$. At Line 8, by the condition at Line 7, $s \in White_i$, hence $s \notin Cyan_i$. Also, at the start of dfs_blue(s,i), we have $\exists C \subseteq \mathcal{S} \colon Cyan_i = C$. We show by induction on the number of nested dfs_blue calls n of worker i that Lemma A.1 holds with this C.

- $n = 1$: If $Cyan_i = C$ at Line 5, then after Line 5, $Cyan_i = C \uplus \{s\}$. Since $n = 1$, Line 8 was not reached, hence also at Line 9, $Cyan_i = C \uplus \{s\}$. This also holds after Line 11, since no states are removed or added from $Cyan_i$ in dfs_red. So $Cyan_i = (C \uplus \{s\}) \setminus \{s\} = C$ after Line 12.
- $n = n' + 1$: Assume Lemma A.1 holds for n' nested dfs_blue calls of worker i. We show that it also holds for call $n' + 1$. If $Cyan_i = C$ at Line 5, then after Line 5, $Cyan_i = C \uplus \{s\}$, and by the induction hypothesis, also after Line 8, so $Cyan_i = (C \uplus \{s\}) \setminus \{s\} = C$ after Line 12.

□

Lemma A.2 (Pre-/post-condition dfs_blue (3)). *dfs_blue adheres to the specification:* $\{s \notin (Blue_i \cup Cyan_i)\}$ *dfs_blue*(s,i) $\{s \in Blue_i\}$

Proof. dfs_blue is only called at Line 2 and Line 8. At Line 8, $t \in White_i$, hence $t \notin (Blue_i \cup Cyan_i)$, and at Line 2, by Definition A.1, $s_i \notin (Blue_i \cup Cyan_i)$. Finally, by Line 12, $s \in Blue_i$ when dfs_blue returns. □

Lemma A.3 (Pre-/post-condition dfs_red (1)). *dfs_red adheres to the specification:* $\{\exists P \subseteq \mathcal{S}\colon Pink_i = P \wedge s \notin Pink_i\}$ *dfs_red*(s,i) $\{Pink_i = P\}$

Proof. Only at Line 11 and Line 19, dfs_red(s,i) can be called. At Line 19, by the condition at Line 18, $s \notin Pink_i$. At Line 4, by Definition A.1, $Pink_i = \emptyset$. At Line 11, still $Pink_i = \emptyset$, since $Pink_i$ is only changed in dfs_red, hence $s \notin Pink_i$. Also, at the start of dfs_red(s,i), we have $\exists P \subseteq \mathcal{S}\colon Pink_i = P$. We show by induction on the number of nested dfs_red calls n of worker i that Lemma A.3 holds with this P.

- $n = 1$: If $Pink_i = P$ at Line 14, then after Line 14, $Pink_i = P \uplus \{s\}$. Since $n = 1$, Line 19 was not reached, hence also at Line 23, $Pink_i = P \uplus \{s\}$. So $Pink_i = (P \uplus \{s\}) \setminus \{s\} = P$ after Line 24.
- $n = n' + 1$: Assume Lemma A.3 holds for n' nested dfs_red calls of worker i. We show that it also holds for call $n' + 1$. If $Pink_i = P$ at Line 14, then after Line 14, $Pink_i = P \uplus \{s\}$, and by the induction hypothesis, also after Line 19, so $Pink_i = (P \uplus \{s\}) \setminus \{s\} = P$ after Line 24.

□

The following lemma expresses the fact that during the (local) blue search, no red search is active.

Lemma A.4 (Pre-/post-condition dfs_blue (2)). *dfs_blue adheres to the specification:* $\{Pink_i = \emptyset\}$ *dfs_blue*(s,i) $\{Pink_i = \emptyset\}$

Proof. First we prove by induction on the number of nested dfs_blue calls n of worker i that (1) $\{\exists P \subseteq \mathcal{S}\colon Pink_i = P\}$ dfs_blue(s,i) $\{Pink_i = P\}$. Then, we show that $P = \emptyset$. At the start of dfs_blue(s,i), we have: $\exists P \subseteq \mathcal{S}\colon Pink_i = P$.

- $n = 1$: If $Pink_i = P$ at Line 5, then since $n = 1$, Line 8 was not reached, hence also at Line 9, and by Lemma A.3 after Line 11, $Pink_i = P$.
- $n = n' + 1$: Assume (1) holds for n' nested dfs_blue calls of worker i. We show that it also holds for call $n' + 1$. If $Pink_i = P$ at Line 5, then by the induction hypothesis, also at Line 9, and by Lemma A.3 after Line 11, $Pink_i = P$.

Finally, by Definition A.1, $\bigcup_i Pink_i = \emptyset$ at Line 1, so $P = \emptyset$ for all dfs_blue calls at Line 2. Furthermore, if $Pink_i = P$ at Line 5, then also $Pink_i = P$ at Line 8 for all nested dfs_blue calls. Hence $P = \emptyset$ for all dfs_blue calls. □

Starting at the following lemma, the notation mc_ndfs$(s,i)@n$ refers to Line n in the code of mc_ndfs in Algorithm 5.3.

Lemma A.5. *mc_ndfs$(s,i)@9$: $s \in \Box(Blue_i \cup Cyan_i \cup Red)$ is an invariant of* Mc-ndfs*.*

Proof. At Line 9, we know for all $t \in \textsc{next-state}_i(s)$ that either (1) $t \notin White_i \vee t \in Red$, or (2) dfs_blue$(t,i)$ was executed at Line 8 since at Line 7, $t \in White_i \wedge t \notin Red$. If (1), then either $t \notin White_i$, hence $t \in (Blue_i \cup Cyan_i)$, or $t \in Red$, hence $t \in (Blue_i \cup Cyan_i \cup Red)$. If (2), then after Line 8, by Lemma A.2, $t \in Blue_i$, hence $t \in (Blue_i \cup Cyan_i \cup Red)$. □

Lemma A.6. *Invariantly in* Mc-ndfs*, for all workers, the successors of blue states are either Red, or Blue or cyan for the same worker: $\forall i$: $Blue_i \subseteq \Box(Blue_i \cup Cyan_i \cup Red)$.*

Proof. Only at Line 12, a state s is added to $Blue_i$. By Lemma A.5, at Line 9, for all $t \in \textsc{next-state}_i(s)$, we have $t \in (Blue_i \cup Cyan_i \cup Red)$. We can show that a state is never removed from $Blue_i \cup Cyan_i \cup Red$. First of all, once set to **true** (Line 23), $s.red$ is never set to **false**. Therefore, a state s is never removed from Red. Second of all, $s.color[i]$ is never set to $white$, hence a state $s \in (Blue_i \cup Cyan_i)$ is never removed from $Blue_i \cup Cyan_i$. Hence $t \in (Blue_i \cup Cyan_i \cup Red)$ also at Line 12. □

Lemma A.7. *mc_ndfs$(s,i)@23-24$: $s \notin \Box(Cyan_i)$ is an invariant of* Mc-ndfs*.*

Proof. By contradiction. Say that $s \in \Box(Cyan_i)$ at Line 24. Then there must exist $t \in \textsc{next-state}_i(s)$ with $t \in Cyan_i$. Since dfs_red does not add states to $Cyan_i$ (Line 19), we also have $t \in Cyan_i$ at Line 15. But then, the condition at Line 16 holds, and Line 17 is reached, therefore Line 24 cannot have been reached, and we have a contradiction. □

We can now reason over the color of states that dfs_red visits.

Lemma A.8 (Pre-/post-conditions dfs_red (2))**.** *dfs_red adheres to:*
$\{s \in \Box(Blue_i \cup Cyan_i \cup Red) \wedge s \in (Blue_i \cup Cyan_i)\}$ *dfs_red(s,i)* $\{s \in Red\}$

Proof. By induction on the number of nested dfs_red calls n by worker i. dfs_red is only called at Line 11 and Line 19.

- $n = 1$: hence this dfs_red(s,i) must have been called at Line 11 in dfs_blue(s,i). Say that at Line 5, $Cyan_i = C$. Then after Line 5, $Cyan_i = C \uplus \{s\}$, and by

Lemma A.1, also after Line 8, $Cyan_i = C \uplus \{s\}$. Hence at Line 11, $s \in Cyan_i$, so $s \in (Blue_i \cup Cyan_i)$. Furthermore, by Lemma A.5, at Line 9, $s \in \Box(Blue_i \cup Cyan_i \cup Red)$. This also holds at Line 11.

- $n = n' + 1$: Assume Lemma A.8 holds for n' nested dfs_red calls of worker i. We show that it also holds for call $n' + 1$. This dfs_red(s,i) must have been called at Line 19. There, by the induction hypothesis, $t \in (Blue_i \cup Cyan_i \cup Red)$, and $t \notin Red$ at Line 18, hence $t \in (Blue_i \cup Cyan_i)$ at Line 19. By Lemma A.7 and the fact that $Cyan_i$ is not changed in dfs_red, $t \notin Cyan_i$ at Line 19, hence $t \in Blue_i$, therefore, by Lemma A.6, $t \in \Box(Blue_i \cup Cyan_i \cup Red)$.

Finally, by Line 23, $s \in Red$ after Line 23. □

The red search thus only visits states that are blue or cyan. This is not surprising, as the red search happens after the blue search backtracks and takes into account the cyan stack of the blue search. As a consequence, red states also have these colors:

Lemma A.9. *Invariantly in Mc-ndfs, red states are also blue or cyan for some worker:* $Red \subseteq \bigcup_i (Blue_i \cup Cyan_i)$.

Proof. Only at Line 23, s is added to Red by worker i. At Line 13, by Lemma A.8, $s \in (Blue_i \cup Cyan_i)$. A state is never removed from $Blue_i \cup Cyan_i$, since $s.color[i]$ is never set to *white*. Hence, also at Line 23, $s \in (Blue_i \cup Cyan_i)$. □

A basic property of dfs is that successors of backtracked states are also backtracked or are still on the stack. The following lemma expresses this fact for the red search, which also takes into account the cyan stack of the blue search.

Lemma A.10. *Invariantly in Mc-ndfs, successors of red states are either red or pink for some worker, but in the latter case, never cyan for that same worker:* $Red \subseteq \Box(Red \cup \bigcup_i (Pink_i \setminus Cyan_i))$.

Proof. Only at Line 23, s is added to Red by worker i. At Line 23, we know for all $t \in$ next-state$_i(s)$ that either (1) $t \in Pink_i \vee t \in Red$, or (2) dfs_red$(t,i)$ was executed at Line 19 since at Line 18, $t \notin Pink_i \wedge t \notin Red$. And (3), $t \notin Cyan_i$ by Lemma A.7. If (1), then either $t \in Pink_i \setminus Cyan_i$, or $t \in Red$, hence $t \in Red \cup \bigcup_i (Pink_i \setminus Cyan_i)$. If (2), then after Line 19, by Lemma A.8, $t \in Red$.

We can also show that a state s is never removed from $Red \cup \bigcup_i (Pink_i \setminus Cyan_i)$. A state is never removed from Red. It may be removed from $Pink_i \setminus Cyan_i$ at Line 24 (by removing pink) or at Line 5 (by marking a pink state cyan). However, at Line 24, s is already red by Line 23, and by the pre-condition of Lemma A.4 a new cyan state cannot be already pink. Hence $s \in Red \cup \bigcup_i (Pink_i)$ after Line 24 and Line 5. □

Corollary A.1. *mc_ndfs*(s,i)@20: $s \in \Box(Red \cup (Pink_i \setminus Cyan_i))$ *is an invariant of mc_ndfs.*

Proof. It follows from the proof Lemma A.10, that $s \in \Box(Red \cup Pink_i \setminus Cyan_i)$ at Line 23. This also holds at Line 20 (the statements between those lines do not modify the color sets). □

Lemma A.11. *Invariantly in* Mc-ndfs*, pick states are blue or cyan for the same worker:* $\forall i\colon Pink_i \subseteq (Blue_i \cup Cyan_i)$.

Proof. After Line 14, $s \in Pink_i$, and by Lemma A.8, $s \in (Blue_i \cup Cyan_i)$. Since $s.color[i]$ is never set to *white*, a state is never removed from $Blue_i \cup Cyan_i$. □

Lemma A.12. *Invariantly in* Mc-ndfs*, successors of pink states are red, or cyan or blue for the same worker:* $\forall i\colon Pink_i \subseteq \Box(Blue_i \cup Cyan_i \cup Red)$.

Proof. When coloring a state s pink at Line 14 in dfs_red(s,i) for some worker i, by Lemma A.8, $s \in \Box(Blue_i \cup Cyan_i \cup Red)$. Furthermore, a state $s \in (Blue_i \cup Cyan_i \cup Red)$ is never removed from $Blue_i \cup Cyan_i \cup Red$, because $s.color[i]$ is never set to *white*, hence a state $s \in (Blue_i \cup Cyan_i)$ is never removed from $Blue_i \cup Cyan_i$, and $t.red$ is never set to **false**, hence a state $t \in Red$ is never removed from Red. □

Lemma A.13. *Invariantly in* Mc-ndfs*, blue accepting states are also red:*
$\forall a \in \mathcal{F}\colon a \in \bigcup_i Blue_i \implies a \in Red$.

Proof. A state is never removed from Red. Furthermore, only at Line 12, a state is added to $Blue_i$ by some worker i. If $s \in \mathcal{F}$, after Line 11, by Lemma A.8, $s \in Red$. □

Accepting states are special in the red search: The search is always launched starting from an accepting seed and *never* visits any other accepting state. This basic insight was used in [Cou+92] to proof the algorithm correct. Of course here we require more detailed lemmas to ensure that the synchronization between parallel workers does not affect this property negatively. Indeed it does not:

Lemma A.14 (Pre-/post-conditions dfs_red (4))**.** *For all* $a \in \mathcal{F}$*, dfs_red adheres to the specification:* $\{Pink_i = \emptyset\}$ *dfs_red*(a,i) $\{Pink_i = \emptyset\}$

Proof. Only at Line 11 and Line 19 can a dfs_red be called. First, we show by contradiction that dfs_red(a,i) cannot be called at Line 19. Say that dfs_red(a,i) is called at Line 19 in some dfs_red(s,i) with $a \in$ next-state(s). Then, at Line 13, by Lemma A.8, $a \in (Blue_i \cup Cyan_i \cup Red)$. Since $a \in \mathcal{F}$, by Lemma A.13, $a \in (Cyan_i \cup Red)$. If $a \in Cyan_i$, the condition in Line 16 holds for a, hence Line 17 is reached, so Line 19 is not reached, and we have a contradiction. If $a \in Red$, then the condition in Line 18 does not hold, hence Line 19 is not reached, and we have a contradition.

At Line 4, by Definition A.1, $Pink_i = \emptyset$. At Line 11, still $Pink_i = \emptyset$, $Pink_i$ is only changed in dfs_red, and by Lemma A.3, at Line 12, also $Pink_i = \emptyset$. □

Corollary A.2. *mc_ndfs(s,i)@20: $s \in \mathcal{F} \implies Pink_i = \{s\}$ is an invariant of Mc-ndfs.*

Proof. After Line 24, since $s \in \mathcal{F}$, by Lemma A.14, $Pink_i = \emptyset$. Therefore, before Line 24, $Pink_i = \{s\}$ ($s \in Pink_i$ at Line 14 and Line 23 by Lemma A.3). □

Lemma A.15. *Invariantly in Mc-ndfs, pink accepting states that are not yet red are cyan for the same worker (they are on the stack of both the blue and the red search, until the latter backtracks): $\forall i\colon\ a \in \mathcal{F} \wedge a \in (Pink_i \setminus Red) \implies a \in Cyan_i$.*

Proof. Since $a \in Pink_i$, by Lemma A.11, $a \in (Blue_i \cup Cyan_i)$. In fact, $a \notin Blue_i$, since if $a \in Blue_i$, by $a \in \mathcal{F}$ and Lemma A.13, also $a \in Red$, which is not the case. Hence $a \in Cyan_i$. □

Lemma A.16. *mc_ndfs(s,i)@20: $s \in \mathcal{F} \implies s \in \Box(Red)$ is an invariant of Mc-ndfs.*

Proof. By Corollary A.2, $Pink_i = \{s\}$. Corollary A.1 then only allows: $s \in \Box(Red \cup \{s\})$ (a self loop over s). However, $s \in Cyan_i$ by Lemma A.15, contradicting Corollary A.1 ($s \notin Cyan_i$). Therefore, this self loop cannot exist and we have: $s \in \Box(Red)$. □

In the following lemmata, the notation $s \xrightarrow{\neg Red}{}^{+} t$ is used to indicate that the path does not contain any red state. Similarly, $s \xrightarrow{Pink}{}^{+} t$ denotes a path that contains only pink states.

Lemma A.17. *Invariantly in Mc-ndfs, for the same worker, all cyan states can reach all pink states: $\forall i, c \in Cyan_i, p \in Pink_i\colon \exists a \in \mathcal{F} \cap Pink_i\colon\ c \to^* a \to^* p$.*

Proof. At Line 5/Line 12, a *successor* state is added/removed to/from $Cyan_i$, thus for all states $c_1, c_2 \in Cyan$, we have a path $c_1 \xrightarrow{Cyan_i}{}^{*} c_2$. There are yet no $Pink_i$ states (Lemma A.4).

$Pink_i$ states can only be introduced by invoking dfs_red at Line 11. There, dfs_red is called for $a \in \mathcal{F}$, which are immediately colored $Pink_i$ at Line 14. By Lemma A.15, also $a \in Cyan_i$. Then, Lemma A.17 holds with $p \equiv a \equiv c$.

In dfs_red(s,i), dfs_red is only called for $t \in$ NEXT-STATE(s) at Line 19, when, by Line 14, $s \in Pink_i$. As long as a dfs_red(t,i) is not finished, also $t \in Pink_i$ by Line 14. This shows that for all $p \in Pink_i$, there exists a path $a \xrightarrow{Pink_i}{}^{*} p$. Since the cyan states are connected, we also have $c \to^* a \to^* p$. □

The counter on accepting states ensures that they are not marked red prematurely (while some red search is still busy):

Lemma A.18. *Invariantly in Mc-ndfs, pink accepting states are non-red, except when all works have decremented the state's counter and are waiting for the counter to become 0:* $\forall i, a \in \mathcal{F} \cap Pink_i \colon a \notin Red \vee \mathsf{dfs\_red}(a,i)@24$.

Proof. Before $a \in \mathcal{F}$ is colored pink by $\mathsf{dfs\_red}(a,i)$ at Line 14, its counter is first incremented at Line 10 because by Corollary A.2 the dfs_red call was made at Line 11. Hence, $Pink_i = \{a\}$. The counter can only be decremented by Line 21, where again $Pink_i = \{a\}$ by Lemma A.3. Thus only after the decrement of the counter, we can have $a \in Red$, but at that same time a is about to be uncolored pink, because we have dfs_red (a,i)@24. □

Lemma A.19. *The following invariant holds for Mc-ndfs:*
$\forall s \in Red, a \in \mathcal{F} \setminus Red, \colon s \rightarrow^{+} a \implies (\exists i, p \in Pink_i, c \in Cyan_i \colon p \xrightarrow{\neg Red}{}^{+} c)$
(For all red states with a path to a non-red accepting state, there is some path from a pink to a cyan state of the same worker without red on it)

Proof. Assume towards a contradiction that $s \rightarrow^{+} a$ for some $s \in Red$, $a \in \mathcal{F}$ and $a \notin Red$. Let $s' \in Red$ be the *last* red state on the path $s \rightarrow^{+} a$. Then, since $s' \neq a$, it has a successor $t \notin Red$ in this path. By Lemma A.10, we obtain $t \in Pink_i$ for some worker i, so let $p := t$.

Note that $t \neq a$, otherwise by Lemma A.15 $t \in Cyan_i$ and by Lemma A.10 $t \notin Cyan_i$. So we find another successor t' such that $s \rightarrow^{*} s' \rightarrow t \rightarrow t' \rightarrow^{*} a$. Assume towards a contradiction that no state on the path $t' \rightarrow^{*} a$ is in $Cyan_i$; recall that $t' \rightarrow^{*} a$ contains no *Red* states either (we started from the *last* red state s' on the path). Then by Lemma A.12, all states on $t' \rightarrow^{*} a$ are in $Blue_i$. But then also $a \in Blue_i$ and by Lemma A.13, $a \in Red$, a contradiction. So there exists a $c \in Cyan_i$ with $s \rightarrow^{*} p \rightarrow^{+} c \rightarrow^{*} a$.

It is easy to demonstrate that the invariant is not invalidated when pink and cyan states are removed. When p' is uncolored at Line 24, we have $p' \in Red$ by Line 23 for which the invariant holds (with $s \equiv p'$). When a state is removed from cyan at Line 12, there are no pink states by Lemma A.4 and the conclusion of the invariant was already invalid.

Also the addition of a red state cannot invalidate the invariant. First of all, if the red state is the accepting state, the premise of the invariant no longer holds. In all the other cases, we have a new *last* red state on the path $s \rightarrow^{+} a$. □

In the following, we reason on the graph module red states and show that any configuration of the colors in this subgraph will lead to eventual detection of an accepting cycle if there exists one. This is necessary, a stronger invariant on red states, such as *"red states do not lie on accepting cycles"* does not hold, as Figure 5.1 shows. $\mathcal{B}'$ represents $\mathcal{B}$ after the removal of red states, i.e. for $\mathcal{B} = (\mathcal{S}, s_I, \textsc{next-state}, \mathcal{F})$, we define $\mathcal{B}' = (\mathcal{S}', s'_I, \textsc{next-state}', \mathcal{F}')$, with $\mathcal{S}' = \mathcal{S} \setminus Red$, $s'_I = s_I$[1.1], $\textsc{next-state}'(s) = \textsc{next-state}(s) \setminus Red$, and $\mathcal{F}' = \mathcal{F} \setminus Red$. State colorings are affected in a similar way, e.g. $Pink'_i = Pink_i \setminus Red$.

Lemma A.20. *The following invariant holds for* Mc-ndfs*:*
$\forall a \in \mathcal{F} \colon s_I \rightarrow^* a \wedge a \rightarrow^+ a \implies \exists a' \in \mathcal{F}' \colon s_I \rightarrow^* a' \rightarrow^+ a' \vee \exists i, p \in Pink'_i, c \in Cyan'_i \colon p \xrightarrow{\neg Red}{}^+ c$
(The büchi automaton sans red states contains a reachable accepting cycle or a non-red path from a pink state to a cyan state for some worker).

Proof. Several cases can be identified upfront:

1 at Line 12, a state is removed from $Cyan_i$,

2 at Line 24, a state is removed from $Pink_i$, and

3 at Line 23, a state is colored Red.

In Case 1, no path $p \xrightarrow{\neg Red}{}^+ c$ with $p \in Pink_i$ and $c \in Cyan_i$ is removed, because at Line 12 there are no $Pink_i$ states (Lemma A.4).

In Case 2, at Line 24 s is removed from $Pink_i$. By Line 23 the state is already red and could not be part of the non-red path.

For Case 3, we have two sub cases: Case 3a, when the state s marked red is on an accepting cycle ($\exists a \in \mathcal{F} \colon a \rightarrow^+ s \rightarrow^+ a$), and Case 3b, when the state s marked on a path $p \xrightarrow{\neg Red}{}^* s \xrightarrow{\neg Red}{}^* c$ with $p \in Pink'_i$ and $c \in Cyan'_i$.

Case 3a: First note that $s \not\equiv a$, because all successors of a must be red (Lemma A.16), hence there can be no path $a \xrightarrow{\neg Red}{}^+ a$. Therefore, after Line 23, we have $s \rightarrow^+ a$ with $s \in Red$ and $a \notin Red$. From this path and Lemma A.19, it follows that there is a path $p \xrightarrow{\neg Red}{}^+ c$ for some $p \in Pink_i$ and $c \in Cyan_i$, which satisfies Lemma A.20.

Case 3b: assume that $s \in \pi$ with $\pi \equiv p \xrightarrow{\neg Red}{}^+ c$, $p \in Pink'_i$ and $c \in Cyan'_i$. State s is about to be colored Red. We witness that there must be an $a'' \in Pink_i \cap \mathcal{F}$ such that $c \rightarrow^* a'' \rightarrow^* p$ (Lemma A.17). It does not matter whether $a = a''$, we either have $a'' \notin Red$ or $\mathsf{dfs\_red}(a'', i)@24$ by Lemma A.18. In the former case, we have a path $s \rightarrow^+ a''$ (again $s \neq a''$) with $a \in \mathcal{F}$ and $s \in Red$, and from Lemma A.19 it follows that there is a path $p \xrightarrow{\neg Red}{}^+ c$ for some $p \in Pink_j$ and $c \in Cyan_j$, which satisfies Lemma A.20. In the latter case, the state $a'' \in \mathcal{F} \cap Pink_i$ is about to be uncolored pink by worker j. By Corollary A.2, $Pink_j = \{a''\}$, therefore $p = a''$ contradicting the assumption that $p \xrightarrow{\neg Red}{}^+ c$. □

[1.1] We assume here that $s_I \notin Red$. If this is not the case, $\mathcal{B} \setminus Red$ is a graph consisting of 0 states.

Lemma A.21 (Post-condition mc_ndfs)**.** *mc_ndfs adheres to the specification:* $\{\ldots\}$ *mc_ndfs*(s_I) $\{\bigcup_i Cyan_i = \bigcup_i Pink_i = \emptyset \wedge \forall x\colon s_I \rightarrow^+ x \implies x \in \bigcup_i Blue_i\}$

Proof. Only at Line 5, a state s is colored cyan by some worker i in dfs_blue(s,i). After Line 2, all blue Dfss are finished, hence also dfs_blue(s,i), and by Lemma A.2, $s \in Blue_i$, therefore $s \notin Cyan_i$. Furthermore, after Line 2, for all i, by Lemma A.3, $Pink_i = \emptyset$, i.e. $\bigcup_i Pink_i = \emptyset$. For all workers i, after Line 2, by Lemma A.2, $s_I \in Blue_i$. Finally, by Lemma A.6 and $\bigcup_i Cyan_i = \emptyset$, $Blue_i \subseteq \Box(Blue_i \cup Cyan_i \cup Red) = \Box(Blue_i \cup Red)$. By Lemma A.9 and $\bigcup_i Cyan_i = \emptyset$, $Blue_i \subseteq \Box(Blue_i \cup \bigcup_j Blue_j) = \Box \bigcup_j Blue_j$. Hence, if $s_I \rightarrow^* x$ then $x \in \bigcup_i Blue_i$. □

Theorem A.1 (Correctness Mc-ndfs)**.** *After mc_ndfs is finished, it holds that:* $C \equiv$ **report error** $\iff \exists a \in \mathcal{F}.s_I \rightarrow^* a \wedge a \rightarrow^+ a$

Proof. We split C as follows: $C \equiv C_{\Longleftarrow} \wedge C_{\Longrightarrow} \equiv C_1 \iff C_2$ and prove it in parts ($C_{\Longleftarrow}, C_{\Longrightarrow}$ are the necessary and sufficient conditions).

$C_{\Longleftarrow}$**:** We show that $\neg C_1 \implies \neg C_2$, which implies $C_2 \implies C_1$. First, assume that $\neg C_1$. Then, mc_ndfs(s_I, N) returns normally (no **exit all**). We show by contradiction that $\neg C_2$ holds. Assume C_2, then there exists an $a \in \mathcal{F}$ such that $s_I \rightarrow^* a \wedge a \rightarrow^+ a$. By Lemma A.21, $a \in \bigcup_i Blue_i$, and since $a \in \mathcal{F}$, by Lemma A.13, $a \in Red$. Finally, by Lemma A.20, either there exists another $a' \notin Red$ with $a' \rightarrow^+ a'$ contradicting Lemma A.21 and Lemma A.13, or there exists a path $p \rightarrow^+ c$ for some i, $p \in Pink_i$ and $c \in Cyan_i$. However, by Lemma A.21, $\bigcup_i Pink_i = \emptyset$, so p cannot exist, hence we have a contradiction.

$C_{\Longrightarrow}$**:** We consider it sufficiently obvious that dfs_red(s,i)@17 implies the existence of an accepting run, because of the stacks of the dfss. □

The above proof shows partial correctness of Mc-ndfs. For complete correctness it is required that the algorithm is guaranteed to terminate. If dfs_red terminated, we can conclude termination of dfs_blue from the fact that for each worker i the set $Blue_i \cup Cyan_i$ grows monotonically (blue is never removed). Eventually, all the states are in the set and the blue search ends. The same cannot immediately be concluded for dfs_red, because of the **await** condition at Line 22. Termination of this waiting state, however, follows from the following basic observations: (1) every worker i can have at most one outstanding pink flag on an accepting state ($a \in \mathcal{F} \in Pink_i$), which is unset at Line 24 before entering the waiting state, hence when worker i is waiting, there can be no other worker is waiting for i. Furthermore, also the shared set $Red \cup Pink_i$ grows monotonically, guaranteeing a completion of all red dfss in a finite amount of time.

B

Proofs for Chapter 10

B.1 Correctness Proof and Corollaries for Ndfs

In the current section, we provide a correctness proof for plain Ndfs with cyan color for cycle detection (Algorithm B.1). This proof serves an independent and thorough demonstration of some corollaries about the algorithm which we used in Chapter 10. While these corollaries can be drawn from papers on the algorithm, cf. [Cou+92; SE05], the algorithms are slightly different and the proofs are more informal. We state these additional corollaries at the end, and use them to explain the algorithm more intuitively. Because the proof is written in the context of timed automata, we use the abstracted transition relation $\Rightarrow$ instead of the normal relation $\rightarrow$, but the two are interchangeable, because we do not consider subsumption yet.

As in Section A.1, we use the modal operator in, e.g. $s \in \Box X$ to express that $\textsc{next-state}(s) \subseteq X$. The set $\Box X$ can thus be interpreted as all states that only have successors in X: $\Box X = \{s \in \mathcal{S} \mid \textsc{next-state}(s) \subseteq X\}$. The *Hoare triple* $\{P\}\ C\ \{Q\}$ [Hoa69] style notation expresses two facts at once: (1) If *pre-condition* P holds before calling C, then *post-condition* Q holds upon return of C, and (2) always when C is called, the pre-condition P holds. Whenever a function reaches a **report** statement, it terminates after reporting, i.e. there is no normal function return, making any post-condition vacuously true. We prove our propositions by doing induction over the number of execution steps. If, for example, we prove that successors of blue states are either blue or cyan, it suffices to show that this holds (1) before the algorithm starts (when all states are white) and (2) after execution of each line where either blue or cyan is modified, assuming that it held before. Theorem B.2 and Theorem B.3 prove Algorithm B.1's soundness and

Algorithm B.1 NDFS with cyan color for cycle detection

```
1:  procedure NDFS
2:      $Cyan := Blue := Red := \emptyset$
3:      $dfsBlue(s_0)$
4:      report no cycle
5:  procedure $dfsRed(s)$
6:      $Red := Red \cup \{s\}$
7:      for all $t$ in NEXT-STATE$(s)$ do
8:          if $t \in Cyan$ then report cycle
9:          if $t \notin Red$ then $dfsRed(t)$
10: procedure $dfsBlue(s)$
11:     $Cyan := Cyan \cup \{s\}$
12:     for all $t$ in NEXT-STATE$(s)$ do
13:         if $t \notin Blue \wedge t \notin Cyan$ then
14:             $dfsBlue(t)$
15:     if $s \in \mathcal{F}$ then
16:         $dfsRed(s)$
17:     $Blue := Blue \cup \{s\}$
18:     $Cyan := Cyan \setminus \{s\}$
```

completeness (Algorithm 10.1 is a copy of that algorithm).

Algorithm B.1 uses separate color sets instead of a multi-valued *color* variable as in e.g. Algorithm 5.2. It is however rather easy to see that the correctness of the latter follows from the former as Corollary B.7 illustrates.

Lemma B.1 (Pre-/post-conditions *dfsBlue*)**.** *In Algorithm B.1, the following pre- and post-conditions hold:*
$\{Cyan = C \wedge s \notin (Blue \cup Cyan)\}\ dfsBlue(s)\ \{Cyan = C \wedge s \in Blue\}$

Proof. $dfsBlue(s)$ is only called on white states by Line 2 and Line 13, hence $s \notin Blue \cup Cyan$. By induction on the number of *dfsBlue* calls, we show that $Cyan = C$ for some $C \subseteq \mathcal{S}$ holds upon return of *dfsBlue*, if $Cyan = C$ at call time.

$n = 1$: $dfsBlue(s)$ is called at Line 10 for $s \notin Cyan$ by Line 2 and Line 13. Let $Cyan = C$ at Line 11, so after Line 11, we have $Cyan = C \uplus \{s\}$. Because Line 14 is not called, we have $Cyan = C \uplus \{s\}$ at Line 18 and $Cyan = C \uplus \{s\} \setminus \{s\} = C$ after.

$n = n' + 1$: Assume Lemma B.1 holds for n' nested *dfsBlue* calls. We show that it also holds for call $n' + 1$. If $Cyan = C$ at Line 11, then after Line 11, $Cyan = C \uplus \{s\}$, and by the induction hypothesis, also after Line 14, so $Cyan = (C \uplus \{s\}) \setminus \{s\} = C$ after Line 11.

Before $dfsBlue(s)$ returns, we have $s \in Blue$ at Line 17. □

Lemma B.2. *In Algorithm B.1, successors of blue states are blue or cyan:*
$Blue \subseteq \Box(Blue \cup Cyan)$.

Proof. Initially, *Blue* is empty and the proposition holds. States are colored blue at Line 17, at which point all successors t have been considered at Line 13–14. If $t \notin Blue \cup$

Cyan, then *dfsBlue*(t) is executed adding t to *Blue* by the post-condition of Lemma B.1. States are only removed from cyan at Line 18, but after being colored blue at Line 17. □

Corollary B.1. *Lemma B.2 holds for s at Line 17, so at Line 15 also:* $s \in \Box(\mathit{Blue} \cup \mathit{Cyan})$.

Lemma B.3. *In Algorithm B.1, when dfsRed is finished, then all red states have red successors:* $\mathit{Red} \subseteq \Box\mathit{Red}$.

Proof. Follows from induction on red searches initiated at Line 16, which perform reachability on non-red states. □

Lemma B.4. *The red search visits only blue states except for the (cyan) seed.*

Proof. The search at Line 16 starts at the seed $s \in \mathcal{F} \cap \mathit{Cyan}$ (by Line 11 and Lemma B.1), with $s \in \Box(\mathit{Blue} \cup \mathit{Cyan})$ (Corollary B.1). Before the recursive *dfsRed* call at Line 9, if a successor t of s is *Cyan*, Algorithm B.1 terminates at Line 8. Hence, $t \in \mathit{Blue}$ at Line 9. □

Corollary B.2. *All red states are blue or cyan:* $\mathit{Red} \subseteq (\mathit{Blue} \cup \mathit{Cyan})$.

Proof. If a red search was launched at Line 16 ($s \in \mathcal{F}$), states visited by *dfsRed* are colored red at Line 6. By Lemma B.4, the red search only visits blue states except for the seed s, which is cyan. Because states are never removed from $\mathit{Blue} \cup \mathit{Cyan}$ (only from cyan at Line 18, but this is after the state is added to blue at Line 17), always: $\mathit{Red} \subseteq (\mathit{Blue} \cup \mathit{Cyan})$. □

Lemma B.5. *Algorithm B.1, when dfsRed is finished, red states have blue, non-cyan successors:* $\mathit{Red} \subseteq \Box(\mathit{Blue} \setminus \mathit{Cyan})$.

Proof. By Corollary B.1, we have $s \in \Box(\mathit{Blue} \cup \mathit{Cyan})$ for the initial *dfsRed* call at Line 16. By Lemma B.4 and Lemma B.2, we have $t \in \Box(\mathit{Blue} \cup \mathit{Cyan})$ for the recursive *dfsRed* call at Line 9. So always $s \in \Box(\mathit{Blue} \cup \mathit{Cyan})$ at Line 6. If a successor $t \in \mathit{Cyan}$ at Line 8, Algorithm B.1 terminates, so: $s \in \Box\mathit{Blue} \setminus \mathit{Cyan}$ once *dfsRed* returns.

Cyan states can only be added in during the blue search. Assume Line 11 colors a state t cyan, while its predecessor $s \in \mathit{Red}$. By Lemma B.3, also $t \in \mathit{Red}$. Hence by Corollary B.2, $t \in \mathit{Blue} \cup \mathit{Cyan}$, contradicting Lemma B.1. Therefore, $\mathit{Red} \subseteq \Box(\mathit{Blue} \setminus \mathit{Cyan})$. □

Lemma B.6. *In Algorithm B.1, invariantly, blue accepting states are red:* $\mathit{Blue} \cap \mathcal{F} \subseteq \mathit{Red}$.

Proof. A state $s \in \mathcal{F}$ is marked blue at Line 17. There, we have $s \in \mathit{Red}$ because Line 6 happens before Line 17 if $s \in \mathcal{F}$ (see Line 15). □

Lemma B.7. *Algorithm B.1 ensures that blue accepting states never lie on accepting cycles: $\mathit{Blue} \cap \mathcal{F} \cap \mathit{Cycle} = \emptyset$.*

Proof. At Line 17, a state $s \in \mathcal{F}$ is colored blue. By Lemma B.6, we must already have $s \in \mathit{Red}$. Also, $s \in \mathit{Cyan}$ by Line 11 and Lemma B.1. Assume towards a contradiction that a cycle $s \Rightarrow^+ s$ exists. By induction on the length of the cycle, using $\mathit{Red} \subseteq \Box\mathit{Red}$ from Lemma B.3, the immediate predecessor t of s on the cycle has to be red. However, with $s \in \mathit{Cyan}$, contradicting Lemma B.5. □

The following theorems demonstrate the algorithm's correctness, because it always terminates with a report: A cycle report entails that the graph contains an accepting cycle (soundness) and a no cycle report entails that the graph does not contain an acceptance cycle.

Theorem B.1 (Termination)**.** *Algorithm B.1 always terminates with a report.*

Proof. The color set $\mathit{Blue} \cup \mathit{Cyan}$ continuously grows, as only states are added to it (except cyan states which are only remove after being colored blue). This reduces the (finite) number of successors that have to be considered at Line 14 and Line 9. Therefore, both *dfsRed* and *dfsBlue* eventually return, including the initial *dfsBlue* call at Line 3, generating a report at Line 4. Unless a cycle is reported earlier at Line 8. □

Theorem B.2 (Soundness)**.** **report cycle** $\Longrightarrow \exists a \in \mathcal{F} \colon s_0 \Rightarrow^* a \Rightarrow^+ a$

Proof. By property of the DFS stacks: If a red search, started from an accepting seed $s \in \mathcal{F}$ at Line 16, finds a path to a state s' on the cyan stack, there is a path $s' \Rightarrow^* s$. Therefore, there is an accepting cycle: $s_0 \Rightarrow^* s \Rightarrow^+ s' \Rightarrow^* s$. □

Theorem B.3 (Completeness)**.** **report no cycle** $\Longrightarrow \nexists a \in \mathcal{F} \colon s_0 \Rightarrow^* a \Rightarrow^+ a$

Proof. At Line 4, $s_0 \in \mathit{Blue}$ by Line 17, and $\mathit{Cyan} = \emptyset$ (always Line 18 after Line 11). By Lemma B.2, all states are blue, hence no accepting cycle exists by Lemma B.7. □

The following corollaries illustrate the working of the NDFS algorithm more intuitively: For each accepting state (Corollary B.4), a red search is launched to find a path back to the cyan stack closing the accepting cycle. Th search may ignore states visited by previous red searches (red states) as these do not lead to accepting cycles Corollary B.5, making the algorithm linear.

Corollary B.3. *At Line 13, $\mathit{Red} \subseteq \mathit{Blue}$.*

Proof. Lemma B.4 and Line 17 happens after the initial *dfsRed* call at Line 16. □

Corollary B.4. *The red search visits one single accepting state: the seed.*

Proof. The search starts at Line 16 in a cyan seed s and then visits only blue states at Line 9 (Lemma B.4). Assuming that it also visits some $t \in \mathcal{F}$ with $s \neq t$, we have $t \in \mathit{Red}$ by Lemma B.6 contradicting the condition $t \notin \mathit{Red}$ at Line 9. □

Corollary B.5. *Outside of dfsRed, no red state leads to an accepting cycle.*

Proof. Outside of *dfsRed*, we have $\mathit{Red} \subseteq \Box\mathit{Red}$ by Lemma B.3. Assume towards a contradiction that there exists a state $s \in \mathit{Red}$ that leads to an accepting cycle. By Lemma B.5, we have $\mathit{Red} \subseteq \Box\mathit{Blue}$. By induction on the lasso from s, we learn that the cycle is both blue and red. However, this contradicts Lemma B.7. □

Corollary B.6. *dfsRed is not dependent on DFS order, it can be implemented with any reachability algorithm (that marks visited states red).*

Corollary B.7. *The correctness of Algorithm 5.2 follows from the correctness of Algorithm B.1.*

Proof. The blue and cyan sets are disjoint except at Line 17. From Corollary B.2, it follows that the color red may override blue and cyan (as happens at Line 18 and Line 9 in Algorithm 5.2). Care has to be taken however that the converse override does not happen (see Line 18 and Line 20 in Algorithm 5.2), and red is also interpreted as blue/cyan (see Line 14 in Algorithm 5.2). □

B.2 Correctness Proof for Ndfs with Subsumption

In the current section, we prove Algorithm B.2 (a copy of Algorithm 10.2) correct, using the same notations as in Section B.1 and also reusing some of the previous lemmas. In fact, we only repeat those lemma that required modification now that subsumption is added to the algorithms. Because we use subsumption, the abstracted transition relation $\Rightarrow$ is used, instead of the normal relation $\rightarrow$ (the two are no longer interchangeable). Notice that the algorithm now avoids red states during the blue search as this might prune the search space Section 10.4.

As in Chapter 10, we write $s \sqsubseteq S$ to express subsumption checks on sets, meaning $\exists s' \in S\colon\ s \sqsubseteq s'$. And $S \sqsubseteq s$, meaning $\exists s' \in S\colon\ s' \sqsubseteq s$. We write $X_{\sqsubseteq}$ for all states *subsumed by* states in X: $X_{\sqsubseteq} = \{s \mid s \sqsubseteq X\}$, i.e. all states that have equal or less behavior than X. We also write $X_{\sqsupseteq}$ for all states *that subsume* states in X: $X_{\sqsupseteq} = \{s \mid X \sqsubseteq s\}$, i.e. all states that have equal or more behavior than X.

Algorithm B.2 NDFS with subsumption

```
1:  procedure NDFS()
2:      $Cyan := Blue := Red := \emptyset$
3:      $dfsBlue(s_0)$
4:      report no cycle
5:  procedure $dfsRed(s)$
6:      $Red := Red \cup \{s\}$
7:      for all $t$ in NEXT-STATE$(s)$ do
8:          if $\underline{Cyan \sqsubseteq t}$ then report cycle
9:          if $\underline{t \not\sqsubseteq Red}$ then $dfsRed(t)$
10: procedure $dfsBlue(s)$
11:     $Cyan := Cyan \cup \{s\}$
12:     for all $t$ in NEXT-STATE$(s)$ do
13:         if $(t \notin Blue \cup Cyan \underline{\wedge t \not\sqsubseteq Red})$
14:         then $dfsBlue(t)$
15:     if $s \in \mathcal{F}$ then
16:         $dfsRed(s)$
17:     $Blue := Blue \cup \{s\}$
18:     $Cyan := Cyan \setminus \{s\}$
```

Lemma B.8 (Pre-/post-conditions *dfsBlue*)**.** *Next to the pre- and post-conditions of Lemma B.1, Algorithm B.2 also ensures that dfsBlue is not called on states subsumed by red:* $\{s \notin (Red_{\sqsubseteq})\}$ *dfsBlue*(s) $\{\}$

Proof. *dfsBlue*(s) is only called on white states by Line 2 and on states not subsumed by red by Line 13. Hence, $s \notin Red_{\sqsubseteq}$. □

Lemma B.9. *In Algorithm B.2, successors of blue states are blue, cyan or subsumed by red:* $Blue \subseteq \Box(Blue \cup Cyan \cup Red_{\sqsubseteq})$.

Proof. Initially, *Blue* is empty and the proposition holds. States are colored blue at Line 17, at which point all successors t have been considered at Line 13–14. If $t \notin$

$Blue \cup Cyan \cup Red_{\sqsubseteq}$ then $dfsBlue(t)$ is executed adding t to *Blue* by the post-condition of Lemma B.1. States are only removed from cyan at Line 18, but after being colored blue at Line 17. □

Corollary B.8. *Lemma B.9 holds for s at Line 17, so at Line 15 also:*
$s \in \subseteq \Box(Blue \cup Cyan \cup Red_{\sqsubseteq})$.

The following shows, that the red search never breaches the bounds of the blue search (the cyan stack). If the blue search where to employ subsumption, this would not be the case, as shown in Figure 10.6.

Lemma B.10. *A pre-condition of* $dfsRed(s)$ *is that* $s \in Blue \cup Cyan$ *and that all successors of s are either blue or cyan, or subsumed by red:*
$\{s \in Cyan \cup Blue \wedge s \in \Box(Blue \cup Cyan \cup Red_{\sqsubseteq})\}\ dfsRed(s)\ \{s \in Red \wedge s \not\in \Box Cyan_{\sqsupseteq}\}$.
While a post-condition is that s is red and its successors do not subsume cyan states.

Proof. By induction on the number of *dfsRed* calls, we show that the pre-conditions hold.

$n = 1$: The initial *dfsRed* call must be from Line 16 with the seed $s \in \mathcal{F} \cap Cyan$ (by Line 11 and Lemma B.1). Also, $s \in (Blue \cup Cyan \cup Red_{\sqsubseteq})$ by Corollary B.8.

$n = n' + 1$ The nth $dfsRed(t)$ call must be from Line 9. Assume that for the previous n'th cal, we have $s \in \Box(Blue \cup Cyan \cup Red_{\sqsubseteq})$. We consider the successors t of s, which are all processed at Line 7. Therefore, $t \in (Blue \cup Cyan \cup Red_{\sqsubseteq})$. Any $t \in Red_{\sqsubseteq}$ it is discarded by Line 9. If $t \in Cyan$, then Algorithm B.2 terminates at Line 8. Hence, $t \in Blue$ at Line 9 for the nth call. And by Lemma B.9, $t \in \Box(Blue \cup Cyan \cup Red_{\sqsubseteq})$.

The post-condition holds by Line 6 and the fact that states are never uncolored red.

In both cases of the induction, $dfsRed(s)$ terminates when a successor $t \in Cyan_{\sqsupseteq}$. Because the red search does not change the cyan set, we have $s \not\in \Box Cyan_{\sqsupseteq}$ upon return of *dfsRed*. □

Corollary B.9. *As states are never removed from* $Blue \cup Cyan$ *nor from Red, it follows from Lemma B.10 that:* $Red \subseteq (Blue \cup Cyan)$.

Lemma B.11. *In Algorithm B.2, when dfsRed is finished, then states subsumed by red states have successor subsumed by red states:*
$\{Red_{\sqsubseteq} \subseteq \Box Red_{\sqsubseteq}\}\ dfsBlue(s)\ \{Red_{\sqsubseteq} \subseteq \Box Red_{\sqsubseteq}\}$.

Proof. We have $s \in \mathit{Red}$ by Line 6 and red states are never uncolored red. For all recursive calls at Line 9 $t \in \mathit{Red}$ by the post-conditions of Lemma B.10. Otherwise, $t \in \mathit{Red}_{\sqsubseteq}$ by Line 9. Therefore, $s \in \mathit{Red} \cap \Box \mathit{Red}_{\sqsubseteq}$ upon return of *dfsRed*. The fact that subsumption $\sqsubseteq$ is a simulation relation (Proposition 10.1) ensures that for states $s' \sqsubseteq s$, also $s' \in \Box \mathit{Red}_{\sqsubseteq}$. Hence, $\mathit{Red}_{\sqsubseteq} \subseteq \Box \mathit{Red}_{\sqsubseteq}$ if *dfsRed* is completed. □

In the following lemma, the notation $\mathit{dfsBlue}(s)@n$ refers to Line n in the code of Algorithm B.2.

Lemma B.12. *Algorithm B.2, when dfsBlue returns, red states have non-cyan successors:* $\mathit{dfsBlue}(s)@17 \implies \mathit{Red}_{\sqsubseteq} \cap \Box \mathit{Cyan}_{\sqsupseteq} = \emptyset$.

Proof. A state s can only be marked red in $\mathit{dfsRed}(s)$. It is never uncolored red. Upon its return, $s \in \mathit{Red}$ and $s \notin \Box \mathit{Cyan}_{\sqsupseteq}$ by Lemma B.10. Because the red search does not tamper with the set of cyan states, we have $\mathit{Red} \cap \Box \mathit{Cyan}_{\sqsupseteq} = \emptyset$ up on completion of the red search, when the control flow returned to Line 17.

Only the blue search can color states cyan. Assume there is some state $r \in \mathit{Red}$, whose successor s is colored cyan at Line 11. By Lemma B.11, we have $s \in \mathit{Red}_{\sqsubseteq}$, contradicting Lemma B.8.

It follows that $\mathit{Red} \cap \Box \mathit{Cyan}_{\sqsupseteq} = \emptyset$. By the fact that $\sqsubseteq$ is a simulation relation (Proposition 10.1), we also have: $\mathit{Red}_{\sqsubseteq} \cap \Box \mathit{Cyan}_{\sqsupseteq} = \emptyset$. Or intuitively: if red states have no successors subsuming cyan states, all states with less behavior than the red states, can also not subsume cyan states. □

Lemma B.13. *Algorithm B.2 ensures that blue accepting states never lie on accepting cycles:* $\mathit{Blue} \cap \mathcal{F} \cap \mathit{Cycle} = \emptyset$.

Proof. At Line 17, a state $s \in \mathcal{F}$ is colored blue. By Lemma B.6, we must already have $s \in \mathit{Red}$. Also, $s \in \mathit{Cyan}$ by Line 11 and Lemma B.8. Assume towards a contradiction that a cycle $s \Rightarrow^{+} s$ exists. By induction on the length of the cycle, using Lemma B.11, we find that the whole cycle must be subsumed by red. Therefore, also for the predecessor s' of s on the cycle, such that $s' \Rightarrow s$, we have $s' \sqsubseteq \mathit{Red}$. With $s \in \mathit{Cyan}$, this contradicts Lemma B.12. □

The following theorems demonstrate the algorithm's correctness, because it always terminates with a report: A cycle report entails that the graph contains an accepting cycle (soundness) and a no cycle report entails that the graph does not contain an acceptance cycle.

Theorem B.4 (Termination)**.** *Algorithm B.2 always terminates with a report.*

Proof. The color sets *Red* and $Blue \cup Cyan$ continuously grows, as only states are added to it (except cyan states which are only remove after being colored blue). This reduces the – by Proposition 10.3 finite – number of successors that have to be considered at Line 14 and Line 9. Therefore, both *dfsRed* and *dfsBlue* eventually return, including the initial *dfsBlue* call at Line 3, generating a report at Line 4. Unless a cycle is reported earlier at Line 8. □

Theorem B.5 (Soundness)**. report cycle** $\implies \exists a \in \mathcal{F}\colon s_0 \Rightarrow^* a \Rightarrow^+ a$

Proof. By Line 8, we now find a path: $s_0 \Rightarrow^* s \Rightarrow^+ s'$ s.t. $s \sqsubseteq s'$ containing an accepting state on the path $s \Rightarrow^+ s'$. By Lemma 10.2, this implies the existence of an accepting cycle. □

Theorem B.6 (Completeness)**. report no cycle** $\implies \nexists a \in \mathcal{F}\colon s_0 \Rightarrow^* a \Rightarrow^+ a$

Proof. By induction on the length of the path $s_0 \Rightarrow^* s_n$ to any reachable state s_n, using Lemma B.9 and Corollary B.9, we show first that $s \sqsubseteq Blue$. For this we also use the fact that at Line 4 $Cyan = \emptyset$ by Line 2 and Lemma B.8.

$n = 0$: At Line 4, $s_0 \in Blue$ by Line 3 and the post-condition of Lemma B.1.

$n = n' + 1$: Assume $s_{n'} \in Blue$. By Lemma B.9, its successor must be: $s_n \in (Blue \cup Red_{\sqsubseteq})$, because $Cyan = \emptyset$. If $s_n \in Red_{\sqsubseteq}$, then $\exists s' \sqsupseteq s\colon s \in Red$. By Corollary B.9 also $s' \in Blue$, because $Cyan = \emptyset$. Therefore, $s_n \sqsubseteq Blue$.

Hence, all reachable states are subsumed by blue.

Now we consider all reachable accepting states $a \in \mathcal{F}$, again at Line 4. It holds that $a \sqsubseteq Blue$, in other words, there exists another accepting state $a' \in Blue$, such that $a \sqsubseteq a'$. By Proposition 10.4: $a, a' \in \mathcal{F}$. Assume, a lies on a cycle. Since $a' \in Blue$, it does not lie on an accepting cycle according to Lemma B.13. This contradicts with the contraposition of Lemma 10.1, which transliterates as: If s' does not lead to an accepting cycle, then any $s \sqsubseteq s'$ does also not lead to an accepting cycle. □

B.3 Correctness Proof for CNDFS with Subsumption

In the current section, we prove the parallel Algorithm B.3 (a copy of Algorithm 10.3) correct, using the same notations as in Section B.2 (also the same preliminary remarks apply). We again reuse the notations from Chapter 10, and also use the subsumption lemmas from that chapter. A proof for the version without subsumption was presented in Chapter 7. The exact parts in the lemmas and proofs that changed is underlined here (except for the cases where we use the abstracted transition relation $\Rightarrow$, instead of the normal relation $\rightarrow$).

Algorithm B.3 Multi-core CNDFS with subsumption on *Red* and over *Cyan*

```
1:  procedure CNDFS($P$)
2:      $Blue := Red := \emptyset$
3:      forall $i$ in $1..P$ do $Cyan_i := \emptyset$
4:      $dfsBlue_1(s_0) \| .. \| dfsBlue_P(s_0)$
5:      report no cycle
6:  procedure $dfsRed_i(s)$
7:      $\mathcal{R}_i := \mathcal{R}_i \cup \{s\}$
8:      for all $t$ in NEXT-STATE$_i(s)$ do
9:          if $\underline{Cyan \sqsubseteq t}$ then
10:             report cycle
11:         if $t \notin \mathcal{R}_i \wedge \underline{t \not\sqsubseteq Red}$ then
12:             $dfsRed_i(t)$
13: procedure $dfsBlue_i(s)$
14:     $Cyan_i := Cyan_i \cup \{s\}$
15:     for all $t$ in NEXT-STATE$_i(s)$ do
16:         if $t \notin Cyan_i \cup Blue \underline{\wedge t \not\sqsubseteq Red}$ then
17:             $dfsBlue_i(t)$
18:     $Blue := Blue \cup \{s\}$
19:     if $s \in \mathcal{F}$ then
20:         $\mathcal{R}_i := \emptyset$
21:         $dfsRed_i(s)$
22:         await $\forall s' \in \mathcal{R}_i \cap \mathcal{F} \setminus \{s\} : \underline{s' \sqsubseteq Red}$
23:         forall $s'$ in $\mathcal{R}_i$ do $Red := Red \cup s'$
24:     $Cyan_i := Cyan_i \setminus \{s\}$
```

Lemma B.14. *States subsumed by red have successors subsumed by red: $\underline{Red_{\sqsubseteq} \subseteq \Box Red_{\sqsubseteq}}$.*

Proof. Initially, there are no red states, hence the lemma holds.

States are colored red when $dfsBlue_p$@23 and are never uncolored red. The set of states $\mathcal{R}_p$ that is colored at Line 23 contains all states reachable from the seed s, but not yet subsumed by red, since $dfsRed_p(s)$ performed a DFS from s over all states not subsumed by red. For the states subsumed by red and reachable from s, the induction hypothesis can be applied, hence there are states subsumed by red that are reachable from s that are not in $\mathcal{R}_p$. As a consequence, always $Red \subseteq \Box Red_{\sqsubseteq}$.

The fact that subsumption $\sqsubseteq$ is a simulation relation (Proposition 10.1) ensures that for states $s' \sqsubseteq s$ with $s \in \Box Red_{\sqsubseteq}$, also $s' \in \Box Red_{\sqsubseteq}$. Hence, it holds that $Red_{\sqsubseteq} \subseteq \Box Red_{\sqsubseteq}$. □

Lemma B.15. *At Line 22, the set $\mathcal{R}_p$ invariably contains (1) the seed s, (2) all non-red states reachable from s and also (3) all states in the set are reachable from the seed s: $dfsBlue_p(s)@22 \Rightarrow (s \in \mathcal{R}_p \wedge (\forall s' \notin Red_{\sqsubseteq} : s \Rightarrow^* s' \Rightarrow s' \in \mathcal{R}_p) \wedge (\forall s'' \in \mathcal{R}_p \Rightarrow s \Rightarrow^* s''))$.*

Proof. At Line 7, we have $s \in \mathcal{R}_p$. For the rest, see proof of Lemma B.14. □

Lemma B.16. *The only accepting state that can be colored red at Line 23 (for the first time) is the current seed s itself: $dfsBlue_p(s)@23 \Rightarrow (\mathcal{R}_p \cap \mathcal{F}) \setminus Red_{\sqsubseteq} \subseteq \{s\}$.*

Proof. Assume $dfsBlue_p(s)@23$ and $\exists a \in (\mathcal{F} \setminus \{s\}) : a \in \mathcal{R}_p$. We show that $a \in Red_{\sqsubseteq}$.

By Lemma B.15, $\mathcal{R}_p$ contains at least s and the states reachable from s and not subsumed by red. After Line 22, all non-seed accepting states in $\mathcal{R}_p$ are subsumed by red: $(\mathcal{R}_p \cap (\mathcal{F} \setminus \{s\})) \subseteq Red_{\sqsubseteq}$. Since, $a \in \mathcal{R}_p \cap (\mathcal{F} \setminus \{s\})$, we have: $a \in Red_{\sqsubseteq}$. □

Proposition B.1. *The initial invocation of $dfsRed_p(s)$ at Line 21 of Algorithm B.3 reports a cycle if and only if the seed s belongs to a cycle.*

Proof. $\Leftrightarrow$ is split into two cases:

Case $\Rightarrow$: Every state $s' \in Cyan_p$ can reach the seed from $dfsBlue_p(s)@21$ by properties of the DFS stack. Similarly, when $dfsRed_p(s'')@10$, $s'' \sqsupseteq Cyan_p$ is reachable from the seed s. Therefore, there is a path: $s \Rightarrow^* s''$ for some $s'' \sqsupseteq c \in Cyan_p$ and $c \Rightarrow^* s$. By Lemma 10.2, there is an accepting cycle.

Case $\Leftarrow$: assume $dfsRed_p(s)$ at Line 21 finishes normally (without cycle report), while s lies on a cycle C. We show this leads to a contradiction. Since *dfsRed* avoids only states subsumed by red (Line 11), there would have to be some $r \in C \cap Red_{\sqsubseteq}$ obstructing the search. In other words, there is a state $r' \sqsubseteq r$ such that $r' \in Red$. The state r' can only have been colored red at Line 23 by some worker. W.l.o.g. we investigate the first worker $dfsRed_{p'}$ to have colored r' red. p' started for an $s' \in \mathcal{F}$ ($dfsBlue_{p'}(s')$@Line 21).

Since r' is not yet red, by Lemma B.14 $C \cap Red_{\sqsubseteq} = \emptyset$. Before r is colored red, it is first stored in $\mathcal{R}_{p'}$. By Lemma B.15, we also have $C \sqsubseteq \mathcal{R}_{p'}$. Either there is some $s'' \in C$ such that $s' \sqsubseteq s''$, then the cycle through s' would have been detected since $s'' \in Cyan_{p'}$ (use contraposition Lemma 10.1). Or else there is no such $s'' \in C$, and then we have $\{s\} \sqsubseteq (\mathcal{R}_{p'} \setminus Red_{\sqsubseteq})$ when $dfsBlue_{p'}(s')@23$, contradicting Lemma B.16. □

Proposition B.2. *Red states never subsume an accepting cycle.*

Proof. Initially, there are no red states, hence the proposition holds.

When $dfsBlue_p(s)@23$, the set of states $\mathcal{R}_p$ is colored red. The only accepting state to be colored red is the seed s (Lemma B.16). By Proposition B.1, this state s does not lie on an accepting cycle. Hence, Proposition B.2 is preserved.

It follows that there is no state subsumed by s, i.e. $s' \sqsubseteq s$, which does lie on an accepting cycle. Otherwise, we would have a contradiction with the contraposition of Lemma 10.1, which transliterates as: If s' does not lead to an accepting cycle, then any $s \sqsubseteq s'$ does also not lead to an accepting cycle. □

Lemma B.17. *Blue states have successors that are blue, subsumed by red, or cyan for some worker p:* $Blue \subseteq \bigcup_p \Box(Blue \cup Cyan_p \cup Red_{\sqsubseteq})$.

Proof. Initially there are no blue states, hence the lemma holds.

Only at Line 18, states are colored blue, after each successor t has been skipped at Line 16 ($t \in Cyan_p \cup Blue \cup Red_{\sqsubseteq}$), or processed by $dfsBlue_p$ at Line 17 (leading to $t \in Blue$). States can be uncolored cyan (Line 24), but only after they have been colored blue (Line 18). □

Lemma B.18. *A blue accepting state, that is not also $Cyan_p$ for some worker p, must be red:* $\forall a \in (Blue \cap \mathcal{F}) : (\forall p \in \{1 \dots P\} : a \notin Cyan_p) \Rightarrow a \sqsubseteq Red$.

Proof. Assume $s \in (\mathcal{F} \cap Blue)$ and $\forall p \in \{1 \dots P\} : s \notin Cyan_p$. We show that $s \sqsubseteq Red$.

State s can only be colored blue when $dfsBlue_p(s)@18$. There, it still retains its cyan coloring from Line 14, it only loses this color at Line 24. But, since $s \in \mathcal{F}$, Line 23 was reached and there $a \in \mathcal{R}_p$ by Lemma B.15. Hence, $s \sqsubseteq Red$ at Line 24. □

Proposition B.3. *Algorithm 1 always terminates with a report.*

Proof. The individual DFSs cannot proceed indefinitely due to a growing set of red and blue states, and the fact that we only consider finite abstractions Proposition 10.3. So eventually a cycle (Line 10) or no cycle is reported (Line 5). However, progress may also halt due to the wait statement at Line 22. We now assume towards a contradiction that a worker p is waiting indefinitely for a state $a \in \mathcal{F}$ to become subsumed by red: $dfsBlue_p(s)@22$, $s \neq a$ and $a \in \mathcal{R}_p$. We will show that either a will be subsumed by red eventually, or a cycle would have been detected, contradicting the assumption that p keeps waiting.

By Lemma B.15, a is reachable from s: $s \Rightarrow^+ a$. And by Line 18, $s \in Blue$. Induction on the path $s \Rightarrow^* a$, using Lemma B.17, tells us that: either all states are blue (1), or there is a cyan state on this path (2), or there is a state subsumed by red on this path (3):

1. $a \in Blue \wedge \forall p \in \{1 \dots P\}$: $a \notin Cyan_p$: by Lemma B.18, $a \in Red$, which contradicts the assumption that p is waiting for a to become red. (Note that $\exists p' \in \{1 \dots P\}$: $a \in Cyan_{p'}$ is handled in Case 2.)
2. $\exists c \in Cyan_{p'} : s \Rightarrow^+ c \Rightarrow^* a$, then depending on the identity of worker p', we have:

A) $p = p'$: but then $\mathit{dfsRed}_p(s)$ would have terminated on cycle detection ($C \equiv s \Rightarrow^+ c \Rightarrow^+ s$), except when dfsRed_p did not reach c in presence of a red state subsuming C. However, this would contradict Proposition B.2.

B) $p \neq p'$: we show that either p' is executing or going to execute $\mathit{dfsRed}_{p'}(a)$. To eventually color state a red, worker p' must not end up itself in a waiting state: $\mathit{dfsBlue}_{p'}(a')@22$. First, consider the case $a \neq a'$. We also have $s \Rightarrow^+ c \Rightarrow^* a'$ (stack $\mathit{Cyan}_{p'}$). Hence, by Lemma B.15, also $a' \in \mathcal{R}_p$. Therefore, we can assume w.l.o.g. that $a = a'$ and only consider $\mathit{dfsBlue}_{p'}(a)@22$. We can repeat the reasoning process of this proof, with $p \equiv p'$ and $s \equiv a$. But since there are finitely many workers, the chain of processes waiting for each other eventually terminates, except the hypothetical configuration of a cyclic waiting dependency, which we consider finally.

3. By induction on the length of the path, using Lemma B.14, we learn that $a \sqsubseteq \mathit{Red}$. Contradicting the assumption that p is waiting for a.

To exclude cyclic dependencies, assume $n \geq 2$ workers are simultaneously waiting for each other's seed to be colored red at Line 22. We have: $\mathit{dfsBlue}_1(s_1)@22 \land \cdots \land \mathit{dfsBlue}_n(s_n)@22 \land s_2 \in \mathcal{R}_1 \land \cdots \land s_1 \in \mathcal{R}_n$. This is only possible if $s_1 \Rightarrow^+ s_n \land \cdots \land s_n \Rightarrow^+ s_1$, hence there is a cycle: $s_1 \Rightarrow^+ \ldots \Rightarrow^+ s_n \Rightarrow^+ s_1$. However, this contradicts that the red DFSs (which terminate anyway) would have detected this cycle (Proposition B.1). □

Theorem B.7. *Algorithm B.3 reports an accepting cycle if and only if one is reachable from s_0.*

Proof. By Proposition B.3, the algorithm is guaranteed to terminate with some report, forming the basis for two cases:

Case $\Rightarrow$: $\mathit{dfsRed}_p(s)@10$ implies that there is an accepting cycle according to Proposition B.1.

Case $\Leftarrow$ (consider the entire case underlined): At Line 5, we have $s_0 \in \mathit{Blue}$ and $\forall p \in \{1 \ldots P\}: \mathit{Cyan}_p = \emptyset$ by Line 18 and by properties of DFS.

By induction on the length n of the path, $s_0 \Rightarrow s_n$, using Lemma B.17 and Lemma B.14, we show that all reachable states s_n are red or blue: $s_n \in (\mathit{Blue} \cup \mathit{Red}_{\sqsubseteq})$.

$n = 0$: At Line 4, $s_0 = s_n$. Therefore, $s_0, s_n \in \mathit{Blue}$.

$n = n' + 1$: Assume $s_{n'} \in (\mathit{Blue} \cup \mathit{Red}_{\sqsubseteq})$. If $s_{n'} \in \mathit{Blue}$, by Lemma B.9, its successor must be: $s_n \in (\mathit{Blue} \cup \mathit{Red}_{\sqsubseteq})$, because the cyan sets are empty. If $s_{n'} \in \mathit{Red}_{\sqsubseteq}$, by Lemma B.14, its successor must be: $s_n \sqsubseteq \mathit{Red}$.

Hence, all reachable states are blue or subsumed by red.

Now we consider all reachable accepting states $a \in \mathcal{F}$, again at Line 4. If $a \in \mathit{Blue}$, then also $a \sqsubseteq \mathit{Red}$ by Lemma B.18. So all accepting states are subsumed by red: $\mathcal{F} \sqsubseteq \mathit{Red}$. By Proposition B.2, it follows that there are no accepting cycles. □

This concludes correctness of Algorithm B.3.

Publications from the Author

12.3 Publications on Formal Methods

[1] F. I. van der Berg and A. W. Laarman. “SpinS: Extending LTSmin with Promela through SpinJa.” In: *Electronic Notes in Theoretical Computer Science* 296 (2013), pp. 95 –105. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2013.07.007.

[2] A. E. Dalsgaard, A. W. Laarman, K. G. Larsen, M. C. Olesen, and J. C. van de Pol. “Multi-core Reachability for Timed Automata.” In: *FORMATS'12*. Ed. by M. Jurdziński and D. Ničković. Vol. 7595. LNCS. Springer, 2012, pp. 91–106. ISBN: 978-3-642-33364-4. DOI: 10.1007/978-3-642-33365-1_8.

[3] T. van Dijk, A. W. Laarman, and J. C. van de Pol. “Multi-core and/or symbolic model checking.” In: *AVoCS 2012*. Ed. by G. Luettgen and S. Merz. Vol. 53. Electronic Communications of the EASST. Bamberg, Germany: EASST, 2012, 773:1–773:7. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/773.

[4] T. van Dijk, A. W. Laarman, and J. C. van de Pol. “Multi-Core BDD Operations for Symbolic Reachability.” In: *Electronic Notes in Theoretical Computer Science* 296.0 (2013). Proceedings of PASM/PDMC, pp. 127 –143. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2013.07.009.

[5] S. Evangelista, A. W. Laarman, L. Petrucci, and J. C. van de Pol. “Improved Multi-Core Nested Depth-First Search.” In: *ATVA'12*. Ed. by S. Ramesh. Vol. 7561. LNCS. Thiruvananthapuram (Trivandrum), Kerala: Springer, 2012, pp. 269–283. DOI: 10.1007/978-3-642-33386-6_22.

[6] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. J. Wijs. "Multi-core Nested Depth-First Search." In: *Automated Technology for Verification and Analysis*. Ed. by T. Bultan and P.-A. Hsiung. Vol. 6996. LNCS. Springer Berlin Heidelberg, 2011, pp. 321–335. ISBN: 978-3-642-24371-4. DOI: 10.1007/978-3-642-24372-1_23.

[7] A. W. Laarman, E. Pater, J. C. van de Pol, and M. Weber. "Guard-Based Partial-Order Reduction." In: *Model Checking Software*. Ed. by E. Bartocci and C. R. Ramakrishnan. Vol. 7976. LNCS. Springer Berlin Heidelberg, 2013, pp. 227–245. ISBN: 978-3-642-39175-0. DOI: 10.1007/978-3-642-39176-7_15.

[8] A. W. Laarman, M. C. Olesen, A. E. Dalsgaard, K. G. Larsen, and J. C. van de Pol. "Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction." In: *Computer Aided Verification*. Ed. by N. Sharygina and H. Veith. Vol. 8044. LNCS. Springer Berlin Heidelberg, 2013, pp. 968–983. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_69.

[9] A. W. Laarman and D. Faragó. "Improved on-the-Fly Livelock Detection." In: *NASA Formal Methods*. Ed. by G. Brat, N. Rungta, and A. Venet. Vol. 7871. LNCS. Springer Berlin Heidelberg, 2013, pp. 32–47. ISBN: 978-3-642-38087-7. DOI: 10.1007/978-3-642-38088-4_3.

[10] A. W. Laarman and J. C. van de Pol. "Variations on Multi-Core Nested Depth-First Search." In: Proceedings 10th International Workshop on *Parallel and Distributed Methods in verifiCation,* Snowbird, Utah, USA, July 14, 2011. Ed. by J. Barnat and K. Heljanko. Vol. 72. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2011, pp. 13–28. DOI: 10.4204/EPTCS.72.2.

[11] A. W. Laarman, J. C. van de Pol, and M. Weber. "Boosting Multi-Core Reachability Performance with Shared Hash Tables." In: *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland*. Ed. by N. Sharygina and R. Bloem. Lugano, Switzerland: IEEE Computer Society, 2010. URL: http://dl.acm.org/citation.cfm?id=1998496.1998541.

[12] A. W. Laarman, J. C. van de Pol, and M. Weber. "Multi-Core LTSmin: Marrying Modularity and Scalability." In: *NASA Formal Methods*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Vol. 6617. LNCS. Pasadena, CA, USA: Springer Verlag, 2011, pp. 506–511. DOI: 10.1007/978-3-642-20398-5_40.

[13] A. W. Laarman, J. C. van de Pol, and M. Weber. “Parallel Recursive State Compression for Free.” In: *Model Checking Software*. Ed. by A. Groce and M. Musuvathi. Vol. 6823. LNCS. Springer Berlin Heidelberg, 2011, pp. 38–56. ISBN: 978-3-642-22305-1. DOI: 10.1007/978-3-642-22306-8_4.

[14] S. van der Vegt and A. W. Laarman. “A Parallel Compact Hash Table.” In: *Mathematical and Engineering Methods in Computer Science*. Ed. by Z. Kotásek, J. Bouda, I. Černá, L. Sekanina, T. Vojnar, and D. Antoš. Vol. 7119. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 191–204. ISBN: 978-3-642-25928-9. DOI: 10.1007/978-3-642-25929-6_18.

12.3 Publications on Software Engineering

[15] A. W. Laarman and I. Kurtev. “Ontological Metamodeling with Explicit Instantiation.” In: *Software Language Engineering*. Ed. by M. Brand, D. Gašević, and J. Gray. Vol. 5969. LNCS. Springer Berlin Heidelberg, 2010, pp. 174–183. ISBN: 978-3-642-12106-7. DOI: 10.1007/978-3-642-12107-4_14.

[16] A. W. Laarman. “Improving a Modular Verification Technique for Aspect Oriented Programming.” In: *5th Twente Student Conference on Information Technology, Enschede, The Netherlands*. Ed. by C. Huijs. Vol. 5. Twente Student Conference on IT. Enschede, The Netherlands: Twente University Press, 2006, pp. 37–40. URL: http://eprints.eemcs.utwente.nl/15698/.

[17] A. W. Laarman. “Achieving QVTO & ATL Interoperability: An Experience Report on the Realization of a QVTO to ATL Compiler.” In: *1st International Workshop on Model Transformation with ATL, MtATL 2009, Nantes, France*. Ed. by F. Jouault. CEUR Workshop Proceedings. Nantes, France: Sun SITE Central Europe, 2009, pp. 119–133. URL: http://eprints.eemcs.utwente.nl/15696/.

12.3 Technical Reports

[18] A. W. Laarman, J. C. van Pol, and M. Weber. “Boosting Multi-Core Reachability Performance with Shared Hash Tables.” In: *ArXiv e-prints* 1004.2772 (Apr. 2010). arXiv: 1004.2772.

[19] A. W. Laarman, J. C. van de Pol, and M. Weber. “Parallel Recursive State Compression for Free.” In: *ArXiv e-prints* abs/1104.3119 (2011). arXiv: 1104.3119.

Supervised Theses by the Author

12.3 Master Theses

[1] F. I. van der Berg. “Model Checking LLVM IR using LTSmin.” MSc. Thesis. University of Twente, 2013. URL: http://fmt.cs.utwente.nl/education/master/190/.

[2] R. Burgman. “Partial-order reduction based on probe sets.” MSc. Thesis. University of Twente, 2012. URL: http://essay.utwente.nl/62506/.

[3] T. van Dijk. “The parallelization of binary decision diagram operations for model checking.” MSc. Thesis. University of Twente, 2012. URL: http://essay.utwente.nl/61650/.

12.3 Bachelor Theses

[4] S. van der Vegt. “A Concurrent Bidirectional Linear Probing Algorithm.” In: *15th Twente Student Conference on Information Technology, Enschede, The Netherlands*. Ed. by C. Heijnen and H. Koppelman. Vol. 15. TSConIT. Enschede, The Netherlands: Twente University Press, 2011, pp. 269–276. URL: http://referaat.cs.utwente.nl/conference/15/paper.

References

[Abd+96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. "General Decidability Theorems for Infinite-State Systems." In: *LICS*. 1996, pp. 313–321. DOI: `10.1109/LICS.1996.561359`.

[ABH13] A. Amighi, D. S. C. C. Blom, and D. M. Huisman. *Resource protection using atomics: patterns and verifications*. Tech. rep. TR-CTIT-13-10. Enschede, the Netherlands, 2013. URL: `http://doc.utwente.nl/85886/`.

[AD94] R. Alur and D. L. Dill. "A theory of timed automata." In: *Theoretical Computer Science* 126.2 (1994), pp. 183 –235. ISSN: 0304-3975. DOI: `http://dx.doi.org/10.1016/0304-3975(94)90010-8`.

[AFS04] L. Alfaro, M. Faella, and M. Stoelinga. "Linear and Branching Metrics for Quantitative Transition Systems." In: *Automata, Languages and Programming*. Ed. by J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella. Vol. 3142. LNCS. Springer Berlin Heidelberg, 2004, pp. 97–109. ISBN: 978-3-540-22849-3. DOI: `10.1007/978-3-540-27836-8_11`.

[Aga+10] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. "Scalable Graph Exploration on Multicore Processors." In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: `10.1109/SC.2010.46`.

[AK74] O. Amble and D. E. Knuth. "Ordered hash tables." In: *The Computer Journal* 17.2 (1974), pp. 135–142. DOI: `10.1093/comjnl/17.2.135`.

[AKT13] J. Alglave, D. Kroening, and M. Tautschnig. “Partial Orders for Efficient Bounded Model Checking of Concurrent Software.” In: *Computer Aided Verification*. Ed. by N. Sharygina and H. Veith. Vol. 8044. LNCS. Springer Berlin Heidelberg, 2013, pp. 141–157. isbn: 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_9.

[Alu99] R. Alur. “Timed Automata.” English. In: *Computer Aided Verification*. Ed. by N. Halbwachs and D. Peled. Vol. 1633. LNCS. Springer Berlin Heidelberg, 1999, pp. 8–22. isbn: 978-3-540-66202-0. doi: 10.1007/3-540-48683-6_3.

[Ami+12] A. Amighi, S. C. C. Blom, M. Huisman, and M. Zaharieva-Stojanovski. “The VerCors Project: Setting Up Basecamp.” In: *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*. PLPV ’12. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 71–82. isbn: 978-1-4503-1125-0. doi: 10.1145/2103776.2103785.

[Amn+01] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D’Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, M. O. Müller, P. Pettersson, C. Weise, and W. Yi. “UPPAAL - Now, Next, and Future.” English. In: *Modeling and Verification of Parallel Processes*. Ed. by F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan. Vol. 2067. LNCS. Springer Berlin Heidelberg, 2001, pp. 99–124. isbn: 978-3-540-42787-2. doi: 10.1007/3-540-45510-8_4.

[AMP06] A. Armando, J. Mantovani, and L. Platania. “Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers.” In: *Model Checking Software*. Ed. by A. Valmari. Vol. 3925. LNCS. Springer Berlin Heidelberg, 2006, pp. 146–162. isbn: 978-3-540-33102-5. doi: 10.1007/11691617_9.

[App13] J. Appleby. *HANAlgorithmics – Efficiency by Design with SAP HANA – Part 1*. http://www.saphana.com/community/blogs/blog/2013/04/18/hanalgorithmics--efficiency-by-design-with-sap-hana--part-1. Last accessed: 20 Apr 2014. 2013.

[Asa+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. “A view of the parallel computing landscape.” In: *Commun. ACM* 52.10 (2009), pp. 56–67. issn: 0001-0782. doi: http://doi.acm.org/10.1145/1562764.1562783.

[Ayg+09] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. “The Design of OpenMP Tasks.” In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2009), pp. 404–418. ISSN: 1045-9219. DOI: `10.1109/TPDS.2008.105`.

[Baa+10] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. “CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell.” In: *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*. 2010, pp. 714–721. DOI: `10.1109/DSD.2010.21`.

[Bac78] J. Backus. “The history of Fortran I, II, and III.” In: *History of programming languages I*. ACM. 1978, pp. 25–74.

[Bai+03] C. Baier, B. Haverkort, H. Hermanns, and J. Katoen. “Model-checking algorithms for continuous-time Markov chains.” In: *Software Engineering, IEEE Transactions on* 29.6 (2003), pp. 524–541. ISSN: 0098-5589. DOI: `10.1109/TSE.2003.1205180`.

[Bar+06] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. “DiVinE – A Tool for Distributed Verification.” In: *Computer Aided Verification*. Ed. by T. Ball and R. Jones. Vol. 4144. LNCS. Springer, 2006, pp. 278–281. URL: `http://dx.doi.org/10.1007/11817963\_26`.

[Bar+08] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. “ProbDiVinE-MC: Multi-core LTL Model Checker for Probabilistic Systems.” In: *QEST ’08: Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 77–78. ISBN: 978-0-7695-3360-5. DOI: `http://dx.doi.org/10.1109/QEST.2008.29`.

[Bar+10] J. Barnat, L. Brim, M. Češka, and P. Ročkai. “DiVinE: Parallel Distributed Model Checker.” In: *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*. IEEE, 2010, pp. 4–7.

[Bar+11a] J. Barnat, P. Bauch, L. Brim, and M. Češka. “Designing Fast LTL Model Checking Algorithms for Many-Core GPUs.” In: *Journal of Parallel and Distributed Computing* 0 (2011), pp. –. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2011.10.015`.

[Bar+11b] J. Barnat, P. Bauch, L. Brim, and M. Ceska. “Computing Strongly Connected Components in Parallel on CUDA.” In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 544–555. ISBN: 978-0-7695-4385-7. DOI: 10.1109/IPDPS.2011.59.

[Bar+13] J. Barnat, L. Brim, V. Havel, J. Havlıcek, J. Kriho, M. Lenco, P. Ročkai, V. Štill, and J. Weiser. “DiVinE 3.0–An Explicit-State Model Checker for Multithreaded C & C++ Programs.” In: *Computer Aided Verification (CAV 2013)* (2013), p. 6.

[Bar10] J. Barnat. “Platform-Dependent Verification.” Habilitation thesis. Faculty of Informatics, Masaryk University, Brno Czech Republic Czech Republic, 2010.

[BB11] L. Brim and J. Barnat. “Platform Dependent Verification: On Engineering Verification Tools for 21st Century.” In: *ArXiv e-prints* (Nov. 2011). arXiv: 1111.0368 [cs.SE].

[BB91] J. C. M. Baeten and J. A. Bergstra. “Real time process algebra.” English. In: *Formal Aspects of Computing* 3.2 (1991), pp. 142–188. ISSN: 0934-5043. DOI: 10.1007/BF01898401.

[BBC03a] J. Barnat, L. Brim, and J. Chaloupka. “Parallel Breadth-First Search LTL Model-Checking.” In: *ASE’03*. IEEE Computer Society, 2003, pp. 106–115.

[BBC03b] J. Barnat, L. Brim, and J. Chaloupka. “Parallel Breadth-First Search LTL Model-Checking.” In: *Proc. 18th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 2003, pp. 106–115.

[BBC05a] J. Barnat, L. Brim, and J. Chaloupka. “From Distributed Memory Cycle Detection to Parallel {LTL} Model Checking.” In: *Electronic Notes in Theoretical Computer Science* 133.0 (2005). Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2004), pp. 21 –39. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2004.08.056.

[BBC05b] J. Barnat, L. Brim, and J. Chaloupka. “From Distributed Memory Cycle Detection to Parallel LTL Model Checking.” In: *Electronic Notes Theoretical Computer Science* 133 (2005), pp. 21–39.

[BBR07] J. Barnat, L. Brim, and P. Ročkai. “Scalable Multi-core LTL Model-Checking.” In: *Model Checking Software*. Vol. 4595. LNCS. Springer, 2007, pp. 187–203. ISBN: 978-3-540-73369-0.

[BBR09a] J. Barnat, L. Brim, and P. Ročkai. “A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties.” In: *Formal Methods and Software Engineering (ICFEM 2009)*. Vol. 5885. LNCS. Springer, 2009, pp. 407–425.

[BBR09b] J. Barnat, L. Brim, and P. Ročkai. “DiVinE 2.0: High-Performance Model Checking.” In: *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*. IEEE Computer Society Press, 2009, pp. 31–32.

[BBR10a] J. Barnat, L. Brim, and P. Ročkai. “Parallel Partial Order Reduction with Topological Sort Proviso.” In: SEFM. IEEE Computer Society, 2010, pp. 222–231. ISBN: 978-0-7695-4153-2. DOI: 10.1109/SEFM.2010.35.

[BBR10b] J. Barnat, L. Brim, and P. Ročkai. “Scalable Shared Memory LTL Model Checking.” In: *STTT*. LNCS 12.2 (2010), pp. 139–153. DOI: 10.1007/s10009-010-0136-z.

[BBS06] L. B. Briones, E. Brinksma, and M. Stoelinga. “A Semantic Framework for Test Coverage.” In: *Automated Technology for Verification and Analysis*. Ed. by S. Graf and W. Zhang. Vol. 4218. LNCS. Springer Berlin Heidelberg, 2006, pp. 399–414. ISBN: 978-3-540-47237-7. DOI: 10.1007/11901914_30.

[BCP08] J. Barnat, J. Chaloupka, and J. C. van de Pol. “Improved Distributed Algorithms for SCC Decomposition.” In: *Electronic Notes in Theoretical Computer Science* 198.1 (2008). Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007), pp. 63 –77. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.02.001.

[BCS07] H. Boudali, P. Crouzen, and M. Stoelinga. “Dynamic Fault Tree Analysis Using Input/Output Interactive Markov Chains.” In: *Dependable Systems and Networks, 2007. DSN ’07. 37th Annual IEEE/IFIP International Conference on*. 2007, pp. 708–717. DOI: 10.1109/DSN.2007.37.

[BD+00] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. “Scalable Distributed On-the-Fly Symbolic Model Checking.” English. In: *Formal Methods in Computer-Aided Design*. Ed. by J. Hunt Warren A. and S. D. Johnson. Vol. 1954. LNCS. Springer Berlin Heidelberg, 2000, pp. 427–441. ISBN: 978-3-540-41219-9. DOI: 10.1007/3-540-40922-X_24.

[BD98a] D. Bošnački and D. Dams. “Discrete-Time Promela and Spin.” In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by A. P. Ravn and H. Rischel. Vol. 1486. LNCS. Springer Berlin Heidelberg, 1998, pp. 307–310. ISBN: 978-3-540-65003-4. DOI: 10.1007/BFb0055359.

[BD98b] D. Bošnački and D. Dams. “Integrating Real Time Into Spin: A Prototype Implementation.” English. In: *Formal Description Techniques and Protocol Specification, Testing and Verification*. Ed. by S. Budkowski, A. Cavalli, and E. Najm. Vol. 6. The International Federation for Information Processing (IFIP). Springer US, 1998, pp. 423–438. ISBN: 978-1-4757-5262-5. DOI: 10.1007/978-0-387-35394-4_26.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. “A Tutorial on Uppaal.” In: *FMDRTS*. Ed. by M. Bernardo and F. Corradini. Vol. 3185. LNCS. Springer, 2004, pp. 200–236. ISBN: 978-3-540-23068-7. DOI: 10.1007/978-3-540-30080-9_7.

[Beh+02] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. “Uppaal Implementation Secrets.” English. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by W. Damm and E.-R. Olderog. Vol. 2469. LNCS. Springer Berlin Heidelberg, 2002, pp. 3–22. ISBN: 978-3-540-44165-6. DOI: 10.1007/3-540-45739-9_1.

[Beh+03] G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. “Static Guard Analysis in Timed Automata Verification.” In: *TACAS* (2003), pp. 254–270.

[Beh+06] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. “Lower and upper bounds in zone-based abstractions of timed automata.” English. In: *International Journal on Software Tools for Technology Transfer* 8.3 (2006), pp. 204–215. ISSN: 1433-2779. DOI: 10.1007/s10009-005-0190-0.

[Beh+11] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi. “Developing Uppaal over 15 years.” In: *Software: Practice and Experience* 41.2 (2011), pp. 133–142. DOI: 10.4204/EPTCS.13.1. eprint: {0912.1897v1,}.

[Beh05] G. Behrmann. “Distributed Reachability Analysis in Timed Automata.” In: *STTT* 7.1 (2005), pp. 19–30.

[Ben+98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. “Partial order reductions for timed systems.” In: *CONCUR’98 Concurrency Theory*. Ed. by D. Sangiorgi and R. Simone. Vol. 1466. LNCS. Springer Berlin Heidelberg, 1998, pp. 485–500. ISBN: 978-3-540-64896-3. DOI: 10.1007/BFb0055643.

[Ben02] J. Bengtsson. “Clocks, DBMs and States in Timed Systems.” PhD thesis. Uppsala University, 2002.

[Ber13] F. I. van der Berg. “Model Checking LLVM IR using LTSmin.” MSc. Thesis. University of Twente, 2013. URL: http://fmt.cs.utwente.nl/education/master/190/.

[BH06] A. Bell and B. R. Haverkort. “Distributed disk-based algorithms for model checking very large Markov chains.” In: *Formal Methods in System Design* 29.2 (2006), pp. 177–196.

[BHR13] J. Barnat, J. Havlíček, and P. Ročkai. “Distributed LTL Model Checking with Hash Compaction.” In: *Electron. Notes Theor. Comput. Sci.* 296 (Aug. 2013), pp. 79–93. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2013.07.006.

[BHV00] G. Behrmann, T. Hune, and F. Vaandrager. “Distributing timed model checking - how the search order matters.” In: *CAV*. 2000, pp. 216–231.

[Bie+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. “Bounded Model Checking.” In: vol. 58. Advances in Computers. Elsevier, 2003, pp. 117 –148. DOI: http://dx.doi.org/10.1016/S0065-2458(03)58003-2.

[Bie+99a] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. “Symbolic Model Checking Using SAT Procedures Instead of BDDs.” In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. DAC ’99. New Orleans, Louisiana, USA: ACM, 1999, pp. 317–320. ISBN: 1-58113-109-7. DOI: 10.1145/309847.309942.

[Bie+99b] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. “Symbolic Model Checking without BDDs.” English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. R. Cleaveland. Vol. 1579. LNCS. Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-65703-3. DOI: 10.1007/3-540-49059-0_14.

[Bin+10] B. Bingham, J. Bingham, F. M. de Paula, J. Erickson, G. Singh, and M. Reitblatt. “Industrial Strength Distributed Explicit State Model Checking.” In: *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*. 2010, pp. 28–36. DOI: 10.1109/PDMC-HiBi.2010.13.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[BKR10] S. C. C. Blom, G. Kant, and A. Rensink. "Distributed Graph-Based State Space Generation." In: *Electronic Communications of the EASST*. Electronic Communications of the EASST 32 (2010). Ed. by J. de Lara and D. Varró, p. 8. ISSN: 1863-2122. URL: `http://journal.ub.tu-berlin.de/eceasst/article/view/521`.

[BL13a] F. I. van der Berg and A. W. Laarman. "SpinS: Extending LTSmin with Promela through SpinJa." In: *Electronic Notes in Theoretical Computer Science* 296 (2013), pp. 95 –105. ISSN: 1571-0661. DOI: `http://dx.doi.org/10.1016/j.entcs.2013.07.007`.

[BL13b] D. Beyer and S. Löwe. "Explicit-State Software Model Checking Based on CEGAR and Interpolation." In: *Fundamental Approaches to Software Engineering*. Ed. by V. Cortellessa and D. Varró. Vol. 7793. LNCS. Springer Berlin Heidelberg, 2013, pp. 146–162. ISBN: 978-3-642-37056-4. DOI: `10.1007/978-3-642-37057-1_11`.

[BLL03] S. C. C. Blom, I. van Langevelde, and B. Lisser. "Compressed and Distributed File Formats for Labeled Transition Systems." In: *Electronic Notes in Theoretical Computer Science* 89.1 (2003). PDMC 2003, Parallel and Distributed Model Checking (Satellite Workshop of CAV '03), pp. 68 –83. ISSN: 1571-0661. DOI: `DOI:10.1016/S1571-0661(05)80097-0`.

[BLLL09] D. Bošnački, S. Leue, and A. Lluch-Lafuente. "Partial-order reduction for general state exploring algorithms." In: vol. 11. 1. 2009, pp. 39–51.

[Blo+07] S. C. C. Blom, J. R. Calamé, B. Lisser, S. Orzan, J. Pang, J. C. van de Pol, M. Torabi Dashti, and A. J. Wijs. "Distributed analysis with μCRL: a compendium of case studies." In: TACAS'07. Braga, Portugal: Springer, 2007, pp. 683–689. ISBN: 978-3-540-71208-4. URL: `http://portal.acm.org/citation.cfm?id=1763507.1763576`.

[Blo+08a] S. C. C. Blom, B. Lisser, J. van de Pol, and M. Weber. "A Database Approach to Distributed State Space Generation." In: *Electron. Notes Theor. Comput. Sci.* 198.1 (2008), pp. 17–32. ISSN: 1571-0661. DOI: `http://dx.doi.org/10.1016/j.entcs.2007.10.018`.

[Blo+08b] S. C. C. Blom, B. R. Haverkort, M. Kuntz, and J. van de Pol. "Distributed Markovian Bisimulation Reduction aimed at CSL Model Checking." In: *Electr. Notes Theor. Comput. Sci.* 220.2 (2008), pp. 35–50.

[BLP03] G. Behrmann, K. G. Larsen, and R. Pelánek. "To Store or Not to Store." In: *Computer Aided Verification*. Ed. by J. Hunt WarrenA. and F. Somenzi. Vol. 2725. LNCS. Springer Berlin Heidelberg, 2003, pp. 433–445. ISBN: 978-3-540-40524-5. DOI: `10.1007/978-3-540-45069-6_40`.

[Blu+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*. Vol. 30. 8. ACM, 1995.

[BNR09] N. H. M. aan de Brugh, V. Nguyen, and T. C. Ruys. "MoonWalker: Verification of .NET Programs." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. LNCS. Springer Berlin Heidelberg, 2009, pp. 170–173. ISBN: 978-3-642-00767-5. DOI: `10.1007/978-3-642-00768-2_15`.

[BO03] S. C. C. Blom and S. Orzan. "Distributed branching bisimulation reduction of state spaces." In: *Electronic Notes in Theoretical Computer Science* 89.1 (2003), pp. 99–113.

[BO05] S. C. C. Blom and S. Orzan. "A distributed algorithm for strong bisimulation reduction of state spaces." English. In: *International Journal on Software Tools for Technology Transfer* 7.1 (2005), pp. 74–86. ISSN: 1433-2779. DOI: `10.1007/s10009-004-0159-4`.

[Boc78] G. V. Bochmann. "Finite state description of communication protocols." In: *Computer Networks (1976)* 2.4–5 (1978), pp. 361 –372. ISSN: 0376-5075. DOI: `http://dx.doi.org/10.1016/0376-5075(78)90015-6`.

[BOS06] V. Braberman, A. Olivero, and F. Schapachnik. "Dealing with practical limitations of distributed timed model checking for timed automata." In: *Formal Methods in System Design* 29 (2 2006). 10.1007/s10703-006-0012-3, pp. 197–214. ISSN: 0925-9856. URL: `http://dx.doi.org/10.1007/s10703-006-0012-3`.

[Bou+08] H. Boudali, P. Crouzen, B. R. Haverkort, M. Kuntz, and M. I. A. Stoelinga. "Architectural dependability evaluation with Arcade." In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. 2008, pp. 512–521. DOI: `10.1109/DSN.2008.4630122`.

[Bou04] P. Bouyer. "Forward Analysis of Updatable Timed Automata." In: *Formal Methods in System Design* 24.3 (2004), pp. 281–320.

[Boš02] D. Bošnački. "A Nested Depth First Search Algorithm for Model Checking with Symmetry Reduction." In: *Formal Techniques for Networked and Distributed Sytems*. Ed. by D. A. Peled and M. Y. Vardi. Vol. 2529. LNCS. Springer Berlin Heidelberg, 2002, pp. 65–80. ISBN: 978-3-540-00141-6. DOI: `10.1007/3-540-36135-9_5`.

[BP02] S. C. C. Blom and J. C. van de Pol. “State Space Reduction by Proving Confluence.” English. In: *Computer Aided Verification*. Ed. by E. Brinksma and K. G. Larsen. Vol. 2404. LNCS. Springer Berlin Heidelberg, 2002, pp. 596–609. isbn: 978-3-540-43997-4. doi: `10.1007/3-540-45657-0_50`.

[BP08] S. C. C. Blom and J. C. van de Pol. “Symbolic Reachability for Process Algebras with Recursive Data Types.” In: *Theoretical Aspects of Computing (ICTAC 2008)*. Ed. by J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün. Vol. 5160. LNCS. Springer, Sept. 2, 2008, pp. 81–95. isbn: 978-3-540-85761-7. doi: `http://dx.doi.org/10.1007/978-3-540-85762-4_6`.

[BPW09] S. C. C. Blom, J. C. van de Pol, and M. Weber. *Bridging the Gap between Enumerative and Symbolic Model Checkers*. Technical Report TR-CTIT-09-30. Enschede: University of Twente, 2009.

[BPW10] S. C. C. Blom, J. C. v. d. Pol, and M. Weber. “LTSmin: Distributed and Symbolic Reachability.” In: *CAV*. Ed. by T. Touili, B. Cook, and P. Jackson. Vol. 6174. LNCS. Edinburgh: Springer, 2010, pp. 354–359.

[BR01] T. Ball and S. K. Rajamani. “The SLAM Toolkit.” English. In: *Computer Aided Verification*. Ed. by G. Berry, H. Comon, and A. Finkel. Vol. 2102. LNCS. Springer Berlin Heidelberg, 2001, pp. 260–264. isbn: 978-3-540-42345-4. doi: `10.1007/3-540-44585-4_25`.

[BR02] T. Ball and S. K. Rajamani. “The SLAM Project: Debugging System Software via Static Analysis.” In: *SIGPLAN Not.* 37.1 (Jan. 2002), pp. 1–3. issn: 0362-1340. doi: `10.1145/565816.503274`.

[BR08] J. Barnat and P. Ročkai. “Shared Hash Tables in Parallel Model Checking.” In: *Elec. Notes in Theor. Comp. Sc.* 198.1 (2008). Proc. of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007), pp. 79 –91. issn: 1571-0661. doi: `DOI:10.1016/j.entcs.2007.10.021`.

[Bra11] A. R. Bradley. “SAT-Based Model Checking without Unrolling.” In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by R. Jhala and D. Schmidt. Vol. 6538. LNCS. Springer Berlin Heidelberg, 2011, pp. 70–87. isbn: 978-3-642-18274-7. doi: `10.1007/978-3-642-18275-4_7`.

[Bri+01] L. Brim, I. Černá, P. Krčál, and R. Pelánek. “Distributed LTL Model Checking Based on Negative Cycle Detection.” In: *FSTTCS 2001*. Vol. 2245. LNCS. Springer, Heidelberg, 2001, pp. 96–107.

[Bri+04] L. Brim, I. Cerná, P. Moravec, and J. Simsa. "Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking." In: *FMCAD*. Ed. by A. J. Hu and A. K. Martin. Vol. 3312. LNCS. Springer, 2004, pp. 352–366. ISBN: 3-540-23738-0. DOI: 10.1007/978-3-540-30494-4_25.

[Bri+05] L. Brim, I. Černá, P. Moravec, and J. Šimša. "Distributed Partial Order Reduction of State Spaces." In: *Electronic Notes in Theoretical Computer Science* 128.3 (2005), pp. 63 –74. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2004.10.019.

[Bri06] L. Brim. "Distributed Verification: Exploring the Power of Raw Computing Power." In: *Formal Methods: Applications and Technology*. Ed. by L. Brim, B. Haverkort, M. Leucker, and J. C. van de Pol. Vol. 4346. LNCS. Springer, 2006, pp. 23–34.

[Bry86] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." In: *IEEE Transactions on Computers* 35 (1986), pp. 677–691.

[BTY97] A. Bouajjani, S. Tripakis, and S. Yovine. "On-the-Fly Symbolic Model Checking for Real-Time Systems." In: *18th IEEE*. RTSS. Washington, DC, USA: IEEE, 1997, pp. 25–34. ISBN: 0-8186-8268-X. URL: http://dl.acm.org/citation.cfm?id=827269.828996.

[Bur+90] 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: *LICS'90*. 1990, pp. 428–439.

[Bur12] R. Burgman. "Partial-order reduction based on probe sets." MSc. Thesis. University of Twente, 2012. URL: http://essay.utwente.nl/62506/.

[CC77] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[CE82] E. Clarke and E. Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic." In: *Logics of Programs*. Ed. by D. Kozen. Vol. 131. LNCS. Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-11212-9. DOI: 10.1007/BFb0025774.

[CES09] E. M. Clarke, E. A. Emerson, and J. Sifakis. "Model checking: algorithmic verification and debugging." In: *Commun. ACM* 52.11 (Nov. 2009), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/1592761.1592781.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications." In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263. ISSN: 0164-0925. DOI: 10.1145/5397.5399.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, volume I of Studies in Logic and the Foundations of Mathematics*. 1958.

[CGR11] S. Cranen, J. F. Groote, and M. Reniers. "A linear translation from CTL* to the first-order modal μ-calculus." In: *Theoretical Computer Science* 412.28 (2011). Festschrift in Honour of Jan Bergstra, pp. 3129 –3139. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/j.tcs.2011.02.034.

[CGS04] E. M. Clarke, A. Gupta, and O. Strichman. "SAT-based counterexample-guided abstraction refinement." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.7 (2004), pp. 1113–1123. ISSN: 0278-0070.

[Cha+04] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables." In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '04. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2004, pp. 30–39. ISBN: 0-89871-558-X. URL: http://dl.acm.org/citation.cfm?id=982792.982797.

[Che+07] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. "Hierarchical cache coherence protocol verification one level at a time through assume guarantee." In: *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*. 2007, pp. 107–114. DOI: 10.1109/HLDVT.2007.4392796.

[CHR91] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. "A calculus of durations." In: *Information processing letters* 40.5 (1991), pp. 269–276.

[Cim+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking." English. In: *Computer Aided Verification*. Ed. by E. Brinksma and K. G. Larsen. Vol. 2404. LNCS. Springer Berlin Heidelberg, 2002, pp. 359–364. ISBN: 978-3-540-43997-4. DOI: 10.1007/3-540-45657-0_29.

[CJ99] H. Comon and Y. Jurski. “Timed automata and the theory of real numbers.” In: *CONCUR*. Vol. 1664. Springer, 1999, pp. 242–257.

[Cla+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. “Counterexample-Guided Abstraction Refinement.” In: *Computer Aided Verification*. Ed. by E. A. Emerson and A. P. Sistla. Vol. 1855. LNCS. Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-67770-3. DOI: 10.1007/10722167_15.

[Cla+02] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. “SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques.” English. In: *Computer Aided Verification*. Ed. by E. Brinksma and K. G. Larsen. Vol. 2404. LNCS. Springer Berlin Heidelberg, 2002, pp. 265–279. ISBN: 978-3-540-43997-4. DOI: 10.1007/3-540-45657-0_20.

[Cla+03] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. “Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems.” In: *International Journal of Foundations of Computer Science* 14.04 (2003), pp. 583–604. DOI: 10.1142/S012905410300190X.

[Cla+96] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. “Exploiting symmetry in temporal logic model checking.” English. In: *Formal Methods in System Design* 9.1-2 (1996), pp. 77–104. ISSN: 0925-9856. DOI: 10.1007/BF00625969.

[Cla08] E. M. Clarke. “The Birth of Model Checking.” In: *25 Years of Model Checking*. Ed. by O. Grumberg and H. Veith. Berlin, Heidelberg: Springer, 2008, pp. 1–26. ISBN: 978-3-540-69849-4. DOI: http://dx.doi.org/10.1007/978-3-540-69850-0_1.

[Cle84] J. G. Cleary. “Compact Hash Tables Using Bidirectional Linear Probing.” In: *IEEE Transactions on Computers* C-33.9 (1984), pp. 828 –834. ISSN: 0018-9340. DOI: 10.1109/TC.1984.1676499.

[Cli07] C. Click. *A Lock-Free Hash Table*. Talk at JavaOne 2007, http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf. 2007.

[Cli08] C. Click. *Where the Lions gone?* Dagstuhl, http://www.dagstuhl.de/Materials/Files/08/08332/08332.BrimLubos.Slides.pdf. 2008.

[CLM85] P. Celis, P.-A. Larson, and J. Munro. “Robin hood hashing.” In: *Foundations of Computer Science, 1985., 26th Annual Symposium on*. 1985, pp. 281–288. DOI: 10.1109/SFCS.1985.48.

[Cob64] A Cobham. “The intrinsic computational difficulty of functions.” In: *International Congress for Logic, Methodology, and Philosophy of Science*. 1964.

[Coo79] S. A. Cook. “Deterministic CFL’s are accepted simultaneously in polynomial time and log squared space.” In: *Proceedings of the eleventh annual ACM symposium on Theory of computing*. STOC ’79. Atlanta, Georgia, USA: ACM, 1979, pp. 338–345. DOI: 10.1145/800135.804426.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd ed. The MIT Press, 2009. ISBN: 0262033844.

[Cou+92] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. “Memory-Efficient Algorithms for the Verification of Temporal Properties.” In: *Formal Methods in System Design* 1.2/3 (1992), pp. 275–288. DOI: 10.1007/BFb0023737.

[Cou13] R. Courtland. *The Status of Moore’s Law: It’s Complicated*. http://spectrum.ieee.org/semiconductors/devices/the-status-of-moores-law-its-complicated. 2013.

[Cou99] J. M. Couvreur. “On-the-fly Verification of Linear Temporal Logic.” In: *FM-99 - Formal Methods*. Ed. by J. Wing, J. Woodcock, and J. Davies. Vol. 1708. LNCS. Springer Berlin / Heidelberg, 1999, pp. 711–711. DOI: 10.1007/3-540-48119-2_16.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. “Terminator: Beyond Safety.” In: *Proceedings of CAV 2006*. 2006, pp. 415–418.

[Cra+13] S. Cranen, J.-F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. Vink, W. Wesselink, and T. A. C. Willemse. “An Overview of the mCRL2 Toolset and Its Recent Advances.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by N. Piterman and S. Smolka. Vol. 7795. LNCS. Springer Berlin Heidelberg, 2013, pp. 199–213. ISBN: 978-3-642-36741-0. DOI: 10.1007/978-3-642-36742-7_15.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. 99th. Wiley-Interscience, 1991. ISBN: 0471062596.

[CVH04] K. Y. Chun and D. Van Hung. “Verifying real-time systems using untimed model checking tools.” In: *The United Nations University. Tech. Report UNU-IIST* 302 (2004).

[Dal+11] A. E. Dalsgaard, R. R. Hansen, K. Jørgensen, K. G. Larsen, M. C. Olesen, P. Olsen, and J. Srba. "opaal: A Lattice Model Checker." In: *NASA Formal Methods*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Vol. 6617. Springer, 2011. Chap. LNCS, pp. 487–493. DOI: 10.1007/978-3-642-20398-5_37.

[Dal+12] A. E. Dalsgaard, A. W. Laarman, K. G. Larsen, M. C. Olesen, and J. C. van de Pol. "Multi-core Reachability for Timed Automata." In: *FORMATS'12*. Ed. by M. Jurdziński and D. Ničković. Vol. 7595. LNCS. Springer, 2012, pp. 91–106. ISBN: 978-3-642-33364-4. DOI: 10.1007/978-3-642-33365-1_8.

[Dat13] N. C. Dat. "Multi-Core Model Checking." B. S. Thesis. National University of Singapore (NUS), 2013.

[DC80] P. W. Dymond and S. A. Cook. "Hardware complexity and parallel computation." In: *Foundations of Computer Science, 1980., 21st Annual Symposium on*. 1980, pp. 360–372. DOI: 10.1109/SFCS.1980.22.

[Dij12] T. van Dijk. "The parallelization of binary decision diagram operations for model checking." MSc. Thesis. University of Twente, 2012. URL: http://essay.utwente.nl/61650/.

[Dij75] E. W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs." In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975.

[Dil89] D. L. Dill. "Timing Assumptions and Verification of Finite-State Concurrent Systems." In: *AVMFSS*. Ed. by J. Sifakis. Vol. 407. Springer, 1989, pp. 197–212. ISBN: 978-3-540-52148-8. DOI: 10.1007/3-540-52148-8_17.

[DJK13] A. Dudás, S. Juhász, and S. Kolumbán. "Recalibrating Fine-Grained Locking in Parallel Bucket Hash Tables." In: *Facing the Multicore-Challenge III*. Ed. by R. Keller, D. Kramer, and J.-P. Weiss. Vol. 7686. LNCS. Springer Berlin Heidelberg, 2013, pp. 60–71. ISBN: 978-3-642-35892-0. DOI: 10.1007/978-3-642-35893-7_6.

[DKR82] D. Dolev, M. M. Klawe, and M. Rodeh. "An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle." In: *J. Algorithms* 3.3 (1982), pp. 245–260.

[DKW08] V. D’silva, D. Kroening, and G. Weissenbacher. “A Survey of Automated Techniques for Formal Software Verification.” In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27.7 (2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410.

[DLL62] M. Davis, G. Logemann, and D. Loveland. “A machine program for theorem-proving.” In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.

[DLP12] T. van Dijk, A. W. Laarman, and J. C. van de Pol. “Multi-core and/or symbolic model checking.” In: *AVoCS 2012*. Ed. by G. Luettgen and S. Merz. Vol. 53. Electronic Communications of the EASST. Bamberg, Germany: EASST, 2012, 773:1–773:7. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/773.

[DLP13] T. van Dijk, A. W. Laarman, and J. C. van de Pol. “Multi-Core BDD Operations for Symbolic Reachability.” In: *Electronic Notes in Theoretical Computer Science* 296.0 (2013). Proceedings of PASM/PDMC, pp. 127 –143. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2013.07.009.

[DM04] P. Dillinger and P. Manolios. “Bloom Filters in Probabilistic Verification.” In: *Formal Methods in Computer-Aided Design*. Ed. by A. Hu and A. Martin. Vol. 3312. LNCS. Springer Berlin Heidelberg, 2004, pp. 367–381. ISBN: 978-3-540-23738-9. DOI: 10.1007/978-3-540-30494-4_26.

[DM09] P. Dillinger and P. Manolios. “Fast, All-Purpose State Storage.” In: *Model Checking Software*. Ed. by C. S. Păsăreanu. Vol. 5578. LNCS. Springer Berlin Heidelberg, 2009, pp. 12–31. ISBN: 978-3-642-02651-5. DOI: 10.1007/978-3-642-02652-2_6.

[DP60] M. Davis and H. Putnam. “A Computing Procedure for Quantification Theory.” In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034.

[DT98] C. Daws and S. Tripakis. “Model Checking of Teal-Time Reachability Properties Using Abstractions.” In: *TACAS*. Vol. 1384. Springer, 1998, pp. 313–329.

[Dun11] J. E. Dunn. “Death of Moore’s Law Will Cause Economic Crisis.” In: *PC World* (2011).

[Dwy+07] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare. “Parallel Randomized State-Space Search.” In: *Proc. ICSE 2007*. IEEE Computer Society Press, 2007, pp. 3–12.

[EH08] J. Esparza and K. Heljanko. *Unfoldings: a partial-order approach to model checking*. Springer, 2008.

[EJP97] E. A. Emerson, S. Jha, and D. Peled. “Combining partial order and symmetry reductions.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by E. Brinksma. Vol. 1217. LNCS. Springer Berlin Heidelberg, 1997, pp. 19–34. ISBN: 978-3-540-62790-6. DOI: 10.1007/BFb0035378. URL: http://dx.doi.org/10.1007/BFb0035378.

[EKP13] S. Evangelista, L. M. Kristensen, and L. Petrucci. “Multi-threaded Explicit State Space Exploration with State Reconstruction.” In: *ATVA’2013*. Vol. 8172. LNCS. Springer, 2013, pp. 208–223.

[EL87] E. A. Emerson and C.-L. Lei. “Modalities for model checking: branching time logic strikes back.” In: *Science of Computer Programming* 8.3 (1987), pp. 275 –306. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/0167-6423(87)90036-0.

[Eme97] E. A. Emerson. “Model checking and the mu-calculus.” In: *DIMACS series in discrete mathematics* 31 (1997), pp. 185–214.

[EP10] S. Evangelista and C. Pajault. “Solving the Ignoring Problem for Partial Order Reduction.” English. In: *STTF* 12 (2 2010), pp. 155–170. ISSN: 1433-2779. DOI: 10.1007/s10009-010-0137-y.

[EPP05] S. Evangelista and J. F. Pradat-Peyre. “Memory Efficient State Space Storage in Explicit Software Model Checking.” In: *Model Checking Software*. Ed. by P. Godefroid. Vol. 3639. LNCS. Springer Berlin / Heidelberg, 2005, pp. 43–57. URL: http://dx.doi.org/10.1007/11537328_7.

[EPY11] S. Evangelista, L. Petrucci, and S. Youcef. “Parallel Nested Depth-First Searches for LTL Model Checking.” In: *Automated Technology for Verification and Analysis*. Ed. by T. Bultan and P.-A. Hsiung. Vol. 6996. LNCS. Springer Berlin Heidelberg, 2011, pp. 381–396. ISBN: 978-3-642-24371-4. DOI: 10.1007/978-3-642-24372-1_27.

[ER08] E. Eide and J. Regehr. "Volatiles Are Miscompiled, and What to Do About It." In: *Proceedings of the 8th ACM International Conference on Embedded Software*. EMSOFT '08. Atlanta, GA, USA: ACM, 2008, pp. 255–264. ISBN: 978-1-60558-468-3. DOI: `10.1145/1450058.1450093`.

[ES96] E. Emerson and A. Sistla. "Symmetry and model checking." English. In: *Formal Methods in System Design* 9.1-2 (1996), pp. 105–131. ISSN: 0925-9856. DOI: `10.1007/BF00625970`. URL: `http://dx.doi.org/10.1007/BF00625970`.

[Esp+13] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. "A Fully Verified Executable LTL Model Checker." In: *Proceedings of CAV 2013*. 2013.

[Eva+12] S. Evangelista, A. W. Laarman, L. Petrucci, and J. C. van de Pol. "Improved Multi-Core Nested Depth-First Search." In: *ATVA'12*. Ed. by S. Ramesh. Vol. 7561. LNCS. Thiruvananthapuram (Trivandrum), Kerala: Springer, 2012, pp. 269–283. DOI: `10.1007/978-3-642-33386-6_22`.

[EW05] E. A. Emerson and T. Wahl. "Dynamic Symmetry Reduction." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by N. Halbwachs and L. D. Zuck. Vol. 3440. LNCS. Springer Berlin Heidelberg, 2005, pp. 382–396. ISBN: 978-3-540-25333-4. DOI: `10.1007/978-3-540-31980-1_25`.

[Fac13] V. Fachin. "Softwarebasierte Fehlertoleranz: Timed Automata Based Performance Analysis." B. S. Thesis. Technical University of Dortmund, 2013. URL: `http://ess.cs.tu-dortmund.de/Teaching/WS2012/SFt/Downloads/ausarbeitungen/Vasco_Fachin.pdf`.

[Far07] D. Faragó. "Model Checking of Randomized Leader Election Algorithms." MA thesis. Universität Karlsruhe, 2007.

[FHV13] W. Fokkink, P. Hijma, and S. Vijzelaar. *Programming Assignment: Multi-core nested depth-first search in Java*. `http://anna.fi.muni.cz/models/cgi/models.cgi`. Last accessed: 14 Jan 2013. 2013.

[Fis+01] K. Fisler, R. Fraer, G. Kamhi, M., and Z. Yang. "Is There a Best Symbolic Cycle-Detection Algorithm?" English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by T. Margaria and W. Yi. Vol. 2031. LNCS. Springer Berlin Heidelberg, 2001, pp. 420–434. ISBN: 978-3-540-41865-8. DOI: `10.1007/3-540-45319-9_29`.

[Fos95] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[FS01] A. Finkel and P. Schnoebelen. "Well-Structured Transition Systems Everywhere!" In: *Theoretical Computer Science* 256.1-2 (2001), pp. 63–92. ISSN: 0304-3975.

[FS09] D. Faragó and P. H. Schmitt. "Improving Non-Progress Cycle Checks." In: *SPIN*. Ed. by C. S. Păsăreanu. Vol. 5578. LNCS. Springer, 2009, pp. 50–67. ISBN: 978-3-642-02651-5. DOI: 10.1007/978-3-642-02652-2_8.

[Gar+07] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. "CADP 2006: a toolbox for the construction and analysis of distributed processes." In: *Proceedings of the 19th international conference on Computer aided verification*. LNCS. Berlin, Germany: Springer, 2007, pp. 158–163. ISBN: 978-3-540-73367-6. URL: http://portal.acm.org/citation.cfm?id=1770351.1770376.

[GGH04] H. Gao, J. F. Groote, and W. H. Hesselink. "Almost wait-free resizable hashtables." In: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. 2004, pp. 50–. DOI: 10.1109/IPDPS.2004.1302969.

[GGH05] H. Gao, J. F. Groote, and W. H. Hesselink. "Lock-free dynamic hash tables with open addressing." English. In: *Distributed Computing* 18.1 (2005), pp. 21–42. ISSN: 0178-2770. DOI: 10.1007/s00446-004-0115-2.

[GGP12a] F. Gava, M. Guedj, and F. Pommereau. "A BSP Algorithm for On-the-fly Checking LTL Formulas on Security Protocols." In: *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*. 2012, pp. 11–18. DOI: 10.1109/ISPDC.2012.10.

[GGP12b] F. Gava, M. Guedj, and F. Pommereau. "Performance Evaluations of a BSP Algorithm for State Space Construction of Security Protocols." In: *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*. 2012, pp. 170–174. DOI: 10.1109/PDP.2012.65.

[GGP12c] F. Gava, M. Guedj, and F. Pommereau. "A BSP algorithm for on-the-fly checking CTL* formulas on security protocols." In: *Proceedings of PDCAT'12*. Best paper award at PDCAT'12. IEEE Computer Society, Dec. 2012, pp. 79–84.

[GHP95] P. Godefroid, G. J. Holzmann, and D. Pirottin. "State-space caching revisited." English. In: *Formal Methods in System Design* 7.3 (1995), pp. 227–241. ISSN: 0925-9856. DOI: 10.1007/BF01384077.

[GHS01] O. Grumberg, T. Heyman, and A. Schuster. "Distributed Symbolic Model Checking for μ-Calculus." English. In: *Computer Aided Verification*. Ed. by G. Berry, H. Comon, and A. Finkel. Vol. 2102. LNCS. Springer Berlin Heidelberg, 2001, pp. 350–362. ISBN: 978-3-540-42345-4. DOI: 10.1007/3-540-44585-4_32.

[GLM+08] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. "Automated Whitebox Fuzz Testing." In: *NDSS*. Vol. 8. 2008, pp. 151–166.

[GMS12] H. Garavel, R. Mateescu, and W. Serwe. "Large-Scale Distributed Verification using CADP: Beyond Clusters to Grids." Anglais. In: *11th International Workshop on Parallel and Distributed Methods in verifiCation*. London, Royaume-Uni, Sept. 2012. URL: http://hal.inria.fr/hal-00730668.

[GMZ04] P. Gastin, P. Moro, and M. Zeitoun. "Minimization of Counterexamples in SPIN." In: *Model Checking Software*. Ed. by S. Graf and L. Mounier. Vol. 2989. LNCS. Springer Berlin Heidelberg, 2004, pp. 92–108. ISBN: 978-3-540-21314-7. DOI: 10.1007/978-3-540-24732-6_7.

[God90] P. Godefroid. "Using Partial Orders to Improve Automatic Verification Methods." In: *CAV*. Ed. by E. M. Clarke and R. P. Kurshan. Vol. 531. LNCS. Springer, 1990, pp. 176–185. ISBN: 3-540-54477-1.

[Gos+05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. 3rd ed. Amsterdam: Addison-Wesley Longman, 2005, p. 688. ISBN: 0321246780.

[Gre96] J.-C. Gregoire. "State Space Compression in SPIN with GETSs." In: *Proc. Second SPIN Workshop, Rutgers Univ*. American Mathematical Society, 1996, pp. 3–2.

[Gro+08] J. F. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. v. Weerdenburg, W. Wesselink, T. Willemse, and J. v. d. Wulp. "The mCRL2 toolset." In: *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques*. WASDeTT'08. 2008.

[GRU08] J. F. Groote, M. A. Reniers, and Y. S. Usenko. “Verification of networks of timed automata using mCRL2.” In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536575.

[GS09] A. Gaiser and S. Schwoon. “Comparison of Algorithms for Checking Emptiness on Büchi Automata.” In: *MEMICS’09*. Ed. by P. Hlinený, V. Matyáš, and T. Vojnar. Vol. 13. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl, Germany, 2009. ISBN: 978-3-939897-15-6. DOI: 10.4230/DROPS.MEMICS.2009.2349.

[GV03] J. Geldenhuys and A. Valmari. “A nearly memory-optimal data structure for sets and mappings.” In: *SPIN’03*. Portland, OR, USA: Springer, 2003, pp. 136–150. ISBN: 3-540-40117-2. URL: http://portal.acm.org/citation.cfm?id=1767111.1767120.

[GV04] J. Geldenhuys and A. Valmari. “Tarjan’s Algorithm Makes On-the-Fly LTL Verification More Efficient.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by K. Jensen and A. Podelski. Vol. 2988. LNCS. Springer Berlin / Heidelberg, 2004, pp. 205–219. URL: http://dx.doi.org/10.1007/978-3-540-24730-2_18.

[GVR99] J. Geldenhuys, P. de Villiers, and J. Rushby. “Runtime Efficient State Compaction in SPIN.” In: *Theoretical and Practical Aspects of SPIN Model Checking*. Ed. by D. Dams, R. Gerth, S. Leue, and M. Massink. Vol. 1680. LNCS. Springer Berlin / Heidelberg, 1999, pp. 12–21. URL: http://dx.doi.org/10.1007/3-540-48234-2_2.

[GZ01] J.-F. Groote and H. Zantema. “Resolution and Binary Decision Diagrams Cannot Simulate Each Other Polynomially.” English. In: *Perspectives of System Informatics*. Ed. by D. Bjørner, M. Broy, and A. V. Zamulin. Vol. 2244. LNCS. Springer Berlin Heidelberg, 2001, pp. 33–38. ISBN: 978-3-540-43075-9. DOI: 10.1007/3-540-45575-2_5.

[Hal95] T. R. Halfhill. “An error in a lookup table created the infamous bug in Intel’s latest processor.” In: *BYTE, March* (1995).

[Han07] H. Hansen. “Alternatives to Büchi automata.” PhD thesis. Tampere, Finland, 2007. URL: http://www.cs.tut.fi/~hansen/thesis/.

[Hav13] J. Havlíček. “Untimed LTL Model Checking of Timed Automata.” (visited on 05/06/2013). MA thesis. Master’s thesis. Masaryk University, Faculty of Informatics, 2013. URL: http://is.muni.cz/th/324943/fi_m/.

[HB07] G. J. Holzmann and D. Bošnački. “The Design of a Multicore Extension of the SPIN Model Checker.” In: *IEEE Trans. Softw. Eng.* 33.10 (2007), pp. 659–674. ISSN: 0098-5589. DOI: http://dx.doi.org/10.1109/TSE.2007.70724.

[Hea11] J. R. Healey. “Toyota deaths reported to safety database rise to 37.” In: *USA Today* (2011).

[HG08] H. Hansen and J. Geldenhuys. “Cheap and Small Counterexamples.” In: *Software Engineering and Formal Methods, 2008. SEFM ’08. Sixth IEEE International Conference on*. 2008, pp. 53–62. DOI: 10.1109/SEFM.2008.18.

[HGP92] G. J. Holzmann, P. Godefroid, and D. Pirottin. “Coverage Preserving Reduction Strategies for Reachability Analysis.” In: *Proceedings of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification XII*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1992, pp. 349–363. ISBN: 0-444-89874-3. URL: http://portal.acm.org/citation.cfm?id=645835.670549.

[HJ04] G. J. Holzmann and R. Joshi. “Model-Driven Software Verification.” In: *Model Checking Software*. Ed. by S. Graf and L. Mounier. Vol. 2989. LNCS. Springer Berlin Heidelberg, 2004, pp. 76–91. ISBN: 978-3-540-21314-7. DOI: 10.1007/978-3-540-24732-6_6.

[HJG08] G. J. Holzmann, R. Joshi, and A. Groce. “Tackling Large Verification Problems with the Swarm Tool.” In: *SPIN 2008*. Vol. 5156. LNCS. Springer-Verlag, 2008, pp. 134–143.

[HJG11] G. J. Holzmann, R. Joshi, and A. Groce. “Swarm Verification Techniques.” In: *Software Engineering, IEEE Transactions on* 37.6 (2011), pp. 845–857. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.110.

[HJN08] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. “Strategies for Solving SAT in Grids by Randomized Search.” In: *AISC/MKM/Calculemus*. Ed. by S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk. LNCS 5144. Springer, 2008, pp. 125–140. ISBN: 978-3-540-85109-7. DOI: 10.1007/978-3-540-85110-3_11.

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming.” In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Hola] G. J. Holzmann. *PROMELA reference*. http://spinroot.com/spin/Man/promela.html.

[Holb] G. J. Holzmann. “Coordination problems in multiprocessing systems.” PhD thesis. Technical University Delft.

[Hol08] G. J. Holzmann. “A Stack-Slicing Algorithm for Multi-Core Model Checking.” In: *Electronic Notes in Theoretical Computer Science* 198.1 (2008). Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007), pp. 3 –16. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.10.017.

[Hol11] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2011. ISBN: 0321773713, 9780321773715.

[Hol12] G. J. Holzmann. “Parallelizing the Spin Model Checker.” In: *Proceedings of the 19th international conference on Model Checking Software*. Ed. by A. Donaldson and D. Parker. Vol. 7385. LNCS. 10.1007/978-3-642-31759-0_12. Springer, 2012, pp. 155–171. ISBN: 978-3-642-31758-3. URL: http://dx.doi.org/10.1007/978-3-642-31759-0_12.

[Hol81] G. J. Holzmann. *PAN: a protocol specification analyzer*. Tech. rep. Technical Report TM81-11271-5, AT&T Bell Laboratories, 1981.

[Hol90] G. J. Holzmann. “Algorithms for automated protocol verification.” In: *AT&T technical journal* 69.1 (1990), pp. 32–44.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991. ISBN: 0-13-539925-4.

[Hol97a] G. J. Holzmann. “The Model Checker SPIN.” In: *IEEE Trans. Softw. Eng.* 23 (5 1997), pp. 279–295. ISSN: 0098-5589. DOI: 10.1109/32.588521.

[Hol97b] G. J. Holzmann. “State Compression in SPIN: Recursive Indexing And Compression Training Runs.” In: *proc. of the third international SPIN workshop*. 1997.

[Hol98] G. J. Holzmann. “An Analysis of Bitstate Hashing.” English. In: *Formal Methods in System Design* 13.3 (1998), pp. 289–307. ISSN: 0925-9856. DOI: 10.1023/A:1008696026254.

[How80] W. A. Howard. “The formulae-as-types notion of construction.” In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[HP94] G. J. Holzmann and D. Peled. “An Improvement in Formal Verification.” In: *Proceedings of the Formal Description Techniques*. Chapman & Hall, 1994, pp. 197–211.

[HP99] G. J. Holzmann and A. Puri. “A minimized automaton representation of reachable states.” English. In: *International Journal on Software Tools for Technology Transfer* 2.3 (1999), pp. 270–278. ISSN: 1433-2779. DOI: 10.1007/s100090050034.

[HPV02] H. Hansen, W. Penczek, and A. Valmari. “Stuttering-Insensitive Automata for On-the-fly Detection of Livelock Properties.” In: *Electronic Notes in Theoretical Computer Science* 66.2 (2002). FMICS’02, 7th International {ERCIM} Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop), pp. 178 –193. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/S1571-0661(04)80411-0.

[HPY96] G. J. Holzmann, D. Peled, and M. Yannakakis. “On Nested Depth First Search.” In: *The SPIN Verification System*. American Mathematical Society, 1996, pp. 23–32.

[HS08] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. ISBN: 0123705916. URL: http://www.worldcat.org/isbn/0123705916.

[HS10] F. Herbreteau and B. Srivathsan. “Efficient On-the-Fly Emptiness Check for Timed Büchi Automata.” In: *ATVA*. Ed. by A. Bouajjani and W.-N. Chin. Vol. 6252. Springer, 2010, pp. 218–232.

[HS99] G. J. Holzmann and M. H. Smith. “Software Model Checking.” English. In: *Formal Methods for Protocol Engineering and Distributed Systems*. Ed. by J. Wu, S. Chanson, and Q. Gao. Vol. 28. IFIP Advances in Information and Communication Technology. Springer US, 1999, pp. 481–497. ISBN: 978-1-4757-5270-0. DOI: 10.1007/978-0-387-35578-8_28.

[HST08] M. Herlihy, N. Shavit, and M. Tzafrir. “Hopscotch Hashing.” In: *Distributed Computing* (2008), pp. 350–364. URL: http://dx.doi.org/10.1007/978-3-540-87779-0_24.

[Hut09] G. D. Hutcheson. “The Economic Implications of Moore’s Law.” English. In: *Into the Nano Era*. Ed. by H. Huff. Vol. 106. Springer Series in Materials Science. Springer Berlin Heidelberg, 2009, pp. 11–38. ISBN: 978-3-540-74558-7. DOI: 10.1007/978-3-540-74559-4_2.

[HW07] M. Hammer and M. Weber. ““To Store or Not To Store” Reloaded: Reclaiming Memory on Demand.” In: *Formal Methods: Applications and Technology*. Ed. by L. Brim, B. Haverkort, M. Leucker, and J. van de Pol. Vol. 4346. LNCS. Springer Berlin Heidelberg, 2007, pp. 51–66. ISBN: 978-3-540-70951-0. DOI: 10.1007/978-3-540-70952-7_4.

[IB02] C. P. Inggs and H. Barringer. “Effective State Exploration for Model Checking on a Shared Memory Architecture.” In: *Electronic Notes Theoretical Computer Science* 68.4 (2002).

[IB06] C. P. Inggs and H. Barringer. “CTL* model checking on a shared-memory architecture.” English. In: *Formal Methods in System Design* 29.2 (2006), pp. 135–155. issn: 0925-9856. doi: `10.1007/s10703-006-0008-z`.

[Int07] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide*. Tech. rep. Intel, 2007, p. 64.

[IO01] S. S. Ishtiaq and P. W. O’Hearn. “BI as an assertion language for mutable data structures.” In: *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’01. London, United Kingdom: ACM, 2001, pp. 14–26. isbn: 1-58113-336-7. doi: `10.1145/360204.375719`.

[Jan+06] G. Janssen et al. *Design of a Pointerless BDD Package*. US Patent 6,993,732. 2006.

[JLX09] Z. Jianhua, W. Linzhang, and L. Xuandong. “A Partial Order Reduction Technique for Parallel Timed Automaton Model Checking.” In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by T. Margaria and B. Steffen. Vol. 17. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2009, pp. 262–276. isbn: 978-3-540-88478-1. doi: `10.1007/978-3-540-88479-8_19`.

[JM09] R. Jhala and R. Majumdar. “Software model checking.” In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 21:1–21:54. issn: 0360-0300. doi: `10.1145/1592434.1592438`.

[Joh13] P. Johnson. “8 famous software bugs in space.” In: *IT World* (2013).

[JP08] B. Jacobs and F. Piessens. *The VeriFast Program Verifier*. 2008.

[JR10] M. d. Jonge and T. Ruys. “The SpinJa Model Checker.” In: *Model Checking Software*. Ed. by J. C. van de Pol and M. Weber. Vol. 6349. LNCS. Springer Berlin / Heidelberg, 2010, pp. 124–128. doi: `10.1007/978-3-642-16164-3_9`.

[KLJ10] I. Konnov and O. A. Letichevsky Jr. “Model Checking GARP Protocol using Spin and VRS.” In: *International Workshop on Automata, Algorithms, and Information Technologies* (2010).

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-time Systems.” In: *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[KP12] G. Kant and J. C. van de Pol. “Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games.” In: *ArXiv e-prints* (Oct. 2012).

[KP88a] S. Katz and D. Peled. “An efficient verification method for parallel and distributed programs.” In: *REX Workshop*. Ed. by J. W. de Bakker, W. P. de Roever, and G. Rozenberg. Vol. 354. LNCS. Springer, 1988, pp. 489–507. isbn: 3-540-51080-X.

[KP88b] S. Katz and D. Peled. “An Efficient Verification Method for Parallel and Distributed Programs.” In: *REX workshop*. Vol. 398. LNCS. Springer, 1988, pp. 489–507. isbn: 978-3-540-51080-2. doi: `10.1007/BFb0013032`.

[Kre+] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal. *An Efficient Distributed Algorithm for Finding Terminal Strongly Connected Components*. url: `http://www.cs.vu.nl/~ekr/papers/tscc.pdf`.

[Kri71] S. A. Kripke. *Semantical Considerations on Modal Logic; Naming and Necessity*. Oxford University Press, 1971.

[Kru05] R. M. Krueger. “Graph Searching.” AAINR07605. PhD thesis. Toronto, Ont., Canada, 2005. isbn: 0-494-07605-4.

[Kur+98] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. “Static partial order reduction.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by B. Steffen. Vol. 1384. LNCS. Springer Berlin Heidelberg, 1998, pp. 345–357. isbn: 978-3-540-64356-2. doi: `10.1007/BFb0054182`. url: `http://dx.doi.org/10.1007/BFb0054182`.

[KZ10] I. V. Konnov and V. A. Zakharov. “Using Adaptive Symmetry Reduction for LTL Model Checking.” In: *International Workshop on Program Semantics, Specification and Verification (PSSV 2010) affiliated with CSR 2010*. Available as `http://lvk.cs.msu.su/~konnov/publications/k10_cheaps_abstract.pdf`. 2010, pp. 5–11.

[Laa] A. W. Laarman. *DiVinE, SPIN and LTSmin Performance Comparison*. `http://fmt.cs.utwente.nl/tools/ltsmin/performance`.

[Laa+11] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. J. Wijs. “Multi-core Nested Depth-First Search.” In: *Automated Technology for Verification and Analysis*. Ed. by T. Bultan and P.-A. Hsiung. Vol. 6996. LNCS. Springer Berlin Heidelberg, 2011, pp. 321–335. ISBN: 978-3-642-24371-4. DOI: `10.1007/978-3-642-24372-1_23`.

[Laa+13a] A. W. Laarman, E. Pater, J. C. van de Pol, and M. Weber. “Guard-Based Partial-Order Reduction.” In: *Model Checking Software*. Ed. by E. Bartocci and C. R. Ramakrishnan. Vol. 7976. LNCS. Springer Berlin Heidelberg, 2013, pp. 227–245. ISBN: 978-3-642-39175-0. DOI: `10.1007/978-3-642-39176-7_15`.

[Laa+13b] A. W. Laarman, M. C. Olesen, A. E. Dalsgaard, K. G. Larsen, and J. C. van de Pol. “Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction.” In: *Computer Aided Verification*. Ed. by N. Sharygina and H. Veith. Vol. 8044. LNCS. Springer Berlin Heidelberg, 2013, pp. 968–983. ISBN: 978-3-642-39798-1. DOI: `10.1007/978-3-642-39799-8_69`.

[Laa11] A. W. Laarman. *LTSmin benchmark results*. `http://fmt.cs.utwente.nl/tools/ltsmin/spin-2011/`. Last accessed: 24 Jan 2011. 2011.

[Lar+97] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. “Efficient Verification of Real-Time Systems: Compact Data Structure and State–Space Reduction.” In: *Proc. of the 18th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1997, pp. 14–24.

[LF13] A. W. Laarman and D. Faragó. “Improved on-the-Fly Livelock Detection.” In: *NASA Formal Methods*. Ed. by G. Brat, N. Rungta, and A. Venet. Vol. 7871. LNCS. Springer Berlin Heidelberg, 2013, pp. 32–47. ISBN: 978-3-642-38087-7. DOI: `10.1007/978-3-642-38088-4_3`.

[LH00] T. Latvala and K. Heljanko. “Coping With Strong Fairness.” In: *Fundam. Inf.* 43.1-4 (Jan. 2000), pp. 175–193. ISSN: 0169-2968.

[Li09] G. Li. “Checking Timed Büchi Automata Emptiness using LU-Abstractions.” In: *FORMATS*. Vol. 5813. LNCS. Springer, 2009, pp. 228–242.

[Lit80] W. Litwin. “Linear hashing: a new tool for file and table addressing.” In: *VLDB ’1980: Proceedings of the sixth international conference on Very Large Data Bases*. Montreal, Quebec, Canada: VLDB Endowment, 1980, pp. 212–223.

[LL95] J. A. Lee and J. A. Lee. *International biographical dictionary of computer pioneers*. Taylor & Francis US, 1995.

[Llu] A. Lluch. *Promela Database*. http://www.albertolluch.com/research/promelamodels.

[Low14] G. Lowe. "Concurrent Depth-First Search Algorithms." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by E. Abrahḿ and K. Havelund. Vol. 8413. LNCS. Springer Berlin Heidelberg, 2014, pp. 202–216. ISBN: 978-3-642-54861-1. DOI: 10.1007/978-3-642-54862-8_14.

[LP11] A. W. Laarman and J. C. van de Pol. "Variations on Multi-Core Nested Depth-First Search." In: Proceedings 10th International Workshop on *Parallel and Distributed Methods in verifiCation,* Snowbird, Utah, USA, July 14, 2011. Ed. by J. Barnat and K. Heljanko. Vol. 72. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2011, pp. 13–28. DOI: 10.4204/EPTCS.72.2.

[LPW10a] A. W. Laarman, J. C. van de Pol, and M. Weber. "Boosting Multi-Core Reachability Performance with Shared Hash Tables." In: *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Switzerland*. Ed. by N. Sharygina and R. Bloem. Lugano, Switzerland: IEEE Computer Society, 2010. URL: http://dl.acm.org/citation.cfm?id=1998496.1998541.

[LPW10b] A. W. Laarman, J. C. van Pol, and M. Weber. "Boosting Multi-Core Reachability Performance with Shared Hash Tables." In: *ArXiv e-prints* 1004.2772 (Apr. 2010). arXiv: 1004.2772.

[LPW11a] A. W. Laarman, J. C. van de Pol, and M. Weber. "Multi-Core LTSmin: Marrying Modularity and Scalability." In: *NASA Formal Methods*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Vol. 6617. LNCS. Pasadena, CA, USA: Springer Verlag, 2011, pp. 506–511. DOI: 10.1007/978-3-642-20398-5_40.

[LPW11b] A. W. Laarman, J. C. van de Pol, and M. Weber. "Parallel Recursive State Compression for Free." In: *ArXiv e-prints* abs/1104.3119 (2011). arXiv: 1104.3119.

[LPW11c] A. W. Laarman, J. C. van de Pol, and M. Weber. "Parallel Recursive State Compression for Free." In: *Model Checking Software*. Ed. by A. Groce and M. Musuvathi. Vol. 6823. LNCS. Springer Berlin Heidelberg, 2011, pp. 38–56. ISBN: 978-3-642-22305-1. DOI: 10.1007/978-3-642-22306-8_4.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. “Uppaal in a Nutshell.” In: *STTT* 1 (1 1997), pp. 134–152.

[LRG03] I. v. Langevelde, J. Romijn, and N. Goga. “Founding FireWire bridges through Promela prototyping.” In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. 2003, 8 pp.–. doi: 10.1109/IPDPS.2003.1213434.

[LS99] F. Lerda and R. Sisto. “Distributed-Memory Model Checking with SPIN.” English. In: *Theoretical and Practical Aspects of SPIN Model Checking*. Ed. by D. Dams, R. Gerth, S. Leue, and M. Massink. Vol. 1680. LNCS. Springer Berlin Heidelberg, 1999, pp. 22–39. isbn: 978-3-540-66499-4. doi: 10.1007/3-540-48234-2_3.

[LSD09a] Y. Liu, J. Sun, and J. S. Dong. “Scalable Multi-core Model Checking Fairness Enhanced Systems.” In: *ICFEM 2009*. Vol. 5885. LNCS. Springer, Heidelberg, 2009, pp. 426–445.

[LSD09b] Y. Liu, J. Sun, and J. Dong. “Scalable Multi-core Model Checking Fairness Enhanced Systems.” In: *Formal Methods and Software Engineering*. Ed. by K. Breitman and A. Cavalcanti. Vol. 5885. LNCS. Springer Berlin Heidelberg, 2009, pp. 426–445. isbn: 978-3-642-10372-8. doi: 10.1007/978-3-642-10373-5_22.

[LSD11] Y. Liu, J. Sun, and J. S. Dong. “PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers.” In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering* 0 (2011), pp. 190–199. issn: 1071-9458. doi: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2011.19.

[LT04] T. Latvala and H. Tauriainen. “Improved On-the-fly Verification with Testers.” In: *Nordic Journal of Computing* 11 (2004), pp. 148–164.

[LT93] N. G. Leveson and C. S. Turner. “An investigation of the Therac-25 accidents.” In: *Computer* 26.7 (1993), pp. 18–41. issn: 0018-9162. doi: 10.1109/MC.1993.274940.

[Luo01] L. Luo. “Software testing techniques.” In: *International Institution for Software Research, Carnegie Mellon University Pittsburgh, PA* 15232.1-19 (2001), p. 19.

[Man+13] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci. “System Level Formal Verification via Model Checking Driven Simulation.” In: *Computer Aided Verification*. Ed. by N. Sharygina and H. Veith. Vol. 8044. LNCS. Springer Berlin Heidelberg, 2013, pp. 296–312. ISBN: 978-3-642-39798-1. DOI: `10.1007/978-3-642-39799-8_21`.

[McM92] K. L. McMillan. “Symbolic model checking: an approach to the state explosion problem.” UMI No. GAX92-24209. PhD thesis. Pittsburgh, PA, USA, 1992.

[McM93] K. L. McMillan. “Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits.” In: *Computer Aided Verification*. Ed. by G. Bochmann and D. Probst. Vol. 663. LNCS. Springer Berlin Heidelberg, 1993, pp. 164–177. ISBN: 978-3-540-56496-6. DOI: `10.1007/3-540-56496-9_14`.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.” In: *ACM Transactions on Computer Systems* 9 (1991), pp. 21–65.

[MD13] R. Macey-Dare. “Economic Consequences of Moore’s Law.” In: *Available at SSRN 2261119* (2013).

[MH00] J. M. Martin and Y. Huddart. “Parallel algorithms for deadlock and livelock analysis of concurrent systems.” In: *Communicating Process Architectures* (2000), pp. 1–14.

[Mic02] M. M. Michael. “High performance dynamic lock-free hash tables and list-based sets.” In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. SPAA ’02. Winnipeg, Manitoba, Canada: ACM, 2002, pp. 73–82. ISBN: 1-58113-529-7. DOI: `10.1145/564870.564881`.

[Mil56] G. A. Miller. “The magical number seven, plus or minus two: some limits on our capacity for processing information.” In: *Psychological review* 63.2 (1956), p. 81.

[Min99] M. Minea. “Partial Order Reduction for Model Checking of Timed Automata.” English. In: *CONCUR’99*. Ed. by J. C. M. Baeten and S. Mauw. Vol. 1664. LNCS. Springer Berlin Heidelberg, 1999, pp. 431–446. ISBN: 978-3-540-66425-3. DOI: `10.1007/3-540-48320-9_30`.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. "Automated analysis of cryptographic protocols using Mur phi;" in: *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. 1997, pp. 141–151. DOI: `10.1109/SECPRI.1997.601329`.

[Moo65] G. E. Moore. "Cramming more Components onto Integrated Circuits." In: vol. 38. 10. IEEE Computer Society, 1965, pp. 114–117.

[MP09] M. Monagan and R. Pearce. "Parallel sparse polynomial multiplication using heaps." In: *ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation*. Seoul, Republic of Korea: ACM, 2009, pp. 263–270. ISBN: 978-1-60558-609-0. DOI: `http://doi.acm.org/10.1145/1576702.1576739`.

[MS03] R. Mateescu and M. Sighireanu. "Efficient On-the-fly Model-checking for Regular Alternation-free Mu-calculus." In: *Sci. Comput. Program.* 46.3 (Mar. 2003), pp. 255–281. ISSN: 0167-6423. DOI: `10.1016/S0167-6423(02)00094-1`.

[MSB11] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[MW09] R. Mateescu and A. J. Wijs. "Hierarchical Adaptive State Space Caching Based on Level Sampling." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. LNCS. Springer Berlin Heidelberg, 2009, pp. 215–229. ISBN: 978-3-642-00767-5. DOI: `10.1007/978-3-642-00768-2_21`.

[Nec02] G. C. Necula. "Proof-Carrying Code. Design and Implementation." English. In: *Proof and System-Reliability*. Ed. by H. Schwichtenberg and R. Steinbrüggen. Vol. 62. NATO Science Series. Springer Netherlands, 2002, pp. 261–288. ISBN: 978-1-4020-0608-1. DOI: `10.1007/978-94-010-0413-8_8`.

[Nee14] *Design of a Scalable Hash Table on a GPU*. Vol. 20. January 01. University of Twente, 2014.

[NG02] R. Nalumasu and G. Gopalakrishnan. "An Efficient Partial Order Reduction Algorithm with an Alternative Proviso Implementation." In: *Formal Methods in System Design* 20 (3 2002), pp. 231–247. ISSN: 0925-9856. DOI: `10.1023/A:1014728912264`.

[NIPD96] C. Norris I. P. and D. L. Dill. "Better verification through symmetry." English. In: *Formal Methods in System Design* 9.1-2 (1996), pp. 41–75. ISSN: 0925-9856. DOI: `10.1007/BF00625968`.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. “Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T).” In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977. ISSN: 0004-5411. DOI: `10.1145/1217856.1217859`.

[NR08] V. Nguyen and T. Ruys. “Incremental Hashing for Spin.” In: *Model Checking Software*. Ed. by K. Havelund, R. Majumdar, and J. Palsberg. Vol. 5156. LNCS. Springer Berlin Heidelberg, 2008, pp. 232–249. ISBN: 978-3-540-85113-4. DOI: `10.1007/978-3-540-85114-1_17`.

[ORY01] P. O’Hearn, J. Reynolds, and H. Yang. “Local Reasoning about Programs that Alter Data Structures.” English. In: *Computer Science Logic*. Ed. by L. Fribourg. Vol. 2142. LNCS. Springer Berlin Heidelberg, 2001, pp. 1–19. ISBN: 978-3-540-42554-0. DOI: `10.1007/3-540-44802-0_1`.

[Ove81] W. T. Overman. “Verification of concurrent systems: function and timing.” AAI8121023. PhD thesis. University of California, Los Angeles, 1981.

[Pat11] E. Pater. “Partial Order Reduction for PINS.” MA thesis. University of Twente, 2011. URL: `http://essay.utwente.nl/61036/`.

[Pel07] R. Pelánek. “BEEM: Benchmarks for Explicit Model Checkers.” In: *Proc. of SPIN Workshop*. Vol. 4595. LNCS. Springer, 2007, pp. 263–267.

[Pel11] R. Pelánek. *The BEEM website*. `http://anna.fi.muni.cz/models/cgi/models.cgi`. Last accessed: 24 Jan 2011. 2011.

[Pen+02] G. Penna, B. Intrigila, E. Tronci, and M. V. Zilli. “Exploiting Transition Locality in the Disk Based Murφ Verifier.” English. In: *Formal Methods in Computer-Aided Design*. Ed. by M. D. Aagaard and J. W. O’Leary. Vol. 2517. LNCS. Springer Berlin Heidelberg, 2002, pp. 202–219. ISBN: 978-3-540-00116-4. DOI: `10.1007/3-540-36126-X_13`.

[PH05] C. Purcell and T. Harris. “Non-blocking Hashtables with Open Addressing.” In: *Distributed Computing* (2005), pp. 108–121. URL: `http://dx.doi.org/10.1007/11561927_10`.

[Pip81] N. Pippenger. “Pebbling with an auxiliary pushdown.” In: *Journal of Computer and System Sciences* 23.2 (1981), pp. 151 –165. ISSN: 0022-0000. DOI: `http://dx.doi.org/10.1016/0022-0000(81)90011-8`.

[Pnu77] A. Pnueli. “The temporal logic of programs.” In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[Pou04] K. Poulsen. “Software bug contributed to blackout.” In: *Security Focus* (2004).

[PR04] R. Pagh and F. F. Rodler. “Cuckoo hashing.” In: *Journal of Algorithms* 51.2 (2004), pp. 122 –144. issn: 0196-6774. doi: DOI:10.1016/j.jalgor.2003.12.002.

[PSS07] F. Putze, P. Sanders, and J. Singler. “Cache-, Hash- and Space-Efficient Bloom Filters.” In: *Experimental Algorithms*. Ed. by C. Demetrescu. Vol. 4525. LNCS. Springer Berlin Heidelberg, 2007, pp. 108–121. isbn: 978-3-540-72844-3. doi: 10.1007/978-3-540-72845-0_9.

[PW02] L. C. Paulson and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer, 2002.

[QS08] J. P. Queille and J. Sifakis. “SPECIFICATION AND VERIFICATION OF CONURRENT SYSTEMS IN CESAR.” In: *25 Years of Model Checking*. Ed. by O. Grumberg and H. Veith. Vol. 5000. LNCS. Springer Berlin Heidelberg, 2008, pp. 216–230. isbn: 978-3-540-69849-4. doi: 10.1007/978-3-540-69850-0_13.

[QS82] J.-P. Queille and J. Sifakis. “Specification and verification of concurrent systems in CESAR.” In: *Proceedings of the 5th Colloquium on International Symposium on Programming*. London, UK, UK: Springer-Verlag, 1982, pp. 337–351. isbn: 3-540-11494-7. url: http://dl.acm.org/citation.cfm?id=647325.721668.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Tech. rep. Cambridge, MA, USA, 1974.

[Rei85] J. H. Reif. “Depth-first Search is Inherently Sequential.” In: *Information Processing Letters* 20.5 (1985), pp. 229–234. doi: 10.1016/0020-0190(85)90024-9.

[RK88] N. V. Rao and V. Kumar. “Superlinear speedup in parallel state-space search.” In: *Foundations of Software Technology and Theoretical Computer Science* (1988), pp. 161–174. url: http://dx.doi.org/10.1007/3-540-50517-2_79.

[Rom99] J. M. T. Romijn. “Analysing Industrial Protocols with Formal Methods.” PhD thesis. Enschede, 1999. url: http://doc.utwente.nl/17910/.

[Saa11] R. T. Saad. “Parallel Model Checking for Multiprocessor Architecture.” PhD thesis. Institut National des Sciences Appliquées, Toulouse, France (December 2011), 2011.

[San97a] P. Sanders. “Lastverteilungsalgorithmen fur parallele Tiefensuche. Number 463.” In: *in Fortschrittsberichte, Reihe 10. VDI*. Verlag, 1997.

[San97b] P. Sanders. “Load Balancing Algorithms for Parallel Depth First Search.” PhD thesis. University of Karlsruhe, 1997.

[Sav70] W. J. Savitch. “Relationships between nondeterministic and deterministic tape complexities.” In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177 –192. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/S0022-0000(70)80006-X.

[Sch+12] S. Schivo, J. Scholma, B. Wanders, R. A. U. Camacho, P. E. van der Vet, M. Karperien, R. Langerak, J. C. van de Pol, and J. N. Post. “Modelling biological pathway dynamics with Timed Automata.” In: *Bioinformatics Bioengineering (BIBE), 2012 IEEE 12th International Conference on*. 2012, pp. 447–453. DOI: 10.1109/BIBE.2012.6399719.

[SD97] U. Stern and D. L. Dill. “Parallelizing the Murφ Verifier.” In: *Computer-Aided Verification, 9th International Conference*. Ed. by O. Grumberg. Vol. 1254. LNCS. Haifa, Israel, June 22-25. Springer, 1997, pp. 256–267. URL: http://sprout.stanford.edu/PAPERS/SD97.ps.

[SD98] U. Stern and D. L. Dill. “Using magnetic disk instead of main memory in the Murφ verifier.” In: *Computer Aided Verification*. Ed. by A. Hu and M. Vardi. Vol. 1427. LNCS. Springer Berlin Heidelberg, 1998, pp. 172–183. ISBN: 978-3-540-64608-2. DOI: 10.1007/BFb0028743.

[SE05] S. Schwoon and J. Esparza. “A Note on On-the-Fly Verification Algorithms.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by N. Halbwachs and L. D. Zuck. Vol. 3440. LNCS. Springer Berlin / Heidelberg, 2005, pp. 174–190. URL: http://dx.doi.org/10.1007/978-3-540-31980-1_12.

[SEK11] D. Sulewski, S. Edelkamp, and P. Kissmann. “Exploiting the computational power of the graphics card: Optimal state space planning on the GPU.” In: *Twenty-First International Conference on Automated Planning and Scheduling*. 2011.

[SG03] H. Sivaraj and G. Gopalakrishnan. “Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking.” In: *Electronic Notes in Theoretical Computer Science* 89.1 (2003), pp. 51–67.

[She+10] A. Sherif, A. Cavalcanti, H. Jifeng, and A. Sampaio. "A process algebraic framework for specification and validation of real-time systems." English. In: *Formal Aspects of Computing* 22.2 (2010), pp. 153–191. ISSN: 0934-5043. DOI: 10.1007/s00165-009-0119-6.

[Ske92] R. Skeel. "Roundoff error and the Patriot missile." In: *SIAM News* 25.4 (1992), p. 11.

[SM79] B. Shneiderman and R. Mayer. "Syntactic/semantic interactions in programmer behavior: A model and experimental results." English. In: *International Journal of Computer & Information Sciences* 8.3 (1979), pp. 219–238. ISSN: 0091-7036. DOI: 10.1007/BF00977789.

[Spi07] D. Spinellis. "Another level of indirection." In: *Beautiful code: leading programmers explain how they think* (2007), pp. 279–291.

[SS06] O. Shalev and N. Shavit. "Split-ordered lists: Lock-free extensible hash tables." In: *J. ACM* 53.3 (May 2006), pp. 379–405. ISSN: 0004-5411. DOI: 10.1145/1147954.1147958.

[Ste+99] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. S. LaPiana, P. J. Rutledge, D. Folta, and R. Sackheim. "Mars Climate Orbiter Mishap Investigation Board Phase I Report, 44 pp." In: *NASA, Washington, DC* (1999).

[Sti13] V. Still. "State space compression for the DiVinE model checker." B. S. Thesis. Masaryk University, Faculty of Informatics, 2013. URL: http://is.muni.cz/th/373979/fi_b/.

[Sul12] D. Sulewski. "Large scale parallel state space search utilizing graphics processing units and solid state disks." PhD thesis. Technische Universität Dortmund, 2012.

[Sut09] G. Sutcliffe. "The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0." In: vol. 43. 4. 2009, pp. 337–362.

[SZB10] R. T. Saad, S. D. Zilio, and B. Berthomieu. "A General Lock-Free Algorithm for Parallel State Space Construction." In: *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*. 2010, pp. 8–16. DOI: 10.1109/PDMC-HiBi.2010.10.

[SZB11] R. T. Saad, S. D. Zilio, and B. Berthomieu. "Mixed Shared-Distributed Hash Tables Approaches for Parallel State Space Construction." In: *Parallel and Distributed Computing (ISPDC), 2011 10th International Symposium on*. 2011, pp. 9–16. DOI: 10.1109/ISPDC.2011.12.

[SZB12] R. T. Saad, S. Zilio, and B. Berthomieu. “An Experiment on Parallel Model Checking of a CTL Fragment.” In: *ATVA*. Ed. by S. Chakraborty and M. Mukund. Vol. 7561. LNCS. Springer, ’12, pp. 284–299. isbn: 978-3-642-33385-9. doi: 10.1007/978-3-642-33386-6_23.

[Tar72] R. Tarjan. “Depth-First Search and Linear Graph Algorithms.” In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160.

[Tim11] M. Timmer. “SCOOP: A Tool for SymboliC Optimisations of Probabilistic Processes.” In: *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*. 2011, pp. 149–150. doi: 10.1109/QEST.2011.27.

[Tim13] M. Timmer. “Efficient Modelling, Generation and Analysis of Markov Automata.” PhD thesis. Enschede: University of Twente, 2013. doi: 10.3990/1.9789036505925.

[Tre99] J. Tretmans. “Testing Concurrent Systems: A Formal Approach.” English. In: *CONCUR’99*. Ed. by J. C. M. Baeten and S. Mauw. Vol. 1664. LNCS. Springer Berlin Heidelberg, 1999, pp. 46–65. isbn: 978-3-540-66425-3. doi: 10.1007/3-540-48320-9_6.

[Tri09] S. Tripakis. “Checking timed Büchi Automata Emptiness on Simulation Graphs.” In: *TOCL* 10.3 (2009), p. 15.

[TSP11] M. Timmer, M. Stoelinga, and J. C. van de Pol. “Confluence Reduction for Probabilistic Systems.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by P. A. Abdulla and K. R. M. Leino. Vol. 6605. LNCS. Springer Berlin Heidelberg, 2011, pp. 311–325. isbn: 978-3-642-19834-2. doi: 10.1007/978-3-642-19835-9_29.

[TYB05] S. Tripakis, S. Yovine, and A. Bouajjani. “Checking Timed Büchi automata Emptiness Efficiently.” English. In: *Formal Methods in System Design* 26.3 (2005), pp. 267–292. issn: 0925-9856. doi: 10.1007/s10703-005-1632-8.

[Unk95] Unknown. “Intel Cuts Prices of Pentium Chips.” In: *New York Times* (1995).

[Val88] A. Valmari. “Error Detection by Reduced Reachability Graph Generation.” In: *APN*. 1988, pp. 95–112.

[Val89] A. Valmari. “Eliminating Redundant Interleavings During Concurrent Program Verification.” In: *PARLE*. Ed. by E. Odijk, M. Rem, and J.-C. Syre. Vol. 366. LNCS. Springer, 1989, pp. 89–103. isbn: 978-3-540-51285-1. doi: 10.1007/3-540-51285-3_35.

[Val91a] A. Valmari. “A Stubborn Attack On State Explosion.” In: *Proceedings of the 2nd International Workshop on Computer Aided Verification*. LNCS. London, UK: Springer, 1991, pp. 156–165. ISBN: 3-540-54477-1. URL: http://portal.acm.org/citation.cfm?id=647759.735025.

[Val91b] A. Valmari. “Stubborn Sets for Reduced State Space Generation.” In: *APN*. Vol. 483. LNCS. Springer, 1991, pp. 491–515. ISBN: 978-3-540-53863-9. DOI: 10.1007/3-540-53863-1_36.

[Val93] A. Valmari. “On-the-fly verification with stubborn sets.” In: *Computer Aided Verification*. Ed. by C. Courcoubetis. Vol. 697. LNCS. Springer Berlin Heidelberg, 1993, pp. 397–408. ISBN: 978-3-540-56922-0. DOI: 10.1007/3-540-56922-7_33.

[Val97] A. Valmari. “Stubborn Set Methods for Process Algebras.” In: *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification*. POMIV ’96. Princeton, New Jersey, USA: AMS Press, Inc., 1997, pp. 213–231. ISBN: 0-8218-0579-7. URL: http://dl.acm.org/citation.cfm?id=266557.266608.

[Val98] A. Valmari. “The State Explosion Problem.” In: *LPN*. London, UK: Springer, 1998, pp. 429–528. ISBN: 3-540-65306-6. URL: http://dl.acm.org/citation.cfm?id=647444.727054.

[Var01] M. Vardi. “Branching vs. Linear Time: Final Showdown.” English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by T. Margaria and W. Yi. Vol. 2031. LNCS. Springer Berlin Heidelberg, 2001, pp. 1–22. ISBN: 978-3-540-41865-8. DOI: 10.1007/3-540-45319-9_1.

[Var11] M. Y. Vardi. *Parallelism a Siren’s Song*. Invited talk, PDMC 2011, Snow Bird, Utah, USA. 2011.

[Veg11] S. van der Vegt. “A Concurrent Bidirectional Linear Probing Algorithm.” In: *15th Twente Student Conference on Information Technology, Enschede, The Netherlands*. Ed. by C. Heijnen and H. Koppelman. Vol. 15. TSConIT. Enschede, The Netherlands: Twente University Press, 2011, pp. 269–276. URL: http://referaat.cs.utwente.nl/conference/15/paper.

[Vis96] W. Visser. “Memory Efficient State Storage in SPIN.” In: *In Proceedings of the 2nd SPIN Workshop*. 1996, pp. 21–35.

[VL12] S. van der Vegt and A. W. Laarman. “A Parallel Compact Hash Table.” In: *Mathematical and Engineering Methods in Computer Science*. Ed. by Z. Kotásek, J. Bouda, I. Černá, L. Sekanina, T. Vojnar, and D. Antoš. Vol. 7119. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 191–204. ISBN: 978-3-642-25928-9. DOI: `10.1007/978-3-642-25929-6_18`.

[VW86] M. Y. Vardi and P. Wolper. “An Automata-Theoretic Approach to Automatic Program Verification.” In: *Proc. 1st Symp. on Logic in Computer Science*. Cambridge, 1986, pp. 332–344. URL: `http://www.cs.rice.edu/~vardi/papers/lics86.pdf.gz`.

[WB14] A. J. Wijs and D. Bošnački. “GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by E. Abrahám and K. Havelund. Vol. 8413. LNCS. Springer Berlin Heidelberg, 2014, pp. 233–247. ISBN: 978-3-642-54861-1. DOI: `10.1007/978-3-642-54862-8_16`.

[Web07] M. Weber. “An Embeddable Virtual Machine for State Space Generation.” In: *SPIN*. Ed. by D. Bošnački and S. Edelkamp. Vol. 4595. LNCS. Springer, 2007, pp. 168–186. ISBN: 978-3-540-73369-0. URL: `http://dx.doi.org/10.1007/978-3-540-73370-6\_12`.

[WH13a] S. Wieringa and K. Heljanko. “Asynchronous Multi-core Incremental SAT Solving.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by N. Piterman and S. Smolka. Vol. 7795. LNCS. Springer Berlin Heidelberg, 2013, pp. 139–153. ISBN: 978-3-642-36741-0. DOI: `10.1007/978-3-642-36742-7_10`.

[WH13b] S. Wieringa and K. Heljanko. “Concurrent Clause Strengthening.” In: *SAT 2013*. Ed. by M. Jãrvisalo and A. Van Gelder. Vol. 7962. LNCS. Springer Berlin Heidelberg, 2013, pp. 116–132. ISBN: 978-3-642-39070-8. DOI: `10.1007/978-3-642-39071-5_10`.

[Wij11] A. J. Wijs. “Towards Informed Swarm Verification.” In: *NASA Formal Methods*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Vol. 6617. LNCS. Springer Berlin Heidelberg, 2011, pp. 422–437. ISBN: 978-3-642-20397-8. DOI: `10.1007/978-3-642-20398-5_30`.

[WL93] P. Wolper and D. Leroy. “Reliable Hashing without Collision Detection.” In: *the 5th International Conference of Computer Aided Verification*. Springer-Verlag, 1993, pp. 59–70.

[WWP09] S. Williams, A. Waterman, and D. Patterson. “Roofline: an insightful visual performance model for multicore architectures.” In: *Commun. ACM* 52 (4 2009), pp. 65–76. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/1498765.1498785.

[XLHS13] Y. L. Xuan-Linh Ha Thanh-Tho Quan and J. Sun. “Multi-core Algorithms for LTL Verification.” In: *The 20th Asia-Pacific Software Engineering Conference*. APSEC 2013. Bangkok, Thailand, 2013, accepted for publication.

[Xu+11] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. “SATzilla: Portfolio-based Algorithm Selection for SAT.” In: vol. abs/1111.2249. 2011.

[YO97] B. Yang and D. R. O’Hallaron. “Parallel breadth-first BDD construction.” In: *PPOPP ’97: Proc. of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*. Las Vegas, Nevada, United States: ACM, 1997, pp. 145–156. ISBN: 0-89791-906-8. DOI: http://doi.acm.org/10.1145/263764.263784.

[Zal11] D. Zalubowski. “Toyota settlement in sudden-acceleration case will top $1 billion.” In: *Los Angelos Times* (2011).

[Zob69] A. L. Zobrist. *A New Hashing Method with Application for Game Playing*. Tech. rep. 88. Computer Sciences Department, University of Wisconsin, 1969.

[ČP03] I. Černá and R. Pelánek. “Distributed Explicit Fair Cycle Detection (Set Based Approach).” In: *SPIN*. Ed. by T. Ball and S. K. Rajamani. Vol. 2648. LNCS. Springer, 2003, pp. 49–73. ISBN: 3-540-40117-2. DOI: 10.1007/3-540-44829-2_4.

Index

Samenvatting

Onze moderne samenleving is in toenemende mate afhankelijk van de correcte werking van digitale systemen. Het is geen triviale exercitie om te garanderen dat deze systemen ook werkelijk correct volgens hun specificatie functioneren. Toch is dit essentieel voor systemen die van levensbelang zijn, zoals een automatische piloot, kerncentrale en de ABS in uw auto.

De hoogste mate van vertrouwen die we kunnen verkrijgen in de correctheid van een systeem, is via wiskundig bewijs. Dit is een arbeidsintensief proces waarbij het gedrag van het systeem eerst formeel beschreven wordt en daarna geanalyseerd wordt. Vooral die laatste stap is tijdrovend en vereist de creativiteit van een wiskundige om te demonstreren dat bepaalde eigenschappen blijven gelden onder de strikte wiskunderegels. Met de ontdekking van 'model-checking' is die laatste taak helemaal geautomatiseerd door het gedrag van het systeem met een computer volledig te doorzoeken.

Desondanks wordt de grootte van de systemen die we kunnen 'model-checken' sterk beperkt door de hoeveelheid beschikbare rekenkracht. De oorzaak hiervan is de zogeheten toestandsexplosie, die ontstaat doordat deze automatische aanpak alleen maar kleine gemechaniseerde stappen kan maken en niet zoals de wiskundige beschikt over de creativiteit om generaliserende (denk)stappen te maken. Daarom is het doel van dit proefschrift om de volledige rekenkracht van moderne multikerncomputers te benutten voor de model-checking-taak (vandaar "multi-core"). De parallelle procedures die wij presenteren, benutten alle beschikbare processorkernen, en behalen een versnelling die proportioneel is aan het aantal kernen, oftewel ze zijn schaalbaar (vandaar "scalable").

Dit proefschrift bereikt de efficiënte parallelisatie van een breed scala aan model-checking-procedures in drie stappen, elk beschreven in een deel van het proefschrift:

Ten eerste passen we lockless hashtabellen aan voor 'explicit-state reachability', het onderliggende zoekalgoritme dat de volledige toestandsruimte van een systeem doorzoekt. Met behulp van een boom (een bepaald soort datastructuur) realiseren we toestandscompressie, wat leidt tot een significante reductie van de hoeveelheid aan gebruikt geheugen. Incrementele wijzigingen in deze boom zorgen voor vergelijkbare performance en schaalbaarheid als de lockless hashtabel, terwijl de combinatie met een compacte hashtabel het geheugen kan comprimeren tot ongeveer 4 bytes per toestand, zelfs bij de opslag van meer dan 10 miljard states. Empirisch bewijs laat zien dat de compressie heel vaak binnen de 110% van dit optimale geval ligt.

Ten tweede hebben we parallelle 'nested depth-first search'-algoritmen ontwikkelt om model-checking van LTL in lineaire tijd te ondersteunen. Voortbordurend op de resultaten van onze algoritmen voor multikern-reachability laten we meerdere processen semi-onafhankelijk van elkaar door de toestandsruimte zoeken. Deze techniek is gebaseerd op zwerm-achtige ('swarm-based') verificatie methoden, die lage communicatiekosten uitbuiten door gebruik te maken van een mogelijk redundante planning ('scheduling') van het werk. Daarom vormt deze methode een mogelijke oplossing voor een toekomstscenario waarin communicatiekosten groeien met de toenemende steilheid van de geheugen hiërarchie in computer systemen. Experimenten op huidige hardware tonen al aan dat deze methode weinig overbodig werk verricht en ook nog goed schaalt.

Ten derde, om uiteindelijk ook de verificatie van real-time systemen te ondersteunen, hebben we onze oplossingen voor de multikern-zoekalgoritmen en het checken van LTL vertaald naar het domein van 'timed-automata'. We hebben daarvoor een lockless 'multimap' ontwikkeld, die toestanden met tijd abstractie kan opslaan. Ook presenteren we algoritmen die kunnen omgaan met de grove subsumptieabstractie voor de verificatie van LTL-eigenschappen, en daardoor grotere probleeminstanties kunnen oplossen. De schaalbaarheid, geheugencompressie en performance worden allemaal behouden in de setting waaraan tijd is toegevoegd. Experimenten laten daarom grote vooruitgang zien in vergelijking tot de state-of-the-art model-checker UPPAAL.

De bovenstaande technieken zijn allemaal geïmplementeerd in de model-checker LTSMIN. Deze is taal-onafhankelijk en leent zich daardoor uitstekend voor directe vergelijking met andere model-checkers. We presenteren experimentele vergelijkingen met de state-of-the-art expliciete model-checkers SPIN en DIVINE. Beide implementeren multikern-algoritmen, terwijl DIVINE ook de focus legt op gedistribueerde verificatie. Deze experimenten tonen aan dat de voorgestelde technieken significante vooruitgang bieden in termen van schaalbaarheid, absolute performance en geheugengebruik.

Huidige trends en voorspellingen vertellen ons dat het aantal processorkernen exponentieel toe zal nemen met de tijd (Moore's Law). Onze resultaten zijn mogelijk in staat te profiteren van deze trend. Of de voorgestelde methoden ook werkelijk de tand des tijds zullen doorstaan blijft nog maar de vraag, maar vooralsnog heeft de versnelling van onze algoritmen de 3-voudige vermeerdering in het aantal kernen kunnen bijhouden gedurende de 4 jaren van dit onderzoek.

Titles in the IPA Dissertation Series since 2008

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06