Strong Connectivity and Shortest Paths for Checking Models

Vincent Bloemen
Strong Connectivity and Shortest Paths for Checking Models

Vincent Bloemen
Graduation Committee:
Chairman: Prof.dr. J. N. Kok
Promotors: Prof.dr. J. C. van de Pol
           Prof.dr.ir. W. M. P. van der Aalst

Members:
Dr.ir. R. Langerak University of Twente
Prof.dr. M. J. Uetz University of Twente
Prof.dr.ir. B. F. van Dongen Eindhoven University of Technology
Prof.dr. L. Petrucci LIPN, Université Paris 13

DSI Ph.D. Thesis Series No. 19–010
Institute on Digital Society, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

IPA Dissertation Series No. 2019–07
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISSN: 2589–7721 (DSI Ph.D. Thesis Series No. 19–010)
Available online at https://doi.org/10.3990/1.9789036547864

Typeset with \TeX
Printed by Ipskamp Printers Enschede
Cover design by Vincent Bloemen
Copyright © 2019 Vincent Bloemen
STRONG CONNECTIVITY AND SHORTEST PATHS FOR CHECKING MODELS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus
Prof. dr. T. T. M. Palstra,
on account of the decision of the Doctorate Board,
to be publicly defended
on Wednesday July 10th, 2019 at 16:45

by

Vincent Bloemen

Born on April 2nd, 1991
in Oldenzaal, The Netherlands
This dissertation has been approved by:

Prof.dr. J. C. van de Pol (promotor)
Prof.dr.ir. W. M. P. van der Aalst (promotor)
Abstract

We study directed graphs and focus on algorithms for two classical graph problems; the decomposition of a graph into Strongly Connected Components (SCCs), and the Single-Source Shortest Path problem. In particular, we concentrate on the development of new graph search algorithms for checking models, i.e. techniques that allow a user to analyse a system and verify whether particular properties are maintained. Our contributions advance the performance of state-of-the-art techniques for model checking and conformance checking. Moreover, we additionally pursue new directions to broaden the horizons of both fields.

We developed a multi-core algorithm for on-the-fly SCC decomposition that scales effectively on many-core systems. In its construction, we additionally developed an iterable concurrent union-find structure that may be used in other applications.

We considered SCCs in the domain of model checking and showed that our SCC decomposition algorithm can be applied to outperform the state-of-the-art techniques. Additionally, we explored how more general automata could be model checked by providing techniques to achieve this.

We studied the shortest path problem in the context of conformance checking, in particular for the computation of alignments. By exploiting characteristic choices for the cost function, we compute alignments via an algorithm based on symbolic reachability. We also consider an alternative cost function and show how this leads to a new data structure and algorithm.

Finally, we studied new problems for Parametric Timed Automata (PTAs), which extend timed automata with unknown constant values, or parameters. We developed algorithms to synthesize parameter values for the best- and worst-case behaviour. For instance, computing all parameter valuations such that a target location is reached in minimal- or maximal time.
In order to be prepared for running a marathon, training should start multiple months in advance. One would first have to start running, then train for an easier challenge, e.g., running for five kilometres without taking a break. Steadily, the distances should be increased until one is ready for the final challenge, the marathon.

To me, there are many similarities between running and pursuing a PhD. It is quite improbable to write a dissertation without first pursuing smaller milestones. Therefore, a structured plan – and following that plan – is key for achieving the final goal. Of course there are obstacles (injuries) such as failed experiments, mistakes, rejected papers, and other hurdles to overcome. After four years of research (and years of training prior to that) I am proud to say that I have finished my thesis. In this feat, I would like to thank everyone that helped me throughout my journey.

The first person that I would like to thank is my main promotor and daily supervisor, Jaco. Prior to my PhD, you already supervised me during my internship and my Master thesis. As both projects resulted in the best possible grade, I could only assume that this was a successful collaboration. Therefore, I was more than happy to pursue a PhD under your supervision. During this time, you gave me a lot of freedom and you have been a great help for the many papers that we published together. Our weekly meetings were efficient and you always made time for me. I am amazed by your ability to quickly context switch and understand a complex problem, and occasionally – within seconds – you even come up with a counterexample to disprove my algorithm. *Who needs a theorem prover if you can talk to Jaco?*

My PhD was part of the 3TU Big Software on the Run project. This quite an-
bitious project was a collaboration between the University of Twente, Delft University of Technology, and Eindhoven University of Technology on the topic of analysing (large) software systems in on-the-fly. I would like to thank everyone involved in the project as they helped me with my own research, but also for teaching me about their topics. One goal of this project was to promote collaborations between the different universities. I am glad that I was able to take this opportunity to dive into the field of process mining.

I would like to thank Wil, my copromotor. He taught me about process mining and guided me towards interesting topics to research. Despite Wil’s busy schedule, he always quickly provided me with great feedback. While we only had a handful of in-person meetings, our collaboration resulted in nice publications (and a best student paper award) that form Chapter 4. Regarding the same topic, I would also like to thank Boudewijn and Bas for their help. Especially for the fruitful discussions that we had, but also for your help with setting up experiments.

Regarding Chapter 2, I could not have achieved the same quality without the help of Alfons. During my Master thesis project and the resulting paper that we wrote afterwards, your help has been invaluable. I can still remember our first discussion on SCCs in 2014, in Vienna. I especially appreciate your attention to detail, as well as your seemingly nondepleatable source of energy for pursuing research. Later in my PhD, I was happy to work together with Ji, another collaboration that originated from the BSR project. Together we looked into the visualization of multi-core algorithms (in particular, my SCC algorithm) and obtained some nice results. Sadly, I have not yet managed to use these results to improve the algorithm, though it is more than clear to me that your technique can aid in detecting performance bottlenecks in complex software.

For Chapter 3, Alexandre was of great help to me. With his help (and the SPOT tool) we were able to model check various types of ω-automata. I have learned a lot from you on this topic and it was great to visit you in Paris.

Another Parisian collaboration resulted in Chapter 5. Laure and Étienne quickly got me up to speed with parametric timed automata and IMITATOR. While my OCaml knowledge is far from perfect, with Étienne’s help it did not take too long before we had a working implementation. On the theoretical side I owe my thanks to Laure as well. You thoroughly analysed the problem and provided great feedback. I really enjoyed working together with you. I also had a great time during my visit in Paris, for which I should thank Mathias as well. Merci!

Most of my time during the PhD was spent in the office, which I shared with excellent colleagues. I greatly enjoyed the (non-)work related discussions that took place. Jeroen, I have learned a lot from you during our time as roommates. Especially whenever I had a programming-related question, you were always there
to help me. I would also like to thank Marcus for all your help, and for the much
needed random discussions to take our minds off work for a moment. Arnd was also
part of the BSR project and was able to help me with any formal methods related
question that I had. I am amazed by your productivity, and let us not forget that
you make an excellent Grünkohl. Whenever I faced low-level programming issues,
I could count on Freark to tell me what was going on. You also joined us in the
BSR project and made sure that we had a great time during all related events.
David, we have been studying together from the first year of University. Back then,
I was always happy to work together with you and my opinion has not changed
since. I am impressed by the number of different projects that you can handle,
with great dedication to each of them. A former roommate that I would like to
thank is Stefano. Your enthusiasm about work, sports, acting, baking, or anything
else is almost contagious. Without you, sadly, there is no Fast Moving Team, but
I greatly enjoyed the events in which I could participate for FMT. I would also like
to thank my former roommate and renewed ex-colleague Tom. With you around,
there is never a dull moment. I am happy for your success in parity game solving,
a topic that I would like to but never been able to study in depth.

The FMT group and ex-FMT members are all great people. I thank all of you for
your help and making sure that there are always nice conversations going on in the
hallway, Waaier, and during social events. I have always felt very welcome here. So,
thank you: Marieke, Hajo, Mehmet, Arend, Mariëlle, Ansgar, Rom, Sebastiaan,
Carlos, Enno, Yeray, Fauzia, Sophie, Wytse (I also had a great time working with
you during our Master thesis project) Güner, Mohsen, Zhiwei, Wei, Qiannan,
Ramon, Djurre, Henk, Ida (organizing and planning would have been a chaos
without you). Thanks also to: Waheed, Afshin, Axel, Machiel (I really enjoyed
my internship at Axini), Steven, Saeed, Dennis, Arnaud, Gijs, Joke, Ruonan,
Mark (thanks again for your help during my Bachelor thesis), Marina, Lesley, and
everyone that I forgot to mention.

I would also explicitly like to thank my graduation committee: Rom, Marc,
Boudewijn, and Laure for taking the time to review my thesis and allowing me to
publicly defend my thesis.

Additionally, I would like to express my gratitude to everyone that enriched my
life and kept me sane in my spare time. I would like to thank every member and
former member of the Aloha triathlon association, many of which I have gotten
to know and consider as close friends. Together we trained, participated in races,
organized events, and had a lot of fun. Thank you as well, Rik, Thijs, Theo, Jefry,
and Matthijs, for the evenings of playing board games.

Daarnaast wil ik mijn familie heel erg bedanken. Mam en Henk, ik ben erg blij dat
jullie altijd voor mij klaar staan en bereid zijn om mij te helpen. Remco, Oscar en
Carmen, jullie hebben mij al vroeg geleerd om te programmeren en ik heb altijd
naar jullie opgekeken. Ook wil ik Chantal’s familie bedanken voor de leuke feestjes en uitstapjes die ik heb mogen meemaken met jullie.

Tot slot wil ik Chantal in het bijzonder bedanken. Ik heb jou tijdens mijn PhD leren kennen en ben jou erg dankbaar voor alles wat we samen hebben meegemaakt. Jij stond voor mij klaar in drukke periodes en we konden samen lekker genieten van de welverdiende vakanties. Ik ben elke dag blij om jou te zien en hoop dat we samen nog lang gelukkig kunnen blijven.

Enschede
June 2019
# Table of Contents

Abstract vii

Acknowledgements ix

1 Introduction 1
   1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   1.2 Graph search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   1.3 Checking models . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   1.5 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
   1.6 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2 Multi-Core Computation of Strongly Connected Components 19
   2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   2.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
      2.2.1 Graphs and on-the-flyness . . . . . . . . . . . . . . . . . . . . 25
      2.2.2 Union-find . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
   2.3 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   2.4 Sequential set-based SCC algorithm . . . . . . . . . . . . . . . . . 31
   2.5 A multi-core SCC algorithm . . . . . . . . . . . . . . . . . . . . . . 35
      2.5.1 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
      2.5.2 Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   2.6 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
      2.6.1 Concurrent union-find structure . . . . . . . . . . . . . . . . . 41
      2.6.2 Worker set for tracking visited states . . . . . . . . . . . . . . 44
      2.6.3 Cyclic linked list for iteration and tracking done states . . . 46
      2.6.4 Runtime complexity . . . . . . . . . . . . . . . . . . . . . . . . 52

xiii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.5 Memory usage</td>
<td>52</td>
</tr>
<tr>
<td>2.7 Experiments</td>
<td>53</td>
</tr>
<tr>
<td>2.7.1 Experiments on model checking graphs</td>
<td>55</td>
</tr>
<tr>
<td>2.7.2 Experiments on synthetic graphs</td>
<td>57</td>
</tr>
<tr>
<td>2.7.3 Experiments on explicit graphs</td>
<td>59</td>
</tr>
<tr>
<td>2.8 Analysing bottlenecks by visualisation</td>
<td>63</td>
</tr>
<tr>
<td>2.8.1 Logging information</td>
<td>64</td>
</tr>
<tr>
<td>2.8.2 Analysis with visualisation</td>
<td>64</td>
</tr>
<tr>
<td>2.8.3 A closer look at the suboptimal behaviour</td>
<td>67</td>
</tr>
<tr>
<td>2.9 Conclusion</td>
<td>69</td>
</tr>
<tr>
<td>2.10 Future work</td>
<td>69</td>
</tr>
<tr>
<td>3 Multi-Core LTL Model Checking for Omega Automata</td>
<td>71</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>72</td>
</tr>
<tr>
<td>3.2 Preliminaries</td>
<td>76</td>
</tr>
<tr>
<td>3.3 Related work</td>
<td>81</td>
</tr>
<tr>
<td>3.3.1 Nested depth-first search</td>
<td>81</td>
</tr>
<tr>
<td>3.3.2 BFS-based approaches</td>
<td>82</td>
</tr>
<tr>
<td>3.3.3 SCC-based algorithms</td>
<td>82</td>
</tr>
<tr>
<td>3.3.4 Rabin automata</td>
<td>83</td>
</tr>
<tr>
<td>3.3.5 Checking different automata</td>
<td>84</td>
</tr>
<tr>
<td>3.4 Fin-less automata</td>
<td>85</td>
</tr>
<tr>
<td>3.5 Algorithm for TGRA emptiness</td>
<td>87</td>
</tr>
<tr>
<td>3.5.1 Checking Rabin pairs</td>
<td>87</td>
</tr>
<tr>
<td>3.5.2 TGRP checking algorithm</td>
<td>87</td>
</tr>
<tr>
<td>3.5.3 Parallel implementation</td>
<td>96</td>
</tr>
<tr>
<td>3.5.4 Checking SBAs, TGBAs and TFLAs</td>
<td>98</td>
</tr>
<tr>
<td>3.6 Experiments for checking SBAs and TGBAs</td>
<td>98</td>
</tr>
<tr>
<td>3.6.1 Experimental setup</td>
<td>98</td>
</tr>
<tr>
<td>3.6.2 Comparison with NDFS</td>
<td>100</td>
</tr>
<tr>
<td>3.6.3 Comparison with CNDFS</td>
<td>101</td>
</tr>
<tr>
<td>3.6.4 Comparison with Renault</td>
<td>103</td>
</tr>
<tr>
<td>3.6.5 Experiments using TGBAs</td>
<td>103</td>
</tr>
<tr>
<td>3.7 Experiments for checking TGRAs and TFLAs</td>
<td>105</td>
</tr>
<tr>
<td>3.7.1 Experimental setup</td>
<td>105</td>
</tr>
<tr>
<td>3.7.2 Main results</td>
<td>107</td>
</tr>
<tr>
<td>3.7.3 Fin-less results</td>
<td>112</td>
</tr>
<tr>
<td>3.7.4 Additional results</td>
<td>114</td>
</tr>
<tr>
<td>3.8 Conclusion</td>
<td>115</td>
</tr>
<tr>
<td>3.9 Future work</td>
<td>115</td>
</tr>
<tr>
<td>4 Alignment-based Conformance Checking for Specific Costs</td>
<td>117</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>Preliminaries</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Petri nets</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Alignments</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Symbolic reachability</td>
</tr>
<tr>
<td>4.3</td>
<td>Synchronous moves and milestones</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Milestones</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Relating the model and event log</td>
</tr>
<tr>
<td>4.4</td>
<td>Preprocessing the reference model</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Milestone transitive closure graph</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Searching for optimal subsequences</td>
</tr>
<tr>
<td>4.5</td>
<td>Symbolic alignment computation</td>
</tr>
<tr>
<td>4.5.1</td>
<td>General idea of the symbolic algorithm</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Bidirectional search</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Detailed algorithm</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Correctness</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Implementation</td>
</tr>
<tr>
<td>4.6</td>
<td>Experiments</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Alignment differences for generated models</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Symbolic algorithm applied on generated models</td>
</tr>
<tr>
<td>4.6.3</td>
<td>MTCG algorithm on models with many traces</td>
</tr>
<tr>
<td>4.7</td>
<td>MTCG structure for diagnostics</td>
</tr>
<tr>
<td>4.8</td>
<td>Related work</td>
</tr>
<tr>
<td>4.9</td>
<td>Conclusion</td>
</tr>
<tr>
<td>4.10</td>
<td>Future work</td>
</tr>
<tr>
<td>5</td>
<td>Optimal-Time and Parameter Synthesis for Parametric Timed Automata</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.2</td>
<td>Preliminaries</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Constraints</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Timed automata</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Parametric timed automata</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Symbolic semantics of PTAs</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Reachability synthesis</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Computing the optimum</td>
</tr>
<tr>
<td>5.3</td>
<td>Related work</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Decidability</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Optimal-time reachability for timed automata</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Reachability and synthesis for PTAs</td>
</tr>
<tr>
<td>5.4</td>
<td>Computability and intractability</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Optimal-time reachability/synthesis for L/U-PTAs</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.4.2 Optimal-parameter reachability and synthesis</td>
<td>187</td>
</tr>
<tr>
<td>5.5 Optimal parameter and time synthesis</td>
<td>189</td>
</tr>
<tr>
<td>5.5.1 Algorithm</td>
<td>189</td>
</tr>
<tr>
<td>5.5.2 Correctness</td>
<td>192</td>
</tr>
<tr>
<td>5.5.3 A subclass for which the solution can be computed</td>
<td>193</td>
</tr>
<tr>
<td>5.5.4 Maximal-parameter and reachability problems</td>
<td>194</td>
</tr>
<tr>
<td>5.5.5 Optimal-time reachability and synthesis</td>
<td>195</td>
</tr>
<tr>
<td>5.6 Improved optimal-parameter and minimal-time synthesis</td>
<td>199</td>
</tr>
<tr>
<td>5.6.1 Algorithm</td>
<td>199</td>
</tr>
<tr>
<td>5.6.2 Correctness</td>
<td>202</td>
</tr>
<tr>
<td>5.6.3 Maximal-parameter, minimal-time and reachability problems</td>
<td>202</td>
</tr>
<tr>
<td>5.7 Experiments</td>
<td>203</td>
</tr>
<tr>
<td>5.7.1 Experimental setup</td>
<td>203</td>
</tr>
<tr>
<td>5.7.2 Results</td>
<td>205</td>
</tr>
<tr>
<td>5.8 Conclusion</td>
<td>207</td>
</tr>
<tr>
<td>5.9 Future work</td>
<td>207</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>209</td>
</tr>
<tr>
<td>6.1 Contributions</td>
<td>209</td>
</tr>
<tr>
<td>6.1.1 Multi-core on-the-fly SCC decomposition</td>
<td>209</td>
</tr>
<tr>
<td>6.1.2 Model checking generalized automata</td>
<td>210</td>
</tr>
<tr>
<td>6.1.3 Efficient alignment computation</td>
<td>212</td>
</tr>
<tr>
<td>6.1.4 Optimal-time and parameter synthesis for PTA s</td>
<td>214</td>
</tr>
<tr>
<td>6.2 Reflection</td>
<td>214</td>
</tr>
<tr>
<td>6.3 Future work</td>
<td>215</td>
</tr>
<tr>
<td>Samenvatting</td>
<td>243</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In 1974, Ernő Rubik invented a challenging 3D puzzle, which we now refer to as the Rubik’s Cube. The well-known Rubik’s Cube is formed from smaller cubes, oriented in a $3 \times 3 \times 3$ layout and each $3 \times 3$ layer of cubes (9 layers in total) can be rotated independently. The faces of the smaller cubes are coloured in 6 different colours, such that when the Rubik’s Cube is in a ‘solved’ state, each of the 9 cube faces on every side all have the same colour.

Given a Rubik’s Cube in a ‘scrambled’ state, i.e. where the cube faces on a side do not all have the same colour, the goal is to solve it by applying rotations to the various layers to reach the solved state. One could regard the state space of a Rubik’s Cube as a graph, where the different states of the cube are states in the graph and edges connect states together as a result of performing transformations. By simply performing rotations to the layers, one can reach more than $4.3 \cdot 10^{19}$ different states. To bring this number in perspective, suppose that one could efficiently transform a Rubik’s Cube to a different (and unique) state in a single millisecond, than it would still take more than a billion years to reach all states.

The beauty of the puzzle is that solving it is far from trivial. Due to the vast scale of states, simply applying random transformations to the cube would likely result in a considerable amount of frustration, and a scrambled cube. Luckily there are algorithms that one may apply to more quickly reach the solved state. By recognizing several colour patterns, one may apply a sequence of transformations to reach a desired (intermediate) result. This way, a solution can be reached by performing only a small number of transformations. In fact, it has been shown that a Rubik’s Cube can be solved with as little as 20 rotations [RKDD14].

We just illustrated the effect that an efficient algorithm may have compared to
a simpler approach (e.g. randomly applying transformations). Solving a Rubik’s Cube could be regarded as solving the reachability problem on a graph, where we determine whether the solved state can be reached. A different view is to regard the puzzle as solving the shortest path problem, in which we attempt to solve the cube with as little rotations as possible. An algorithm is a set of instructions to solve a particular problem. One algorithm may be favourable to another, e.g. due to its performance or applicability. In the development of algorithms, there are various aspects to consider. We highlight a couple of these aspects.

**Complexity.** The performance of an algorithm can be measured in its time and space efficiency. These aspects can be derived from the algorithm and are often represented using the Big O notation, which describes the algorithm’s asymptotic worst-case behaviour as a function of its input size. Given a large enough input, a better asymptotic complexity generally leads to a more time/space efficient algorithm in practice.

Consider for instance the problem of sorting a list of numbers. One solution is called selection sort. It operates by looking for the lowest value in the list, put it at the start of the list, and repeat the same process for the remaining elements. It’s time complexity could be expressed in the number of comparisons, which is $O(n^2)$ for a list of $n$ numbers. Consider now the merge sort algorithm, which divides the list in $n$ smaller lists that each contain a single number, and repeatedly merges these lists together to form new sorted lists until a single list remains. Merge sort operates in $O(n \log n)$ time, which means that if a computer can perform a billion instructions per second, then it may take more than a day to sort an array of a ten million numbers using selection sort and only a fraction of a second when merge sort is used.

**Parallelism.** With performance in mind, we simply cannot ignore that multicore processors are commonplace nowadays. Therefore, ideally, a high-performing algorithm should be designed such that all the available cores are used for improving its runtime. We use the term scalability to denote how fast an algorithm performs using $n$ processing cores, compared to the performance of the same algorithm (or a baseline algorithm) that uses a single core for its computation. For some applications it may be possible to divide the input into distinct parts that may be processed independently, and therefore in parallel. We call such problems embarrassingly parallel as only little effort is required to solve the problem efficiently in parallel.

Unfortunately, many problems are not embarrassingly parallel. To still benefit from the full computation power of multiple processors, algorithms have to be designed such that a proportion of the total workload can be run in parallel.
A parallel algorithm may introduce an increased total workload compared to its sequential version, which is acceptable as long as its performance gain from parallelism outweighs the time to perform the additional work.

**Implementation.** Theoretical results are important for deciding which technique is best suited for solving a problem, but one should not neglect results from empirical studies. There are several reasons that cause theory and practice to be different from each other. By abstracting the algorithm too much, seemingly straightforward operations may actually turn out to be large performance bottlenecks in practice. One may thus fail to notice several crucial aspects, that remain undiscovered until the technique is implemented. Even in case an algorithm closely matches its implementation, the hardware used for performing an experiment has intricacies of its own. Accessing memory is not achieved in constant time due to e.g. caching. It is especially difficult to predict an algorithm’s performance when multiple processors are used, since it is infeasible to consider all possible interleavings of the different threads.

**Context.** The context of the problem to be solved has a large impact on the choice of an appropriate algorithm. On the one hand, the context may limit the range of techniques that are applicable. For instance, traversing a large graph becomes significantly more difficult if only a small amount of memory is available. On the other hand, an algorithm may be able to exploit the context to its benefit. Consider for instance a shortest path problem on a graph with only zero-cost weights. While applying an out-of-the-box shortest path algorithm certainly does the job, it is likely more efficient to simply use a reachability algorithm. The context itself can also be the focus, where an algorithm is modified to be applicable in a specific context. Hence, the application of an algorithm may be (part of) the contribution. By looking at a problem from a different perspective, interesting ideas may result in the birth of new research directions.

1.1 Motivation

With the ever-growing complexity of computer programs it becomes increasingly more difficult to reason about a system’s correctness, or be able to understand some of its internal procedures at all. This trend is combined with the fact that we are more and more dependent on technology, for daily usage, but also in situations where a system error may be life-threatening. Software products keep growing in scale to the point where individual developers only know a small part of its inner workings. In addition, there has been quite some interest in neural networks (e.g.
deep learning) over the last couple of years, with the result that the software itself becomes an incomprehensible generated artefact. Given this increase in scale, the development of better techniques for understanding and formally analysing systems should therefore be of great importance.

**Systems and models.** We make the distinction between systems, models, and graphs. With a *system*, we refer to a set of components and interactions between these components. For instance, a computer program may be described as a system, where different parts of the program form distinct components and sets of instructions form interactions to perform a certain task. Alternatively, a traffic light forms a system, where the various lights interact with each other, and detection systems, to control the flow of traffic. We describe a *model* as an abstraction of a system, in which certain components and interactions are omitted. A model may therefore highlight aspects of interest and neglect other elements, often with the goal to better understand the actual system. Finally, we may use a *graph* to represent a system or model, where states and edges serve as components and interactions, respectively. A multitude of different graph formalisms may be used to more clearly express particular aspects of a model.

**Analysing systems.** In practice, we often cannot formally analyse a system itself, but rely on an abstracted version of reality in the form of a model. By representing this model as a graph, we obtain a clearly defined structure that may be analysed. It may be possible to manually examine the graph-based representation and study whether there are any problems, but this quickly becomes infeasible. Automated approaches search the graph for defects and report them to the user. We may for instance consider the problem whether a particular erroneous state is reachable in the model. Graph traversal algorithms can be used to solve this, and similar problems. We may be interested in particular structural properties of the graph that go beyond reachability. In particular, we study strong connectivity, i.e. the property that two states can reach each other, and shortest paths, i.e. the minimal number of steps required to reach a certain state.

**Increasing scale.** When systems become larger and more complex, accurate models of these systems become more intricate and grow larger as well. Analysis techniques are therefore faced with increasingly complex and larger inputs. As a result, it takes much longer, or even becomes too difficult in practice, to analyse properties of the model with existing techniques. A solution is to improve these techniques by reducing their computation time, which may be achieved by more efficient algorithms.
Our contributions. In this dissertation we study directed graphs and focus on algorithms for two classical graph problems; the decomposition of a graph into Strongly Connected Components (SCCs), and the Single-Source Shortest Path (SSSP) problem, which we describe in Section 1.2. In particular, we concentrate on the development of new graph search algorithms for checking models, i.e. techniques that allow a user to analyse a system and verify whether particular properties are maintained. Our contributions improve the performance of state-of-the-art techniques for model checking and conformance checking, which we explain in Section 1.3. Moreover, we additionally pursued new directions to broaden the horizons of both fields.

In Section 1.2 and Section 1.3, we provide background information on graphs and techniques to check models. We then summarize our contributions in Section 1.4.

1.2 Graph search

A directed graph consists of states (or nodes/vertices) and directed edges (or arcs/transitions) that form connections between the states. Given a graph, we can consider particular computation problems, such as deciding whether there is a directed path in the graph from \( a \) to \( b \) (reachability), or request the minimal number of steps required to reach \( b \) from \( a \) (shortest path).

To solve such graph problems we may employ graph traversal algorithms. In Figure 1.1 we illustrate two basic techniques for traversing a graph, called Depth-First Search (DFS) and Breadth-First Search (BFS). The difference between the two algorithms is in the order in which new states are processed. With DFS we maintain a LIFO (Last-In First-Out) stack of states. When we take one state from the top of the stack, we compute its successors and push all undiscovered states onto the stack, and pick the next state from the top of the stack. This process is
continued until the stack is empty and correspondingly all reachable states have been discovered. With BFS we instead maintain a FIFO (First-In First-Out) queue of states. The process is the same as for DFS, but states are now inserted at the back of the queue and thus the ‘oldest’ state in the queue is picked first.

There are of course various alternative approaches to extract properties from the graph. We could for instance order the states in ascending order of their in-degree (the number of edges pointing to the current state) and process them in that order. However, this would require full knowledge of the entire graph (all states \textit{and} edges) in advance, which may not always be feasible in practice.

\textbf{On-the-fly graph exploration.} There are several reasons why storing an entire graph in memory is impractical. Given a large enough graph, it may simply require too much space to explicitly store all states and edges, making it impossible to store directly. Alternatively, we may not know the entire graph in advance, which is possible in scenarios where the graph is computed \textit{on-the-fly}. On-the-fly graph computation implies that there exists a function that, given a source state, calculates its successors. Such graphs are accompanied by at least one known initial state and we call such graphs \textit{implicit} (as opposed to \textit{explicit} graphs that are known in advance).

Consider for instance a graph of the different configurations of a chess board, where edges represent the possible moves. Storing the entire graph would likely take up too much space in memory (there are more than $10^{40}$ different chess board configurations), but it is certainly feasible to construct a function that calculates all successor states from a given configuration.

When traversing an implicit graph, it may still be the case that the graph itself might not fit in memory (which may or may not be known in advance). Therefore, instead of storing the graph in its entirety when traversing it, a common approach is to only explicitly store the visited states in memory and discard the edges. It is often the case that sufficient information about the encountered edges and paths can be tracked during the graph traversal, making it unnecessary to track edges.

\textbf{Strongly connected components.} A \textit{Strongly Connected Component} (SCC) is a set of states in the graph for which any two states in the set can reach each other. Additionally, an SCC is \textit{maximal} in the sense that there does not exist a larger SCC that contains states from another SCC. As a result, there is exactly one way to decompose a graph into SCCs. We give an example of an SCC decomposition in Figure 1.2. Here, the component $\{b, c, e\}$ forms an SCC, as it is possible to form a path between any two states in the set.
SCCs reveal interesting structural properties of a graph. For instance, every cycle in the graph (path for which the first and last states are the same) is contained in an SCC. Correspondingly, if we contract all states in each SCC to a single ‘supernode’, we form a quotient graph that is a directed acyclic graph (a graph that contains no cycles). If a graph does contain cycles, we may construct paths of infinite-length, which are useful e.g. in model checking (see Section 1.3).

There are many different algorithms that decompose a graph into SCCs. Arguably the most famous algorithm is Tarjan’s algorithm [Tar72]. It performs a DFS traversal through the graph, while maintaining some additional information per state. Upon encountering an edge pointing to an already visited state that is part of the current DFS path, it detects a cycle. The algorithm tracks (via the additional state information) which states belong to the same SCC. Dijkstra [Dij82] proposed a variant that tracks a stack of SCC roots, which is updated on the detection of cycles. A different approach enumerates all states while performing a DFS, then reverses the direction of all edges and performs another DFS to detect SCCs [Sha81]. Yet another technique selects a pivot state and intersects all forward reachable states with the set of all backward reachable states to form an SCC [FHP00]. Each algorithm has its own advantages and drawbacks, making it context-dependent to decide on an appropriate technique.

**Shortest paths.** The *Single-Source Shortest Path* (SSSP) problem, or simply shortest path problem, is to find a path between two states such that its total cost is minimized. Here, the cost of a path is determined by the number of edges that must be traversed, or e.g. by the sum of associated edge costs (or weight) for each edge on the path. Edge costs could be regarded as an extension to directed graphs, where a number (positive or negative) is attached to each edge in the graph. A simple example (for which all edge costs are 1) is given in Figure 1.2, where the
cost of reaching each state from the initial state is depicted.

Shortest paths are useful for solving various optimization problems. Consider for instance the fastest way of transportation from one location to another. Alternatively, we may apply shortest paths to optimally schedule a process execution such that its computation time is minimized. Another application is conformance checking (see Section 1.3) to combine observed behaviour with a reference model and minimize their discrepancies.

For graphs that have no negative edge costs, Dijkstra’s algorithm [Dij59] provides a solution to the shortest path problem. It works by performing a BFS from the initial state, where a priority queue is used instead of a standard queue. The states in the priority queue are ordered on the total cost of reaching them, such that the state with the lowest associated cost is selected first. This technique still forms the basis for many shortest path solutions and it is improved by applying heuristics to ‘direct’ the search in the A* algorithm [HNR68]. In case a graph contains negative edge costs (which may lead to negative-cost cycles), the less efficient but more versatile Bellman-Ford algorithm [Bel58] can be applied instead.

1.3 Checking models

When designing a piece of software, one (perhaps implicitly) maintains a set of requirements that the software should follow. Such requirements describe what the system should or should not do. For instance, the program should not reach an erroneous state or raise an error, or the program should provide a correct answer to a particular computation problem. The basis for verification is to analyse if such properties are actually realised in the implementation.

To analyse software, one could consider every possible state of the system, i.e. program states or ‘snapshots’ that contain the values for every variable stored in memory at the current point in the program’s execution (including the program’s input). Such program states could be seen as states in a directed graph, and edges are formed by the atomic steps in the program’s execution that cause one state to be transformed to another (by e.g. performing a variable assignment). For this graph, one may check whether certain properties from the specification hold. For instance, one may traverse the graph to check or validate whether an erroneous state is reachable. If this is the case, the program contains a bug.

In practice, there may be near-infinite possible program states in a piece of software, considering that we may provide it with arbitrary input values. Therefore, it is generally infeasible to analyse software by considering every possible program state. An alternative is to construct a model of the system, i.e. an abstracted
representation of the complete system. We may omit certain variables that are not of interest for the specific properties to check, and thereby reduce the size of the execution graph. However, we would like to emphasise that constructing an appropriate model, that is both concise and a good reflection of reality, is by no means a trivial task.

We are not limited to checking properties on software programs. Consider for instance a business process. There may be an incoming order, several procedures to process the order (e.g. by building a product), and finalizing the order (e.g. shipping the product, handling invoices). This process may be combined with individual tasks for the various employees. By describing this combined process, we also form an abstracted representation of reality, i.e. a model, that may be represented by a graph as well. However, one should not blindly trust the model, as it may very well be an incorrect abstraction of reality. The business process should also comply with certain requirements, e.g. for each shipped product, there should be an invoice.

We can extend the model by combining it with additional properties. For instance, it may be useful to consider the time taken at certain states in the model. One may be concerned with optimizing the flow of a process by analysing the time taken in every step of the process, and thus focus on reducing the total time taken.

An alternative method to analyse a model is to combine it with logged information, e.g. execution traces of a program or a log trace of a business process. We may then analyse whether the logged information and the model express the same system. If this is not the case, then either the logged information or the model (or both) may incorrectly represent reality. Moreover, models could be learned from the observed behaviour and one could also use the logged information to detect and highlight outliers.

**Model checking.** In model checking [CHVB18], one is given a model, which we assume is represented by directed graph, and a property to check. Such a property may for instance specify that an erroneous state should never be reached. We call this a safety property and via graph traversal we may be able to detect whether this property holds or not.

Alternatively, we may be interested in liveness properties to analyse infinitely running systems. Consider for instance a traffic light system. One could ask whether each light always turns green at some point in the future. Given a system where this property does not hold, then there is at least one light that, from some point onwards, never turns green.

Checking whether liveness properties hold is more involved than checking reachability. We need to check if the property holds for all possible infinite runs of the
model. Checking this directly is impractical as every possible infinite run should be considered. A better way is to check for the non-existence of a faulty run, i.e. an infinite run that invalidates the property. Thus, if we can traverse the model such that a cycle is reached for which the property does not hold, then we encountered a counterexample. Otherwise, if we cannot encounter such a cycle in the entire graph, then the property holds.

When checking liveness properties, the model and the negated property are combined together. A cycle in the combined model is called ‘accepting’ if the negated property holds on the cycle (indicated by marking particular states as being accepting). Algorithms for checking liveness, therefore, search for an ‘accepting cycle’. A well-known algorithm to detect accepting cycles is Nested Depth-First Search (NDFS) [CVWY92], which performs an ‘outer’ DFS to explore the state space and launches ‘inner’ DFS instances to search for a cycle from every accepting state that the outer DFS encounters. An alternative method is to decompose the combined model into SCCs and detect if a reachable non-trivial SCC contains an accepting state, by e.g. using Tarjan’s algorithm [Cou99]. The NDFS algorithm uses less memory, but an SCC-based algorithm may be applied to check more generalized acceptance conditions efficiently.

**Parametric timed automata.** In some scenarios, it is useful to specify additional properties in the model. One such property is the notion of time. A Timed Automaton (TA) [AD91] extends a standard model by tracking real-time clocks. The system may then have constraints on the clocks that specify when an edge may be taken. For instance, when making tea, one first has to wait until the water has boiled before it should be put in the cup.

However, one may not know certain delays in advance, e.g. we might not know the time until the water has boiled. An extension to a TA, called a Parametric Timed Automaton (PTA) [AHV93] allows the user to model uncertainties as parametric constant values. We may, for instance, model the time to boil water as a parameter. Then, by performing reachability we can synthesize the parameter values. This may e.g. be combined with safety properties to determine how fast the water must be boiled such that the cup of tea is ready in 5 minutes.

Checking properties on a PTA becomes significantly more complex compared to standard models. Because a clock may be assigned any positive value, the model contains an infinite number of states. In practice, constraints are combined with locations (i.e. the states in the models when clocks are omitted) to form symbolic states. A new graph is formed from taking these symbolic states and computing their successors. Properties may then be checked on these symbolic states, such that e.g. time and parameter values can be extracted. However, reachability and
parameter synthesis are undecidable in general for PTAs \cite{AHV93}, meaning that these problems can only be solved on a subset of PTAs.

**Conformance checking.** In conformance checking \cite{CvDSW18}, we are concerned with differences between modelled and observed behaviour. Here, we check if the model describes the same behaviour as observed in practice. A log trace is a sequence of events (that may be combined with additional properties) that reflects an observed run of the real system or process. Since the model or the log trace may reveal undesired behaviour, we study the discrepancies between them.

Alignments \cite{Adr14} combine (observed) log traces and (derived) runs from the model by considering each pair of log and model events separately. Ideally, the entire log trace exactly forms a run through the model, implying that there are no differences between modelled and observed behaviour. Otherwise, there may be observed events that cannot be mapped onto the model, or events in the ‘best-fitting’ run through the model that are not part of the log trace. An alignment is formed by constructing a run through the synchronized product of the model and log trace, such that the number of discrepancies is minimized.

An alignment is constructed by performing a shortest path search in the model and log trace, where synchronized events (alignment pairs for which the observed events equal modelled ones) may e.g. have a cost of 0 and model or log moves (events that are only observed in the model or log trace) have a cost of 1. By minimizing the total cost, we effectively minimize the total number of differences between the observed and modelled behaviour. The alignment itself can then be used to further study the cause for the differences.

### 1.4 Contributions

In this dissertation, we focus on improving the current state-of-the-art techniques for checking models. In particular, we are concerned with improvements from a performance point of view. Additionally, we present new directions that augment the corresponding fields.

We can group our contributions in two categories: techniques that are based on SCC decomposition and techniques that are based on a shortest path computation. Both categories can be further divided into context-driven performance improvements and applications to explore new directions for verification.

We provide an overview of our contributions in Figure 1.3 and summarize these as follows.
Multi-core on-the-fly SCC decomposition. As discussed in Section 1.2, there are several existing algorithms for the decomposition of SCCs. We are interested in on-the-fly SCC decomposition, i.e. without prior knowledge of the entire graph, with the intention to apply it for model checking. Given that we nowadays have multiple processors, ideally, we would like to take full advantage of them to improve the performance of SCC decomposition.

For SCC decomposition in our scenario, there already are solutions that scale (improve in performance for multiple cores compared to a single core version) [Low16, RDKP17]. However, both algorithms only scale for the subset of graphs that only contain small SCCs (in the number of states). In Chapter 2, which is based on the work from [BLvdP16], we propose a technique that is also scalable for graphs with large SCCs.

For our algorithm, we extend the concurrent union-find data structure with a cyclic list to iterate over states. We then use this structure for storing partially detected SCCs and share this globally for all threads. This way, different threads can collaboratively decompose a graph into SCCs, for both small and large SCCs.

In our empirical study, we demonstrate scalability on a 64-core machine and show that our algorithm significantly outperforms existing work. In general, the performance of our algorithm is equally well or better than related work, and given a graph with large SCCs, our approach is typically 10 to 30 times faster. In addition to the work from [BLvdP16], we employ a novel visual analytics tool to investigate the inner workings of our algorithm.
1.4. Contributions

Model checking generalized automata. In Chapter 3 we apply our parallel SCC decomposition algorithm for model checking and show in [BvdP16] that it also performs better than the state-of-the-art, i.e. multi-core version of nested depth-first search [ELPvdP12]. We show that our technique is in particular advantageous for larger models. Moreover, we outperformed alternative approaches in the 2016 Model Checking Contest [KGH+16].

The standard approach to model checking is to specify the to-be-checked property in a Büchi Automaton, or occasionally a Transition-based Generalized Büchi Automaton (TGBA). In [BDLvdP17] we consider whether yet another type of automata, called a Transition-based Generalized Rabin Automaton (TGRA), that may be beneficial to model checking. This more complex automaton can decrease the size of the negated property automaton as well as the size of the combined graph. The downside is that the model checking procedure becomes significantly more complex. We present a new algorithm, also extending on our SCC decomposition technique, that model checks based on TGRA. We show how model checking TGRA is significantly different from traditional approaches. We were, however, not able to improve on our earlier work from [BvdP16].

We also introduce a new type of automaton in [BDLvdP19], that we call Fin-less automata, which can be derived from a TGRA. We show that we can outperform our earlier technique, and therefore also related work, by model checking based on these Fin-less automata.

Efficient alignment computation. When analysing systems for which logged information is available, we may consider conformance checking. For large models and/or many log traces, the computation of alignments becomes a time-consuming task. The current best-practice for computing alignments is to perform an A* shortest path search [Adr14]. In terms of algorithmic complexity, there is little performance to be gained. Instead of improving the performance for alignment computation in the general case, in Chapter 4 we focus on specific cost functions. With a cost function, one assigns cost values for the different move types. A ‘standard’ approach, that is commonly used, is to assign a cost of 0 for synchronous moves, and a cost of 1 to model and log moves.

In [BvdPvdA18] we introduce an alignment computation algorithm specifically for a subset of cost functions, which includes the standard one. The algorithm is based on symbolic reachability, meaning that instead of single states, we store sets of states and compute the successors for every state in a set at once. We show that it outperforms the state-of-the-art, i.e. [Adr14] and [vD18] (the latter comparison is new compared to [BvdPvdA18]).

We also consider an alternative cost function in [BvZvdA+18]. Here, instead of
minimizing the number of discrepancies, we set up the cost function to instead maximize the number of synchronous moves. We show what effect this has on the computed alignments and exploit this cost function in a new algorithm for computing alignments. We construct a Transitive Closure Graph (TCG) from the model, and use this structure to more efficiently compute alignments when there are many log traces to align.

We have extended the work from \([BvZvdA+18]\) in \([BvZvdA^+]\) (which is currently under submission) to also include so-called milestone events, which are events that may never be processed as a log- or model moves. Milestones may be used to guide the construction of alignments and refine the model. We extended the TCG structure to include milestones and in addition we show that this structure may also be useful as a diagnostic tool, since it is less sensitive to the non-determinism of optimal alignments when compared to alternative approaches.

**Optimal time and parameter synthesis for PTAs.** In Chapter 5 we study parametric timed automata and extend this field by considering optimal time and parameter synthesis. In \([ABPvdP19]\) we consider the problem of reaching a particular location of a PTA in minimal total time. This means that the result is a set of parameter constraints that allow the system to reach the location in minimal time, and that parameter valuations for which this is not possible are omitted.

We show that this is computable via a shortest path search on the symbolic states (which consist of locations and associated constraints). When compared to standard reachability synthesis from related work \([JLR15]\), we found that our restriction to minimize time may actually reduce the computation time of synthesizing a set of parameter valuations. Additionally, we also focus on reachability of a single minimal parameter valuation, and on the problem of minimizing a particular parameter value.

In Chapter 5 we extend the work from \([ABPvdP19]\) by also considering maximal time and maximal parameter synthesis. We show that a shortest path algorithm cannot be used for the maximal time variant, because the value for the maximal time is not necessarily monotonic.

### 1.5 Publications

This dissertation is built upon the following publications (in order of the chapter structure). The author of this thesis is responsible for the main content of each paper.
1.5. Publications


This paper presents our multi-core SCC decomposition algorithm, which is an extension and significant improvement on the author’s Master thesis [Blo15], by achieving scalability for more than 8 threads and evaluating the algorithm with a considerably larger quantity of experiments in combination with a more in-depth analysis. We received an artifact evaluation badge for the reproducibility of our results.


Here, we applied the SCC algorithm from [BLvdP16] to perform model checking. We show that our algorithm outperforms the state-of-the-art techniques in model checking.


For this paper, we extended the algorithm from [BvdP16] to check TGRAs. While we achieved this feat, our evaluation (using several TGRA generators) showed no improvement over checking TGRAs.


This paper is an extended journal version of [BDLvdP17], in which we extend the set of experiments and also introduce Fin-less automata. We show that Fin-less automata can be derived from TGRAs and can be checked efficiently.


In this paper we designed an alignment computation algorithm (with a restriction on the cost function) based on symbolic reachability. Using a set of generated models we showed that this technique outperforms the A* algorithm.
Here, we investigate alignments for a cost function that maximize synchronous moves, instead of minimizing log- and model moves. We study the differences and also present a new algorithm, that computes the transitive closure of a model to speed up alignment computations for many log traces.

This paper is still under submission and is an invited journal paper for a special issue of the Information Systems journal, containing extended versions of selected papers from BPM 2018. We extended our previous work [BvZvdA+18] with the notion of milestone events in alignments and use the transitive closure graph for diagnostic information.

This paper investigates the problems of minimal-time reachability and synthesis for PTAs. We show that this is possible\textsuperscript{1} via a shortest path computation and show that this can be computed faster than standard reachability and synthesis. We received an artifact evaluation badge for the reproducibility of our results.

\textbf{Other contributions by the author.} The author of this dissertation has also contributed to the following publications.


\textsuperscript{1} These problems are also undecidable in general for PTAs, hence we present a \textit{semi-algorithm}, which only outputs a correct result if the algorithm terminates.
1.6. Overview

This is a short paper for the FM doctoral symposium, in which we present research directions for model checking. With our SCC decomposition algorithm as a basis, we show how it may be used to model check more general automata compared to Büchi automata.


This book chapter presents the state-of-the-art in parallel model checking. The author of this thesis has partially contributed, in the discussion of parallel SCC-based techniques.


This paper, which is currently under submission, presents a visual analytics tool for analysing parallel data sequences. Our contribution is in the form of a case study where we apply the tool to examine our multi-core SCC decomposition algorithm on a particularly interesting model. We included the case study in Section 2.8.

1.6 Overview

The remainder of this dissertation is organised in the following way.

Chapter 2 presents our multi-core SCC decomposition algorithm along with its internal iterable union-find data structure.

Chapter 3 applies the SCC decomposition algorithm from Chapter 2 for model checking, and investigates how more general automata (compared to Büchi automata) can be model checked.

Chapter 4 includes our symbolic reachability algorithm for computing alignments. We also study alignments that maximize synchronous moves with milestone events, and present our TCG structure and the accompanying alignment computation algorithm.

Chapter 5 studies the subject of optimal-time reachability and synthesis for PTAs and we show how this can be achieved in practice.

Chapter 6 concludes the thesis with a discussion of our contributed work and directions for future research.
We provide an overview of the main chapters in Figure 1.3. Each chapter can be read and understood independently. However, we do suggest reading Chapter 2 prior to Chapter 3, as the algorithms from Chapter 3 extend the SCC decomposition algorithm that is presented in Chapter 2.
Chapter 2

Multi-Core Computation of Strongly Connected Components

In this chapter we focus on the problem of decomposing graphs into strongly connected components (SCCs) by using parallelism. SCC decomposition is a fundamental technique in graph theory, and it has applications in e.g. compiler analysis, data mining, and model checking.

The main advantages of Tarjan’s strongly connected component algorithm are its linear time complexity and ability to return SCCs on-the-fly, while traversing or even generating the graph. Until now, most parallel SCC algorithms sacrifice both: they run in quadratic worst-case time and/or require the full graph in advance.

In this chapter we present a novel parallel, on-the-fly SCC algorithm. It maintains a quasi-linear linear-time complexity (which is practically linear) by letting workers explore the graph randomly while carefully communicating partially completed SCCs, without sacrificing correctness. For efficiently communicating partial SCCs, we develop a concurrent, iterable disjoint set structure, which combines the union-find data structure with a cyclic list.

We demonstrate scalability with respect to Tarjan’s algorithm on a 64-core machine using 75 real-world graphs (from model checking and explicit data graphs), synthetic graphs (combinations of trees, cycles and linear graphs), and random graphs. Previous work did not show speedups for graphs containing a large SCC. We observe that our parallel algorithm is typically 10-30× faster compared to Tarjan’s algorithm for graphs containing a large SCC. Comparable performance (with respect to the current state-of-the-art, i.e. [RDKP15]) is obtained for graphs containing many small SCCs.
The main contents of this chapter are based on the author’s master thesis [Blo15] and a published conference paper [BLvdP16] (joint work with Alfons Laarman and Jaco van de Pol), which significantly improves and extends the work from [Blo15] by maintaining scalability for more than 8 threads and by performing more experiments on on-the-fly and explicitly given graphs. We also studied a particular scenario in more detail by using a novel visual analytics tool (Section 2.8), which formed a case study in [QBW+19] (currently under submission, main work by Ji Qi).

2.1 Introduction

Sorting states in depth-first search (DFS) postorder has turned out to be important for efficiently solving various graph problems. Tarjan first showed how to use DFS to find biconnected components and SCCs in linear time [Tar72]. Later it was used for planarity [HT74], spanning trees [Tar76], topological sort [CLRS09], fair cycle detection [CVWY92] (a problem arising in model checking [VW86]), state covers [Sav82], etc.

Due to irreversible trends in hardware, parallelizing these algorithms has become an urgent issue. In the current chapter we focus on solving this issue for SCC decomposition, improving SCC decomposition for large SCCs. But before we discuss this contribution, we address the problem of parallelizing DFS-based algorithms more generally.

**Traditional parallelization.** Direct parallelization of DFS-based algorithms is a challenge. Lexicographical, or ordered DFS is \( \mathcal{P} \)-complete [rei85], thus likely not parallelizable as under commonly held assumptions, \( \mathcal{P} \)-complete and \( \mathcal{NC} \) problems are disjoint, and the latter class contains all efficiently parallelizable problems (i.e. given a problem of size \( n \), are there constants \( c \) and \( k \) such that the problem can be solved in time \( \mathcal{O}(\log n)^c \) for \( \mathcal{O}(n^k) \) processors?).

Therefore, many researchers, when parallelizing these graph problems, have diverted to phrasing them as fix-point problems, which can be solved with highly parallelizable breadth-first search (BFS) algorithms.

**BFS for SCC decomposition.** The strategy of parallelizing via BFS is also effective for decomposing SCCs. For instance, we can rephrase the problem of finding all SCCs in \( G = (V,E) \), to the problem of finding the SCC \( C \subseteq V \) to which a state \( v \in V \) belongs, remove its SCC \( G' = G \setminus C \), and reiterate the process on \( G' \). Here, \( C \) is equal to the intersection of states reachable from \( v \) and states reachable from \( v \) after reversing the edges \( E \) (backward reachability).
Both reachability queries can be computed via BFS and this yields the quadratic \( \mathcal{O}(n \cdot (n + m)) \) \textit{Forward-Backward} (FB) algorithm (here, \( n = |V| \) and \( m = |E| \)). Researchers repeatedly and successfully improved FB [FHP00, Orz04, HR013, CFHm05, Sch08], which reduced the worst-case complexity to \( \mathcal{O}(m \cdot \log n) \). The FB algorithm may also be efficiently run in parallel, as we will show in Algorithm 1.

A negative side effect of the fix-point approach for SCC decomposition is that the backward search requires that all edges in the graph are stored, or at least that the successors and predecessors of a given state can be computed efficiently. Storing the edges can be done using e.g. adjacency lists or incidence matrices [BM76] in various forms [Kav14, Sec. 1.4.3], however this takes at least \( \mathcal{O}(m + n) \) memory.

**On-the-fly SCC decomposition.** Contrary to the FB solution, Tarjan’s algorithm can run \textit{on-the-fly} using an implicit graph definition \( I_G = (s_0, \text{Succ}()) \), where \( s_0 \in V \) is the initial state and \( \text{Succ}(v) = \{ v' \in V \mid (v, v') \in E \} \), and requires only \( \mathcal{O}(n) \) memory and \( \mathcal{O}(n) \) time to store visited states and associated data. The on-the-fly property is important when handling large graphs that occur in e.g. verification [CGP01], because it may allow the algorithm to terminate early, after processing only a fraction \( (\ll n) \) of the graph. It also benefits algorithms that rely on SCC decomposition but do not require an explicit graph representation, e.g. computing the transitive closure [Nuu95].

**Parallel Randomized DFS (PRDFS).** A novel approach has shown that the DFS-based algorithms can be parallelized more directly, without sacrificing complexity and the on-the-fly property [LLvdP+11, EPY11, ELPvdP12, Low16, RDKP15, LSD09, Lla14, LW14, LF13, Blo15]. The idea is simple: (1) start from naively running the sequential algorithm on \( P \) independent threads, and (2) globally prune parts of the graph where a local search has completed. This way, information is shared between the different workers.

For scalability, the PRDFS approach relies on introducing randomness to direct threads to different parts of a large graph. Hence the approach cannot be used for algorithms requiring lexicographical, or ordered DFS. But interestingly, none of the algorithms mentioned in the first paragraph require a fixed order on outgoing edges of the states (except for topological sort, but for some of its applications the order is also irrelevant [BBR10]), showing that the oft-cited (cf. [BCvdP11, FHP00, BBBJ11, BBR10, vP03a, BvKP01]) theoretical result from Reif [Rei85] does not apply directly. In fact, it might even be the case that the more general problem of non-lexicographical DFS is in NC.

For correctness, the pruning process in PRDFS should carefully limit the influence on the search order of other threads. Trivially, in the case of SCCs, a thread
Table 2.1: Complexities of fix-point (e.g. [CFHm05]) and PRDFS solutions (e.g. [RDKP15]) for the problem of SCC decomposition. Here, $n$ and $m$ represent the number of states and edges in the graph and $P$ the number of processors.

<table>
<thead>
<tr>
<th></th>
<th>Best-case ($O$)</th>
<th>Worst-case ($O$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Work</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$n + m$</td>
</tr>
<tr>
<td>Traditional</td>
<td>$\frac{n + m}{P}$</td>
<td>$n + m$</td>
</tr>
<tr>
<td>PRDFS</td>
<td>$\frac{n + m}{P}$</td>
<td>$n + m$</td>
</tr>
</tbody>
</table>

$^\dagger$ Under the assumption that successors and predecessors of a given state can be computed in constant time. Otherwise, if all edges are stored explicitly, the memory becomes $n + m$.

running Tarjan’s algorithm can remove an SCC from the graph as soon as it is completely detected [LSD09], as long as done atomically [RDKP15]. This would result in limited scalability for graphs consisting of a single SCC [RDKP15].

Time and work trade-off play an import role in parallelizing many of the above algorithms [Spe91]. In the worst case, e.g. with a linear graph as input, the PRDFS strategy cannot deliver scalability – though neither can a fix-point based approach for such an input. The amount of work performed by the algorithm in such cases is $O(P \cdot (n + m))$, i.e.: a factor $P$ (the number of processors) compared to sequential algorithm ($O(n + m)$). However, the runtime never degrades beyond that of the sequential algorithm $O(n + m)$, under the assumption that the synchronization overhead is limited to a constant factor. This is because the same strategy can be used that makes the sequential algorithm efficient in the first place. Table 2.1 compares the two parallelization approaches. We hypothesize that a scalable PRDFS-based SCC algorithm can solve parallel on-the-fly SCC decomposition.

**Contribution: PRDFS for SCCs.** We provide a novel PRDFS algorithm for detecting SCCs capable of pruning partially completed SCCs. Prior works either lose the on-the-fly property, or show no scalability for graphs containing a large SCC. Our proof of correctness shows that the algorithm indeed prunes in such a way that the local DFS property is preserved sufficiently. Experiments show good scalability on real-world graphs, obtained from model checking benchmarks, but also on random and synthetic graphs. We furthermore show practical instances (on explicitly given graphs) for which existing work seems to suffer from the quadratic worst-case complexity. Finally, we examine an instance in detail for which our algorithm only exhibits limited scalability. We employed a visual analytics tool to
Figure 2.1: Schematic of the concurrent, iterable queue, which operation resembles a closing camera shutter (on the right). White nodes (invariably on a cycle) are still queued, whereas grey nodes have been dequeued (contracting the cycle), but can nonetheless be used to query queued nodes, as they invariably (perhaps indirectly) point to white nodes.

analyse logged runs of the algorithm to better understand the bottlenecks of our approach.

**Efficient communication.** Our algorithm works by communicating partial SCCs via a shared data structure based on a union-find forest for recording disjoint subsets [TvL84]. In a sense therefore it is based on SCC algorithms before Tarjan’s [Pur70, Mun71]. The overhead however is limited to a factor defined by the inverse Ackermann function $\alpha$ (rarely grows beyond a constant 6), yielding a quasi-linear solution.

We avoid synchronization overhead by designing a new *iterable* union-find data structure that allows for concurrent updates. The subset iterator functions as a queue and allows for elements to be removed from the subset, while at the same time the set can grow (disjoint subsets are merged). This is realized by a separate cyclic linked list, containing the nodes of the union-find tree. Removal from the list is done by collapsing the list like a shutter, as shown in Figure 2.1. All nodes therefore invariably point to the sublist, while *path compression* makes sure that querying queued states always takes an amortized constant time. Multiple workers can concurrently explore and remove nodes from the sublist, or merge disjoint sublists.

2.2 Preliminaries

Given a directed graph $G = (V, E)$, we denote an edge $(v, v') \in E$ as $v \rightarrow v'$. For the graph’s size, we denote $n = |V|$ and $m = |E|$. A *path* is a sequence of states
\begin{equation}
\langle v_0, \ldots, v_k \rangle, \text{ s.t. } \forall 0 \leq i < k : v_i \in V \text{ and } \forall 0 \leq i < k : v_i \rightarrow v_{i+1}. \text{ With } v \rightarrow^* w \text{ we denote that there is a path } \langle v, \ldots, w \rangle \in V^*. \text{ If a state } v \text{ reaches itself via a non-empty path (with at least 2 states), the path is called a cycle, e.g. } \langle v, v \rangle \text{ is a cycle if } v \rightarrow v. \text{ States } v \text{ and } w \text{ are strongly connected iff } v \rightarrow^* w \land w \rightarrow^* v, \text{ written as } v \leftrightarrow w, \text{ and we define partial-, fitting-, and strongly connected components as follows.}
\end{equation}

**Definition 2.1: Partial Strongly Connected Component (PSCC)**

A partial strongly connected component (PSCC or partial SCC) is a non-empty state set $C \subseteq V$ such that $\forall v, w \in C : v \leftrightarrow w$, i.e. every two states in $C$ are strongly connected.\(^{a}\)

\(^a\) For a PSCC $C$ we do not require that a cycle must be formed using only states from $C$. For example, given cycle $\langle v, w, x, v \rangle \in V^*$ and $C = \{v, w\}$, then $C$ is a PSCC.

**Definition 2.2: Fitting Strongly Connected Component (FSCC)**

A fitting strongly connected component (FSCC or fitting SCC) is a PSCC $C$ for which all cycles can be formed by only using states from $C$, i.e. $\forall v, w \in C : (\exists \langle x_0, \ldots, x_i \rangle \in C^* : \langle v, x_0, \ldots, x_i, w \rangle \text{ is a path}).$

**Definition 2.3: Strongly Connected Component (SCC)**

A strongly connected component (SCC) is a maximal FSCC, i.e. $C \subseteq V$ is an SCC iff $C$ is an FSCC and there is no FSCC $C' \subseteq V$ such that $C \subset C'$. In case that $|C| = 1$ such that $v \in C$ and $v \not\rightarrow v$, we call $C$ a trivial SCC.

Thus, given an SCC or FSCC $C$, any non-empty subset $C' \subseteq C$ is a PSCC. Also, any SCC is an FSCC, but an FSCC is not necessarily an SCC. In Figure 2.2 we give an example of a PSCC, FSCC, and SCC.

We note that an SCC can be also be defined (more precisely) as a pair that contains both a set of states and a set of edges, where the latter set consists of all edges.
that are part of cycles in the SCC. For this chapter we are not concerned with the corresponding set of edges, hence the simplified definition. In Chapter 3, we will however be interested in sets of edges for (partial) SCCs, and redefine an SCC in Definition 3.4.

2.2.1 Graphs and on-the-flyness

Explicit and implicit graphs. From an implementation point of view, we will distinguish between three different definitions of a graph representation, namely:

- **Explicit graph:** $G_E = (V, E)$, where the nodes $V$ and edges $E$ are explicitly stored and can be accessed directly. Notably, we assume that backwards edges, $E^{-1}$, are also stored or can be computed efficiently.

- **Forward-only implicit graph:** $G_F = (v_0, Succ())$, where $v_0$ is an initial state and $Succ(v)$ is a successor function: $Succ(v) = \{ w \mid v \rightarrow w \}$. It is therefore not possible to efficiently compute backward edges in a forward-only implicit graph. Note that we may only access states that can be reached from $v_0$.

- **Implicit graph:** $G_I = (v_0, Succ(), Prev())$, where $v_0$ is an initial state, $Succ(v)$ is a successor function, and $Prev(v)$ is a predecessor function: $Prev(v) = \{ w \mid w \rightarrow v \}$. Here, we may not access states from disjoint subgraphs that do not contain $v_0$.

Notably, the most restricted representation is the forward-only implicit graph and the explicit graph is the least restricted in terms of usability as $Succ(v)$ and $Prev(v)$ can be computed trivially for arbitrary states. Also note that while it is possible to directly access or pick a state $v \in V$ in an explicit graph, this is not possible in a (forward-only) implicit graph as one first needs to traverse the graph from $v_0$ until state $v$ is encountered, and then may compute e.g. $Succ(v)$. For all types of graphs, the structural definition $G = (V, E)$ remains being used for e.g. correctness proofs and w.l.o.g. we assume that all states in $V$ can be reached from $v_0$, i.e. $\forall v \in V : v_0 \rightarrow^* v$.

**Example 2.1: Motivation for the different graph types**

For each of the presented graph types there are several practical use-cases, which are not necessarily disjoint.

An explicit graph is the ‘standard’ way of representing a graph, and may be used for representing fully known networks, such as social networks or road networks. For such examples it may be impractical to use an implicit representation as there is no easy way to give a successor function.
An implicit graph may be used for all graphs for which we can compute successor states with a function. Examples include communication protocols and hardware models, but also games and puzzles, e.g., one can use states to describe different board states in a game of checkers and edges to represent the possible moves to successor states or from predecessor states.

A forward-only implicit graph can be used for all cases where an implicit graph is used, but it is mainly useful in cases for which a predecessor state is difficult or even impossible to obtain. An example is to model a (non-deterministic abstraction of a) software program. By simply executing the program from a given state we can compute successor states, but it is far more difficult to obtain all states that precede the current program state (in particular those that are reachable from the initial state).

Notions of on-the-flyness. Depending on the application area, it may be valuable for an algorithm to report completely explored SCCs, or even fitting SCCs or cycles, before the algorithm terminates. In LTL model checking it is particularly useful to apply on-the-fly algorithms, as detection of an accepting cycle (a cycle in a Büchi automaton containing an accepting state) is sufficient to terminate the algorithm early. In Chapter 3 we focus on this topic, and in fact apply our SCC decomposition in the context of LTL model checking for different types of automata. Barnat et al. [BBR12] define three levels of “on-the-flyness” for LTL model checking algorithms; given that the automaton contains an accepting cycle: (M₁) the algorithm never terminates early, (M₂) the algorithm may terminate early, and (M₃) the algorithm is guaranteed to terminate early. While early termination is not exactly defined, the intended meaning is that only part of the state space has to be explored before the algorithm may terminate (the nested depth-first search algorithm [CVWY92] falls in this category). Notably, algorithms of type M₃ are the most valuable for model checking.

We adapt the “on-the-flyness” notion for the subject of SCC decomposition, where we assume that the algorithm eventually decomposes all SCCs, as follows.

O₁. The algorithm reports all SCCs after the entire graph is explored.
O₂. The algorithm reports a completed SCC once it has been fully explored.
O₃. The algorithm reports an FSCC $C$ when a new cycle is detected, such that the cycle is included in $C$, and $C$ also includes all previously reported cycles that form a non-empty intersection with states from $C$.

Here, $O₃$ is more useful than $O₂$ (since $O₃$ will eventually report completed SCCs), and $O₂$ is more useful than $O₁$. We note that an algorithm of the form $Oᵢ$ can be
modified to comply with $M_i$, for $i \in \{1, 2, 3\}$ (see Chapter 3).

The notion of on-the-flyness for an SCC algorithm may be combined with the types of graph on which the algorithm can be applied, where the combination of $O_3$ on a forward-only implicit graph is the most applicable scenario (the same algorithm can be applied on every other graph type, and notion of on-the-flyness). We call algorithms that operate in this scenario ideally applicable.

We remark that our notions of on-the-flyness are not exhaustive, but sufficient for this chapter. One can for instance consider algorithms that report every encountered simple cycle (a cycle in which no state may be repeated, aside from the first state being the last state), which resembles notion $O_3$ for which previously reported cycles are omitted. Another direction is to report all simple cycles, which may be exponentially many compared to the number of total states.

### 2.2.2 Union-find

The disjoint set union or union-find, originally designed by Galler and Fischer [GF64] and improved in e.g. [HU73, TvL84], is a data structure for storing and retrieving a disjoint partition of a set of nodes. It also supports merging of these partitions, allowing an incremental coarsening of the partition. A set is identified by a single node, the root or representative of the set. Other nodes in the set use a parent pointer to direct towards the root (in an inverted tree structure). There are two main functions that can be used in this structure:

- The $\text{Find}(a)$ operation recursively searches for the root of the set containing $a$, and updates the parent pointer of $a$ and parent pointers for all intermediate nodes, to directly point to the root (path-compression [HU73]).

- The $\text{Unite}(a, b)$ operation involves directing the parent pointer from the root of $a$ to the root of $b$, or vice versa. By choosing the root with the highest identifier (assuming a uniform distribution) as the ‘new’ root, the tree remains asymptotically optimally structured [GKL14].

**Example 2.2: Illustrating the union-find structure**

Consider the example union-find structure from Figure 2.3a. Here, $e$ and $a$ are the representatives of the sets $\{e, f, g\}$ and $\{a, b, c, d\}$, respectively. The operation $\text{Find}(e)$ returns $e$, and $\text{Find}(d)$ returns $a$. For the latter operation, with path-compression, we would update the parent pointer of $d$ to directly point to $a$.

The operation $\text{Unite}(a, b)$ will not modify the structure as $a$ and $b$ already belong to the same set. When applying $\text{Unite}(f, d)$, however, the
structure is updated to that of Figure 2.3b, where the parent pointer of \( e \) is updated to direct towards \( a \), causing all states to be part of the same set. Alternatively, we could have chosen to direct the parent pointer of \( a \) to point to \( e \).

Operations on the union-find take amortized quasi-constant time, bounded by the inverse Ackermann function \( \alpha \), which is practically constant. Union-find is therefore well-suited to maintain the SCC state sets as the algorithm coarsens them (e.g. by collapsing cycles).

2.3 Related work

Sequential DFS-based algorithms. Arguably, the most famous SCC decomposition algorithm is Tarjan’s algorithm [Tar72]. The algorithm tracks an index and lowlink value per state, where the index corresponds to the order in which the states are visited and the lowlink represents the smallest index reachable from the current state. SCCs are formed by updating the lowlink when a cycle is encountered and during the backtrack procedure. Besides Tarjan’s algorithm, there have been several different approaches that also rely on DFS.

The Kosaraju-Sharir algorithm [Sha81] performs two DFS instances, one forward and one backward, to decompose SCCs (note that this approach is inapplicable on a forward-only graph). Dijkstra proposed a version of Tarjan’s algorithm that instead of tracking lowlink values, maintains a stack of possible root candi-
Algorithm 1: Parallel Forward-Backward algorithm [FHP00].

```
input : Graph \( G = (V,E) \) and initial state \( v_0 \in V \).
output: Disjoint set union \( S \) of strongly connected components in \( G \).

1 \( S := \emptyset \) // set for tracking SCCs
2 FB(V) // main function call
3 function FB(A)
4     if \( A \neq \emptyset \) then
5         \( p := \text{Pivot}(A) \) // pick a state \( p \in A \), initially pick \( v_0 \in V \)
6         \( F := \text{FWD}(p,A) \) // compute forward reachable states
7         \( B := \text{BWD}(p,A) \) // compute backward reachable states
8         report SCC \( F \cap B \) // optionally, report each completed SCC
9         \( S := S \cup \{F \cap B\} \)
10        do in parallel // (NB: the three sets are disjoint)
11            FB(F \ B)
12            FB(B \ F)
13            FB(A \ (F \cup B))
```

dates [Dij76, Dij82]. When a back-edge is detected, the algorithm pops states from this stack until the ‘root’ of the cycle is found.

One of the early SCC algorithms, before Tarjan’s algorithm, was a set-based algorithm developed by Purdom [Pur70], and later optimized by Munro [Mun71]. Like Tarjan’s algorithm it uses DFS, but this is not explicitly mentioned (Tarjan was the first to do so). Moreover, instead of keeping states on the stack, the set-based algorithm stores partial SCCs (originally together with a set of all outgoing states of these sets). These sets can be handled in amortized, quasi-constant time by storing them in a union-find data structure, as noted by Gabow [Gab00] and presented in e.g. [RDKP13]. We provide a variant of this algorithm in Section 2.4, which forms the basis for our multi-core algorithm.

**Fix-point based algorithms.** We already discussed the Forward-Backward (FB) algorithm [FHP00] in Section 2.1, which finds SCCs by taking the intersection of all forward reachable states with the slice of backward reachable ones. We provide a version of the algorithm in Algorithm 1. Note that besides the stated parallelization, one could also perform the forward and backward reachability in parallel. A notable improvement to FB is the trimming procedure [IHPR05], which removes trivial SCCs without fix-point recursion. Trimming (recursively) removes states from the graph that contain no successors or no predecessors, since these
states form trivial SCCs. OBF [BCvdP11] further improves this by subdividing the graph in a number of independent sub-graphs. A different approach is the Colouring algorithm [Orz04]. This technique assigns a colour (number) to each state and these colours are recursively propagated to successors with a lower colour. This divides the graph in disconnected subgraphs and their backwards slices identify SCCs.

Barnat et al. [BBBJ11] provide CUDA implementations for FB, OBF, and Colouring. Coppersmith et al. [CFHm05] present and prove an $O(m \cdot \log n)$ serial runtime version of FB, Schudy [Sch08] further extended this to obtain an expected sequential runtime bounded by $O(\log^2 n)$ reachability queries.

Hong et al. [HRO13] improved FB for practical instances, by also trimming SCCs of size 2. After the first SCC is found (which is assumed to be large in size), the remaining components are detected (with Colouring) and decomposed with FB. Slota et al. [SRM14] propose orthogonal optimization heuristics and combine them with Hong’s work, which is further improved by Ji et al. [JLH18]. We compare our work with Hong’s algorithm (in an environment where the graph is explicitly given) and show that the techniques are complementary, in the sense that there are instances for which Hong’s algorithm outperforms ours and vice versa.

**Parallel on-the-fly algorithms.** Renault et al. [RDKP15] were the first to present a PRDFS algorithm that spawns multiple instances of Tarjan’s algorithm and communicates completely explored SCCs via a shared union-find structure. We improve this work by communicating partial SCCs. Lowe [Low16] runs multiple synchronized instances of Tarjan’s algorithm, without overlapping stacks. Instead a worker is suspended if it meets another’s stack and stacks are merged if necessary. While in the worst case this might lead to a quadratic time complexity, Lowe’s experiments show decent speedups on model checking inputs, though not for large SCCs.

**Parallel DFS.** Parallel DFS is a related problem. It has long been researched intensively, though there are two types of works [Fre91]: one discussing parallelizing DFS-like searches (often using terms like ‘backtracking’, load balancing and stack splitting), and theoretical parallelization of lexicographical (strict) DFS, e.g. [AAK90, dTK91]. Träff [Trä13] proposes a strategy to processes incoming edges, allowing them to be handled in parallel. The resulting complexity is $O(\frac{mn}{P} + n)$, providing speedup for dense graphs. Research into the characterization of search orders provides other new venues to analyse these algorithms [CK08, BKS05].
Union-find. The investigation of Van Leeuwen and Tarjan settled the question which of many union-find implementations were superior [TvL84]. Goel et al. [GKLT14] however later showed that a simpler randomized implementation can also be superior. Other versions exist that support deletion [KST02, ATG+14]. A wait-free concurrent union-find structure was introduced in [AW91] and later improved by Jayanti and Tarjan [JT16]. To the best of our knowledge, we are the first to combine both iteration and removal in the union-find structure. This allows the disjoint sets to be processed as a cyclic queue, which enables concurrent iteration, removal and merging.

2.4 Sequential set-based SCC algorithm

We update Munro’s algorithm [Mun71] to collapse cycles into fitting SCCs immediately, as in [Dij76, RDKP13]. We refer to this as the Set-Based SCC algorithm, which is ideally applicable and presented in Algorithm 2. Its essence is to perform a DFS and collapse cycles to supernodes, which constitute fitting SCCs. The strongly connected components are tracked in a union-find structure, which we represent using a map, \( S: V \rightarrow 2^V \). For this structure, we maintain the invariant:

\[ \forall v, w \in V : w \in S(v) \iff S(v) = S(w). \]

As a consequence, all the mapped sets are disjoint and retrievable in the map via any of its members. This construction allows us later to iterate over the sets. With the Unite function, we merge two mapped sets, such that the invariant is maintained. As an example, let \( S(v) = \{v\} \) and \( S(w) = \{w, x\} \), then Unite\((v, w)\) yields \( S(v) = S(w) = S(x) = \{v, w, x\} \) while keeping all other mappings the same.

The algorithm also maintains two strictly increasing sets, Visited and Explored, to keep track of encountered states and completely explored SCCs. It also keeps track of a stack \( R \), which contains a subset of states from the program stack, in the same order, while maintaining the property that states on the stack have paths to other states that are higher on the stack. We denote a stack as a sequence, e.g. \( R = \langle a, b \rangle \) with \( R.\text{Top}() = b \).

Algorithm details. We now explain Algorithm 2. The algorithm’s base DFS can be seen in lines 7, 9 and 11, i.e. for state \( v \) we mark it as visited, then consider all its successors, and recursively search its unvisited successors. If a visited state is encountered (that is not explored), then a cycle has been detected (lines 12 to 16). A cycle is collapsed by iteratively uniting the top two states from the \( R \) stack until eventually \( v \) and \( w \) belong to the same partial SCC (which forms a fitting SCC at line 16 since the complete cycle is united). Because cycles are collapsed
Figure 2.4: A graph for Example 2.3 with initial state $a$.

\begin{algorithm}[h]
\caption{Sequential set-based SCC algorithm.}
\begin{algorithmic}[1]
\Statex \textbf{input :} Graph $G = (V,E)$ and initial state $v_0 \in V$.
\Statex \textbf{output:} Disjoint set union $S$ of strongly connected components in $G$.
\State $\forall v \in V : S(v) := \{v\}$ // union-find structure for SCCs
\State $\text{Explored} := \emptyset$ // set of completely explored states
\State $\text{Visited} := \emptyset$ // set of visited states
\State $R := \emptyset$ // stack of SCC roots
\State $\text{SetBased}(v_0)$ // main function call
\Function{SetBased}{$v$}
\State $\text{Visited} := \text{Visited} \cup \{v\}$
\State $R$.Push($v$)
\ForAll{$w \in \text{Succ}(v)$}
\If{$w \in \text{Explored}$} continue // already completed SCC
\ElseIf{$w \notin \text{Visited}$} $\text{SetBased}(w)$ // unvisited node $w$
\Else // cycle found, merge states (NB: $\exists w' : w' \in R \land w \in S(w')$
\While{$S(v) \neq S(w)$} do // loop until $v$ and $w$ are united
\State $r := R$.Pop() // in the first iteration $r = v$
\State $\text{Unite}(r, R$.Top()) // unite the top two states from $R$
\EndWhile
\State \textbf{report FSCC} $S(v)$ // optionally, report fitting SCCs
\EndIf
\EndFor
\If{$v = R$.Top()} // completely explored SCC
\State \textbf{report SCC} $S(v)$ // optionally, report each completed SCC
\State $\text{Explored} := \text{Explored} \cup S(v)$
\State $R$.Pop()
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

Immediately, the stack $R$ is maintained independently of the program stack and invariently contains a subset of the latter (in the same order). The \texttt{Explored} set tracks all completely explored SCCs and states from this set are ignored when searching for successors (line 10).
Example 2.3: Example run of Algorithm 2

Consider the graph \( G \) from Figure 2.4 and run \texttt{SetBased}(a). We describe the process as follows.

1. After line 8 we have \( S = \{\{a\},\{b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\}\} \), \( R = \langle a \rangle \), \( \text{Visited} = \{a\} \), and \( \text{Explored} = \emptyset \).
2. Since \( b \) is a successor of \( a \), \texttt{SetBased}(b) gets called at line 11, and at line 8 in the recursive call we have that \( S \) is unchanged, \( R = \langle a, b \rangle \), \( \text{Visited} = \{a, b\} \), and \( \text{Explored} = \emptyset \).
3. From state \( b \), we first observe successor \( a \), which is part of the \texttt{Visited} set, hence we execute lines 12 to 16. State \( b \) gets popped from \( R \) and is united with \( a \) and the FSCC \( S(a) = \{a, b\} \) is reported.
4. The other successor of \( b \) (which is \( c \)) is processed recursively, from \( c \) we first visit \( e \), then \( f \), \( g \), and \( h \). We now have \( S = \{\{a, b\},\{c\},\{d\},\{e\},\{f\},\{g\},\{h\},\{i\}\} \), \( R = \langle a, c, e, f, g, h \rangle \), \( \text{Visited} = \{a, b, c, e, f, g, h\} \), and \( \text{Explored} = \emptyset \).
5. We encounter the edge \( h \to f \) and collapse the cycle until \( S(h) = S(f) \), by popping \( h \) and \( g \) from the stack and unifying these states with \( S(f) \). We report the FSCC \( \{f, g, h\} \) and have \( R = \langle a, c, e, f \rangle \).
6. There are no other successors of \( h \), and since \( h \neq R.\text{Top}() \), we backtrack. Then we also backtrack from state \( g \).
7. Now, we have processed the successor of \( f \) and observe in line 17 that \( f = R.\text{Top}() \). We are now certain that \( \{f, g, h\} \) is a complete SCC, which can be reported. We then mark all these states as \texttt{Explored} (s.t. \( \texttt{Explored} = \{f, g, h\} \)) and pop \( f \) from \( R \).
8. Then we observe that \( e = R.\text{Top}() \), thus report \( \{e\} \) as a trivial SCC and include it in the \texttt{Explored} set. We now have \( R = \langle a, c \rangle \), \( S = \{\{a, b\},\{c\},\{d\},\{e\},\{f, g, h\},\{i\}\} \), \( \text{Visited} = \{a, b, c, e, f, g, h\} \), and \( \text{Explored} = \{e, f, g, h\} \).
9. We now explore successor \( d \) of \( c \), which subsequently visits \( i \) and \( i \to e \).
   Since \( e \in \texttt{Explored} \), we can ignore this edge and backtrack from \( i \) (thereby also reporting and removing the SCC \( \{i\} \)).
10. We now explore \( d \to b \), which forms a cycle. We remove and unite states from \( R.\text{Top}() \) until \( S(b) = S(d) \). First \( d \) and \( c \) are combined to form \( S(d) = \{c, d\} \), which is then merged with the \( S(a) \) (since \( R.\text{Top}() = a \)) to obtain and report the FSCC \( \{a, b, c, d\} \).
11. After backtracking from states \( d \), \( c \), and \( b \), we are in the initial state \( a \). Here we have \( a = R.\text{Top}() \) and thus report and remove the SCC \( \{a, b, c, d\} \). After \( a \) is popped from \( R \), the algorithm ends with \( S = \{\{a, b, c, d\},\{e\},\{f, g, h\},\{i\}\} \), \( R = \emptyset \) and both \texttt{Explored} and \texttt{Visited} contain all states.
Correctness of Algorithm 2. We first provide useful properties of the algorithm. The algorithm partitions states in three sets, such that $\forall v \in V$, either:

1. $v \in \text{Explored}$, implying that $v$ is part of a completely explored (and reported) SCC, we also have $v \in \text{Explored} \implies v \in \text{Visited}$,

2. $v \in \text{Unseen}$, with $\text{Unseen} = V \setminus \text{Visited}$, (hence also $v \notin \text{Explored}$) implying that $v$ has not been encountered before by the algorithm, or

3. $v \in \text{Live}$, with $\text{Live} = \text{Visited} \setminus \text{Explored}$, implying that $v$ has been visited, but is not yet part of a completely explored SCC.

Every state is initially Unseen. During the algorithm, the states become Live and eventually Explored. After the algorithm, every state is Explored.

It can be shown that the algorithm ensures that all supernodes for the states in $R$ are disjoint and together contain all Live states:

$$\bigcup_{v \in R} S(v) = \text{Live}$$

(2.1)

(where $\bigcup$ is the disjoint union of sets, i.e. $A \cup B$ for which $A \cap B = \emptyset$). As a consequence, the Live states can be reconstructed using $S$ and $R$, so we do not actually need a Visited set except for ease of explanation. Furthermore, it can be shown that all Live states have a unique representation on $R$:

$$\{S(v) \cap R \mid v \in \text{Live}\} = \{\{r\} \mid r \in R\}.$$  

(2.2)

Or, for each Live state $v$, its mapped supernode $S(v)$ has exactly one representative $r \in R \cap S(v)$ such that $S(v) = S(r)$. Both equations play a role in ensuring that the algorithm returns complete SCCs, as explained next.

From the above, we can illustrate how the algorithm decomposes SCCs. If a successor $w$ of $v$ is Live, it holds that $\exists w' \in R: S(w') = S(w)$. Such a Live state is handled by lines 12 to 16, where the top two states from $R$ are united, until $w'$ is encountered, or rather until it holds that $S(w') = S(v) = S(w)$. Because $w'$ has a path to $v$ (an inherited invariant of the program stack), all united components are indeed strongly connected, i.e. lie on a cycle. Moreover, because of $r$’s uniqueness (Equation (2.2)), there is no other Live component part of this cycle (eventually this guarantees completeness, i.e. that the algorithm reports a maximal strongly connected component).

A completed SCC $S(v)$ is reported and marked Explored if $v$ remains on top of the $R$ stack at line 17, indicating that $v$ could not be united with any other state in $R$ (lower on $R$).
The current section describes our multi-core SCC algorithm. The algorithm is based on the PRDFS concept, where \( \mathcal{P} \) workers randomly process the graph, starting from \( v_0 \), and prune each other’s search space by communicating parts of the graph that have been processed. This dynamically, but not perfectly, partitions the graph across the workers, by trading redundancy for communication. To the best of our knowledge, the current approach of doing this is by ‘removing’ completed SCCs from the graph once identified by one worker [RDKP15]. The random exploration strategy then would take care of work load distribution. However, such a method would not scale for graphs consisting of single large SCC. We demonstrate a method that is able to communicate partial SCCs and can therefore scale on more graphs.

The basis of the parallel algorithm is the sequential set-based algorithm from the previous section, which stores strongly connected states in supernodes. For the time being, we do not burden ourselves with the exact data structures and focus on solutions that ease explanation. Afterwards, we show how the set operations are implemented.

**Communication principles.** We assume that each parallel worker \( p \) starts a PRDFS search with a local stack \( R_p \) and randomly searches the graph independently, starting from \( v_0 \). The upcoming algorithm is based on four core principles about state communication:

1. If a worker encounters a cycle, it communicates this cycle globally by uniting all states on the cycle to one supernode, i.e. \( S \) becomes a shared structure. As a consequence, worker \( p \) might unite multiple states that remain on the stack \( R_{p'} \) of some other worker \( p' \), violating Equation (2.2) for now.

2. Because, the workers can no longer rely on the uniqueness property, without additional care the algorithm looses its completeness property: It may wrongly report partial SCCs as complete SCCs because the collapsing stops early (before the highest connected supernode on the stack is encountered). To remedy this, we iterate over the contents of the supernodes until all of its states have been processed.

3. Since other workers constantly grow partial SCCs in the shared \( S \), it is no longer useful to retain a local \texttt{Visited} set. Equation (2.1) showed that indeed the \texttt{Live} states can be deduced from \( S \) and \( R \) in the sequential algorithm. We choose to use an implicit definition of \texttt{Live} so that a worker \( p \) only explores a state \( v \) if there is no \( r \in R_p \), s.t. \( v \in S(r) \), we say \( v \) is not
part of an \( R_p \) supernode. Thus states connected to a supernode on \( R_p \) by some other worker \( p' \) can be pruned by \( p \).

4. If a worker \( p \) backtracks from a state, i.e. it finds that all successors are either \textbf{Explored} or part of some supernode in \( R_p \), it records said state in a global set \textbf{Done} for all other workers to disregard. Here, a \textbf{Done} state implies that it is fully explored. Once all states in an SCC are \textbf{Done}, the SCC is marked \textbf{Explored}.

\subsection*{2.5.1 Algorithm}

The multi-core SCC algorithm, provided in Algorithm 3, is similar to the sequential set-based one (Algorithm 2) and follows the four discussed principles. We highlighted the main differences between Algorithm 3 and Algorithm 2. Each worker \( p \) maintains a local search stack \( (R_p) \), while \( S \) is shared among all workers for globally communicating partial SCCs. We also globally share the \textbf{Done} and \textbf{Explored} sets. For now we assume that all lines in the algorithm can be executed atomically\(^1\).

First of all, instead of performing only a DFS, the algorithm also iterates over states from \( S(v) \), at lines 8 to 17, until every \( v' \in S(v) \) is marked \textbf{Done} (principles 2 and 4). Therefore, once this while-loop is finished, we have that \( S(v) \subseteq \textbf{Done} \), hence the SCC may be marked \textbf{Explored} in line 18. The if-condition at that line succeeds if worker \( p \) wins the race to add \( S(v) \) to \textbf{Explored}, ensuring that only one worker reports the SCC. Thus, when \( UFSCC(v) \) terminates, we ensure that \( v \in \textbf{Explored} \) and that \( v \) is removed from the local stack \( R_p \) (line 21).

For every \( v' \in (S(v) \setminus \textbf{Done}) \), the successors of \( v' \) are considered in a randomized order via the \texttt{Random()} function at line 10 – according to PRDFS’ random search and prune strategy. After the forall-loop from lines 10 to 16, we considered each successor \( w \) of \( v' \) and infer that \( w \) is either in \textbf{Explored} or in the current SCC, i.e. \( \text{Succ}(v') \subseteq (\textbf{Explored} \cup S(v)) \). Therefore we mark \( v' \) to be \textbf{Done} (principle 4).

Compared to the \texttt{Visited} set from Algorithm 2, this algorithm uses \( S \) and \( R_p \) to derive whether or not a state \( w \) has been ‘visited’ before (principle 3). Here, ‘visited’ implies that there exists a state \( w' \) on \( R_p \) (hence also on the local search path) which is part of the same supernode as \( w \). Section 2.6 shows how an iteration on \( R_p \) in line 12 can be avoided. The recursive procedure for \( w \) is called for a successor of \( v' \), and we have that \( v' \in S(v) \). Assuming that \( \forall a, b \in S(v) : a \leftrightarrow b \)

\(^1\) We also assume that the combinations of lines 8 to 9 and lines 18 to 19 are executed atomically. In Section 2.6 we show that our data structures do not exactly guarantee that each line of the algorithm is executed atomically, but that race conditions are prevented to guarantee correctness.
Algorithm 3: The UFSCC algorithm (code for worker $p$).

1. $\forall v \in V : S(v) := \{v\}$ // global union find structure for SCCs
2. $\text{Done} := \emptyset$ // global set of completely explored states
3. $\text{Explored} := \emptyset$ // global set of completely explored SCCs
4. $\forall p \in [1 \ldots P] : R_p := \emptyset$ // local stack $R_p$ for each worker
5. $\text{UFSCC}_1(v_0) || \ldots || \text{UFSCC}_P(v_0)$ // main function in parallel for each worker

6. function $\text{UFSCC}_p(v)$
   7. $R_p$.Push($v$)
   8. while $(S(v) \setminus \text{Done}) \neq \emptyset$ do // keep picking states from $S(v)$
      9. pick $v' \in (S(v) \setminus \text{Done})$
      10.forall $w \in \text{Random}(\text{Succ}(v'))$ do
          11. if $w \in \text{Explored}$ then continue // already completed SCC
          12. else if $\nexists w' : w \in S(w')$ then $\text{UFSCC}_p(w)$ // (locally) new
          13. else // cycle found, merge states globally
              14. while $S(v) \neq S(w)$ do
                  15. $r := R_p$.Pop()
                  16. $\text{Unite}(r, R_p.\text{Top}())$ // concurrent state merging
          17. end while
      18. end forall
   19. end while
   20. $\text{Done} := \text{Done} \cup \{v'\}$
   21. if $S(v) \not\subseteq \text{Explored}$ then // all states in $S(v)$ are Done
      22. $\text{Explored} := \text{Explored} \cup S(v)$
      23. report SCC $S(v)$
   24. end if
   25. if $v = R_p.\text{Top}()$ then $R_p.\text{Pop}()$ // remove explored SCCs from the stack

holds, i.e. $S(v)$ is a PSCC, we have $v \rightarrow^* v' \rightarrow w$ and thus $w$ is reachable from $v$. This ensures that for any two consecutive states $a$ and $b$ in $R_p$, we have $a \rightarrow^* b$.

As an example, consider Figure 2.5, where we show a possible $R_p$ scenario for an execution of the algorithm. Here, $b$ is a successor of $a$, and a different worker has united states $b$ and $c$ together. Worker $p$ then picks a state, $c$, from $S(b)$ and finds successor $d$. From state $d$ we find the successor $e$, which is marked $\text{Done}$ (indicated in grey) and pick state $f \in S(e)$ to finally reach $g$.

We note that Algorithm 3 is not ideally applicable, i.e. it does not precisely respect the $O_3$ property. As an example, consider two workers that are traversing the graph, and both workers detect a different cycle. Suppose now that the first worker has finished uniting all states on this cycle and reports the PSCC. Then, it may be the case that the second worker is in the process of uniting states in the same PSCC, meaning that it may contain an incomplete cycle. Hence, the
first worker cannot be certain that the reported PSCC is an FSCC. However, for early termination (e.g., upon detecting an accepting cycle), we can force the other workers to halt after merging cycles and report an FSCC.

Essence of scalability. The scalability in the algorithm comes partly, but not completely from the PRDFS-based search, i.e., successors of a state are randomized to steer the various workers into different parts of the state space. Since the algorithm maintains a global set of PSCCs, $S$, whenever a worker encounters and contracts a cycle, this cycle is also communicated to all other workers. This on its own speeds up the detection of cycles, but it does not help with determining when an SCC is completely explored. If no further action is taken, then every worker would still have to explore the entire state space to ensure correctness. However, we mark every state $v$ as being Done when it is fully handled. That is, when the successors of $v$ either are part of an already explored SCC, or part of the current SCC, $S(v)$. When every state in $S(v)$ is marked as being Done, the SCC is fully explored. Since every worker can process different states concurrently, and workers are able to pick states from $S(v) \setminus \text{Done}$ (we assume for now that this is achieved efficiently), the algorithm also scales for fully exploring SCCs.

The idea of globally communicating cycles and marking individual states as being Done is especially useful for dealing with large SCCs, ideally with many internal cycles. Notably, this is a bottleneck in prior PRDFS-based solutions.

2.5.2 Correctness

The soundness defined here implies that reported SCCs are strongly connected, whereas completeness implies that they are maximal and that we find all reachable SCCs. We demonstrate soundness (in Theorem 2.1) by showing that states added to $S(v)$ are always strongly connected with $v$ (or any other state in $S(v)$). Completeness (see Theorem 2.2) follows from the requirements to mark an SCC Explored (Lemmas 2.2 to 2.4). Complete correctness follows from termination in
2.5. A multi-core SCC algorithm

Theorem 2.3.

**Lemma 2.1: Reachability on the stack**

Every state in $R_p$ can reach the top state of $R_p$, i.e. $\forall r \in R_p: r \rightarrow^* R_p.\text{Top}()$.

**Proof.** Follows from $R_p$ being a subset of the program stack.

**Theorem 2.1: Soundness**

The set $S$ always consists only of PSCCs, i.e. for any two states $v$ and $w$ in $S(v)$, we have that $v \leftrightarrow w$.

**Proof.** Initially the hypothesis holds, since $\forall v: S(v) = \{v\}$. Assuming that the theorem holds for $S$ before executing a statement of the algorithm, we show it holds after, considering only the non-trivial case, i.e. line 16. Before line 14, we have $S(v) = S(R_p.\text{Top}())$, since either $v = R_p.\text{Top}()$ or a recursive procedure has united $v$ with the current $R_p.\text{Top}()$. The while-loop (line 14-16) continuously pops the top state of $R_p$ and unites it with the new $R_p.\text{Top}()$. Since the condition at line 12 did not hold, we can assume that $\exists w' \in R_p: w \in S(w')$.

The while-loop ends iff $S(R.\text{Top}()) = S(v) = S(v') = S(w) = S(w')$. Because $v' \rightarrow w$, $w \rightarrow^* w'$, and $w' \rightarrow^* R.\text{Top}()$ (Lemma 2.1), we have that the cycle $v' \rightarrow^* w \rightarrow^* v'$ is merged, thus preserving the hypothesis in the updated $S$. 

**Lemma 2.2: Termination guarantees exploration**

After $UFSCC_p(v)$ terminates, $v \in \text{Explored}$.

**Proof.** This follows directly from lines 18 to 19 and the fact that $v$ is never removed from $S(v)$.

**Lemma 2.3: Successors of Done states**

Successors of Done states are Explored or in a supernode on $R_p$:

$$\forall v \in \text{Done}, \forall w \in \text{Succ}(v): (w \in \text{Explored}) \lor (\exists r \in R_p: w \in S(r)).$$
Proof. The only place where Done is modified is in line 17. States are never removed from Explored or from $S(v)$ for any $v$. In lines 10 to 16, all successors $w$ of $v'$ are considered separately:

1. $w \in \text{Explored}$ is discarded (trivially adheres to the lemma).
2. $\nexists w' \in R_p : w \in S(w')$. Here, $w$ is not in a supernode of $R_p$ and UFSCC is recursively invoked for $w$. From Lemma 2.2, we obtain that $w \in \text{Explored}$ upon return of UFSCC$_p(w)$.
3. $\exists w' \in R_p : w \in S(w')$. There is some $w' \in R_p$ s.t. $S(w) = S(w')$. Supernodes of states on top of $R_p$ are united until $S(v) = S(v') = S(w)$.

All three cases thus satisfy the conclusion of the hypothesis before $v'$ is added to Done at line 17.

\textbf{Lemma 2.4: Explored states are Done}

If a state $v$ is Explored, then all states in $S(v)$ are Explored and Done:

$$v \in \text{Explored} \implies \forall v' \in S(v) : v' \in \text{Explored} \land v' \in \text{Done}.$$  

Proof. Follows from the while-loop exit condition at line 8.

\textbf{Theorem 2.2: Completeness}

After UFSCC$_p(v)$ finishes, for every reachable state $t$ (i.e. $v \rightarrow^* t$) we have:

(a) $t \in \text{Explored}$, and
(b) $S(t)$ is an SCC.

Proof. ((a) every reachable state $t$ is Explored). We have that $v \in \text{Explored}$ (Lemma 2.2) and therefore also $\forall v' \in S(v) : v' \in \text{Done}$ (Lemma 2.4). By Lemma 2.3, we deduce that every successor of $v'$ is Explored (as $v' \in S(v)$ implies $v' \in \text{Explored}$ by line 18). Therefore, by applying the above reasoning recursively on the successors of $v'$ we obtain that every reachable state from $v$ is Explored.

((b) $S(t)$ is an SCC). We show that $S(t)$ is a maximal FSCC. Assume by contradiction that for disjoint Explored sets $S(v) \neq S(w)$ we have $v \rightarrow^* w \rightarrow^* v$. W.l.o.g., we can assume $v \rightarrow w$. Since $S(w) \neq S(v)$, we deduce that $w \in \text{Explored}$ when this edge is encountered (Lemma 2.3). However, since $w \rightarrow^* v$ we must also have $v \in \text{Explored}$ (Theorem 2.2a), contradicting our assumption. Hence, every Explored $S(t)$ is a maximal SCC.
Theorem 2.3: Termination

Algorithm 3 terminates for finite graphs.

Proof. The while-loop terminates because states can only be added once to $S(v)$ due to a strictly increasing Done in every iteration. The recursion stops because the states part of Live supernodes in $R_p$ only increase: as per line 7 and line 16. □

2.6 Implementation

For the implementation of Algorithm 3, we require the following:

- Data structures for $R_p$, $S$, Explored and Done.
- A mechanism to iterate over $v' \in (S(v) \setminus \text{Done})$ (line 9).
- Means to check, for a state $v$, if there is some $r \in R_p$, s.t. $r \in S(v)$ (line 12).
- An implementation of the Unite() procedure (line 16).
- A technique to add states to Done and Explored (lines 17 to 19).

We first explain how each aspect is implemented and provide detailed algorithms in Technicalities 2.1 to 2.3.

2.6.1 Concurrent union-find structure

The local stack, $R_p$, can be implemented with a standard stack structure. $S$ can be implemented with a variation of the wait-free union-find structure [AW91]. Its implementation employs the atomic Compare&Swap (CAS) instruction, which is shown in Algorithm 4. If the value for $x$ equals $a$, it is updated atomically to $b$. The function returns whether $x$ was successfully updated.

<table>
<thead>
<tr>
<th>Algorithm 4: Atomic Compare and Swap instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 function CAS($x, a, b$)</td>
</tr>
<tr>
<td>2    if $x = a$ then</td>
</tr>
<tr>
<td>3        $x := b$</td>
</tr>
<tr>
<td>4        return True</td>
</tr>
<tr>
<td>5    else return False</td>
</tr>
</tbody>
</table>
We use the CAS instruction to update the parent pointer for a state in the union-find structure without interfering with operations from other workers. The union-find structure itself is implemented as an array, UF, which we index using unique identifiers of states (in our implementation, these are represented by hash values). In a Unite\((a,b)\) procedure, the new root of \(S(a)\) and \(S(b)\) is determined by selecting either the root of \(S(a)\) or \(S(b)\), whichever has the highest index, i.e. essentially random in our setting using hashed locations. This mechanism for uniting states preserves the quasi-constant time complexity for operations on the union-find structure [GKL14].

The Explored set is implemented with a status field in the union-find structure. We have that \(S(v)\) is Explored if the status for the root of \(v\) is Explored. Thus marking an SCC Explored (using a CAS instruction) on the root indeed implements line 18 and line 19 of Algorithm 3 atomically. This marking is achieved in amortized quasi-constant time (a Find() call with a status update).

### Technicalities 2.1: Concurrent union-find

We show how the Find and Unite functions are implemented in Algorithm 5. Note that the union-find structure will be updated in later sections, causing the Unite procedure to be modified in Algorithm 8.

The Find operation in our concurrent structure works the same as its sequential version, i.e. we traverse the parent-pointers until we reach the root state, and update the traversed parents to point directly to the root. Concurrent interleaveings do not affect correctness because the sets only increase, i.e. a state is never removed from the union-find structure. Of course, the root might be updated right before Find returns the found root.

The Unite function is implemented by first computing the roots of the to-be-united states, \(a\) and \(b\). In case these are the same, then the states are already part of the same set. Otherwise, we select the root with the highest index in the UF array, \(r\), and choose this as the new root\(^a\). Using a CAS instruction, we update the parent of the lower root, \(q\), to point to \(r\). It may be possible that \(q\) has been updated in the meantime. For instance, when a different worker united \(q\) with some state \(s\) such that \(q < r < s\), before line 11, then \(UF[q].parent = s\). In this case the CAS fails, and we retry the Unite procedure. After the Unite procedure terminates, we guarantee that states \(a\) and \(b\) are part of the same set.

In order to implement line 14 from Algorithm 3 (checking whether \(S(v)\) equals \(S(w)\)), we provide a SameSet procedure that checks if two states are part of the same set. Similar to Unite, it computes the roots of both states and returns True if they are equal. Otherwise we check if the roots
have been updated in the mean time. In case the root for \(a\) is unchanged, we know that \(a\) and \(b\) were disjoint at the start of the \texttt{SameSet} procedure, which is sufficient for Algorithm 3. Otherwise, we retry the procedure.

We note that there are some intricacies to the \texttt{SameSet} procedure. In fact, it may report a wrong result in case \(b\)’s root is updated by another worker to point to \(\text{root}_a\) between line 16 and 17. Because the procedure relies on the most up-to-date results for the roots of \(a\) and \(b\), it is actually not possible to check if they do \textit{not} belong to the same set (assuming no locking mechanism is used). However, when \texttt{SameSet} returns \texttt{False}, it does guarantee that at least up to some point in the procedure, \(a\) and \(b\) were in distinct sets. This is sufficient for the main algorithm, Algorithm 3, because then we know that the top two states of the \(R_p\) stack may be merged together.

\begin{algorithm}
\begin{algorithmic}[1]
\caption{Find, Unite, and SameSet functions}
\Function{Find}{$a$}
\State if \(\text{UF}[a].\text{parent} \neq a\) then \State \hspace{1em} \(\text{UF}[a].\text{parent} := \text{Find}((\text{UF}[a].\text{parent}) \) \text{// path compression}
\State \hspace{1em} \Return \(\text{UF}[a].\text{parent}\) \text{// return the root}
\EndFunction
\Function{Unite}{$a,b$}
\State \(\text{root}_a := \text{Find}(a)\)
\State \(\text{root}_b := \text{Find}(b)\)
\If{\(\text{root}_a = \text{root}_b\)} \Return \text{// already united} \EndIf
\State \(r := \max(\text{root}_a, \text{root}_b)\) \text{// largest index is the new root}
\State \(q := \min(\text{root}_a, \text{root}_b)\)
\If{\(\neg \text{CAS}((\text{UF}[q].\text{parent}, q, r))\) \then} \State \hspace{1em} \Return \text{// retry if failed}
\EndIf
\EndFunction
\Function{SameSet}{$a,b$}
\State \(\text{root}_a := \text{Find}(a)\)
\State \(\text{root}_b := \text{Find}(b)\)
\If{\(\text{root}_a = \text{root}_b\)} \Return \text{True} \EndIf
\If{\(\text{UF}[\text{root}_a].\text{parent} = \text{root}_a\)} \Return \text{False} \EndIf
\State \Return \text{SameSet}(\text{root}_a, \text{root}_b)$
\EndFunction
\end{algorithmic}
\end{algorithm}

\footnote{It is necessary to enforce a globally agreed upon scheme for selecting the new root (i.e. in our case the highest index), since otherwise we may get the situation in which two workers unite two (sets of) states together and have their parent pointers direct to each other (e.g. \(\text{UF}[a].\text{parent} = b\) and \(\text{UF}[b].\text{parent} = a\)). As a result, the \texttt{Find} operation will never terminate for states in this set.}
Figure 2.6: Illustration of the worker set before (left) and after (right) uniting the two PSCCs. The top side depicts the graph, the union-find structure with the worker set is shown below for workers $R$ and $B$.

### 2.6.2 Worker set for tracking visited states

We show how $\forall w' \in R_p : w \in S(w')$ (line 12 from Algorithm 3) is implemented. In the union-find structure, for each node we keep track of a bitset of size $\mathcal{P}$. This worker set contains a bit for each worker and represents who is currently exploring a partial SCC. We say that worker $p$ has visited a state $w$ if the worker set for the root of $S(w)$ has the bit for $p$ set to true. If otherwise worker $p$ encounters an unvisited successor state $w$, it sets its worker bit in the root of $S(w)$ using CAS and recursively calls UFSCC($w$) (line 12). In the `Unite(a,b)` procedure, after updating the root, the worker sets for $a$ and $b$ are combined (with a logical OR, implemented with CAS) and stored on the new root to preserve that a worker bit is never removed from a set. The process is repeated if the root is updated while updating the worker set. Since only worker $p$ can set worker bit $p$ (aside from combining worker sets in the `Unite()` procedure), only partial SCCs visited by worker $p$ (on $R_p$) can and will contain the worker bit for $p$. 
Example 2.4: Worker set illustration

Consider Figure 2.6. Here, worker B has found the cycle $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$, which are all united: $S(b) = \{a,b,c,d\}$, and worker R explored $a \rightarrow e \rightarrow f \rightarrow e$ (with $S(f) = \{e,f\}$). Worker R now considers the edge $f \rightarrow c$ and wants to know if it has previously visited a state in $S(c)$ (which would imply a cycle). Because the worker set is managed at the root of $S(c)$ (which is node b), worker R simply has to check at b if its worker bit is set. Since this is the case, the sets $S(f)$ and $S(e)$ can be united.

Technicalities 2.2: Visiting states using a worker set

We may now distinguish between multiple types of states while traversing the state space, e.g. unvisited states, completely explored states, and states that were visited (perhaps implicitly) before. To quickly determine in which category a given state belongs, we make use of a MakeClaim function.

We show how the MakeClaim function from Algorithm 6 works and how it can be used in Algorithm 3 to check whether $\exists w' \in R_p : w \in S(w')$ holds. Two worker sets are merged together in the Unite procedure, which then requires a locking mechanism. We show how the extended Unite procedure is implemented in Algorithm 8.

Algorithm 6: Determining the status of a state with the worker set

```plaintext
1 function MakeClaim(a, p)
2     root_a := Find(a)
3     if UF[root_a].uf_status = Explored then
4         return claim_explored // completely explored SCC
5     if p \in UF[root_a].workers then
6         return claim_found // set already contains worker ID
7     while p \not\in UF[root_a].workers do
8         UF[root_a].workers := UF[root_a].workers \cup \{p\} // add worker ID
9         root_a := Find(root_a) // ensure that the worker ID is added
10    return claim_new
```

The MakeClaim procedure first checks whether the set of states is already marked Explored, by referencing the uf_status field of its root. Then, in case the SCC is not yet Explored, we check if the worker ID is already present in the set, i.e. if the current worker has previously visited any state in the set. Otherwise, we add the worker ID to the set, by adding it to the root of the set. Afterwards we check if the root has been correctly updated.
with the worker ID info, since it may be the case that another worker executed a \textit{Unite} procedure that would change the root before the worker ID was added. The function returns \texttt{claim\_explored}, \texttt{claim\_found}, or \texttt{claim\_new} if \( S(a) \) is \texttt{Explored}, if it contains the current worker ID, or if the worker ID was added, respectively. Note that these results address the conditionals from lines 11 to 13 in Algorithm 3.

### 2.6.3 Cyclic linked list for iteration and tracking done states

The most challenging aspect of the implementation is to keep track of the \texttt{Done} states and iterate over \( S(v) \setminus \texttt{Done} \). We do this by making the union-find iterable by adding a separate cyclic list implementation. That is, we extend the structure with a \texttt{next} pointer and a list status flag that indicates if a state is either \texttt{Busy} or \texttt{Done} (initially, the flag is set to \texttt{Busy}).

We refer to Figure 2.7 for a state diagram of the different flags that we use internally. The \texttt{uf\_status} field refers to the status for an entire PSCC, which is initially \texttt{Live}, and the \texttt{list\_status} field refers to the status of an individual state, which is initially set to \texttt{Busy}. After a PSCC has been completely explored (i.e. when all states in the PSCC have been marked \texttt{Done}), the PSCC (or rather SCC) becomes \texttt{Explored}. For now, we can disregard the \texttt{Lock} states in both diagrams.

The idea is that the cyclic list contains all \texttt{Busy} states. States marked \texttt{Done} are skipped for iteration, yet not physically removed from the cycle. Rather, they will be lazily removed once reached from their predecessor on the cycle (this avoids maintaining doubly linked lists). All \texttt{Done} states maintain \texttt{next} pointers that direct towards the cyclic list, to ensure that the cycle of \texttt{Busy} states is reachable.
from every state $v' \in S(v)$. If all states from $S(v)$ are marked \texttt{Done}, the cycle becomes empty (we end up with a \texttt{Done} state directing to itself) and $S(v)$ can be marked \texttt{Explored}.

\begin{example}[Cyclic list illustration]
To illustrate the cyclic list structure, consider the example from Figure 2.8. In the partial SCC $\{a, b, c, d\}$, states $a$ and $c$ are marked \texttt{Done} (grey), and states $b$ and $d$ are \texttt{Busy} (white). We further assume that $\{e\}$ is an \texttt{Explored} SCC. The states $a$ and $c$ are marked \texttt{Done} since their successors are in $(\texttt{Explored} \cup S(a))$ (see also Lemma 2.3). Note that $b$ and $d$ (and $f$) may not yet be marked \texttt{Done} because their successors have not been fully explored yet. The right side depicts a possible representation for the cyclic list structure (where the blue arrows are next pointers).

In the algorithm, line 8 and 9 (iterating over $S(v) \setminus \texttt{Done}$) can be implemented by recursively traversing $v$.\texttt{next} to eventually encounter a state $v' \in (S(v) \setminus \texttt{Done})$ or otherwise it detects that there is no such state and the while-loop ends. line 17 is implemented by setting the status for $v'$ to \texttt{Done} in the union-find structure, which is lazily removed from the cyclic list.

The \texttt{Unite}(a, b) procedure is extended to combine two cyclic lists into one (without losing any states). We show this procedure using the example from Figure 2.9. Here \texttt{Unite}(a, f) is called ($b$ becomes the new root of the partial SCC). The next pointers (solid blue arrows) for $a$ and $f$ are swapped with each other (note that

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{cyclic_list.png}
\caption{Illustration of the cyclic list in practice. The left side depicts the graph, the right side depicts the corresponding cyclic list structure. White nodes are \texttt{Busy} and grey nodes are \texttt{Done}. The blue arrows depict next pointers.}
\end{figure}
Chapter 2. Multi-Core Computation of SCCs

this cannot be done atomically), which creates one list containing all states. It is important that both $a$ and $f$ are Busy states (as otherwise part of the cycle may be lost), which is ensured by first searching and using light-weight, fine-grained locks (one per node) on Busy states $a$ and $f$. The locks also protect the non-atomic pointer swap (see Algorithm 7 and Algorithm 8).

Removing a state from the cyclic list is a two-step process. First, the status for the state is set to Done. Then, during a search for Busy states, if two subsequent Done states are encountered, the next pointer for the first one is updated (Figure 2.10). Note that the ‘removed’ state remains pointing to the cyclic list. One may observe a resemblance between this procedure and path compression (or more precisely, path halving [vLvdW77]), which is used to compress parent pointers in the union-find structure.

We use a lock on the union-find structure (only on the root for which the parent gets updated) and a lock on list nodes encoded as status flags. The structure for union-find nodes is shown in Figure 2.11. We chose to use fine-grained locking instead of wait-free solution, for two reasons: (1) we do not require the stronger guarantees of the latter, and (2) we expect that the pointer indirection needed for a wait-free design would decrease performance compared to the current node layout with its memory locality.
### Technicalities 2.3: Iterable union-find and complete algorithm

We show how the iterable union-find is implemented in Algorithm 7, how the `Unite` procedure is updated to accommodate the extension in Algorithm 8, and finally give a complete version of the UFSCC algorithm in Algorithm 9.

The `PickFromList(a)` procedure is used to determine if $S(a) \subseteq \text{Done}$ (by returning `NULL`), otherwise, it returns a state $s \in (S(a) \setminus \text{Done})$. We traverse the list via the `next` pointers and return the first available `Busy` state.

**Algorithm 7: The list part of the iterable union-find**

```plaintext
function PickFromList(a)
    do // wait until $a$ is not locked (another worker may be uniting)
        if UF[a].list_status = Busy then return a
    while UF[a].list_status \neq Done
    b := UF[a].next // $a$.list_status = Done (exit from do-while)
    if a = b then // self-loop of a Done state => all states are Done
        root_a := Find(a) // mark the SCC Explored, if not already done
        if CAS(UF[root_a].uf_status, Live, Explored) then
            report SCC root_a // optionally, report the SCC root
        return NULL // no state to pick from an Explored SCC
    do // also make sure that $b$ is not locked, return $b$ if it’s Busy
        if UF[b].list_status = Busy then return b
    while UF[b].list_status \neq Done // exit if status = Done
    c := UF[b].next // we have $a \rightarrow b \rightarrow c$ via next pointers
    UF[a].next := c // $a$ and $b$ are Done => shrink the list: $a \rightarrow c$
    return PickFromList(c) // recursively traverse the cyclic list

function RemoveFromList(a)
    while UF[a].list_status \neq Done do // try again if $a$ is locked
        CAS(UF[a].list_status, Busy, Done) // only remove Busy state
```

Figure 2.11: The node record of the iterable union-find data structure.
In case we encounter a locked state (a state’s list_status gets set to Lock during the Unite procedure) we wait until it gets unlocked again (lines 4 and 13). If we encounter that \( a . \text{next} = a \), we found that all states in \( S(a) \) are marked Done, mark \( S(a) \) as Explored and return NULL in lines 6 to 10.

In case both \( a \) and \( b \) are Done states, we recursively continue the search from \( b \)'s successor, \( c \). Before that, we update the next pointer from \( a \) to \( c \) to reduce the length of the cyclic list.

The RemoveFromList\( (a) \) procedure continuously tries to mark the list_status of \( a \) to Done (it may be locked now). It may also be the case that some other worker marks the state as Done.

**Algorithm 8:** The Unite procedure of the iterable union-find

1. function \( \text{LockRoot}(a) \)
2.   \[ \text{if CAS(}UF[a].\text{uf_status, Live, Lock) then} \]
3.     \[ \quad \text{if } UF[a].\text{parent} = a \text{ then return True} \]
4.     \text{return False} // unable to lock root (unnecessary to unlock)
5. 
6. function \( \text{LockList}(a) \)
7.   \[ s := \text{PickFromList}(a) \text{ // pick a Busy state} \]
8.   \[ \text{if } s = \text{NULL then return NULL} \text{ // cannot lock an explored SCC} \]
9.   \[ \text{if CAS(}UF[s].\text{list_status, Busy, Lock) then return } s \]
10.  \text{return LockList}(s) \text{ // locked by another worker, try again}
11. 
12. function \( \text{Unite}(a,b) \)
13.   \[ \text{root}_a := \text{Find}(a) \]
14.   \[ \text{root}_b := \text{Find}(b) \]
15.   \[ \text{if root}_a = \text{root}_b \text{ then return} \text{ // already united} \]
16.   \[ r := \text{max(root}_a, \text{root}_b) \text{ // largest index is the new root} \]
17.   \[ q := \text{min(root}_a, \text{root}_b) \]
18.   \[ \text{if } \neg \text{LockRoot}(q) \text{ then return Unite(root}_a, \text{root}_b) \text{ // try again} \]
19.   \[ a_{\text{List}} := \text{LockList}(a) \text{ // lock the list states} \]
20.   \[ b_{\text{List}} := \text{LockList}(b) \]
21.   \[ \text{Swap}(UF[a_{\text{List}}].\text{next}, UF[b_{\text{List}}].\text{next}) \text{ // non-atomic instruction} \]
22.   \[ \text{UF[q].parent := } r \text{ // update parent (to some state in } S(r)) \]
23.   \[ \text{do} \text{ // update the worker set for the root} \]
24.     \[ r := \text{Find}(r) \]
25.     \[ \text{UF[r].workers := UF[r].workers } \cup \text{UF[q].workers} \text{ // atomic} \]
26.     \[ \text{while } \neg \text{UF[r].parent} = r \text{ // ensure that we update the root} \]
27.     \[ \text{UF[a_{\text{List}}].list_status := Busy} \text{ // unlock the locked list states} \]
28.     \[ \text{UF[b_{\text{List}}].list_status := Busy} \]
In Algorithm 8 we show the complete Unite procedure (that also updates the worker set and the cyclic list). The procedure remains the same as in Algorithm 5 for lines 11 to 15. Then, we attempt to lock \( q \) (and make sure that we have locked the root of \( S(q) \)), if this fails we simply try again. Once \( q \) is locked, no other worker may lock it and we also ensure that no list_status fields for states in \( S(q) \) have been set to Lock (we may however have locked list_status fields for some states in \( S(r) \)).

The procedure continues by locking a list state from \( S(b) \) and \( S(b) \). Then, their next pointers get swapped and we combine the worker sets of \( S(q) \) and \( S(r) \). Once this is done, the list_status fields are set back to Busy and the procedure terminates. Since \( q \) is no longer a root, we do not have to unlock it. In previous work [Blo15], we had to lock both \( S(q) \) and \( S(r) \) whereas now we only have to lock \( S(q) \). This led to a significant performance improvement as it allows to concurrently unite multiple SCCs together.

The complete version of the UFSCC algorithm is given in Algorithm 9. This version relies on the implementations from Algorithms 5 to 8.

**Algorithm 9: Implementation of the UFSCC algorithm**

```
\[\forall p \in [1...P] : R_p := \emptyset \] // local stack for each worker

function UFSCC_main(v_0, P)

forall p \in [1...P] do

    MakeClaim(v_0, p) // set worker IDs for state \( v_0 \)

UFSCC_1(v_0) || ... || UFSCC_P(v_0) // run in parallel

function UFSCC_p(v)

R_p.Push(v)

v' := PickFromList(v) // pick a Busy state from \( S(v) \)

while v' \neq \text{NULL} do

    forall w \in \text{Random}(\text{Succ}(v')) do

        claim := MakeClaim(w, p) // (ignore claim_explored)

        if claim = claim_new then UFSCC_p(w) // (locally) new

        else if claim = claim_found then // cycle detected

            while \neg \text{SameSet}(v, w) do

                r := R_p.Pop()

                Unite(r, R_p.Top()) // merge top two stack states

            RemoveFromList(v') // fully handled all states from Succ(v')

        v' := PickFromList(v') // pick next state

    if v = R_p.Top() then R_p.Pop() // remove completed SCC
```
2.6.4 Runtime complexity

Our treatment of the complexity here is not as formal as in e.g. [dlt91], and relies on practical assumptions. We argue that, while the theoretical complexity study of parallel algorithms offers invaluable insights, it is also remote from the practical solution that we focus on. For instance, we cannot within the foreseeable future obtain \( n \) parallel processors for \( n \)-sized graphs.

We assume that overhead from synchronization and successor randomization is bound by a constant factor. Because the worker sets need to be updated, \texttt{Unite} operations take \( \mathcal{O}(\mathcal{P}) \) time and work (instructions). Assuming again that \( \mathcal{P} \) is small enough for the worker set to be stored in one or few words (e.g. 64), \texttt{Unite} becomes quasi constant (now the \texttt{Find} operation it calls dominates). Finding a \texttt{Busy} state in the cyclic list is bounded by \( \mathcal{O}(n) \) (the maximal sequence of \texttt{Done} states before reaching a \texttt{Busy} state). However, we conjecture that the lazy removal of \texttt{Done} nodes performs equivalent to path halving [vLvdW77], which has been shown to have an amortized quasi-constant time complexity [TvL84]. Therefore, supernode iteration is amortized quasi constant under the condition that \( \mathcal{P} \ll n \). Though if this not the case, we could always use a more complicated doubly linked list instead [Wy179, p. 63]. So amortized, all supernode operations are bounded by \( \mathcal{O}(\alpha) \) time and work, i.e. the inverse Ackermann function representing quasi constant time.

Since every worker may visit every edge and state, performing a constant number of supernode operations on the state, the worst-case work complexity for the algorithm is bounded by \( \mathcal{O}(\alpha \cdot \mathcal{P} \cdot (n + m)) \). As typically, the inverse Ackermann function is bounded by a constant, we obtain \( \mathcal{O}(\mathcal{P} \cdot (n + m)) \). The runtime, however, remains bounded by \( \mathcal{O}(m + n) \) (ignoring the quasi factor), as useless work is done in parallel. The best-case runtimes from Table 2.1, \( \mathcal{O}\left(\frac{n+m}{\mathcal{P}}\right) \), follow when assuming that approximately each state is processed by only one worker.\(^2\)

2.6.5 Memory usage

The memory usage of Algorithm 3, using the implementation discussed in the current section, comprises two aspects. First, each worker has a local stack \( R_p \), which contains at most \( n \) items (e.g. states on a path or within the same SCC when drawn from the iterable union-find structure). Second, the global union-find contains: a parent pointer, a worker set, a next pointer, and status fields for both union-find and its internal list (Figure 2.11). Assume that states and pointers in

\(^2\) We acknowledge that for fair comparison with Table 2.1, the impact of the worker set (a factor \( \mathcal{P} \)) cannot be neglected, e.g. for large \( \mathcal{P} \) approaching \( n \), yielding \( \mathcal{O}(\mathcal{P} \cdot \max(\alpha, \mathcal{P}) \cdot (n + m)) \) work and \( \mathcal{O}(\max(\alpha, \mathcal{P}) \cdot (n + m)) \) runtime.
the structure can be stored in constant amount of space, their storage requires 3 units. Also assuming again that $P \ll n$, the worker set takes another unit, e.g., one 64-bit machine word for $P = 64$. Then, the algorithm uses at most $(P \cdot n) + 4n$ units (Stacks plus shared data structures).

The worst-case space complexity becomes $O(P \cdot n)$ (also when $P$ is not constant, e.g., approaching $n$, and the worker set cannot be stored in a constant anymore). The local replication on the stacks is the main cause of the worst-case memory usage of PRDFS algorithms as shown in Table 2.1 (other PRDFS algorithms, such as [RDKP15, ELPvdP12, EPY11, LF13], do not require a worker set). In practice however, the replication is proportional to the contention occurring on states which is arguably low when the algorithm processes large graphs and $P \ll n$. Hence the actual memory usage will be closer to the best-case complexity that occurs when stacks do not overlap, i.e.: $O(n)$ (unless, again, $P$ is not constant and the worker set dominates memory usage).

## 2.7 Experiments

The current section presents an experimental evaluation of UFSCC. Results are evaluated compared to Tarjan’s sequential algorithm [Tar72], the on-the-fly PRDFS algorithm from Renault et al. [RDKP15], and the offline fix-point algorithm from Hong et al. [HRO13], using model checking inputs, synthetic graphs and explicitly given graphs.

**Experimental setup.** All experiments were performed on a machine with 4 AMD Opteron™ 6376 processors, each with 16 cores, forming a total of 64 cores. There is a total of 512GB memory available. We performed every experiment at least 50 times and calculated their means and 95% confidence intervals.

We implemented UFSCC in the LTSMin model checker [KLM+15],³ which we use for decomposing SCCs on implicitly given graphs. LTSMin’s post() implementation applies measures to permute the order of a state’s successors so that each worker visits the successors in a different order. We compare against a sequential version of Tarjan’s algorithm and the PRDFS algorithm from Renault et al. [RDKP15] (see Section 2.3), which we refer to as Renault. Both are also implemented in LTSMin to enable a fair comparison. Unfortunately, we were unable to implement Lowe’s algorithm [Low16], nor successful to use its implementation for a comparison.

³The implementations for the algorithms, the benchmark graphs, and results are available online via https://github.com/utwente-fmt/ppopp16.
Table 2.2: Graph characteristics for model checking, synthetic and explicit graphs. Here, \( M \) denotes millions and columns respectively denote the number of states, transitions, SCCs, the maximal SCC size, and an approximation of the diameter (the greatest distance between any two vertices).

<table>
<thead>
<tr>
<th>graph</th>
<th>states</th>
<th>trans</th>
<th>SCCs</th>
<th>max SCC</th>
<th>diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader-filters.7</td>
<td>26.3M</td>
<td>91.7M</td>
<td>26.3M</td>
<td>1</td>
<td>71</td>
</tr>
<tr>
<td>bakery.6</td>
<td>11.8M</td>
<td>40.4M</td>
<td>2.6M</td>
<td>8.6M</td>
<td>176</td>
</tr>
<tr>
<td>cambridge.6</td>
<td>3.4M</td>
<td>9.5M</td>
<td>8,413</td>
<td>3.3M</td>
<td>418</td>
</tr>
<tr>
<td>lup.3</td>
<td>1.9M</td>
<td>3.6M</td>
<td>1</td>
<td>1.9M</td>
<td>134</td>
</tr>
<tr>
<td>resistance.1</td>
<td>13.8M</td>
<td>32.1M</td>
<td>3</td>
<td>13.8M</td>
<td>60,391</td>
</tr>
<tr>
<td>sorter.3</td>
<td>1.3M</td>
<td>2.7M</td>
<td>1.2M</td>
<td>278</td>
<td>300</td>
</tr>
</tbody>
</table>

| L1751L1751T1     | 9.2M   | 24.5M | 3    | 3.1M    | 3501     |
| L351L351T4       | 3.8M   | 11.3M | 31   | 123,201 | 704      |
| L5L5T16          | 3.3M   | 9.8M  | 131,071 | 25 | 24       |
| Li10Lo200        | 4.0M   | 15.2M | 100  | 40,000  | 416      |
| Li50Lo40         | 4.0M   | 15.8M | 2,500 | 1,600   | 176      |
| Li200Lo10        | 4.0M   | 16.0M | 40,000| 100     | 416      |
| livej            | 4.8M   | 69.0M | 971,232 | 3.8M | 19       |
| patent           | 3.8M   | 16.5M | 3.8M | 1       | 24       |
| pokec            | 1.6M   | 30.6M | 325,893| 1.3M | 14       |
| random           | 10.0M  | 100.0M | 877  | 10.0M   | 11       |
| rmat             | 10.0M  | 100.0M | 2.2M | 7.8M    | 9        |
| ssc2             | 1.0M   | 157.9M | 31   | 1.0M    | 133      |
| wiki-links       | 5.7M   | 130.2M| 2.0M | 3.7M    | 284      |

For a direct comparison to offline algorithms, we also implemented UFSCC in the SCC environment provided by Hong et al. [HRO13]. This implementation is compared against benchmarks of Tarjan and Hong’s concurrent algorithm (both provided by Hong et al.). While Slota et al. [SRM14] improve upon Hong’s algorithm, we did not have access to an implementation. However, considering the fact that both are quadratic in the worst case, we can expect similar performance pitfalls as Hong’s algorithm for some inputs. We have not tested Ji’s algorithm [JLH18].

We chose this more tedious approach because the results from on-the-fly experiments are hard to compare directly against offline implementations. Offline implementations benefit from having an explicit graph representation and can directly index based on the state numbering, whereas on-the-fly implementations both generate successor states, involving computation, and need to hash the states. This results in a factor of \( \pm 25 \) in the number of states processed per second, for the same graphs using the sequential version of the algorithms: Too much for a meaningful comparison of speedups.

Table 2.2 summarizes graph sizes from respectively: a selection of graphs from the
2.7. Experiments

2.7.1 Experiments on model checking graphs

We use model checking problems from the BEEM database [Pel07], which consists of a set of benchmarks that closely resemble real problems in verification. From this suite, we select all inputs (62 in total) large enough for parallelization (with search spaces of at least $10^6$ states), and small enough to complete within a timeout of 100 seconds using our sequential Tarjan implementation. Though our implementation in reality generates the search space during exploration (on-the-fly), we refer to it simply as ‘the graph’.

Figure 2.12 (left) compares the runtimes for UFSCC on 64 workers against Tarjan’s sequential algorithm in a speedup graph (note the log-log scale). The x-axis represents the total time used for Tarjan’s algorithm and the y-axis represents the observed speedup for the UFSCC algorithm on the same graph instances. We observe that in most cases UFSCC performs at least $10 \times$ faster than Tarjan. The performance improvements for all model checking graphs range from 4.1 (sorter.3) to
35.5× faster (leader\textunderscore filters.7) and the geometric mean performance increase for UFSCC over Tarjan is 14.84. Compared to the sequential runtime of UFSCC, its multi-core version exhibits a geometric mean speedup of 19.53. We also witness a strong trend towards increasing speedups for larger graphs (dots to the right represent longer runtimes in Tarjan, which is proportional to the size of the search space).

Our results do not show an ideal 64× improvement over Tarjan's algorithm, which partly results from the fact that graphs cannot be split up perfectly for all workers (there are necessarily some overlaps in the concurrent graph traversal procedures). Additionally, we simply cannot expect an ideal speedup due to the intricacies of our internal structures that are combined with suboptimal concurrent interleavings in practice.

Figure 2.12 (right) compares UFSCC with Renault on model checking graphs, both using 64 workers. For a number of graphs Renault performs slightly better (up to 1.2× faster than UFSCC). These graphs all contain many small SCCs and thus communicating completed SCCs is effective. Since UFSCC applies a similar communication procedure while also maintaining the cyclic list structure, a slight performance decrease is to be expected. For most other graphs however we see that UFSCC significantly outperforms Renault. These graphs indeed contain large SCCs. In some cases Renault even performs worse than Tarjan's sequential algorithm, which we attribute to contention on the union-find structure (where multiple workers operate on a large SCC at the same time, thus all requiring the root of that SCC). On average, we observe again that UFSCC scales better for larger graphs. The geometric mean performance increase for UFSCC over Renault (considering every examined model checking graph) is 6.42.

We select 6 graphs to examine more closely. We choose the graphs to range from the best and worst results when UFSCC is compared to Tarjan and Renault. We indicated the selected graphs with red crosses (×) in Figure 2.12. Information about these graphs is provided in Table 2.2 (the first 6 graphs).

The scalability for UFSCC and Renault compared to Tarjan is depicted in Figure 2.13, and Table 2.3 quantifies the performance results for 64 workers. In Figure 2.13, while we generally see a fairly linear performance increase, we notice two peculiarities when the number of workers is increased for UFSCC. First, in some graphs the performance increase levels out (or even drops a bit) for a certain number of workers and thereafter continues in increasing performance (bakery.6, cambridge.6, lup.3). These results occur from high variances in the performance results, e.g. where UFSCC executes in 1 second for about 75% of the cases and in 4 seconds for the other 25%. We argue that this is partly due to an unfortunate random successor selection and partly due to intricacies of our internally used structure. In Section 2.8 we analyse the performance for lup.3 in more detail.
Table 2.3: Comparison of sequential on-the-fly Tarjan with concurrent UFSCC and Renault on 64 cores for a set of model checking inputs and synthetic graphs.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Execution time (s)</th>
<th>UFSCC speedup vs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tarjan</td>
<td>Renault</td>
</tr>
<tr>
<td></td>
<td>68.602</td>
<td>1.995</td>
</tr>
<tr>
<td>leader_filters.7</td>
<td>40.747</td>
<td>68.526</td>
</tr>
<tr>
<td>bakery.6</td>
<td>15.162</td>
<td>19.965</td>
</tr>
<tr>
<td>cambridge.6</td>
<td>5.645</td>
<td>7.941</td>
</tr>
<tr>
<td>lup.3</td>
<td>39.452</td>
<td>58.247</td>
</tr>
<tr>
<td>resistance.1</td>
<td>12.189</td>
<td>5.793</td>
</tr>
<tr>
<td>sorter.3</td>
<td>8.095</td>
<td>1.275</td>
</tr>
<tr>
<td>L1751L1751T1</td>
<td>15.144</td>
<td>5.127</td>
</tr>
<tr>
<td>L351L351T4</td>
<td>13.473</td>
<td>3.973</td>
</tr>
<tr>
<td>L5LST6</td>
<td>12.906</td>
<td>4.438</td>
</tr>
</tbody>
</table>

Another peculiarity is that for most graphs the performance does not increase as much for 32 workers or more (and even drops for sorter.3). We determined that this effect likely results from a high contention on the internal union-find structure; where a large number of workers try to access the same memory locations (the same reason for why Renault with 64 workers performs worse compared to Tarjan for graphs with large SCCs). In a previous design of UFSCC, which used a more drastic locking scheme, this effect was observed to be more prevalent [Blo15].

For each model checking graph, we compared the total number of visited states for UFSCC using 64 workers with the number of unique states. We observe that 0.5% (leader_filters.7) up to 128% (resistance.1) re-explorations occurred in the experiments. With re-exploration we mean the number of times that states are explored by more than one worker, which ideally should be minimized. As a geometric mean (over the averaged relative re-exploration for each graph), the total number of explored states is 21.1% more than the number of unique states, implying that on average each worker only visits \( \frac{121.1}{64} \approx 1.9\% \) of the total number of states.

### 2.7.2 Experiments on synthetic graphs

We experimented on synthetically generated graphs (equivalent to the ones used by Barnat et al. [BCvdP11]) to find out how particular aspects of a graph influence UFSCC’s scalability. The first type of graph, called \( LxLxTy \), is the Carte-
Figure 2.13: Scalability for the on-the-fly UFSCC and Renault implementations relative to Tarjan’s algorithm on a set of model checking inputs, on a log-log scale.
sian product for two cycles of \( x \) states with a binary tree of depth \( y \) (generated by taking the parallel composition of processes with said control flow graphs: \( \text{Loop}(x) \parallel \text{Loop}(x) \parallel \text{Tree}(y) \)). This graph has \( 2^{y+1} - 1 \) components of size \( x^2 \).

The second type is called \( \text{Li}_x \text{Lo}_y \) and is a parallel composition of two sequences of \( x \) states with two cycles of \( y \) states (\( \text{Line}(x) \parallel \text{Line}(x) \parallel \text{Loop}(y) \parallel \text{Loop}(y) \)). This graph has \( x^2 \) components of size \( y^2 \). Table 2.2 summarizes these graphs. We provide a comparison of Tarjan, Renault and UFSCC in Table 2.3 and show how UFSCC and Renault scale compares to Tarjan in Figure 2.14.

In Figure 2.14, we notice that the scalability of UFSCC is more consistent than the results in Figure 2.13. This likely follows from the even distribution of successors and SCCs in these graphs. While UFSCC’s speedup compared to both Tarjan and Renault are not as impressive as we see in the model checking experiments, it still outperforms Tarjan and Renault significantly (though less impressively for \( \text{L5L5T16} \) and \( \text{Li200Lo10} \), which both consist of many small SCCs). Additional experiments on inputs with increased out-degree (generated by putting more processes in parallel), showed that speedups indeed improved.

### 2.7.3 Experiments on explicit graphs

We experimented with an offline implementation of UFSCC and compare its results with (an offline implementation of) Tarjan’s and Hong’s algorithm. The explicit graphs are stored in a CSR adjacency matrix format (c.f. [HRO13]). In a CSR structure, we track an array of pointers (one for each state) and an array of all edges (ordered by their source IDs), such that every state points to their first outgoing edge. We benchmarked several real-world and generated graph instances. Information about these examined graphs can be found in Table 2.2 (the bottom seven graphs). The \text{livej}, \text{patent} and \text{pokec} graphs were obtained from the SNAP [LK14] database and represent the LiveJournal social network [BHKL06], the citation among US patents [LKF05], and the Pokèc social network [TZ12]. The graph \text{wiki-links}\footnote{Obtained from http://haskell.org/wiki/.} represents Wikipedia’s page-to-page links. The \text{random}, \text{rmat} and \text{ssca2} graphs represent random graphs with real-world characteristics and were generated from the GTGraph [BM06] suite using default parameters. We also constructed explicit graphs from the selected on-the-fly model checking and synthetic experiments (by storing all edges during an on-the-fly search in CSR format), these graphs are prefixed with “e-” in Table 2.4. We used the GreenMarl [HRO13] framework to convert the graphs to a binary format suitable for the implementation.

The results of these experiments can be found in Table 2.4. We provide the speedup
Figure 2.14: Scalability for selected synthetic graphs of UFSCC and Renault relative to Tarjan’s algorithm, on a log-log scale.
graphs for six of these graphs in Figure 2.15. Again, the results from Table 2.3 and Table 2.4 cannot be compared, since the on-the-fly experiments perform more work due to the dynamic generation of successors. We notice some interesting
Table 2.4: Comparison of sequential offline Tarjan with concurrent UFSCC and Hong on 64 cores for a set of model checking inputs, synthetic graphs and real-world graphs (all explicitly represented in CSR adjacency matrix format).

<table>
<thead>
<tr>
<th>graph</th>
<th>execution time (s)</th>
<th>UFSCC speedup vs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tarjan</td>
<td>Hong</td>
</tr>
<tr>
<td>e-leader_filters.7</td>
<td>2.771</td>
<td>1.532</td>
</tr>
<tr>
<td>e-lup.3</td>
<td>0.418</td>
<td>0.100</td>
</tr>
<tr>
<td>e-bakery.6</td>
<td>1.733</td>
<td>5.110</td>
</tr>
<tr>
<td>e-cambridge.6</td>
<td>0.277</td>
<td>0.330</td>
</tr>
<tr>
<td>e-sorter.3</td>
<td>0.091</td>
<td>3.710</td>
</tr>
<tr>
<td>e-resistance.1</td>
<td>2.040</td>
<td>5.974</td>
</tr>
<tr>
<td>e-L1751L1751T1</td>
<td>0.878</td>
<td>139.636</td>
</tr>
<tr>
<td>e-L351L351T4</td>
<td>0.247</td>
<td>30.216</td>
</tr>
<tr>
<td>e-L5L5T16</td>
<td>0.201</td>
<td>-</td>
</tr>
<tr>
<td>e-Li10Lo200</td>
<td>0.251</td>
<td>30.627</td>
</tr>
<tr>
<td>e-Li50Lo40</td>
<td>0.221</td>
<td>22.452</td>
</tr>
<tr>
<td>e-Li200Lo10</td>
<td>0.223</td>
<td>39.934</td>
</tr>
<tr>
<td>livej</td>
<td>2.963</td>
<td>0.278</td>
</tr>
<tr>
<td>patent</td>
<td>0.914</td>
<td>0.060</td>
</tr>
<tr>
<td>pokec</td>
<td>1.372</td>
<td>0.113</td>
</tr>
<tr>
<td>random</td>
<td>10.081</td>
<td>0.573</td>
</tr>
<tr>
<td>rmat</td>
<td>8.327</td>
<td>0.498</td>
</tr>
<tr>
<td>scca2</td>
<td>1.657</td>
<td>0.208</td>
</tr>
<tr>
<td>wiki-links</td>
<td>4.237</td>
<td>0.352</td>
</tr>
</tbody>
</table>

results from the offline experiments, which we summarize as follows:

- On model checking graphs, UFSCC generally performs best (with two exceptions) and Hong performs slower than Tarjan for 4 of the 6 graphs.

- On synthetic graphs, UFSCC struggles to gain performance over Tarjan but remains to perform similarly, while Hong significantly performs worse (and always crashed on L5L5T16).

- On real-world graphs, UFSCC shows increased performance compared to Tarjan, but Hong clearly outperforms UFSCC.

The synthetic graphs seem to exhibit instances of Hong’s quadratic worst-case performance. We also examined that the performance for UFSCC with 1 worker is significantly worse compared to that of Hong (up to a factor of 4), indicating that the overhead for our internal structure affects the performance on offline graphs.
2.8 Analysing bottlenecks by visualisation

In the previous section we showed that the UFSCC algorithm exhibits good scalability in general. However, for some experiments we observed only limited scalability when using a specific amount of workers. Our goal in this section is to better understand the reason for this limited scalability. To achieve this, we carefully instrumented the algorithm’s implementation with timing information, without significantly impacting the performance. By visualizing the log trace, we can interpret the data to better understand the reason for the suboptimal performance. We select the lup.3 model, since this model has the most notable scalability issues (as shown in Figure 2.13).

We take a closer look at the lup.3 model, which functions as a model for a hardware router’s lookup machine [AŘK04]. We executed our algorithm with 1, 2, 4, 6, 8, 12, 16, 24, 32, 48, and 64 workers, and measured 25 execution times. The graph itself consists of 1,948,617 states and 3,621,672 transitions, with all states belonging to a single SCC.

Figure 2.16: Execution times for UFSCC on the graph lup.3 (from [AŘK04]). The ○ and △ symbols show individual execution times for respectively the algorithm without and with logging active. The red and blue lines denote the corresponding means.
We show the results of our experiments in Figure 2.16 (the graph is cropped a bit for clarity). We can observe (from the □ symbols) that execution times are very similar for the 24-threaded experiments, but they vary greatly for the 8-threaded experiments. This effect is so significant that the mean execution time for the 8-threaded experiments is actually slower than that for the 4-threaded instances (which is of course undesired behaviour). We therefore focus on the 8-threaded execution of our algorithm.

We carefully logged the runtime information for different runs of the algorithm. We use a recently proposed visualisation technique [QBW+] (in submission) to get a better understanding of what is causing the performance differences between a fast and a slow run of the algorithm.

2.8.1 Logging information

When logging information from the algorithm there is a trade-off between the granularity of the logged events and the impact on performance. When we log in too much detail, it becomes unclear whether the performance differences originate from the algorithm itself or from the logging process.

The best middle-ground we found was to track the time (in ns), thread ID, event name, and state ID (a hash value) for each event. Here, the event name could be either the start or end point of one of the following operations:

- **Successor**: a successor state is being explored.
- **Backtrack**: the algorithm backtracks from a state.
- **Union**: two states are (globally) merged together.

Every thread tracks a local array of events, which is printed at the end of the execution to minimize logging I/O influence. Of course, these additional operations may affect the performance (also indirectly, due to e.g. additional caching). The performance impact is visible in Figure 2.16 in blue (with the △ symbols). We observe a similar performance dip as in the ‘standard’ case. We take a closer look at the best and worst performing instances of 25 executions. In the 8-threaded case this results in a log trace for an execution taking 6.1 seconds and one that takes 9.9 seconds to compute.

2.8.2 Analysis with visualisation

With a state space being of a size of almost two million, visualising the graph is practically useless. In total there were 17,095,188 events recorded for the
2.8. Analysing bottlenecks by visualisation

Figure 2.17: Example of the visualization. In our context, the red box highlights a single thread, while the blue box shows relations between four different threads.

(a) Starting the algorithm.  
(b) ‘Ideal’ working scenario.  
(c) Suboptimal behaviour.  
(d) One thread continues to work.

Figure 2.18: Screenshots of the visualisation tool from Qi et al., used on log traces generated by 8-threaded instances of the UFSCC algorithm.

8-threaded instance that performed quickest, making it also infeasible to perform inefficient computations while processing the log trace.

This visualization is designed for tracing and comparing multiple sequences in parallel. In the context of analysing our UFSCC algorithm, a sequence corresponds to a sequence of events generated by one worker, indexed by time. The visual design is illustrated in Figure 2.17 and describes the following:

- Sequences are binned by time windows in fixed size, where the time windows are shown as white boxes in the background. We use a window size of 10ms.
- Events within time windows are represented as bins (circles in Figure 2.17)
and greyscale of the filling colour of bins shows the event frequency inside (the darker the more events, i.e. the more states processed).

- Bins belonging to the same sequence are linked horizontally.
- Within the same window, similar bins from different sequences are linked by arcs (in our case, multiple workers processing the same state).
- Greyscale of the horizontal links/arcs shows the degree of similarity of linked bins (the darker the more similar, i.e. the more overlap in processed states).

When two workers operate on the same state, we argue that the performance is suboptimal as one of the threads could have worked on a different state. In an ideal case, the state space is ‘divided’ perfectly across the threads with minimal overlap between the visited states\(^5\).

**Analysis.** We focus on the 8-threaded log traces for the *fast* and *slow* executions. Figure 2.18 depicts four screenshots of different scenarios in the log trace:

- **Figure 2.18a.** At the start of the algorithm, every worker is started from the initial state, hence the overlap in states for the first few iterations. The threads slowly explore disjoint parts of the state space. Only a handful of states get processed in each interval as the algorithm is still ‘starting up’ (indicated by the white bins).

- **Figure 2.18b.** After a couple of intervals the threads work on mostly disjoint parts of the state space and up to 10,000 states per interval get processed. There is some overlap between two subsequent intervals of the same worker, which are caused by backtrack and union operations. Most of the algorithm’s execution resembles this instance. We argue that this is a practically ideal scenario.

- **Figure 2.18c.** We observe some suboptimal behaviour (in this case we used a window size of 50ms to better highlight the situation), in which two or three threads work for about 100 milliseconds on mostly the same states. This behaviour is mostly present in the *slow* execution, but also occurs sometimes in the *fast* one. Note that only 7 workers are active during most time frames.

- **Figure 2.18d.** This case is only present in the *slow* execution. We observe that after most of the workers seem to have finished exploration, a single worker remains processing remaining states in the SCC. This takes a full second of computation time and forms the major bottleneck of the algorithm.

---

\(^5\)It is actually necessary to have some overlap between the visited states to exploit our cycle detection mechanism for gaining performance. But for the most part the algorithm should strive for as little overlap as possible.
2.8.3 A closer look at the suboptimal behaviour

The major bottleneck of the algorithm is shown in Figure 2.18d, in which only a single thread keeps working at the end of the execution. We were however unable to pinpoint the exact cause for this effect. We found that the remaining worker performed many backtrack events. This could imply that the search stack for that worker was still significantly large when all other workers finished their explorations. A reason for this is that the slow worker traversed a large path before detecting a cycle (note that the entire graph consists of a single SCC). Because other workers also traverse the graph and detect cycles, the slow worker might only have to merge a few states to end up in a large partial SCC. The slow worker would then first fully explore the partial SCC (i.e. the entire state space) before it backtracks from its original path. Note that with more workers, the probability of traversing a long path, before a cycle is detected, decreases – which might explain that the performance stabilises for 16 or more threads.

We also look at the scenario from Figure 2.18c. The problem that we are seeing is that multiple threads (in this case three) are working on mostly the same states. Effectively this causes these three workers to behave as if they were a single worker, thereby hampering scalability. Something else to notice is that a single worker has practically no overlap with itself in a future window. This is quite different from the behaviour in Figure 2.18b. An overlap in states from a previous window is mostly caused by merging states; a path is explored in the first window, and the worker detects that the path forms a cycle in the succeeding window, and merges these states together.

However, for Figure 2.18c there is almost no overlap in states being processed for consecutive windows. This gives us an indication that the algorithm is not just exploring the state space, but that it is picking states from the cyclic list of Busy states. This may indicate a bottleneck of our cyclic list implementation. In future work, it may therefore be interesting to consider alternative techniques for ‘removing’ Done states from the list.

In Figure 2.19 we provide an example of this list. Once a state is processed, it gets marked as being Done (grey in the figure) and the worker continues with the next state. Once a worker finds that the next state is already Done, it will update the list pointer to the subsequent state. This process continues until the list becomes ‘empty’. Then, all states from the SCC have been explored, and the workers may backtrack from it.

Multiple workers can work simultaneously on the cyclic list. Initially, we would like every worker to work on a distinct part of the list, which we illustrate in the example from Figure 2.19a. Then, for instance worker 1 and 2 may finish exploring states a and d respectively. These states get marked as being explored
and the workers continue to the next states in the list (Figure 2.19b). Finally, worker 3 finishes exploring state f and continues to work on state b. Now, both worker 1 and 3 are working on the same state, so the workers perform duplicate work (they both explore state b), which is not erroneous but it is undesired from a performance point of view. Both workers perform duplicate work on state b, but after that state is explored, they will both move to state c and so on. This is also clearly visible in Figure 2.18c as this effect remains for multiple windows.

We make a note that our analysis should be regarded as a preliminary study on the algorithm’s performance, mainly to provide some meaningful insights. Since we explicitly focused on applying our algorithm on a single graph, and explicitly in an 8-threaded scenario (which resulted in an abnormal performance for our algorithm), our findings cannot be regarded as conclusive.

In terms of improving the algorithm, there are several options. One way is to track which states are currently being worked on. Once a worker observes this, it should attempt to move to the next state. The state should not be avoided completely, since a large part of the state space may be reached from the ignored state. Another option is to let the different workers skip a few states (e.g. with the number of steps based on their worker ID) in the cyclic list to avoid working on the same state as another worker. However, this may lead to a significant overhead on merely picking the next state. Future work will have to show which solution works best in practice.
2.9 Conclusion

We presented a new quasi-linear multi-core on-the-fly SCC algorithm. The algorithm communicates partial SCCs using a new *iterable* union-find structure. Its internal cyclic linked list allows for concurrent iteration and removal of nodes – enabling multiple workers to aid each other in decomposing an SCC. Experiments demonstrated a significant speedup ($10\times$ to $30\times$ faster for 64 workers) over the current best known approaches, on a variety of graph instances. Unlike previous work, we retain scalability on graphs containing a large SCC.

We also looked into a single experiment to investigate bottlenecks. Due to the intricate structure of the algorithm, it remains difficult to make concrete statements on what exactly affects the performance. However, by means of visualising the execution data we found promising directions to study in the future.

In Chapter 3 we show how the algorithm can be applied in the context of model checking. We show that it can be extended to check for multiple acceptance conditions.

2.10 Future work

While we have shown good performance for the SCC decomposition algorithm, there are still some interesting directions for future work.

**Performance improvement.** There are several ways to (attempt to) improve the performance of the algorithm. A good starting point would be to continue our preliminary study on a ‘bad performing’ experiment. One could experiment with e.g. the following:

- Variations to the cyclic list, e.g. using a doubly-linked list or a skip list. A goal is to prevent multiple workers from visiting the same states in the list and spread out the workload as best as possible (without introducing a significant performance penalty).

- One could also experiment with lightweight thread spawning to further parallelize the algorithm. For instance, instead of uniting two partial SCCs together, a worker could first check if the to-be-united SCC completes the cycle, and then spawn a thread that performs the `Unite` procedure. The worker could continue searching the state space in the meantime.
• Especially for explicit graphs, we may be able to combine the UFSCC algorithm with existing techniques to prune the search space, such as the trimming procedures in recent algorithms [HRO13, SRM14, JLH18].

Theoretical contribution. It may be interesting to investigate parallel DFS from a theoretical point of view. While the result from Reif [Rei85] states that ordered DFS is $\mathcal{P}$-complete, we have shown that strictly ordered DFS is not necessary for decomposing SCCs. We could investigate whether (a variant of) our algorithm is scalable in theory. With the iterable union-find in mind, we could also investigate if this also holds for different graph search algorithms.
Chapter 3

Multi-Core LTL Model Checking for Omega Automata

In this chapter we focus on the automata-theoretic approach to Linear-time Temporal Logic (LTL) model checking. Given a model of a system and a property (specified in LTL), a model checker automatically checks whether the property is maintained in the model. The LTL model checking problem can be reduced to the detection of a so-called accepting cycle, which can be solved using an SCC decomposition algorithm. In this chapter, we extend our multi-core SCC decomposition algorithm from Chapter 2 in various ways to perform model checking, and show that we outperform state-of-the-art techniques.

A synchronized product is created for the model and an automaton that represents the negated LTL formula, which is checked for emptiness. Traditionally, a State-based Büchi Automaton (SBA) is used for representing the formula, which represents the ‘incorrectness’ of the formula as accepting states. More general automata formalisms exist that represent the acceptance condition on transitions, require multiple acceptance conditions to be met, or prevent accepting cycles to contain certain states or transitions. Such more general automata may be able to represent a formula using fewer states, or guarantee that the resulting automaton is deterministic. However, checking for emptiness becomes more complex for these automata, which raises the question whether it is more efficient to perform an emptiness check on e.g. an SBA or on a more general automata.

The main contents of this chapter have been formed from the following papers.

- In [BvdP16] we applied the SCC decomposition algorithm from Chapter 2 for checking SBA s and Transition-based Generalized Büchi Automata (TGBA s).
We showed that our algorithm is able to outperform the state-of-the-art (especially for larger models) and studied how various graph characteristics relate to the results.

- In [BDLvdP17] and its extended journal version [BDLvdP19] we further extended the algorithm to check for emptiness on Transition-based Generalized-Rabin Automata (TGRAs). We were however not able to improve on the TGBA checking performance using our TGRA checking algorithm. We also introduced Fin-less automata, which can be derived from TGRAs, and showed that we can check Fin-less automata equally fast or faster than TGBAs.

3.1 Introduction

Model Checking. Model checking is a way to ensure that a system modelled by a Kripke Structure $K$ satisfies some behavioural properties. There exist many formalisms that can be used for expressing such behavioural properties. For the remainder of this chapter, we focus on Linear-time Temporal Logic (LTL) [Pnu77] to express properties. An LTL formula specifies which infinite-length runs of the model should be accepted or rejected. We briefly introduce LTL in Section 3.2 and refer to Clarke et al. [CHVB18] for an extensive overview on LTL and model checking.

In the automata theoretic approach to LTL model checking [Var95], the formula $\varphi$ (specified in LTL) is first transformed into a Büchi automaton $B_{\neg \varphi}$ (a Büchi automaton only accepts runs that visit an accepting state infinitely often), capturing forbidden behaviours. We then construct a synchronous product of the Büchi automaton and the model $K$ (i.e. $K \otimes B_{\neg \varphi}$), and the procedure then amounts to testing whether the language of this synchronized product is empty: $L(K \otimes B_{\neg \varphi}) = \emptyset$. If the language of the product is non-empty, it means that there exists a counterexample: an execution of $K$ that does not satisfy $\varphi$.

When performed explicitly (i.e. not using any kind of symbolic representation of those automata) the procedure is limited by the well-known state space explosion problem, where the product automaton $K \otimes B_{\neg \varphi}$ becomes too large to handle. We will focus on the emptiness check procedure, i.e. the algorithm that takes an automaton $A = K \otimes B_{\neg \varphi}$ as input, and decides if its language $L(A)$ is empty. We abstract away the fact that in a model checker $A$ is a product, but we have to account for the fact that $A$ can be quite large. Generally, $K$ is of large scale (and often is a parallel composition of components). While the formula, $\varphi$, is usually small, $B_{\varphi}$ can be exponential in the size of $\varphi$. 
On-the-fly model checking mitigates the state space memory constraints by only storing the states (not the transitions) encountered during the emptiness check. The search procedure is launched from an initial state. Reachable states are computed on demand via a successor function, and in case a counterexample is detected the search may end well before the entire state space is explored (see also Section 2.2.1). A consequence of on-the-fly model checking is that this puts a restriction on the algorithms that may be employed, i.e. we cannot efficiently enumerate over the states and edges.

With current hardware systems one can further improve the model checking performance by using multiple cores. This way, the time to model check can be significantly reduced; related work shows that even though the problem is difficult to parallelize, in practice an almost linear improvement with respect to the number of cores can be obtained [BvMv04, BBBv12, Hol12, ELPvdP12, RDKP17].

Beyond standard Büchi automata. Another way to reduce the size of the product automaton is to keep the sizes of the system’s state space and the negated property automaton as small as possible. In particular, smaller property automata can be obtained by using more complex acceptance conditions.

LTL formulas can be translated to ω-automata, i.e. finitely sized automata that accept runs of infinite length. The automata-theoretic approach to LTL model checking is often performed using State-based Büchi Automata (SBAs). An SBA is the simplest form of an ω-automaton and accepts infinite runs that visit an accepting state infinitely often (e.g. see $A_1$ from Figure 3.1 and Example 3.1). An emptiness checking algorithm would have to search for a cycle in the product automaton that contains an accepting state. There are a number of different (sequential and multi-core) techniques that can be used for detecting accepting cycles on SBAs, e.g. [CVWY92, ELPvdP12, Cou99, BBR12].

Instead of representing the acceptance condition on states, it allows for more modelling freedom to have accepting transitions, i.e. Transition-based Büchi Automata (TBAs). An SBA can be converted into in TBA with at most the same number of states (by marking every outgoing transition of an accepting state as an accepting transition). By also allowing for multiple acceptance sets we form Transition-based Generalized Büchi Automata (TGBAs), which can be constructed from using an LTL to ω-automata translations tool (e.g. [DLF+16, KMSZ18, BBKS13]). Here, an accepting cycle must contain a transition from every acceptance set (e.g. see $A_2$ from Figure 3.1). A TGBA allows for more freedom than a TBA, which may therefore reduce the number of states in the automaton. The emptiness checking procedure does become more complex as a consequence, though SCC-based techniques can still be applied [CDLP05, RDKP17] (but techniques based on nested depth-first search [Cou99] are generally limited to SBAs).
Figure 3.1: Three automata that all represent the LTL formula $GFa \land GFb \land FGc$. Here, $(A_1)$ is a non-deterministic SBA, $(A_2)$ is a non-deterministic TGBA, and $(A_3)$ is a deterministic TGRA. With $\bar{a}$ we denote that atomic proposition $a$ does not hold and $\top$ denotes the transition that may always be taken.

A Transition-based Generalized Rabin Automaton (TGRA) extends a TGBA in two ways: (1) the acceptance condition is given by generalized Rabin pairs, which extend TGBAs by also containing transition sets that may only be visited finitely often, and (2) a TGRA allows for a disjunction of multiple generalized Rabin pairs as its acceptance condition (see $A_3$ in Figure 3.1 and Example 3.1). Again, this more complex scheme for acceptance may be able to reduce the number of states in the automaton. A TGRA can also be constructed to always be deterministic (as opposed to SBAs and TGBAs). This latter property may further aid in reducing the size of the product automaton. However, model checking TGRAs has only been studied to a limited extent; a non-generalized variant is studied by Wijs [Wij16] and Chatterjee et al. [CGK13] consider TGRAs in the context of probabilistic model checking. It remains unclear whether the state space reduction that TGRAs may provide outweighs the added complexity of checking TGRAs, when compared to checking TGBAs or SBAs. We will study this problem by designing a model checking algorithm for TGRAs, and evaluate its performance with the current state-of-the-art on sets of available and generated benchmark problems.

Example 3.1: An example SBA, TGBA, and TGRA

Consider the example from Figure 3.1, which depicts an SBA, TGBA, and TGRA for the same formula. The formula $GFa \land GFb \land FGc$ implies that an accepting run visits $a$ and $b$ infinitely often, and from some point onwards $c$ always holds.

In $A_1$ we see an SBA with the accepting state (1). First, a finite number of arbitrary actions may be taken in the self-loop of state (0), e.g. we allow for a transition where $c$ does not hold. Then, at some point we must take the transition to state (1) and ensure that this state is visited infinitely often.
Note that from this point onwards we always have that $c$ holds. Since $a$ and $b$ do not always have to hold we may traverse to states (2) and (3), given that state (1) is visited infinitely often.

In $\mathcal{A}_2$ we see how a TGBA functions. Here we have the same procedure for the initial state as in $\mathcal{A}_1$. Then, we must ensure that we infinitely often visit acceptance marks $\varnothing$ and $\mathbb{1}$. Again, $c$ must always hold in state (1) and e.g. we may choose to alternate between $abc$ and $\bar{a}bc$ to visit both acceptance marks infinitely often.

In $\mathcal{A}_3$ we see a TGRA, with a single generalized Rabin pair, $\text{Fin}(\varnothing) \land \text{Inf}(\varnothing) \land \text{Inf}(\mathbb{1})$. This means that the acceptance marks $\varnothing$ and $\mathbb{1}$ must be visited infinitely often (as in $\mathcal{A}_2$), but the mark $\mathbb{2}$ may only be visited finitely many times. In other words, after a finite number of steps, $c$ must hold indefinitely. Note that $\mathcal{A}_3$ is deterministic and that the number of states is decreased from $\mathcal{A}_1$ to $\mathcal{A}_2$ and from $\mathcal{A}_2$ to $\mathcal{A}_3$, illustrating the benefits of using a more complex acceptance condition. However, the size of a minimal TGRA is not always smaller than a TGBA or a SBA and may thus be of equal size.

We will also introduce a new type of automata in Section 3.4, called \textit{Fin-less}, which can be constructed from a TGRA. Due to its construction, it is larger than a TGRA. However, its acceptance condition becomes significantly more efficient to check as the acceptance is formed by a disjunction of multiple TGBA acceptance instances.

Contributions. In this chapter we compare different types of $\omega$-automata from a model checking performance perspective. In particular, we focus on the time to model check in a multi-core environment and investigate the reason why a particular approach may be beneficial to another one. We make the following contributions.

- We extend the SCC decomposition from Chapter 2 in order to check for SBA and TGBA acceptance.
- We further extend the SCC decomposition procedure to check TGRA, by ‘avoiding’ transitions that may only be visited finitely often.
- We introduce a new type of acceptance, called \textit{Fin-less}, which is a disjunction of TGBAs and can be constructed from a TGRA. We show how the TGBA emptiness check can be extended to support Fin-less acceptance.
- We empirically evaluate our algorithms and related work on the various $\omega$-automata using benchmark models and properties from the 2015
Model Checking Contest [KGH15] and the BEEM database [Pel07]. We also use different tools (Spot [DLF16], Rabinizer 3 [KK14, EKS16], LTL3DRA [BBKS13], and LTL3TELA) for the generation of TGRAs from formulas to study the differences between the automata (and cross-products) that are produced from the various tools.

- We investigate how structural properties of the $\omega$-automaton and the resulting product graph influence the model checking performance.

3.2 Preliminaries

Linear-time Temporal Logic. Given a set of atomic propositions an assignment maps these atomic propositions to either true or false. For instance, an assignment may map atomic propositions $a$ and $b$ to true and $c$ to false, represented by $abc$. A Linear-time Temporal Logic (LTL) formula is used to specify which infinite-length sequences of such assignments should be accepted and rejected. An LTL formula is constructed from the boolean values true and false, the atomic propositions, the boolean operators $\neg$, $\land$, $\lor$, and the temporal operators $X$, $F$, $G$, and $U$. For the temporal operators we have the following: $X\varphi$ means that $\varphi$ holds at the next state, $F\varphi$ means that $\varphi$ holds at some state in the future, $G\varphi$ states that $\varphi$ holds at all states in the future, and $\varphi_1 U \varphi_2$ means that $\varphi_2$ holds at some state in the future and $\varphi_1$ holds for all states until $\varphi_2$ holds. For a detailed overview on the precise semantics, we would like to refer the reader to e.g. Clarke et al. [CHVB18].

Given a model $A_m$ (modelled in a Kripke Structure), and an LTL formula $\varphi$, with model checking we can verify whether $\varphi$ holds on $A_m$. If we were to check the property $\varphi$ directly on $A_m$, we would have to check that $\varphi$ holds for all infinite-length runs in $A_m$. An alternative approach is to check if there is an infinite run in $A_m$ for which the negation of $\varphi$ holds. If this is the case, then we found a counterexample to the original property. Otherwise, in case the model does not allow for an infinite run where $\neg \varphi$ holds, we verified that the original property holds in $A_m$.

Hence, we first construct an $\omega$-automaton from the negation of $\varphi$, i.e. $A_{\neg \varphi}$, which accepts infinite-length runs that falsify the original property $\varphi$. A synchronized product is then constructed from $A_m$ and $A_{\neg \varphi}$, i.e. $A = A_m \otimes A_{\neg \varphi}$ (see e.g. Clarke et al. [CHVB18] for details). Then, $A$ is checked for emptiness by verifying that there is no accepting run. If there is an accepting run in $A$, then this forms a counterexample that violates the original property $\varphi$. 
On-the-fly model checking. In Section 2.2.1 we discussed various types of graphs and levels of on-the-flyness. For model checking, it is generally favourable to also compute the state space of the synchronized product on-the-fly, i.e. restricting the search algorithms to be run on Forward-only implicit graphs\(^1\).

Considering the different levels of on-the-flyness is also important for a model checking algorithm. From the three levels of on-the-flyness \([BBR12]\), it is particularly valuable if an algorithm is of type \(M_3\), which states that the algorithm is guaranteed to terminate early (before exploring the entire state space) if the automaton contains an accepting cycle. This means that once an accepting cycle is found, the algorithm does not have to continue exploring the remaining states. This may prevent visiting a large part of the state space and an \(M_3\) algorithm can thereby significantly speed up model checking compared to algorithms that do not fit in this class, in the presence of counterexamples. The SCC decomposition algorithm from Chapter 2 is sufficiently on-the-fly to be extended to an \(M_3\) model checking algorithm, as we will see in our upcoming algorithm.

\(\omega\)-automata. We define \(\omega\)-automata using acceptance conditions that are positive Boolean formulas over the terms \(\text{Fin}(T)\) (the transitions in \(T\) should be seen finitely often) and \(\text{Inf}(T)\) (infinitely often). This convention, inspired from the HOA format \([BBD^{+}15]\), allows us to express all traditional acceptance conditions and is similar to the formalism used by Emerson and Lei \([EL85]\) using state-based acceptance.

\begin{definition}
\textbf{Definition 3.1: TELA}

A Transition-based Emerson-Lei Automaton (TELA) is a tuple \(A = (\Sigma, Q, q_0, \delta, \text{acc})\), such that:

- \(\Sigma\) is an alphabet that assigns boolean values to atomic propositions,
- \(Q\) is a finite set of states,
- \(q_0 \in Q\) is the initial state,
- \(\delta \subseteq Q \times \Sigma \times Q\) is a transition relation, and
- \(\text{acc}\) is a positive Boolean function over terms over the terms \(\text{Fin}(T)\) and \(\text{Inf}(T)\) for any subset \(T \subseteq \delta\).

For a transition \(t \in \delta\) we note \(t^s\) its source, \(t^\ell\) its label, and \(t^d\) its destination:
\(t = (t^s, t^\ell, t^d)\).
\end{definition}

\(^1\)It may not always be possible or efficient to compute predecessors, as certain actions in the model may be irreversible. Therefore, the convention is to rely on forward-only implicit graphs, as opposed to e.g. the implicit and explicit graphs presented in Section 2.2.1.
Table 3.1: Acceptance condition formulas corresponding to classical names. $F_i$ and $I_i$ denote sets of transitions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B) Büchi</td>
<td>$\text{Inf}(I_1)$</td>
</tr>
<tr>
<td>(GB) gen.-Büchi</td>
<td>$\bigwedge_i \text{Inf}(I_i)$</td>
</tr>
<tr>
<td>(C) co-Büchi</td>
<td>$\text{Fin}(F_1)$</td>
</tr>
<tr>
<td>(R) Rabin</td>
<td>$\bigvee_i \text{Fin}(F_i) \land \text{Inf}(I_i)$</td>
</tr>
<tr>
<td>(GR) gen.-Rabin</td>
<td>$\bigvee_i \text{Fin}(F_i) \land \text{Inf}(I_1^i) \land \text{Inf}(I_2^i) \land \ldots \land \text{Inf}(I_n^i)$</td>
</tr>
<tr>
<td>(FL) Fin-less</td>
<td>$\bigvee_i \text{Inf}(I_1^i) \land \text{Inf}(I_2^i) \land \ldots \land \text{Inf}(I_p^i)$</td>
</tr>
<tr>
<td>(EL) Emerson-Lei</td>
<td>any positive formula of $\text{Fin}(F_i)$ and $\text{Inf}(I_i)$</td>
</tr>
</tbody>
</table>

**Definition 3.2: Acceptance of a TELA**

*Runs* of $\mathcal{A}$ are infinite sequences of consecutive transitions:

$$\text{Runs}(\mathcal{A}) = \{ \rho \in \delta^\omega \mid \rho_0^\theta = q_0 \land \forall i \geq 0 : \rho_i^d = \rho_{i+1}^s \}$$

The *acceptance* of a run $\rho$ is defined by evaluating the acceptance condition $\text{acc}$ over $\rho$ such that:

- $\text{Fin}(T)$ is true iff all the transitions in $T$ occur finitely often in $\rho$.
- $\text{Inf}(T)$ is true iff some transitions in $T$ occur infinitely often in $\rho$.

Let $\ell^\rho \in \Sigma^\omega$ be the word recognized by a run $\rho \in \text{Runs}(\mathcal{A})$ defined by $\ell_i^\rho = \rho_i^s$ for all $i \geq 0$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all words $\ell^\rho$ recognized by some accepting run $\rho$.

Some shape of acceptance conditions $\text{acc}$ are given names, as shown in Table 3.1. We use the abbreviation $TXA$, were $X$ is any value of the first column of Table 3.1, to denote a TELA whose acceptance condition has the shape given in the last column.

For instance, a Transition-based Generalized Büchi Automaton (TGBA) is a TELA where $\text{acc} = \text{Inf}(T_1) \land \text{Inf}(T_2) \land \ldots \land \text{Inf}(T_n)$ for some $n$, meaning that any accepting run has to visit infinitely often one transition from each set $T_i$.

As an example, automaton $A_1$ from Figure 3.2 represents a TGBA for the formula $\text{GF}a \land \text{GF}b$. Here, transitions are labelled by all possible assignments of $a$ and $b$, i.e. elements of $\Sigma = \{\bar{a}b, ab, \bar{a}b, ab\}$ ($\bar{a}$ denotes the negation of $a$), and transitions are also marked using $\circ$ and $\bullet$ to denote their membership to the sets used in the acceptance condition. A run of $A_1$ is accepted if it visits both acceptance marks $\circ$ and $\bullet$ infinitely often.

A Transition-based Generalized Rabin Automaton (TGRA) is a TELA where $\text{acc}$. 

3.2. Preliminaries

has the form $\bigvee_{i=1}^{n}(\text{Fin}(F_i) \land \text{Inf}(I^1_i) \land \text{Inf}(I^2_i) \land \ldots \land \text{Inf}(I^p_i))$ for some values of $n$, and $p_1, p_2, \ldots, p_n$. This is a generalization of Rabin acceptance in the sense that in Rabin acceptance $p_i = 1$ for all $i$. Each conjunctive clause of the form $\text{Fin}(F_i) \land \text{Inf}(I^1_i) \land \text{Inf}(I^2_i) \land \ldots \land \text{Inf}(I^p_i)$ is called a transition-based generalized Rabin pair (TGRP). A transition-based co-Büchi automaton is a TGRA with $n = 1$ and $p_1 = 0$; co-Büchi acceptance consists of a single clause $\text{Fin}(F_1)$.

The two automata $A_2$ and $A_3$ from Figure 3.2 represent the formula $\text{FG}a$ using the alphabet $\Sigma = \{\bar{a}, a\}$ and different acceptance conditions: $A_2$ is a Transition-based Büchi Automaton while $A_3$ is a Transition-based co-Büchi Automaton. Both automata are minimal in their number of states and illustrate that allowing Fin acceptance can reduce the size of an automaton. Moreover, $A_3$ is a deterministic automaton, whereas no equivalent deterministic SBA exists.

Figure 3.3 depicts a deterministic TGRA ($A_4$) and a non-deterministic TGBA ($A_5$), both representing the property $\text{FG}((\text{Fa}) \cup \text{b})$. $A_4$ is accepting if either is visited infinitely often without visiting $\circ$ infinitely often, or if is visited finitely often and both $\bullet$ and $\circ$ are visited infinitely often. Only one of the two TGRPs has to be satisfied. In this case, by comparing $A_4$ and $A_5$ we can (again) observe that Fin acceptance aids in reducing the size of the automaton.

Since generalized Rabin acceptance is just a disjunction of TGRPs, it can serve as a normal form for any acceptance condition. Any acceptance condition can be converted into generalized Rabin acceptance by distributing $\land$ over $\lor$ to obtain a disjunctive normal form, and then replacing any conjunctive clause of the form $\text{Fin}(F^1) \land \text{Fin}(F^2) \land \ldots \land \text{Fin}(F^n) \land \text{Inf}(I^1) \land \text{Inf}(I^2) \land \ldots \land \text{Inf}(I^p)$ by the TGRP $\text{Fin}(\bigcup_{i=1}^{n} F^i) \land \text{Inf}(I^1) \land \text{Inf}(I^2) \land \ldots \land \text{Inf}(I^p)$. This conversion can be done without changing the transition structure of the automaton; but the downside is that it may introduce an exponential number of TGRPs.
**Figure 3.3**: Two automata recognizing $FG((Fa)Ub)$. $(A_4)$ is a non-deterministic TGRA with two pairs and $(A_5)$ is a non-deterministic TGBA.

**Strong connectivity.** We defined strongly connected components (SCCs) in Section 2.2 on sets of states. We redefine an SCC and a fitting SCC to also include transitions, as follows.

**Definition 3.3: Fitting Strongly Connected Component (FSCC)**

Given a TELA of the form $A = (\Sigma, Q, q_0, \delta, acc)$, a *fitting Strongly Connected Component* (FSCC or fitting SCC) is a pair $C := (C_Q, C_\delta) \in 2^Q \times 2^\delta$ with $C_Q \neq \emptyset$ such that any ordered pair of states of $C_Q$ can be connected by a sequence of consecutive transitions from $C_\delta$.

**Definition 3.4: Strongly Connected Component (SCC)**

Given a TELA of the form $A = (\Sigma, Q, q_0, \delta, acc)$, a *strongly connected component* (SCC) is a *maximal* FSCC $C$ with respect to inclusion. Thus the case where both $C_Q$ and $C_\delta$ cannot be extended without losing strong connectivity. An SCC is called *trivial* if $C_\delta = \emptyset$, and hence $C_Q$ consists of a single state.

In a TGBA whose acceptance condition has $n$ acceptance sets of the form $Inf(T_1) \land \ldots \land Inf(T_n)$, finding an accepting run boils down to searching for a trace from the initial state to a reachable fitting SCC $C$ for which $\forall 1 \leq i \leq n : T_i \cap C_\delta \neq \emptyset$ holds, i.e. a fitting SCC that intersects each acceptance set.

In a TGRA, an accepting run has to satisfy one TGRP. A TGRP $\text{Fin}(F) \land \text{Inf}(I^1) \land \ldots \land \text{Inf}(I^p)$ has an accepting run if there is a trace from the initial state to a reachable fitting SCC $C$ with $F \cap C_\delta = \emptyset$ and $I_i \cap C_\delta \neq \emptyset$ for all $1 \leq i \leq p$. In other words a fitting SCC that contains a transition from every Inf set and no transition from the Fin set.

Note that in the case of a TGBA, it is always valid to replace the search for a fitting SCC intersecting all acceptance sets by the search for a *maximal* SCC intersecting
these sets. However this cannot be done when the acceptance condition uses \textit{Fin} sets. For instance consider the automaton $A_4$ in Figure 3.3 checked against the TGRP \textit{Fin}(0) $\land$ \textit{Inf}(1): the automaton has a unique maximal SCC, consisting of both states and every transition, which does not satisfy $0 \cap C_\delta = \emptyset \land 1 \cap C_\delta \neq \emptyset$. However, those constraints hold on the fitting SCC that consists of state 0 and the loop above it. For this reason our TGRA algorithm will build fitting SCCs that do not include transitions labelled by \textit{Fin} sets.

\textbf{LTL classification.} One can consider classifying LTL formulas by using the temporal hierarchy of Manna and Pnueli [MP90]. For half of these classes (called safety, guarantee, and obligation) there is no benefit in using more complex acceptance conditions compared to the SBA acceptance condition, as it can be reduced to a simple DFS [vP03b]. For the remaining classes (called recurrence, persistence, and reactivity), more complex acceptance conditions such as TGBA and TGRA acceptance may result in a smaller or more deterministic automaton. We omit details for these classes of LTL, but for more information on LTL classification, we would like to refer the reader to the Habilitation thesis by Duret-Lutz [DL17].

3.3 Related work

There has been a lot of work on emptiness checking algorithms over the years. For an overview on model checking in general, we would like to refer the reader to the recent handbook by Clarke et al. [CHVB18]. For a more comprehensive study on parallel algorithms for model checking, there is also a good overview by Barnat et al. [BBD+18].

3.3.1 Nested depth-first search

The \textit{Nested Depth-First Search} (NDFS) algorithm was originally proposed by Courcoubetis et al. [CVWY92] and operates by performing a ‘blue’ DFS to detect accepting states (in an SBA setting) and launches a ‘red’ DFS from each accepting state to check whether a cycle can be formed. Since its inception, several improvements have been made to e.g. faster detect non-emptiness, reduce the size of accepting runs, or reduce the memory footprint [HPY96, GMZ04, ELL01, SE05]. There has been some work on extending NDFS to be applicable on generalized Büchi automata [Tau06], but since its complexity depends on the number of acceptance marks, it is more beneficial to use an SCC-based solution.
In a multi-core setting, several algorithms have been proposed that scale on multiple threads in practice [ELPvdP12, EPY11, LvDP11, LLvdP11]. These techniques are based on swarm verification [HJG11]. The idea is that all workers start from the initial state, but the list of successor states is permuted for each worker. This way, distinct workers will explore different parts of the state space with a high probability. Since the CNDFS algorithm from Evangelista et al. [ELPvdP12] outperforms the other techniques, we consider this technique to be the state-of-the-art for NDFS-based model checking and use it to benchmark our algorithms.

### 3.3.2 BFS-based approaches

An alternative direction is to make use of BFS for model checking, which, compared to DFS-based solutions, is highly scalable and does not rely on randomized search algorithms. However, in order to check for accepting cycles, existing approaches (e.g. [BBBv12, BvMv04]) lose the on-the-fly property to a certain extent. One technique is the One-Way-Catch-Them-Young (OWCTY) algorithm [vP03a], which iteratively removes states without predecessors as they cannot be part of an accepting cycle. A counterexample is implicitly detected if there is an accepting state remaining in the remaining SCCs. The Maximal-Accepting-Predecessor (MAP) algorithm [BvMv04] efficiently performs forward reachability for all states. When an accepting state is reachable by itself, a counterexample is detected. An interesting result is that both techniques can be combined to further improve performance [BBR12].

### 3.3.3 SCC-based algorithms

An SCC-based algorithm detects SCCs and checks whether the acceptance criteria is met in these components. Tarjan’s algorithm [Tar72] is extended for tracking accepting conditions and improved to more quickly report accepting cycles [Cou99, GV04]. Notably, SCC-based techniques can be extended to support generalized Büchi automata [CDLP05, SE05], by tracking a set of acceptance marks in each partial SCC.

As already discussed in Chapter 2, there are several multi-core SCC-based algorithms. In this setting, we restrict to on-the-fly algorithms that operate on forward-only implicit graphs. The algorithm by Lowe [Low16] is based on spawning multiple synchronized instances of Tarjan’s algorithm. Here, each state may only be visited by one worker and a work-stealing-like procedure is used to handle conflicts that arise. Experimental evaluation shows that Lowe’s algorithm performs well if the graph contains many small SCCs, but this seems to deteriorate quickly when the SCC sizes grow. Lowe’s algorithm has a quadratic worst-case
3.3. Related work

complexity. The algorithm by Renault et al. [RDKP15, RDKP17] is based on swarm verification and has a quasi-linear complexity. Here, completely explored SCCs are shared globally and avoided by other workers. This approach also does not communicate partial SCCs, but acceptance marks are shared globally whenever a cycle is detected. Therefore, in the presence of counterexamples, Renault’s algorithm may be able to quickly detect them, even for large SCCs. Our algorithm from [BLvdP16] improves on both algorithms by also communicating partial SCCs. We show that this is especially favourable when the automaton does not contain a counterexample, because then the entire state space has to be explored.

3.3.4 Rabin automata

For probabilistic model checking, working with deterministic automata is important, as otherwise the resulting product automaton might not be a Markov chain [BK08]. Since it is well known that not all SBAs can be determinized, probabilistic model checkers use Rabin automata instead. More recently, order-of-magnitude speedups were reported when performing probabilistic model checking using a generalized acceptance condition, i.e. TGRAs [KE12, CGK13]. Also, there has been a lot of interest into building tools such as Spot [DLF+16], LTL3DRA [BBKS13], and Rabinizer 3&4 [KK14, EKS16, KMSZ18] for translating LTL formulas into small deterministic TGRAs and related automata. While a smaller automaton or even a deterministic one does not guarantee a smaller synchronized product, in practice this seems to be a good approach [BDKS14]. For our TGRA checking algorithm we perform experiments using various TGRA generators and show their impact on the model checking performance.

Chatterjee et al. [CGK13] present an algorithm to check deterministic TGRA conditions in the context of (offline) probabilistic model checking. The idea is to consider each generalized Rabin pair \((F_i, \{I_{ij}^1, \ldots, I_{ij}^{l_i}\})\) separately and for each pair: (1) remove the set of states \(F_i\) from the state space, (2) Compute the maximal end-component (MEC) decomposition, and (3) check which MECs have a non-empty intersection with every \(I_{ij}^j\), for \(j = 1, \ldots, l_i\). These sets are then used for computing maximal probabilities. The paper reports significant improvements over checking a non-generalized variant of deterministic Rabin automata. They also present improvements for computing a winning strategy in \(\text{LTL}(F, G)\) games by using a fixpoint algorithm for generalized Rabin pairs. Our algorithm is different in that it operates on-the-fly and in a multi-core setting.

Wijs [Wij16] recently presented an on-the-fly GPU algorithm for checking LTL properties for non-generalized deterministic Rabin automata. Here, the choice for (deterministic) Rabin automata, instead of non-deterministic Büchi automata, is motivated by the observations that it can speed up the successor construction.
and that it can reduce the state space of the cross-product. In that paper, a BFS-based search is used, in particular a variation on the heuristic piggybacking search [Hol12, FH14]. This approach is incomplete due to situations referred to as shadowing and blocking, but these cases can be detected and resolved with a depth-bounded DFS. Our approach differs in that we allow (generalized) TGRAs and do not require repair procedures.

3.3.5 Checking different automata

Livelock detection algorithms such as the DFS$_{fifo}$ algorithm [FS09, LF13] search the state space for cycles that avoid “progress transitions”. If we label all progress transitions with $\bullet$, such a search amounts to the emptiness check of an automaton with acceptance $\text{Fin}(\bullet)$, i.e. a transition-based co-Büchi automaton. DFS$_{fifo}$ detects non-progress cycles by performing a DFS that is restricted to non-progress transitions, but that remembers the set $F$ of all states that are the destination of an ignored progress transition. Once the “restricted” DFS terminates, it is started again from one of the $F$ sets. This approach is similar to our upcoming algorithm for TGRAs, which makes sense as co-Büchi acceptance is just a generalized Rabin pair without any Inf set. In our case, the DFS has to track (fitting) strongly connected components simply to ensure we visit each Fin set infinitely often.

Emerson and Lei [EL85] show that the emptiness check of an $\omega$-automaton with arbitrary acceptance condition is NP-complete. They also present a polynomial algorithm for the case where the acceptance condition is provided as a disjunction of Streett acceptance conditions. Streett acceptance is the negation of Rabin acceptance, a conjunction of $\text{Fin}(I) \lor \text{Inf}(F)$ instances (or equivalent, $\text{Inf}(I) \Rightarrow \text{Inf}(F)$), and Streett acceptance closely relates to fairness checking.

Duret-Lutz et al. [DPC09] present a sequential algorithm for checking Streett objectives by performing an SCC decomposition and tracking thresholds to prevent ‘rejecting’ cycles from occurring in the SCCs. In a multi-core setting, the algorithm by Liu et al. [LSD09] performs an initial SCC decomposition and for every SCC a new instance is launched in parallel that ignores certain transitions. The algorithm is later improved to employ Lowe’s algorithm for the SCC decomposition [WXG+16].
3.4 Fin-less automata

At the cost of introducing non-determinism, any generalized Rabin automaton can be converted into what we have called Fin-less automaton\(^2\) (TFLA) in Table 3.1. We present a procedure to obtain a TFLA from a TGRA in Proposition 3.1.

Proposition 3.1: Fin-removal

Given a TGRA \(A = (\Sigma, Q, q_0, \delta, \bigvee_{i=1}^{n} \text{Fin}(F_i) \land \text{Inf}(I_1^i) \land \ldots \land \text{Inf}(I_p^i))\), the TFLA \(B = (\Sigma, Q', q_0, \delta', \bigvee_{i=1}^{n} \text{Inf}(J_1^i) \land \text{Inf}(J_2^i) \land \ldots \land \text{Inf}(J_p^i))\)

where:

- \(Q' = Q \cup Q \times \{1, 2, \ldots, n\}\)
- \(\delta' = \delta \cup \{(t^s, t^f, (t^d, i)) \mid i \in \{1, 2, \ldots, n\} \land t \in (\delta \setminus F_i)\} \cup \{(t^s, t^f, (t^d, i)) \mid i \in \{1, 2, \ldots, n\} \land t \in (\delta \setminus F_i)\}\)
- \(J_k^i = \{(t^s, t^f, (t^d, i)) \in \delta' \mid t \in (I_k^i \setminus F_i)\} \text{ for } i \in \{1, 2, \ldots, n\} \text{ and } j \in \{1, 2, \ldots, p_i\}\)

is such that \(L(A) = L(B)\).

Proof sketch. Any accepting run of \(A\) will eventually reach a point where all the transitions it visited satisfy one of the generalized Rabin pairs. The above construction, illustrated by Figure 3.4, works by introducing non-determinism to guess this point and the pair satisfied. The non-deterministic transitions (pictured with grey lines) connect to clones of the original automaton\(^3\), in which taking the transitions of \(F_i\) (for pair \(i\)) is forbidden from now on, and the acceptance condition is set so that the other Inf sets still have to be visited infinitely often.

Given a TFLA \(B\) with acceptance \(\bigvee_{i=1}^{n} \text{Inf}(J_1^i) \land \text{Inf}(J_2^i) \land \ldots \land \text{Inf}(J_p^i)\), an accepting run has to satisfy one Inf conjunction. This means that there is an accepting run if, for some \(1 \leq i \leq n\), there is a trace from the initial state to a reachable (fitting) SCC \(C\) such that \(J_k^i \cap C_{\delta} \neq \emptyset\), for all \(1 \leq k \leq p_i\).

A TGBA checking algorithm can be extended to also check for TFLA emptiness. This is achieved by tracking all acceptance sets in each found fitting SCC. Thus, given the automaton \(B_i\), we would track whether \(\bullet, \text{ and } \circ\) have a non-empty intersection in each SCC \(C\). Then, when checking whether a fitting SCC \(C\) contains

\(^2\)Strictly speaking, a Fin-less automaton could use any formula using only Inf terms. We assume that the formula is under disjunctive normal form for simplicity and because this is what the construction of Proposition 3.1 produces.

\(^3\)When implementing this construction, the number of non-deterministic jumps can be reduced: only one such jump is needed per cycle of the sub-automaton created for each generalized Rabin pair.
Figure 3.4: Application of Proposition 3.1 to transform the TGRA $A_4$ of Figure 3.3 into a TFLA $B_4$. The grey transitions correspond to non-deterministic jumps to some copies of the original automaton. Each copy handles one generalized Rabin pair of the original acceptance condition. In the original acceptance was used in both pairs, but since Proposition 3.1 creates a different set of each use, we distinguish these two sets with $1$ and $2$ here.

an accepting cycle, the algorithm iterates over $1 \leq i \leq n$ to search for an $i$ such that $J^k_i \cap C_\delta \neq \emptyset$ holds for all $1 \leq k \leq p_i$. Note that, as with the case for TGBAs, it is also always valid to only check for maximal SCCs instead of FSCCs.

The Fin-removal construction procedure, in combination with the extended TGBA emptiness checking algorithm, could be regarded as a method for checking TGRAs. In fact, the non-determinism introduced by the Fin-removal construction is very similar to non-deterministically choosing to check for one of the TGRPs. An accepting run on a TFLA starts by first forming a trace from the initial state of a TGRA to an arbitrary state in the TGRA. Then, a non-deterministic choice is made to decide which TGRP will be checked. Finally, the trace is continued to a reachable fitting SCC that does not contain any Fin transitions from the chosen TGRP. Note that this accepting run is also accepting in the corresponding TGRA. We will see in our algorithm for checking TGRAs that each TGRP is addressed separately, i.e. we explore the entire state space (in case no accepting cycle is found) to check the acceptance of a TGRP. A TFLA checking procedure is able to check all `TGRPs' at once, since there are no Fin edges in its state space. The downsides of using the Fin-removal construction are that it introduces non-determinism in the automaton, and it linearly increases the size of the automaton (in the number
3.5 Algorithm for TGRA emptiness

In this section, we present a direct algorithm for checking emptiness on TGRAs, without removing Fin sets first. We start by splitting up the TGRA acceptance into individual TGRPs, and show how these can be checked. TGBA and TFLA checking algorithms are simplified variants of the algorithm (where we do not have to track Fin sets).

3.5.1 Checking Rabin pairs

Checking TGRAs can be achieved by checking each Rabin pair separately, as shown in Algorithm 10. In case an accepting cycle is found by TGRPAcc, that sub-procedure should report acc and exit. Thus, in case none of the TGRPAcc sub-procedures reports acceptance, the algorithm returns with No_Acc. The TGRPAcc procedure itself will be explained later.

Algorithm 10: Checking TGRA by checking each TGRP

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>function TGRACheck ((Q, q_0, \text{TGRA} = {\text{TGRP}_1, \ldots, \text{TGRP}_n}))</td>
</tr>
<tr>
<td>2</td>
<td>for all (i \in {1, \ldots, n}) do</td>
</tr>
<tr>
<td>3</td>
<td>\quad TGRPAcc((Q, q_0, \text{TGRP}_i))</td>
</tr>
<tr>
<td>4</td>
<td>return No_Acc // no TGRPAcc call reported Acc</td>
</tr>
</tbody>
</table>

We assume that prior to each TGRPAcc call, we have no knowledge on the individual TGRPs and therefore treat them equally and separately. In theory, this assumption may lead to missed opportunities, for example, if TGRP\(_1\) = TGRP\(_2\). Even an overlap in the Fin and/or Inf fragments of the TGRPs might offer an opportunity to combine gained information.

3.5.2 TGRP checking algorithm

Throughout this section we consider checking a single TGRP with acceptance of the form \(\text{acc} = (F, \{I^1, \ldots, I^p\})\). We note that a TGRP can be seen as an extension of a TGBA, in which a Fin constraint is added. The algorithm that we propose is an extension of the SCC decomposition algorithm from Chapter 2. We present the algorithm’s sequential execution and show how it can be parallelized.
Abstract idea of the algorithm. The general idea of the algorithm, which we present in Algorithm 12 (a simplified version of the algorithm is given in Algorithm 11), is to perform an SCC decomposition of the automaton without allowing any $F$ transitions from being part of the SCCs. As a result, we obtain SCCs that contain all edges except those in $F$. Formally, we have that each SCC $C$ is a fitting SCC of $A$ that is maximal on the TGRP $A_{δ\setminus F} := (Σ, Q, q₀, δ \setminus F, acc)$. $C$ is an accepting FSCC if $C ∩ I_i \neq \emptyset$ for each $1 \leq i \leq p$, i.e. $C$ contains transitions such that every $I_i$ can be visited infinitely often. By definition of $A_{δ\setminus F}$, we have that $C ∩ F = \emptyset$. If $C$ is also reachable from $q₀$ via transitions from $δ$ (including $F$ transitions), it can be reported that a counterexample exists.

Preventing $F$ transitions from being considered. The algorithm detects the aforementioned ‘constrained’ SCCs in linear time and in an on-the-fly setting. It does so by performing a constrained SCC decomposition of $A$ from $q₀$. Once a transition $t = (t_s, t_ℓ, t_d) ∈ F$ is encountered, the state $t_d$ is stored in a so-called $F$-states set and $t$ is further disregarded since $t$ cannot appear in any accepting cycle. Once this search is finished, all states are marked as $Explored$. Now, all SCCs are decomposed on the automaton $A'$, which is formed by a reachability from $q₀$ over the transitions $δ \setminus F$. In case a non-trivial SCC contains transitions from all $I_i$ sets we have detected a counterexample. Otherwise, we pick a state $s$ from the $F$-states set and consider the following two cases:

1. $s$ is marked $Explored$, meaning that it was added to $F$-states but it was also reachable in $A'$ (without taking any $F$ transitions). Thus, we have already explored this state and can ignore it.

2. $s$ is not marked $Explored$, meaning that $s$ is not part of $A'$ and we launch a new SCC decomposition from $s$.

Example 3.2: Illustrating the search procedure

The search procedure is illustrated in Figure 3.5. Here, two TGRPs are checked separately. Note that due to the way how $①$ and $②$ are located in the automaton, the initial searches for the first and second pair lead to different components.

Workers doing the emptiness check for the first pair will first explore $C_1 ∪ C_3$, attempting to find a cycle satisfying $\text{Inf}(①) ∨ \text{Inf}(②)$ without crossing the $③$-transitions leading to $u$ and $t$. If no accepting cycles are found, they will continue their exploration in $C_2$, starting in states $u$ and $t$, and ignoring all transitions going back to $C_1 ∪ C_3$. Both have not been explored so suppose that $u$ is arbitrarily chosen as a ‘new’ initial state. We assume that the search from $u$ visits all states in $C_2^u$. If we now find the edge from $v$ to $r$, 

thus from $C_2$ to $C_1$, we should not report an accepting cycle as it would contain the $\mathbb{1}$ mark. This is guaranteed, since the search from $u$ is initiated after the search from $q_0$ is complete; so $r$ is already marked Explored and thus ignored. In fact, even when we allow the search from $u$ to start before all states in $C_1 \cup C_3$ are marked Explored it may in the worst case only add redundant explorations. This is because the edge from $s$ to $u$ is not included as an edge in the SCC decomposition and hence no cycle can be formed with a $\mathbb{1}$ mark.

Workers doing the emptiness for the second pair, will similarly first look for cycles satisfying $\text{Inf}(\mathbb{3})$ in $C_1 \cup C_2$, postponing the exploration of $C_3$ that is only accessible via a $\mathbb{3}$-transition.

Consider for example what happens when there is no path from $u$ to $t$. After the search for $u$ ends, all reachable states from $u$ are marked Explored and the search from $t$ is started. Once it observes an Explored state it will not continue searching that state, hence no redundant states are explored.

**Simplified algorithm.** In Algorithm 11 we present a simplified version of the TGRP checking algorithm, resembling the Set-based SCC algorithm from Section 2.4. The function is initially called with $s := q_0$ and it recursively explores the graph in a depth-first order. A state is marked visited, then its successors are processed, and finally the state gets marked Explored. All Explored states are ignored when processing successors. If the successor $t$ has not been visited yet,
Algorithm 11: Simplified sequential algorithm for checking a single TGRP.

1. $\text{Visited} := \emptyset$ // Set of visited states
2. $\text{Explored} := \emptyset$ // Set of explored states
3. $\text{Fstates} := \emptyset$ // Set of (ignored) $F$ states

4. $\text{TGRPAccSimple}(Q, q_0, \text{TGRP})$ // main function call

5. function $\text{TGRPAccSimple}(Q, s, \text{TGRP} = (F_M, I_M))$
6. $\text{Visited} := \text{Visited} \cup \{s\}$ // mark $s$ as visited
7. forall $(\text{acc}_t, t) \in \text{Succ}(s)$ do // explore successors
8. if $t \not\in \text{Visited}$ then continue // ignore fully explored states
9. else if $t \not\in \text{Visited}$ then $\text{unseen state} t$
10. if $\text{acc}_t \cap F_M \neq \emptyset$ then $\text{Fstates}.\text{addState}(t)$ // store $F$ states
11. else $\text{TGRPAccSimple}(Q, t, \text{TGRP})$ // recursively explore otherwise
12. if $t$ is visited before, we found a cycle
13. else if $\text{acc}_t \cap F_M = \emptyset$ then // no $F$ transition allowed in the cycle
14. Unite states and combine acc sets on cycle
15. if $I_M \subseteq \text{combined acc set}$ then
16. report acc and exit // accepting cycle found
17. $\text{Explored} := \text{Explored} \cup \{s\}$ // explored state $s$
18. if $\text{Explored} = \text{Visited}$ then // backtracking from the 'initial' state
19. if $\neg \text{Fstates.isEmpty()}$ then // check if there are stored $F$ states
20. $f := \text{Fstates.pickState()}$ // try the next $F$ state
21. $\text{TGRPAccSimple}(Q, f, \text{TGRP})$ // new search
22. else return // no accepting cycle found in this TGRP

Depending on whether it is reached via an $F$ transition, we either store it in the $\text{Fstates}$ set, or we recursively explore the state. If $t$ has been visited before and it is not reached via an $F$ transition, then we detected a cycle. We then want to combine the state and transition marks on this cycle to check if this is an accepting cycle. Once the initial state is fully explored, we check if there are stored states in $\text{Fstates}$, and if so, we start searching from these states until we processed the entire reachable state space.

Data Structures. To represent the $\text{Fin}$ and $\text{Inf}$ fragments of a TGRP, we use a set of accepting marks per transition. We assign a unique mark to each $F$ and $I^i$ set, for $1 \leq i \leq p$, and refer to these marks with $F_M$ and $I^i_M$. We denote the set of all $\text{Inf}$ marks by $I_M$, i.e., $I_M := \bigcup_{1 \leq i \leq p} I^i_M$. The complete set of markings $M$ is
thus defined as $M := \{F_M, I_1^M, \ldots, I_p^M\}$. Each transition $t$ is associated to a set of acceptance marks $\text{acc} \subseteq M$, indicating whether $t \in F$ or $t \in I^i$ for $1 \leq i \leq p$.

We define $S$ as a mapping to store strongly connected components from states to pairs, consisting of a set of states and a set of marks (as opposed to just a mapping from states to states, c.f. Chapter 2). Thus $S(s) = (\text{states}, \text{acc})$ and formally, $S: Q \to 2^Q \times 2^M$. We implement $S$ with the union-find structure from Chapter 2 to maintain the following invariant:

$$\forall u, v \in Q: u \in S(v).\text{states} \iff S(v) = S(u)$$

In the algorithm, we use $S$ to associate each state $u$ to its fitting SCC that contains the states $S(u).\text{states}$ and visits all the marks in $S(u).\text{acc}$. $S$ pairs can be combined using a $\text{Unite}$ function. As an example, let $S(u) := (\{u, w\}, \{I_1^M, I_2^M\})$ and $S(v) := (\{v\}, \{I_2^M, I_3^M\})$, we can use the $\text{Unite}$ function to combine the two structures. After calling $\text{Unite}(u, v)$ we have $S(u) = S(v) = (\{u, v, w\}, \{I_1^M, I_2^M, I_3^M\})$, while keeping all other mappings the same. We use an additional function $\text{AddAcc}$ to only add acceptance marks to the set, thus $\text{AddAcc}(v, \{I_2^M, I_4^M\})$ will ensure that $S(v).\text{acc}$ becomes $\{I_1^M, I_2^M, I_3^M, I_4^M\}$.

The $\text{Fstates}$ structure is implemented as a cyclic list that contains all states added to the list (by means of $\text{Fstates.addState}$) in a similar way as the cyclic list implementation from our union-find structure (Section 2.6.3). $\text{Fstates.pickState}$ returns a state from the list in case the list is non-empty and otherwise returns NULL. States are removed from the list by calling $\text{Fstates.removeState}$.

**Detailed Algorithm.** The detailed sequential algorithm for checking a TGRP is presented in Algorithm 12. We maintain a stack $R$, which can be regarded as an extension to the stack in our SCC decomposition algorithm. Here, in addition to the states, we also track the acceptance sets from the transitions that led to the state.

First, all data structures are initialized in lines 2 to 6. Then, the algorithm continuously picks a state $s$ (initially $q_0$) and calls the $\text{TGRPAccRecur}$ procedure. After the $\text{TGRPAccRecur}$ is finished, $s$ is removed from the $\text{Fstates}$ list and a new state is picked from the list. If the list is empty, then the complete reachable state space must have been visited and since no $\text{acc}$ was reported, we can conclude that no counterexample exists for this TGRP.

In the $\text{TGRPAccRecur}$ procedure, state $s$ is marked as visited and pushed on top of the $R$ stack, along with the accompanying acceptance set $\text{acc}_s$ (note that since there is no transition to the initial state, the empty set is given in line 9). All successors of $s$ are considered in lines 15 to 27. For each successor $t$ we consider three cases:
• $t \in \text{Explored}$ (line 16), this implies that $t$ has already been completely explored and can thus be disregarded.

• $t$ is unvisited (lines 17 to 19), meaning that $t$ has not been encountered yet. If $t$ is part of the $\text{Fin}$ set, we add it to the $\text{Fstates}$ list and ignore it for the current search. Otherwise, we recursively search $t$.

• $t$ is not $\text{Explored}$ but it has been visited before (lines 20 to 27). This implies that there is some state $r'$ on the $R$ stack such that $t \in S(r')$.states and hence a cycle can be formed. The algorithm then continuously takes the top two states from the $R$ stack and unites them (and adds the acceptance mark) until $S(s)$ and $S(t)$ are the same. Finally, the acceptance marks from $\text{acc}_t$ are added. At line 26, $S(s)$ contains all states in the cycle from $s$ to $t$ and forms a fitting SCC. It is then checked if $S(s)$ contains all $\text{Inf}$ acceptance marks. If so, an accepting SCC is found and reported.

After all successors are explored, the algorithm backtracks. In case $s$ equals the top of the $R$ stack (line 28), $s$, we mark $S(s)$ as $\text{Explored}$.

In Figure 3.6 and Figure 3.7 we show an example run of the algorithm on a small automaton. The TGRP has the acceptance condition: $\text{Fin}(0) \land \text{Inf}(1) \land \text{Inf}(2)$, and thus the algorithm avoids edges marked with $\circ$. Eventually, the algorithm will detect the accepting cycle containing states 2, 3, and 6.

**Outline of Correctness.** We argue that the TGRPAcc algorithm decomposes the TGRP automaton in maximal SCCs when defined over the transitions $\delta \setminus F$ and that it correctly reports accepting cycles; it reports $\text{acc}$ when a reachable SCC contains a transition from each $I_i$ sets, for $1 \leq i \leq p$, and no transition from $F$. Due to the conditions of line 18 and 20, for a transition with $\text{acc}_t \cap F \neq \emptyset$ it is not possible to start a recursive call with $\text{acc}_t$ (thus $\text{acc}_t$ never appears on the $R$ stack) nor is it possible to call $\text{AddAcc}$ with $\text{acc}_t$ as an argument. All such transitions are ‘avoided’ and unvisited successors are added to $\text{Fstates}$. We thus conclude that no $F$ transition can be contained in any formed SCC.

Because we do allow and explore all other (non-$F$) transitions during the search, assuming a correct SCC algorithm, the acceptance set of each SCC cannot be further extended without also having to include an $F$ transition.

Since all states that did not get visited were added to the $\text{Fstates}$ list, and each state from this list is eventually picked as an initial state, we argue that the complete reachable state space has been explored after the algorithm terminates on line 11.
 Algorithm 12: Algorithm for checking a TGRP.

1 function $\text{TGRPAcc} (Q,q_0,TGRP = (F_M,I_M))$
2 \hspace{1em} $\forall s \in Q : S(s) := ({s},\emptyset)$ // initialize map
3 Visited := $\emptyset$ // visited set
4 Explored := $\emptyset$ // explored set
5 $R := \emptyset$ // roots stack of (acc,states) pairs
6 $\text{Fstates} := \{q_0\}$ // add initial state to Fstates
7 while $\neg \text{Fstates.isEmpty()}$ do
8 \hspace{1em} $s := \text{Fstates.pickState}()$
9 \hspace{1em} if $s \notin \text{Explored}$ then $\text{TGRPAccRecur}(\emptyset,s)$
10 \hspace{1em} $\text{Fstates.removeState}(s)$
11 return // no Acc got reported in the search
12 function $\text{TGRPAccRecur}(\text{acc}_s,s)$
13 \hspace{1em} Visited := Visited $\cup \{s\}$
14 \hspace{1em} $R.\text{push}(\text{acc}_s,s)$ // also add the acceptance set to reach $s$
15 \hspace{1em} forall $(\text{acc}_t,t) \in \text{Succ}(s)$ do
16 \hspace{2em} if $t \in \text{Explored}$ then continue // explored states are ignored
17 \hspace{2em} else if $t \notin \text{Visited}$ then // 'new' state
18 \hspace{3em} if $\text{acc}_t \cap F_M \neq \emptyset$ then $\text{Fstates.addState}(t)$ // avoid $F$
19 \hspace{2em} else $\text{TGRPAccRecur}(\text{acc}_t,t)$ // otherwise recursively visit $t$
20 \hspace{1em} else if $\text{acc}_t \cap F_M = \emptyset$ then // cycle detected
21 \hspace{2em} while $S(s) \neq S(t)$ do
22 \hspace{3em} $(\text{acc}_r,r) := R.\text{pop}()$
23 \hspace{3em} Unite($r,R.\text{top}().\text{state}$) // unite the top two stack states
24 \hspace{3em} $\text{AddAcc}(r,\text{acc}_r)$ // also add the stored acceptance set
25 \hspace{2em} $\text{AddAcc}(s,\text{acc}_t)$ // finally, also add accs to $S(s)$
26 \hspace{2em} if $I_M \subseteq S(s).\text{acc}$ then // accepting cycle detected
27 \hspace{3em} report Acc and exit
28 \hspace{1em} if $s = R.\text{top}()$ then // mark completed SCC as explored
29 \hspace{2em} $\text{Explored} := \text{Explored} \cup S(s).\text{states}$
30 \hspace{2em} $R.\text{pop}()$
Figure 3.6: Example of running the algorithm on a small automaton and a single TGRP: $\text{Fin}(0) \land \text{Inf}(1) \land \text{Inf}(2)$. In this case, transitions labelled by $0$ should be avoided. Unexplored transitions are dotted. States and transitions on the DFS stack are in bold. Explored states are coloured. The grey background represents the current view of the SCCs as stored in the union-find data structure; for instance $\{u\}$ means that $S(u) = S(v) = (\{u, v\}, 1)$.
3.5. Algorithm for TGRA emptiness

(Figures 3.7: Example (continued from Figure 3.6).

Complexity. One can observe that every state and transition is visited at most once in the algorithm. The TGRPAccRec procedure will mark a state as visited and will never be called twice for the same state. The bottleneck of the algorithm becomes maintaining the $S$ structure. From Chapter 2 we know that the union-find structure (without tracking acceptance marks) causes the complete algorithm to operate in quasi-linear time. By assuming that the number of acceptance marks, $|M|$, is a small constant (which holds in practice), tracking the acceptance can be achieved in constant time per modification to the structure. Hence, the total time complexity is upper bounded by $O(|M| \cdot |\delta| \cdot \alpha(\delta))$ for each TGRP (where $\alpha$ is the inverse Ackermann function, which is basically a small constant).

The space complexity is limited by the sizes of the $R$, $S$, and $F$ states structures. $R$
may contain up to $|Q|$ states and acceptance marks in the worst case (by visiting every state in a single path). $S$ can be implemented as an array of length $|Q|$ of structs that are of constant size, plus $|M|$ bits for tracking acceptance, and $Fstates$ can be implemented as an array of $|Q|$ elements. In total $\mathcal{O}(|Q| \cdot |M|)$ memory is used.

### 3.5.3 Parallel implementation

We now present two ways in which parallelism can be used to speed up the checking process for TGRAs.

**Parallel TGRA checking.** We first consider Algorithm 10 for checking each TGRP separately. After a TGRPAcc call has finished, the next Rabin pair is selected and a new sub-procedure is started, until all $n$ pairs have been checked. Since we are working in a multi-core environment, we can assign different worker instances to different Rabin pairs. Suppose there are $P$ workers available, we can choose to either use all $P$ workers for checking a single Rabin pair, or we can distribute the workers over the different pairs. By distributing the workers evenly, for $n$ Rabin pairs, each pair is checked by $\frac{P}{n}$ workers.

A disadvantage of the latter setup is that each of the $n$ groups of $\frac{P}{n}$ workers processing the same TGRP needs its own copy of the shared data structure. This means that by checking all pairs in parallel, approximately $n$ times more memory is required$^4$.

However one advantage of checking all pairs in parallel is that these jobs are completely independent, so this could be the basis for a distributed algorithm. We expect better scalability, since parallel workers on a single pair have to synchronize on a shared data structure (our implementation uses a concurrent hash table and a shared union-find data structure). Also, when $P$ gets large compared to the size of the (remaining) graph, the probability that workers have to wait for each other, or perform duplicate work, increases.

Another advantage could be that counterexamples may be detected faster in the latter setting. Suppose for example that only the $n^{th}$ pair contains a counterexample that is detected by visiting only part of the state space. Then the parallel pairs approach could prevent traversing the complete state space $n - 1$ times.

We have experimented with these two extreme parallelisation approaches, but it is conceivable that a more flexible job scheduler with load balancing leads to even

---

$^4$All global (shared) data structures have to be copied for the $n$ pairs, but the memory overhead for the local data structures remains the same.
higher speedups. Note, however, that in practice the number of Rabin pairs is quite limited.

**Parallel TGRP checking.** Algorithm 12 can be parallelized by *swarming* the search instances; by starting multiple worker instances from the initial state and using a randomized successor function to steer the workers towards different parts of the state space. The TGRPAccRec function can be seen as an extension to the multi-core SCC algorithm from Chapter 2. The key to this algorithm is to *globally* communicate *locally* detected cycles. This way, multiple workers can cooperatively decompose SCCs. Additionally, (partly) unexplored states in an SCC are tracked globally and once a worker fully explores a state, none of the other workers has to explore this state again. Once all states of an SCC are fully explored, the entire SCC must be fully explored and thus can be marked as being *Explored*.

During *Unite* procedures, the involved parts of the union-find structure are briefly locked to guarantee correctness. During this locking phase, the acceptance set can be updated atomically without interfering with other parts.

The *Fstates* list is implemented by using a fine-grained locking mechanism to add states to the list, such that all states remain on the cycle (similar to the internal cyclic list of the union-find structure). The reason for implementing *Fstates* as a cyclic list becomes clear in the next example. Suppose the *Fstates* list contains two states, *u* and *v*. To avoid contention, the algorithm attempts to divide the workload by assigning half of the workers to search from *u* and the other half to search from *v*. Now, assume that *u* does not have any successors and a large part of the state space is reached from *v*. If the search from *u* completes, we ideally want to let the workers aid in the search from *v*. By maintaining *Fstates* as a cyclic list, without much effort we can check which searches have not been completed yet.

The time complexity of the algorithm is in the worst case increased by a factor *P*, for *P* workers, since the algorithm explicitly manages the worker IDs for all workers in each state. This brings the total time complexity of checking a TGRA with *n* TGRPs to $O(P \cdot n \cdot |M| \cdot |\delta| \cdot \alpha(|\delta|))$. However, in practice we observe a significant performance improvement over the sequential implementation.

For the same reason the memory complexity is also increased by *P*, as every worker also maintains its own *R* stack, forming a total of $O(P \cdot |Q| \cdot |M|)$. Moreover, if all TGRPs are checked simultaneously, a copy of the global data structures has to be made for each group of workers that process a different TGRP. As a result, *n* times more memory is required for these structures in case there are *n* TGRPs.

Though, as mentioned before, checking TGRPs simultaneously could reduce the computation time (compared to checking every TGRP one-by-one).
3.5.4 Checking SBAs, TGBAs and TFLAs

We can apply the current algorithm for checking TGRAs directly for checking simpler automata, since it does not matter if there are no Fin transitions in the state space. In fact, we do not need to maintain an \textsf{Fstates} list for checking SBAs, TGBAs, and TFLAs.

The acceptance criteria for a TFLA consists of a disjunction of Inf sets. While it is possible to directly apply Algorithm 10 to check every set of Inf acceptance marks, this is not necessary. Since there are no Fin transitions that restrict the state space exploration, it is sufficient to visit the entire state space once for all Inf set. Checking whether a (fitting) SCC is accepting does become slightly more involved in this way, as we would need to iterate over the required markings for every Inf set. In practice, every such check can be implemented with a single bitwise AND operation and thus results in practically no performance penalty.

For a TGBA, we only have to check whether a fitting SCC contains a single set of acceptance markings (instead of multiple sets in the TFLA case), so the checking procedure becomes even more efficient. For an SBA, an accepting cycle is detected as soon as an accepting state is united with another state, since this implies that an accepting cycle exists.

3.6 Experiments for checking SBAs and TGBAs

We split up the experiments in two parts. In the current section, we analyse our SCC-based SBA and TGBA checking algorithm against the state-of-the-art techniques for model checking. In the next section we analyse the TGRA and TFLA checking algorithms, using various TGRA generators, and compare their performance against our TGBA checking algorithm.

3.6.1 Experimental setup

All experiments were performed on a machine with 4 AMD Opteron\textsuperscript{TM} 6376 processors, each with 16 cores, forming a total of 64 cores. There is a total of 512GB memory available.

\textbf{Implementation.} Our model checking algorithms for SBAs (UFSCC-SBA) and TGBAs (UFSCC-TGBA) are implemented in the \texttt{LTS\textsc{min}} toolset \cite{KLM+15}. We furthermore extended \texttt{LTS\textsc{min}} to use the Spot library \cite{DLF+16} for directly generating Büchi automata (both SBA and TGBA) from LTL formulas.
We compare our algorithms with (sequential) NDFS [CVWY92], CNDFS [ELPvdP12] and the SCC-based algorithm by Renault et al. [RDKP15] which we further refer to as Renault. We attempt to minimize performance differences caused by effects other than those resulting from the algorithmic differences, hence each algorithm is implemented in the LTS\textsc{min} toolset. All multi-core algorithms make use of LTS\textsc{min}'s internal shared hash tables [LvdPW11], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \(2^{28}\) states.

Models and formulas. We used models and LTL formulas from three existing benchmark sets and describe these as follows.

- **BEEM-orig\footnote{Available at http://fi.muni.cz/~xstrejc/publications/spin2014.tar.gz.}**: This consists of the complete collection of original (DVE) models and formulas from the BEEM database [Pel07]. Additionally, a number of realistic formulas were added for several parameterized models (see Blahoudek et al. [BDKS14] for details), forming a total of over 807 formulas.

- **BEEM-gen\footnote{Available at https://www.lrde.epita.fr/~renault/benchs/TACAS-2015/results.html.}**: These are the same models and LTL formulas as used by Renault et al. [RDKP15, RDKP17]. The (DVE) models are a subset of the BEEM database [Pel07] such that every type of model from the classification of Pelánek [Pel08] is represented. A total of 3,268 randomly generated formulas were selected such that the number of states, transitions and number of SCCs were high in the synchronized cross products.

- **MCC\footnote{Available at http://mcc.lip6.fr/2015/.}**: We used a selection of the 2015 Model Checking Contest problems [KGH+15]. This consists of Petri net instances (specified in PNML) of both academic and industrial models, forming a total of 928 models. For each model, 48 different LTL formulas were provided that check for fireability (propositions on firing a transition) and cardinality (comparing the number of tokens in places), forming a total of \(928 \times 48 = 44,544\) experiments. We performed an initial selection using our TGBA checking algorithm with 64 cores and selected instances taking between one second and one minute to check\footnote{We acknowledge, retrospectively, that this selection is biased in favour of our algorithm. Nevertheless, we argue that the selection remains suitable for the benchmarks.}. This resulted in 1,107 experiments.

We combined all datasets for the experiments, totalling 5,128 experiments of which 2,950 contain a counterexample. Each configuration was performed at least 5 times and we computed the results by using the mean value for the 5 computation times. The algorithms were not always able to successfully perform an experiment...
in the maximum allowed time of 10 minutes. When comparing two configurations, we only consider experiments where both algorithms performed successfully and within the time limit.

All results and means to reproduce the results are publicly available online at https://github.com/utwente-fmt/LTL-HVC16. We compare our algorithms with respectively NDFS, CNDFS, and Renault (all performed on BAs) in the upcoming sections. The results are summarized in Table 3.2 and Table 3.3. In the context of validation, we are pleased to note that we won in the LTL category of the 2016 Model Checking Contest (MCC) [KGC16], where we employed our UFSCC-TGBA algorithm. In the 2017 and 2018 editions we ranked second.\footnote{While we are happy with the results, we would like to note that the results for the MCC editions only partially reflect the performance of the underlying LTL model checking algorithms. Other aspects, such as the efficiency of generating the state space, or the ability to apply reductions and exploit symmetries in the models can greatly affect the overall runtime.}

More importantly, we have not encountered a single erroneous result for all experiments in all three MCC editions.

### 3.6.2 Comparison with NDFS

We first compare the results of UFSCC-SBA with the sequential NDFS algorithm, which is arguably the fastest sequential algorithm. Figure 3.8 shows the speedup of UFSCC (using 64 workers) compared to NDFS. One thing to note is that all results are presented on a log-log scale. Here, the point $(x = 10, y = 20)$ implies that the NDFS algorithm took 10 seconds to complete and UFSCC is $20 \times$ faster

![Figure 3.8: Time comparison of UFSCC with 64 workers and sequential NDFS.](image)
3.6. Experiments for checking SBAs and TGBAs

(thus taking only 0.5 seconds). We first consider the experiments that do not contain counterexamples (Figure 3.8a).

The colored marks depict the ‘origins’ of the models. When relating this to time and speedup, the different classes are dispersed similarly, though the BEEM-gen models are more clustered.

Generally, UFSCC performs at least $10 \times$ faster than NDFS. In Table 3.2 we observe that the average speedup is 14.16. For larger models (where NDFS requires more time), the speedup increases. We observed that the improvement in time to model check closely relates to the size of the model. This effect is also visible in Table 3.2, where the speedup grows from 13.24 to 24.35 when comparing the smallest and largest class of models.

The results are a bit different for experiments that do contain counterexamples (Figure 3.8b). Here, UFSCC is actually slower than NDFS for some ‘smaller’ instances (where NDFS completes within 1 second). This is explained by the extra setup time required for UFSCC, combined with an additional bookkeeping on the data structures, which becomes purposeless in trivial cases.

For increasingly larger models, the speedup for UFSCC improves rapidly. This speedup becomes superlinear (more than $64 \times$ faster using 64 workers), which is explained by the fact that multiple workers are more likely to find a counterexample due to randomization [ELPvdP12, RK88].

When relating the results to the origins of the models, we more clearly observe differences. NDFS performs (on average) the fastest on the BEEM-orig experiments, and the slowest on the MCC experiments. For all benchmarks with counterexamples, UFSCC performs on average 4.87 times faster than NDFS.

3.6.3 Comparison with CNDFS

We compare the results of UFSCC with CNDFS, where both algorithms use 64 workers and operate on SBAs. Results for models without counterexamples are depicted in Figure 3.9a.

In most cases, the performance of UFSCC and CNDFS is comparable, and the time difference rarely exceeds a factor of 2. The figure classifies the experiments by the number of transitions in the model. From this classification it becomes clear that UFSCC’s relative performance improves for larger models. In Table 3.2 we find that the relative speedup of UFSCC versus CNDFS increases from 0.95 to 1.31. One can also observe in this table that UFSCC slightly outperforms CNDFS in graphs containing large SCCs.

\[10\] When we discuss averages over the experiments, we always take the geometric mean.
For models with counterexamples (Figure 3.9b) CNDFS clearly performs better for most of the models. On average, UFSCC is 0.79 times as fast as CNDFS. However, the techniques do complement each other since UFSCC outperforms CNDFS in 14% of the instances, in particular the experiments where CNDFS performs slowest.
### 3.6. Experiments for checking SBAs and TGBAs

Table 3.2: Comparison of geometric mean execution times (in seconds) on models without counterexamples. \( T \) denotes the number of transitions in the state space and \( S \) denotes the ratio of the largest SCC size compared to the state space. The numbers between parentheses denote how many times faster UFSCC-SBA is compared to the other algorithm.

<table>
<thead>
<tr>
<th>( T )</th>
<th>( 0 \ldots 1E7 )</th>
<th>( 1E7 \ldots 1E8 )</th>
<th>( 1E8 \ldots 1E9 )</th>
<th>( 1E9 \ldots 1E10 )</th>
<th>( 0% \ldots 1% )</th>
<th>( 1% \ldots 50% )</th>
<th>( 50% \ldots 100% )</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NDFS</td>
<td>CNDFS</td>
<td>Renault</td>
<td>UFSCC-TGBA</td>
<td>UFSCC-SBA</td>
<td>NDFS</td>
<td>CNDFS</td>
<td>Renault</td>
<td>UFSCC-TGBA</td>
</tr>
<tr>
<td>13.55 (13.24)</td>
<td>2.32 (2.27)</td>
<td>1.02 (0.99)</td>
<td>1.02</td>
<td>183.37 (24.35)</td>
<td>9.89 (1.31)</td>
<td>147.44 (19.58)</td>
<td>7.76 (1.03)</td>
<td>7.53</td>
</tr>
<tr>
<td>25.47 (18.71)</td>
<td>6.30 (4.63)</td>
<td>1.36 (1.00)</td>
<td>1.36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>183.37 (24.35)</td>
<td>9.89 (1.31)</td>
<td>147.44 (19.58)</td>
<td>7.76 (1.03)</td>
<td>7.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 3.6.4 Comparison with Renault

We compare the results of UFSCC with Renault, where both algorithms use 64 workers and operate on SBAs. Recall that both algorithms are SCC-based. Figure 3.10a depicts the results for experiments without counterexamples.

Here, the coloured marks categorize the relative size of the largest SCC in the state space. We observe a clear distinction when relating the results with the SCC characteristics. A significant speedup is observed for all models containing a largest SCC that consists of at least 1% of the total state space. This is explained by the fact that Renault does not communicate partially found SCCs as explained in Section 3.3.3, whereas UFSCC does achieve this.

From Table 3.2, we notice that UFSCC's speedup increases for larger models (19.58× speedup for the largest class of models), this can be mainly explained by the differences in SCC sizes.

For models with counterexamples (Figure 3.10b), we observe that Renault performs similar to UFSCC for most of the models. This is to be expected since accepting cycles are detected in the same manner. However, the same effect concerning SCC sizes as in Figure 3.10a seems present (here we have no knowledge on the relative SCC sizes, since we only partially explore the graph). We further analysed some of the experiments where Renault performs relatively poor and indeed found that these instances contain large SCCs.

#### 3.6.5 Experiments using TGBAs

In the preliminaries, we touched upon the LTL classification hierarchy from Manna and Pnueli [MP90] and concluded that only for half of the classes it makes sense
with counterexamples. The numbers between parentheses denote how many times faster UFSCC-SBA is compared to the other.

<table>
<thead>
<tr>
<th></th>
<th>NDFS</th>
<th>CNDFS</th>
<th>Renault</th>
<th>UFSCC-TGBA</th>
<th>UFSCC-SBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>1.52 (4.87)</td>
<td>0.25 (0.79)</td>
<td>0.37 (1.17)</td>
<td>0.31 (1.00)</td>
<td>0.31</td>
</tr>
</tbody>
</table>

![Graph](image1.png)  ![Graph](image2.png)

(a) Without counterexamples.  (b) With counterexamples.

Figure 3.11: Time comparison of UFSCC using SBAs with its TGBA variant, both using 64 workers. Here, |Transitions| compares (a) the total number of transitions in the complete cross-product and (b) the average number of uniquely explored transitions by the algorithms.

to use a more complex acceptance criteria, compared to SBAs. We made a comparison with two versions of UFSCC, where one is implemented for checking on SBAs and the other for TGBAs. We found that there are indeed instances for which it is more beneficial to construct TGBAs.

While the results do differ per model, the TGBA and SBA implementations perform equally well on average in Figure 3.11a and Figure 3.11b. We can observe in Figure 3.11a that in most cases the size of the cross-product is equal for the SBA and TGBA versions. A consequence is that a TGBA should not provide any benefit over a SBA in these cases.

Perhaps surprisingly, using TGBAs instead of SBAs does not generally benefit the model checking performance in our experiments. Even when the TGBA version does provide a smaller cross-product, the algorithms still perform similarly. This may be explained by the additional overhead and bookkeeping for tracking acceptance sets. However, the problem is context-dependent, i.e. the results should
3.7 Experiments for checking TGRAs and TFLAs

Table 3.4: Tools used in the experimental evaluation.

<table>
<thead>
<tr>
<th>tool</th>
<th>version</th>
<th>ref.</th>
<th>web page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spot (ltl2tgba, autfilt)</td>
<td>2.5.2</td>
<td>[DLF+16]</td>
<td><a href="https://spot.lrde.epita.fr/">https://spot.lrde.epita.fr/</a></td>
</tr>
<tr>
<td>Rabinizer</td>
<td>3.1</td>
<td>[KK14, EKS16]</td>
<td><a href="https://www7.in.tum.de/~kretinsk/rabinizer3.html">https://www7.in.tum.de/~kretinsk/rabinizer3.html</a></td>
</tr>
<tr>
<td>LTL3DRA</td>
<td>0.2.6</td>
<td>[BBKS13]</td>
<td><a href="https://sourceforge.net/projects/ltl3dra/">https://sourceforge.net/projects/ltl3dra/</a></td>
</tr>
<tr>
<td>LTL3TELA</td>
<td>1.1.1</td>
<td></td>
<td><a href="https://github.com/jurajmajor/ltl3tela/">https://github.com/jurajmajor/ltl3tela/</a></td>
</tr>
<tr>
<td>LTSmin</td>
<td>3.0</td>
<td>[KLM+15]</td>
<td><a href="http://ltsmin.utwente.nl/">http://ltsmin.utwente.nl/</a></td>
</tr>
</tbody>
</table>

Table 3.5: Tool options for generating an automaton from an LTL formula $\varphi$.

<table>
<thead>
<tr>
<th>Key</th>
<th>Automaton</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rabinizer 3</td>
<td>TGRA</td>
<td>java -jar Rabinizer3.jar -format=hoa -in=formula -out=std -silent $\varphi$</td>
</tr>
<tr>
<td>LTL3DRA</td>
<td>TGRA</td>
<td>ltl3dra -f $\varphi$</td>
</tr>
<tr>
<td>LTL3TELA</td>
<td>TGRA</td>
<td>ltl3tela -f $\varphi$</td>
</tr>
<tr>
<td>Fin-less</td>
<td>TFLA</td>
<td>ltl3tela -f $\varphi$</td>
</tr>
<tr>
<td>TGBA</td>
<td>TGBA</td>
<td>ltl2tgba -f $\varphi$</td>
</tr>
</tbody>
</table>

be more in favour of the TGBA version if the to-be-checked properties result in significantly smaller automata for TGBAs when compared to SBAs.

3.7 Experiments for checking TGRAs and TFLAs

In this section, we compare the performance of the TGRA (using three different TGRA generators) and TFLA checking algorithms with the TGBA checking algorithm.

3.7.1 Experimental setup

All experiments were performed on a machine with 4 AMD Opteron™ 6376 processors, each with 16 cores, forming a total of 64 cores. There is a total of 512GB memory available.

In preliminary experiments we observed scalability issues for the TGRA checking algorithm when using 64 cores, likely caused by the overhead of maintaining the cyclic list of F states. While we have not yet been able to determine the exact cause of the scalability issues, we did observe good scalability for up to 16 cores. In order to compare our TGRA and TGBA algorithms with each other to some extent, we performed all experiments using 16 cores.
Implementation. References to all the tools involved in the experiments are given in Table 3.4. The TGRA checking algorithm is implemented in the LTSMIN toolset. Additionally, we used several external tools and libraries for generating and parsing the automata:

- We used Rabinizer 3 to generate deterministic transition-based generalized Rabin automata.
- The tool LTL3DRA was also used to generate deterministic automata with transition-based generalized Rabin acceptance. However LTL3DRA only supports a subset of LTL, called LTL¬GUX in [BKS13], which is slightly stricter than the set of LTL formulas where no until (U) operator may occur in the scope of any always (G) operator.
- We used some tools from Spot: ltl2tgba for generating TGBAs, and autfilt for converting automata with other accepting conditions into TGRAs or TFLAs.
- We used LTL3TELA to generate nondeterministic automata with an arbitrarily complex transition-based acceptance. We then used Spot’s autfilt --generalized-rabin to convert these automata to TGRAs.
- We used the cpphoafparser library to parse a HOA automaton [BBD+15] and create an internal representation for LTSMIN.

The commands used to generate automata from LTL are summarized in Table 3.5.

Experiments. For our models and properties, we used the same initial selection as we did in the previous section (Section 3.6.1), by obtaining them from the 2015 Model Checking Contest [KGH+15], and from the BEEM benchmark [Pel07].

Because we are interested in checking properties for which TGRAs may be beneficial to simpler automata, we reduced the set of MCC formulas as follows. The MCC formula set was restricted to those that fall into the classes of ‘more complex’ LTL properties, i.e. for which a non-SBA acceptance criterion can improve on an SBA. This selection is further reduced by selecting only the instances where the ‘TGRA generators’ (LTL3DRA, Rabinizer 3 and LTL3TELA) create TGRAs with at least one non-empty Fin set. Otherwise, a TGRP is the same as a TGBA, and hence the TGBA emptiness check could be used instead. For this selection, we report results on the experiments (118 in total) for which the time to model check using the TGBA checking algorithm is between 1 second and 10 minutes. We remark that this selection is in favour of the TGBA checking algorithm, since all cases where timeouts and memory errors occurred in the TGBA algorithm were filtered out as a result of our selection criteria.
Table 3.6: Comparison of the geometric mean execution times (in seconds). The numbers between parentheses denote how many times faster the TGBA checking algorithm is compared to the other configuration. We only used the experiments that were checked in all TGRA and TGBA configurations without running out of memory or time (39 in total for the MCC experiments, with 13 counterexamples, and 1,167 in total for the BEEM experiments, with 699 counterexamples).

<table>
<thead>
<tr>
<th></th>
<th>MCC</th>
<th>BEEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LTL3TELA</td>
<td>LTL3DRA</td>
</tr>
<tr>
<td>Counterexample</td>
<td>1.51 (1.89)</td>
<td>1.69 (2.12)</td>
</tr>
<tr>
<td>No counterexample</td>
<td>7.68 (1.60)</td>
<td>5.67 (1.18)</td>
</tr>
<tr>
<td>Total</td>
<td>4.47 (1.69)</td>
<td>3.79 (1.43)</td>
</tr>
</tbody>
</table>

The second set of experiments consists of the BEEM models accompanied by randomly generated formulas from [SDKT14] (the same as the BEEM-gen experiments from the previous section). For the TFLAs in all experiments, we only considered those that contain a disjunction of multiple TGBA acceptance sets, hence we filtered out all TFLAs generated from a TGRA with a single Rabin pair.

For each pair of model $M$ and formula $\varphi$ we solved the model checking task using 5 configurations (as presented in Table 3.5) that were repeated 10 times. We take the average of the results to mitigate randomness introduced by the multi-core search procedure. Every task was run with a timeout of 10 minutes. In total the experiments took approximately 10 days to complete.\(^\text{11}\)

### 3.7.2 Main results

The main results of the experiments are presented in Figure 3.12 and are summarized in Table 3.6. We also depict information regarding the state space (for both the automaton and the cross-product) in relation to the TGRA generators and number of TGRPs, in Table 3.7, Figure 3.13, and Figure 3.14. As with our

\(^{11}\) All our results are available on https://github.com/utwente-fmt/Rabin-STTT.
Figure 3.12: Time (in seconds) comparisons of the TGBA (x-axis) and the TGRA and TFLA emptiness checks (y-axis), for various LTL to TGRA and TFLA translations. Each point represents the time to perform an emptiness check using 16 cores. The TGRA and TFLA algorithms performed faster for instances below the x=y line.

In previous experiments, we note that all results are presented on a log-log scale. The (16-core) experiments for the TGBA checking algorithm are provided on the x-axis and the results for the TGRP and TFLA checking experiments are shown on the y-axis. All TGRA
tas are checked by considering each TGRP sequentially, i.e. all workers are assigned to the first TGRP and continue to the second pair (if there is one) when the first TGRP is fully explored.

We encountered some errors in the experiments. There were a number of instances that resulted in a memory error, meaning that the data structures became too large to fit in the memory during the model checking procedure. These errors only occurred for the TGRA checks and were presumably caused by the additional allocation of the \texttt{Fstates} data structure. There are also several instances that resulted in timeouts for some of the configurations.
Table 3.7: Geometric mean sizes of the automata and products. $|Aut|$ denotes the number of states in the LTL automaton, $|Pairs|$ the number of TGRPs in the TGRA, and $|States|$ and $|Trans|$ provide the sizes of the product automaton. We only used data from experiments without a counterexample and that were checked in all TGRA and TGBA configurations without running out of memory or time (26 in total for MCC and 468 in total for BEEM).

|       | $|Aut|$ | $|Pairs|$ | $|States|$ | $|Trans|$ |
|-------|--------|----------|-----------|----------|
| MCC   | 1.00   | 1.53     | 0.71 • 10^6 | 4.26 • 10^6 |
| LTL3TELA | 1.00 | 1.00     | 0.47 • 10^6 | 2.78 • 10^6 |
| LTL3DRA  | 1.38 | 1.00     | 0.47 • 10^6 | 2.78 • 10^6 |
| Rabinizer 3 | 1.31 | 1.00     | 0.51 • 10^6 | 3.11 • 10^6 |
| TGBA   | 2.85   | 1.07     | 4.02 • 10^6 | 13.86 • 10^6 |
| BEEM   | 4.00   | 1.27     | 10.61 • 10^6 | 42.17 • 10^6 |
| LTL3TELA | 3.56 | 1.24     | 10.29 • 10^6 | 40.86 • 10^6 |
| LTL3DRA  | 3.23 | 1.00     | 3.80 • 10^6 | 13.74 • 10^6 |

We first analyse the performance of the TGRA checking algorithm and how the different TGRA generators influence the results, then we consider the translation to Fin-less automata, and finally we discuss some additional results.

**Comparison between the three TGRAs and TGBA.** We first take a look at Figure 3.12 and Table 3.6. It becomes clear that for most experiments it is more beneficial to model check using TGBAs compared to TGRAs (using our algorithm). In the scatter-plot we observe that in some cases the TGBA checking algorithm is more than two orders of magnitude faster. That being said, there are instances where the TGRA checking algorithm is faster, especially for the TGRAs generated by LTL3TELA.

We can observe that there are significant differences between the TGRAs generated by the three TGRA generators. The (non-deterministic) TGRAs from the LTL3TELA generator leads to the most favourable results. Both deterministic TGRA generators (especially Rabinizer 3) generate automata for which the model checking algorithm performs significantly worse. Hence we argue that in our setting we do not benefit much from deterministic TGRAs compared to nondeterministic ones.

**Analysing the TGRAs.** We analyse how the different TGRA generators influence the automaton sizes in Table 3.7. For the MCC experiments we observe
that the automaton sizes for LTL3TELA and LTL3DRA are on average smaller than a TGBA. Moreover, the product automaton for both deterministic automata are smaller than that of a TGBA. The product size of the LTL3TELA TGRAs are larger than for the other automata, and in Table 3.6 we see that LTL3TELA performs the worst on the MCC experiments.

However, the automata for the BEEM experiments are quite different from the MCC ones. Here, both deterministic TGRA generators generate significantly larger automata and product automata (a factor of three larger) compared to both LTL3TELA and TGBA. Here it seems that LTL3TELA leads to more suitable automata for emptiness checking TGRAs. As can be observed in Table 3.6, the product automaton size seems to correlate well with the TGRA emptiness check performance, as LTL3TELA is clearly the favourite of the three.

In Figure 3.13 we show how the number of Rabin pairs correlates with the size of the automaton. Note that this figure only shows the majority of the generated
3.7. Experiments for checking TGRAs and TFLAs

Figure 3.14: Time (in seconds) comparisons of the emptiness checks in the same format as Figure 3.12, but here the plots in the left column only contain TGRAs that have a single Rabin pair. The middle column only shows results for TGRAs with exactly two Rabin pairs, and the right column shows results for the remaining automata, i.e. automata with more than two Rabin pairs.

automata; there are automata with more than 30 states and more than 4 Rabin pairs. It is clear that most of the automata contain a single Rabin pair and have less than five states. Most of LTL3TELA TGRAs seem to have the same number of states as the TGBAs, and LTL3TELA also produces small automata for multiple Rabin pairs. The TGRAs from both deterministic TGRA generators have similar characteristics. There are fewer automata with a single Rabin pair, and the automaton sizes are larger when there are multiple Rabin pairs. This makes sense as multiple Rabin pairs indicate that the LTL formula is more complex, which causes the deterministic TGRA generators to require more states to remain deterministic.
Table 3.8: Comparison of the geometric mean execution times (in seconds) in the same format as Table 3.6. We only used data from experiments that were checked in the TFLA, LTL3TELA-TGRA, and TGBA configurations without running out of memory or time (195 BEEM experiments in total, with 89 counterexamples).

<table>
<thead>
<tr>
<th></th>
<th>LTL3TELA</th>
<th>Fin-less</th>
<th>TGBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counterexample</td>
<td>0.20 (1.55)</td>
<td>0.13 (0.96)</td>
<td>0.13</td>
</tr>
<tr>
<td>No counterexample</td>
<td>15.02 (3.70)</td>
<td>4.52 (1.11)</td>
<td>4.06</td>
</tr>
<tr>
<td>Total</td>
<td>2.11 (2.49)</td>
<td>0.88 (1.04)</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Influence of the number of Rabin pairs and Fin transitions. In Figure 3.14 we split up the results from Figure 3.12 and show how the performance is affected when the TGRAs contain more Rabin pairs. We can see that in general, the TGRA checking algorithm performs relatively worse for automata with multiple Rabin pairs compared to TGRAs with a single pair. We can also see that most of the timeouts occur for two or more Rabin pairs. This result is not surprising, as checking a TGRA with two pairs involves traversing the entire state space of the product automaton twice, assuming there is no counterexample.

We found that for most product automata (with TGRAs), the number of Fin transitions is either a very small or very large proportion of the total number of transitions. In Figure 3.14 we coloured three classes: automata for which the number of Fin transitions is 0-10% of the total number of transitions (blue), automata for which this ratio is between 10% and 90% (black), and automata in which 90% or more transitions are Fin transitions (red).

It becomes clear that this aspect greatly influences the TGRA model checking performance. Its performance is actually comparable to TGBA for a small fraction of Fin transitions. Furthermore, there are almost no instances where the TGRA checking algorithm performs well for cases with many Fin transitions.

3.7.3 Fin-less results

We now discuss how the emptiness check for Fin-less automata compares with the check for TGBAs. As noted before, we only considered TFLAs with at least two Inf sets (and hence are obtained from TFLAs with at least two Rabin pairs). In Figure 3.12 we can see that the performance of the TFLA checking algorithm is comparable to that of the TGBA checking one. While there is only a small difference in the emptiness procedure, the TFLAs are quite different from TGBAs. In Table 3.8 we summarize the performance results and indeed see that there is
3.7. Experiments for checking TGRAs and TFLAs

Table 3.9: Geometric mean sizes of the automata and products in the same format as Table 3.7. Here, $|Pairs|$ denotes the number of Inf sets in the TFLA. We only used data from experiments without a counterexample and that were checked in the TFLA, LTL3TELA-TGRA, and TGBA configurations without running out of memory or time (106 in total for BEEM).

|         | $|Aut|$ | $|Pairs|$ | $|States|$ | $|Trans|$ |
|---------|--------|----------|-----------|----------|
| LTL3TELA| 3.31   | 2.21     | $5.22 \cdot 10^6$ | $11.30 \cdot 10^6$ |
| Fin-less| 5.79   | 2.17     | $2.74 \cdot 10^6$ | $9.06 \cdot 10^6$ |
| TGBA    | 3.76   | 1.00     | $2.37 \cdot 10^6$ | $7.43 \cdot 10^6$ |

only a small difference between the two emptiness checks.

It could be expected that a TFLA emptiness check performs worse than the corresponding TGBA version, as there is a slightly greater overhead in verifying the acceptance condition. Also, as a result of their construction process, contain (linearly, in the number of TGRPs) more states than the original TGRAs. However, since more complex acceptance conditions can be expressed by TFLAs compared to TGBAs, we may be able to construct smaller TFLAs than TGBAs (though, as shown in Table 3.9, this is not reflected by our construction mechanism). A reason for why a TFLA may be checked faster than a TGBA is because multiple acceptance sets can be checked simultaneously, increasing the probability that a counterexample is detected earlier.

In Table 3.9 we see that a TFLA has significantly more states than the corresponding TGBA, which holds as well for its product automaton. The effect of the TFLA construction procedure is visible in Figure 3.13. TFLAs are constructed from the TGRAs generated by LTL3TELA. One can observe that as a result of this construction, the number of states in the automata shifts up when comparing the TFLA with its corresponding TGRA. However, even though the TFLAs are larger and contain more non-determinism than TGRAs, the product automaton is on average smaller.

Figure 3.15 shows how the performance of the TFLA checking algorithm compares with that for the TGRA version of the automata. The TFLA version is significantly faster in practically all instances. The difference is especially large in cases where the TGRA state space consists of many Fin transitions. We can argue that it is more advantageous to first construct a TFLA from a TGRA and check the TFLA, than to check the TGRA directly (using our algorithm).
Figure 3.15: Time (in seconds) comparisons of the emptiness checks in the same format as Figure 3.14, but here the x-axis represents the time required for \text{LTL3TELA-TGRA} and the y-axis shows the time needed for \text{TFLA}.

3.7.4 Additional results

Checking TGRPs in Parallel. In preliminary experiments we experimented with checking TGRPs in parallel, as suggested in Section 3.5.3. We performed experiments to compare the two. In the case for products without counterexamples, there was no observable difference. In case there were counterexamples, the results varied more, but there does not seem to be a clear winner. Because the ‘parallel’ version does allocate significantly more memory (the memory consumption was almost doubled), we prefer to check the TGRPs sequentially.

Scalability. The SCC decomposition extensions for checking SBAs, TGBAs, and TFLAs all achieved good scalability when increasing the number of workers, at least up to 64 cores. Initial experiments for the TGRA checking algorithm showed similar improvements, but the performance improvement starts to drop when increasing beyond 16 cores. The bottleneck of the algorithm is most likely caused by inserting and selecting states from the \text{Fstates} list, especially in case a large part of the state space consists of Fin transitions. Future work could investigate whether the \text{Fstates} list can be further improved, or point out whether
the bottleneck is a structural problem in the algorithm.

3.8 Conclusion

We showed that the SCC decomposition algorithm from Chapter 2 is well-suited for multi-core on-the-fly LTL model checking. We showed that our SBA and TGBA checking variants scale better compared to existing work when the state space increases.

We also presented an algorithm for directly checking TGRAs. However, our experiments show that, in general, our TGBA checking algorithm outperforms our new TGRA algorithm. This seems to be true in particular for cases where a large proportion of the product state space is part of a Fin set for the TGRA. In general we conclude that using TGRAs is therefore not advantageous over TGBAs for checking emptiness, at least when using our algorithms.

We also introduced Fin-less automata, which can be constructed from TGRAs, and showed that the emptiness check for such a TFLA can be implemented quite efficiently. Experiments showed that emptiness checks for TFLAs and TGBAs perform comparably. From our findings we conclude that it is more beneficial to construct a Fin-less automaton and check that for emptiness than to check the TGRA directly.

3.9 Future work

There are several directions for future work for our model checking algorithms. In our experiments we observed that UFSCC-SBA (and UFSCC-TGBA) outperform the state-of-the-art techniques. There is however still room for improvements. First of all, by improving the underlying SCC decomposition algorithm (c.f. Section 2.10), we automatically improve the model checking algorithms.

Another way of improving the performance is to consider applying Partial Order Reduction (POR). By exploiting symmetries in the synchronous product, we may achieve an exponential reduction in the state space [Val96]. In order to implement POR, we would have to consider a sufficient proviso to maintain correctness, e.g. [LPvdPH16, EP10, NWBvdP16]. An initial idea is to ensure that at least one cycle in each terminal SCC gets fully explored, while remaining states may be retrieved from a (reduced) partial order set. While POR has been applied for SBAs, it remains an open problem whether POR can be implemented efficiently for TGBAs or more complex automata.
A different direction is to take a look at counterexamples. One should be able to adapt our algorithm (for SBA checking, and likely also for TGBA checking) to detect the shortest counterexample (e.g. in the length of unique sequence of states). This is not a trivial problem, since a shortest path to an accepting cycle does not guarantee a shortest counterexample; the length of the cycle should also be considered. It should be possible to let the algorithm run until a certain depth (which is updated when an accepting cycle is found), and construct a shortest counterexample once all states up to this depth are explored. Alternatively, it may be interesting to return 'all' counterexamples, or at least all counterexamples that form distinct cycles in the original Büchi automaton.

Regarding the TGRA algorithm, there is still a lot to improve in terms of performance. We observed that our approach with the F states cyclic list is not scalable when a large portion of the state space belongs to that set. It may be interesting to consider alternatives for which Fin transitions are not completely avoided by the initial search. One could think of an NDFS-based solution, where an outer DFS traverses the entire graph (regardless whether the transitions are part of the Fin set), and launch SCC decomposition instances (that avoid Fin transitions) that search for an accepting cycle.

There is likely also much performance to be gained by checking multiple TGRPs at the same time. Currently, we cannot do this because the set of Fin transitions from one pair may overlap with an accepting cycle from another one. But if we alleviate the avoidance of Fin transitions to a certain extent, it may be possible to check multiple (if not all) TGRPs at once.

As a final remark, one would have to consider whether focusing on improving the TGRA checking algorithm is worthwhile. With our current approach we observed a few instances for which the performance is better than the TGBA checking algorithm, but for the most part its performance is worse. These experiments are however of artificial nature and may be unrealistic in practice. Further studies with checking TGRAs would likely have to rely on complex LTL formulas, such that the state space reduction from a TGRA outweighs its model checking complexity compared to a simpler approach.
Alignment-based Conformance Checking for Specific Costs

In this chapter we focus on the single-source shortest path problem. We consider shortest paths in the context of computing alignments and propose new algorithms that are more efficient compared to the state-of-the-art, by exploiting the associated cost-function for alignments.

Given a process model and an event log, conformance checking aims to relate the two together, e.g. to detect discrepancies between them. For the synchronous product net of the process and a log trace, we use a cost function to assign a value for different types of transitions. The transitions are distinguished by reflecting whether the original process, or the log trace, or both, made progress. We refer to these types of transitions by ‘moves’ and distinguish between a synchronous move, a model move, and a log move. By computing a path through this synchronized model, whilst minimizing the total cost, we create a so-called alignment – which is considered to be the primary target result for conformance checking.

Traditional alignment-based approaches (1) have performance problems for larger logs and models, and (2) do not provide reliable diagnostics for non-conforming behaviour (e.g. bottleneck analysis is based on events that did not happen). This is the reason to explore an alternative approach that maximizes the use of observed events. We also introduce the notion of milestone activities (which may not be skipped) and show how the different approaches relate to each other.

The main contents of this chapter have been formed from the following two papers.

- In [BvZvdA+18], which is extended in [BvZvdA+] (in submission), we propose a data structure, that can be computed from the process model, which
can be used for (1) computing alignments of many log traces that maximize synchronous moves, and (2) as a means for showing diagnostic information.

- In [BvdPvdA18] we propose a symbolic algorithm for computing shortest-paths on graphs restricted to 0- and 1-cost edges (which is typical for alignments), and show that this technique is better suited for larger models compared to traditional methods.

### 4.1 Introduction

In the field of process mining [vdA16] there are three main branches: process discovery, conformance checking, and enhancement of processes, all use event data recorded during process execution. In process discovery, we aim to discover process models based on traces of executed event data. In conformance checking, we assess to what degree a process model (potentially discovered) is in line with recorded event data. Finally, in process enhancement, we aim at improving or extending the process model based on facts derived from event data.

Modern information systems allow us to track, often in great detail, the behaviour of the process it supports. Moreover, instrumentation and/or program tracing tools allow us to track the behavioural profile of the execution of enterprise-level software systems [LvDAvdA16, LvdA15]. Such behavioural data are often referred to as an event log, which can be seen as a set of log traces, i.e. sequences of observed events in the system. However, it is often the case, due to noise or under/over-specification, that the observed behaviour does not conform precisely to a valid process instance, i.e. it deviates from its intended behaviour as specified by the model. Additionally, the model itself is not necessarily an accurate representation of reality and therefore may also contain bad behaviour. Hence, given a deviation, this may either be caused by a defect in the model, the log trace, or both.

Conformance checking [CvDSW18] assesses to what degree the event log and model conform to each other. Early conformance checking techniques [RvdA08] are limited in the sense that deviations between observed behaviour and the model may not always be accurately presented. As a consequence, it may be difficult to relate the deviations to their causes, especially in case there are many differences between the observed and modelled behaviour.

**Alignments** [ASvD11, vdaAvD12] were introduced to overcome the limitations of early conformance checking techniques. Alignments map observed behaviour onto behaviour described by the process model. As such, we identify four types of relations between the model and event log in an alignment:
4.1. Introduction

Figure 4.1: Example Petri net model (blue) combined with a log trace (orange), with the corresponding synchronous actions in green. Below the model we show the standard cost function and an optimal alignment $\gamma$ using the standard cost function (with a total cost of $3 + \varepsilon$).

1. A log move, in which we are unable to map an observed event, recorded in the event log, onto the execution of an action as in the model.

2. A model move, in which an action is described by the model, yet this is not reflected in the event log.

3. A synchronous move, in which we are able to map an event, observed in the event log, to a corresponding action described by the model.

4. A silent move, in which the model performs a silent or invisible action (denoted with $\tau$). We are unable to observe such actions.

**Example 4.1: Example alignment**

Consider the example given in Figure 4.1. The Petri net represents the product of an event log $\langle A, C, E, B \rangle$ (depicted in orange) and the model, which is portrayed in blue (model moves) and grey (silent moves). The green transitions illustrate synchronous moves. An alignment is a path from the initial marking on the product, $[\mathbf{p}_0, \mathbf{q}_0]$ to the final marking $[\mathbf{p}_7, \mathbf{q}_4]$, which
An alignment is optimal if it minimizes a given cost function. In the example, \( \gamma \) is optimal for the so-called standard-cost function, which assigns a cost of 0 to synchronous moves, a small value \( \varepsilon > 0 \) to silent moves, and a cost of 1 to model- and log moves. The goal is to search for optimal alignments, which is an NP-hard problem (in the size of the Petri net). An optimal alignment can be found by computing the shortest path on the state space, or marking graph, of the product Petri net.

The current best practice uses an A* [HNR68] approach to find such optimal alignments. Several optimizations are used to improve the performance of the algorithm, such as computing the state space on-the-fly. The marking equation of Petri nets is exploited to prune the search space (remove states that cannot reach the final state any more) and the marking equation can also be used as a good heuristic function for A* [Adr14].

There are however some limitations to the current approach of computing alignments. Due to the large state spaces of some models, computing an alignment with the A* algorithm is not fast enough for practical purposes. Another issue that may occur in practice, is that an event log may contain millions of traces for which such optimization problems need to be solved. While the alignment of a single log-model combination is not computationally intensive, a large number of such alignment problems may become too time-consuming. Lastly, while an alignment is well-suited for showing discrepancies between the model and a log trace, the result may not always be what we want. The result may for instance be conflicting with domain knowledge or implicit assumptions (e.g. we are certain that a certain log or model activity must have taken place). It may also be difficult to interpret multiple alignments for a given model that all have the same cost.

In this chapter, we study each of the above problems. While we are not able to completely solve them, we are able to mitigate the problems to some extent, or at least provide a new insight on the issue. We propose two innovations that are discussed next.
Symbolic alignment computation. Searching for alignments in an explicit-state fashion can still take a long time for large state spaces. In the field of model checking, such a ‘state space explosion’ is a common and well-studied problem [Val96]. One of the techniques to deal with large state spaces is to employ symbolic model checking [BCM+92]. A symbolic representation in the form of decision diagrams is used to efficiently store a set of states. This is combined with operations such as intersect, unite, and a symbolic next-state function to explore the state space. Especially in structured models\(^1\), symbolic reachability is generally able to explore state spaces that are orders of magnitudes larger than explicit techniques are capable of [BCM+92, vDvdP15].

There exist symbolic algorithms for computing shortest paths [Saw04, EK09]. However, there is as of yet no symbolic algorithm to compute alignments. Since Petri nets tend to have structured state spaces, a symbolic solution may be well-suited for computing alignments. In [BvdPvdA18] we developed a symbolic algorithm for computing alignments and compare its performance to A* by using over 4,000 experiments that exhibit various characteristics. We show that our symbolic algorithm is favourable for larger models when compared to the A* approach, especially in cases where the state space of the Petri net tends to blow up.

Maximizing synchronous moves. In [BvZvdA+18] we argue that the standard cost function is not always the best-suited function for computing alignments. As an alternative, we propose a cost function that instead maximizes the number of synchronous moves. We illustrate this concept in Example 4.2.

---

Example 4.2

Consider the process model in Figure 4.2, with the trace \(\sigma' = \langle B \rangle\). An optimal alignment using the standard cost function would result in \(\gamma^1\). Considering that event \(B\) is observed behaviour, i.e. the system logged \textit{parse file}, it seems illogical to map this behaviour with a path in the model indicating that the file was not found. In case we set up the cost function such that the number of synchronous moves is maximized, an optimal alignment would result in \(\gamma^2\). Here, the \textit{parse file} event is mapped as a synchronous move and model moves are inserted accordingly. One may prefer this function, as it gives an explanation for the logged behaviour.

---

\(^1\)With a structured model, we mean that a model state can be encoded in compact manner, by e.g. using few variables and exploiting locality and symmetries. This keeps the decision diagrams small, which correspondingly reduces the memory and time usage per operation compared to an unstructured model.
Chapter 4. Conformance Checking for Specific Costs

Figure 4.2: Example process model (in Petri net formalism) for a simple file reading system. We provide two optimal alignments for the trace $\sigma = \langle B \rangle$, using the standard ($\gamma^0$) and variant ($\gamma^1$) cost functions.

how different cost functions affect the resulting alignments. Regarding the cost-function where we maximize synchronous moves, we employ a data structure, called a Milestone Transitive Closure Graph (MTCG), that can be computed from the model. Note that the MTCG needs to be computed only once and can be used to align all traces in the log. We propose an algorithm for computing alignments, which exploits the structure and is compatible with the state-of-the-art when computing alignments for small models and many log traces.

**Milestones.** We also introduce the notion of milestone activities, which are events on which the log and the model must agree, i.e. if the event occurs in the log trace, it must be synchronized with the model. There may be several reasons for introducing milestone activities. Milestone activities need to be observed and cannot be skipped. They serve as minimal requirements for an alignment, avoiding that an arbitrary path is taken. To illustrate the concept, reconsider Figure 4.2 and assume activity $E$ to be a milestone activity, i.e. disallow alignments to include the pair ($\gg$, $t_4$). Now, the standard cost function (with milestones) would also result in the desired alignment as the shortcut through the model is now unavailable. Obviously, milestones can make it impossible to align a trace with a model (i.e. in case an alignment must include at least one milestone activity as a model move). Milestones help to address a major limitation of the cost function that maximizes synchronous moves, which occurs when the process model contains a large cycle. We show that milestones can be used to prevent such cycles from being traversed carelessly (by marking a particular event on the cycle as a milestone).

**Transitive closure graph as a diagnostic tool.** Finally, we consider the MTCG structure from a diagnostic point of view. Visualizing alignments by projecting them on the model or its corresponding marking graph may not always be desired. An issue is that model moves are not observed in the log trace, but are
'generated' in the diagnostics. When fabricating events that did not really happen (i.e. model moves), timestamps need to be created. This may lead to misleading results. By mapping alignments on a MTCG we only depict synchronous moves, i.e. only moves that really happened, which are easier to interpret and trust. We discuss the advantages and disadvantages of this method.

4.2 Preliminaries

We assume familiarity with the basics of automata theory and Petri nets (otherwise, we refer to [vdA16]).

A sequence is an ordered list of events, which we denote by \( \sigma = \langle \sigma_0, \sigma_1, \ldots, \sigma_{|\sigma|-1} \rangle \). Two sequences are concatenated using the \( \cdot \) operation. Given a sequence \( \sigma \) and a set of elements \( S \), we refer to \( \sigma \setminus S \) as the sequence without any elements from \( S \), e.g. \( \langle a, b, b, c, a, f \rangle \setminus \{b, f\} = \langle a, c, a \rangle \). For two sequences \( \sigma_1 \) and \( \sigma_2 \), we call \( \sigma_1 \) a subsequence of \( \sigma_2 \) (denoted with \( \sigma_1 \sqsubseteq \sigma_2 \)) if \( \sigma_1 \) is formed from \( \sigma_2 \) by deleting elements from \( \sigma_2 \) without changing its order, e.g. \( \langle c, a, t \rangle \sqsubseteq \langle a, c, r, a, t, e \rangle \). Similarly, \( \sigma_1 \sqsubseteq \sigma_2 \) implies that \( \sigma_1 \) is a strict subsequence of \( \sigma_2 \).

A multiset (or bag) is an unordered set that may contain multiple instances of the same element. For example, \([a^2, b, c^3]\) is a multiset in which \( a \) occurs 2 times (we represent multiple occurrences of an element in superscript). Two multisets are combined with a \( \uplus \) operation, e.g. \([a^2, b, c^3] \uplus [b, c^2, d^2] = [a^2, b^2, c^5, d^2]\). We will use multisets to represent markings and event logs. In figures, we use a more concise notation to denote multisets, e.g. we may write \([a^2, b, c^3]\) as \( a^2bc^3 \). Given a set of elements \( S \), we denote the set of all possible multisets as \( \mathcal{B}(S) \), and its power-set by \( 2^S \).

Throughout this chapter, we will assume the existence of a globally defined alphabet \( \Sigma \) containing actions, which does not contain the skip event (\( \gg \)) nor the invisible action or silent event (\( \tau \)). Traces (or log traces) are sequences \( \sigma \in \Sigma^* \), for which each element is contained in the alphabet \( \Sigma \). An event log \( L \in \mathcal{B}(\Sigma^*) \) is a multiset of traces.

4.2.1 Petri nets

Petri nets are a mathematical formalism that allow us to describe processes, technically they allow us to represent parallel behaviour in a relatively compact manner. Consider Figure 4.2 which is a simple example of a Petri net. The Petri net consists of places, visualized as circles, that allow us to express the state (or marking) of the Petri net. Furthermore, it consists of transitions, visualized as
boxes, that allow us to manipulate the state of the Petri net. It is not allowed to connect a place with a place nor a transition with a transition. Thus, from a graph-theoretical perspective, a Petri net is a bipartite graph.

**Definition 4.1: Petri net, marking**

A Petri net is defined as a tuple $\mathcal{N} = (P, T, F, \Sigma_\tau, \lambda, m_0, m_F)$ such that:

- $P$ is a finite set of places,
- $T$ is a finite set of transitions such that $P \cap T = \emptyset$,
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the flow relation,
- $\Sigma_\tau$ is a set of activities, with $\Sigma_\tau = \Sigma \cup \{\tau\}$ (and $\tau \notin \Sigma$),
- $\lambda : T \to \Sigma_\tau$ is a labelling function for each transition,
- $m_0 \in B(P)$ is the initial marking of the Petri net,
- $m_F \in B(P)$ is the final marking of the Petri net.

A marking is defined as a multiset of places, denoting where tokens reside in the Petri net. For instance, in Figure 4.2, $p_0$ contains a token (represented by a black dot), and the marking $[p_0]$ represents the initial marking. A transition $t \in T$ can be fired if, according to the flow relation, all predecessor places of $t$ contain a token. After firing a transition, one token is removed from these places and all places with an incoming arc from $t$ receive a token. It may be possible for a place to contain more than one token.

**Definition 4.2: Marking graph**

The corresponding marking graph or state space for a Petri net $\mathcal{N} = (P, T, F, \Sigma_\tau, \lambda, m_0, m_F)$ is given by the deterministic automaton $\mathcal{M} = (Q, \Sigma_\tau, \delta, q_0, q_F)$, such that:

- $Q \subseteq B(P)$ is the (possibly infinite) set of vertices in $\mathcal{M}$, which corresponds to the set of reachable markings from $m_0$ (obtained from firing transitions),
- $\delta \subseteq (Q \times T \times Q)$ is the set of edges in $\mathcal{M}$, i.e. $(m, t, m') \in \delta$ iff there is a $t \in T$ such that $m'$ is obtained from firing transition $t$ from $m$,
- $q_0 = m_0$ is the initial state of the graph,
- $q_F = m_F$ is the final state of the graph.

For an edge $e = (m, t, m') \in \delta$, we write $\lambda(e)$ to denote $\lambda(t)$ and use the notation $m \xrightarrow{a} m'$ to represent an edge$^a$ $e$ for which $\lambda(e) = a$. The source and target markings of edge $e$ are respectively denoted by $\text{src}(e)$ and $\text{tgt}(e)$, thus we have $\text{src}(e) = m$, and $\text{tgt}(e) = m'$. We refer to the transition of $e$ by $\text{trans}(e) = t$.

---

$^a$ There may be multiple distinct edges that are represented by $m \xrightarrow{a} m'$. 
4.2. Preliminaries

**Definition 4.3: Path, language**

Given a Petri net $\mathcal{N}$ and its corresponding marking graph $\mathcal{M} = (Q, \Sigma, \delta, q_0, q_f)$, a sequence of edges $\rho = \langle e_0, e_1, \ldots, e_n \rangle \in \delta^*$ is called a path in $\mathcal{N}$ (and $\mathcal{M}$) if it forms a path on the marking graph of $\mathcal{N}$: $\text{src}(e_0) = m_0 \land \text{tgt}(e_n) = m_f \land \forall_{0 \leq i < n} : \text{tgt}(e_i) = \text{src}(e_{i+1})$. The set of all paths in $\mathcal{N}$ is denoted by $\text{Paths}(\mathcal{N})$. With $\lambda(\rho)$ we refer to the sequence of labels visited in $\rho$, i.e. $\lambda(\rho) = \langle \lambda(e_0), \lambda(e_1), \ldots, \lambda(e_n) \rangle^\dagger$. We define the language $L$ of a Petri net $\mathcal{N}$ by $L(\mathcal{N}) = \{ \lambda(\rho) \mid \rho \in \text{Paths}(\mathcal{N}) \}$. We can denote paths and languages for marking graphs directly, i.e. $\text{Paths}(\mathcal{M}) = \text{Paths}(\mathcal{N})$ and $L(\mathcal{M}) = L(\mathcal{N})$.

$^\dagger$ There may be different paths $\rho$ and $\rho'$ such that $\lambda(\rho) = \lambda(\rho')$.

For the construction of alignments, we use the notion of a synchronized product. This product is constructed from a log trace and a model, both represented as a Petri net. We show how a log trace is converted to a Petri net, and how this is synchronized with the model.

**Definition 4.4: Trace to Petri net**

Given a trace $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle \in \Sigma^*$, its corresponding Petri net is defined as $\mathcal{N}_\sigma = (P, T, F, \Sigma, \lambda, m_0, m_f)$ with

- $P = \{ p_0, p_1, \ldots, p_n \}$,
- $T = \{ t_1, t_2, \ldots, t_n \}$,
- $F = \{ (p_0, t_1), (p_1, t_2), \ldots, (p_{n-1}, t_n) \} \cup \{ (t_1, p_1), (t_2, p_2), \ldots, (t_n, p_n) \}$,
- $\Sigma = \{ \sigma_1, \sigma_2, \ldots, \sigma_n \}$,
- $\forall_{1 \leq i \leq n} : \lambda(t_i) = \sigma_i$,
- $m_0 = [p_0]$, and $m_f = [p_n]$.

**Definition 4.5: Product of Petri net model and log trace**

Given a model Petri net $\mathcal{N}_M = (P, T, F, \Sigma, \lambda, m_0, m_f)$ and a Petri net from a log trace $\mathcal{N}_\sigma = (P', T', F', \Sigma', \lambda', m'_0, m'_f)$, the synchronous product of the two is defined as $\mathcal{N} = (P'', T'', F'', \Sigma'', \lambda'', m''_0, m''_f)$ with:

- $P'' = P \cup P'$,
- $T'' = T \cup T' \cup \{ t_1 t_2 \mid t_1 \in T \land t_2 \in T' \land \lambda(t_1) = \lambda(t_2) \}$,
- $F'' = F \cup F' \cup \{ (p, t_1 t_2) \mid t_1 t_2 \in T'' \land \left( (p, t_1) \in F \lor (p, t_2) \in F' \right) \}$ \,$\cup$ \,$\{ (t_1 t_2, p) \mid t_1 t_2 \in T'' \land \left( (t_1, p) \in F \lor (t_2, p) \in F' \right) \}$,
- $\Sigma'' = \Sigma \cup \Sigma'$,
- $\lambda = \lambda \cup \lambda'$ \,$\cup$ \,$\{ t_1 t_2 \rightarrow \lambda(t_1) \mid t_1 t_2 \in T'' \}$,
- $m''_0 = m_0 \uplus m'_0$, and $m''_f = m_f \uplus m'_f$. 
Chapter 4. Conformance Checking for Specific Costs

As we will be interested in the transitions from the model, we refer to synchronous transitions \((t_1t_2)\) by their model-part \((t_1)\). As an example of a synchronous product, we would like to refer to the Petri net from Figure 4.1.

We underline the distinction between the Petri net of the model and the Petri net of the synchronous product. In later sections, we explicitly rely on having either a model (Section 4.4) or a synchronous product (Section 4.5).

4.2.2 Alignments

With alignments we map observed behaviour from an event log onto the behaviour that is described in the reference model. An alignment is represented by a sequence of log-model pairs.

**Definition 4.6: Alignment**

Let \(\sigma \in \Sigma^*\) be a log trace and let \(\mathcal{N}\) be a Petri net model such that \(\mathcal{N} = (P, T, F, \Sigma_\tau, \lambda, m_0, m_F)\), for which we obtain the marking graph \(\mathcal{M} = (Q, \Sigma_\tau, \delta, q_0, q_F)\). We refer to \(\Sigma_{\gg}\) as the alphabet containing skips: \(\Sigma_{\gg} = \Sigma \cup \{\gg\}\) (and \(\tau \notin \Sigma_{\gg}\)). We define \(T_{\gg}\) as the set of transitions and the skip event, i.e. \(T_{\gg} = T \cup \{\gg\}\). Now, let \(\gamma \in (\Sigma_{\gg} \times T_{\gg})^*\) be a sequence of log-model pairs. For \(\gamma = (\gamma_0^0, \gamma_0^1, \ldots, \gamma_{|\gamma|-1}^0, \gamma_{|\gamma|-1}^1)\), we define \(\gamma^\ell\) as the log trace events, i.e. \(\gamma^\ell = (\gamma_0^0, \gamma_1^0, \ldots, \gamma_{|\gamma|-1}^0) \setminus \{\gg\}\) and \(\gamma^m\) as the transitions without skips, by \(\gamma^m = (\gamma_0^1, \gamma_1^1, \ldots, \gamma_{|\gamma|-1}^1) \setminus \{\gg\}\).

We refer to the corresponding sequence of actions from \(\gamma^m\) by \(\lambda(\gamma^m)\), i.e. \(\lambda(\gamma^m) = (\lambda(\gamma_0^1), \lambda(\gamma_1^1), \ldots, \lambda(\gamma_{|\gamma|-1}^1))\).

We call \(\gamma\) an *alignment* if the following conditions hold:

1. \(\gamma^\ell = \sigma\) (the activities of the log-part, equals to \(\sigma\)),
2. \(\lambda(\gamma^m) \in L(\mathcal{N})\) (\(\gamma^m\) forms a path in \(\mathcal{N}\)),
3. \(\forall a \in \Sigma, \forall b \in T: a \neq \lambda(b) \Rightarrow (a, b) \notin \gamma\) (we disallow distinct labels),
4. \((\gg, \gg) \notin \gamma\), (the ‘empty’ move may not exist in \(\gamma\)).

**Definition 4.7: Alignment cost**

Let \(\gamma \in (\Sigma_{\gg} \times T_{\gg})^*\) be an alignment for \(\sigma \in \Sigma^*\) and the Petri net \(\mathcal{N}\). The cost function maps elements of \(\gamma\) onto alignment costs of individual moves, \(c : (\Sigma_{\gg} \times T_{\gg}) \rightarrow (R_{\geq 0} \cup \{\infty\})\), and we overload \(c\) for assigning costs to alignments; \(c : (\Sigma_{\gg} \times T_{\gg})^* \rightarrow (R_{\geq 0} \cup \{\infty\})\), such that \(c(\gamma) = \sum_{i=0}^{|\gamma|-1} c(\gamma_i)\).
4.2. Preliminaries

We call an alignment $\gamma$ under cost function $c$ a successful alignment if $c(\gamma) \neq \infty$, otherwise, if $c(\gamma) = \infty$, we call $\gamma$ a failed alignment. We call a successful alignment $\gamma$ optimal iff $\not\exists \gamma': c(\gamma') < c(\gamma)$, i.e. there does not exist an alignment $\gamma'$ with a smaller cost.

**Definition 4.8: Standard cost function**

The standard cost function $c_{st}$ is defined for an alignment pair $(\ell, m) \in (\Sigma \gg \times T \gg)$ as follows (assuming some small $\varepsilon > 0$):

$$c_{st}(\ell, m) = \begin{cases} \varepsilon & \ell = \gg \land m \in T \land \lambda(m) = \tau \text{ (silent move)} \\ 0 & \ell \in \Sigma \land m \in T \land \lambda(m) = \ell \text{ (synchronous move)} \\ 1 & \ell \in \Sigma \land m = \gg \text{ (log move)} \\ 1 & \ell = \gg \land m \in T \land \lambda(m) \neq \tau \text{ (model move)}. \end{cases}$$

In the standard cost function, we penalize log and model moves equally. Synchronous moves are the preferred choice in alignments, since they are free. We note that silent moves also have a small cost, which is there in case there are so-called $\tau$-cycles in the model ($\tau$ transitions in the model that compose a cycle in the marking graph). An alignment could otherwise contain an infinite sequence of silent moves, by continuously traversing these $\tau$-cycles.

**4.2.3 Symbolic reachability**

**Definition 4.9: Binary decision diagram**

An (ordered) Binary decision diagram (BDD) [Bry86] is a rooted directed acyclic graph that represents a boolean function. Each decision node is labelled by a boolean variable $v_i$ and has two child nodes: a low child and a high child, representing the assignment of respectively False and True to $v_i$. A BDD has two terminal nodes, 0 and 1, and a path from the root to a 0 (or 1) terminal represents a variable assignment for which the represented boolean function evaluates to False (or True, respectively).

A set of Petri net markings can be represented as a BDD by encoding the token count of each place as (possibly multiple) boolean variables. Figure 4.3 gives an example of how a simple 1-safe Petri net (i.e. every place can have at most one token) can be encoded as a BDD. Here, a single boolean variable is assigned to every place in the Petri net. In case the Petri net is not 1-safe, multiple boolean variables are required for representing a set of Petri net markings.
Figure 4.3: From left to right: A simple Petri net model, a BDD representing the state $p_0$, a BDD representing the state $p_1p_2$, a BDD representing the union of states $p_0$ and $p_1p_2$, and a BDD representing the application of transition $A$. Here, green solid arrows represent assignments to True and red dotted arrows are assignments to False.

The size (number of nodes) of a BDD is in the worst-case exponential in the number of boolean variables. In practice, the size greatly depends on the variable ordering, i.e., choosing to decide on variable $v_i$ before deciding $v_j$ may greatly reduce the BDD size as a result of locality or symmetries.

Set operations such as $A \cup B$ are possible with BDDs, an example of this is given in Figure 4.3. It is also possible to compute successors, by encoding a transition relation in the form of a BDD (e.g., the rightmost image). This relation then checks (part of) a variable assignment and, by using auxiliary variable nodes (in the example, $q_0$, $q_1$, and $q_2$), sets the corresponding assignment for the successors. Given marking $[p_0]$ in the example, applying transition $A$ and renaming the variables results in $[p_1,p_2]$. If we combine both markings, we obtain the set of markings $\{[p_0],[p_1,p_2]\}$. This makes it possible to perform reachability in a symbolic fashion. However, note that most operations on BDDs are expensive in the sense that they have linear or quadratic complexity in the number of nodes, and exponential in the number of variables.

To represent Petri nets that are not 1-safe, instead of using multiple boolean variables for representing a single place, we may also use List Decision Diagrams (LDDs) [BvdP08, vDvdP15]. Using an LDD, the possible values for a place can be
represented by a list of values. Each of these assignments is then combined with an edge that points to the LDD node of the next place. This structure will be useful for traversing the state space of a Petri net, as the LDD nodes are dynamically sized – we do not have to specify beforehand what the maximum number of tokens is for each place.

4.3 Synchronous moves and milestones

The standard cost function from Section 4.2.2 is the most commonly used cost function in literature [vdA16, vZBvD17, Adr14], though note that any cost function could be used (e.g. see [KSLR17] for an alternative). The standard cost function may lead to undesired results, as illustrated by the example from Figure 4.2. We consider a new cost function that maximizes the number of synchronous moves, since it explains as many log moves as possible. We define the alternative cost function as follows.

**Definition 4.10: The max-sync cost function**

We define the max-sync cost function \( c_{\text{sync}} \) for an alignment pair as follows (assuming some small \( \varepsilon > 0 \)):

\[
c_{\text{sync}}(\ell, m) = \begin{cases} 
\varepsilon & \ell = \ggg \land m \in T \land \lambda(m) = \tau \text{ (silent move)} \\
0 & \ell \in \Sigma \land m \in T \land \lambda(m) = \ell \text{ (synchronous move)} \\
1 & \ell \in \Sigma \land m = \ggg \text{ (log move)} \\
\varepsilon & \ell = \ggg \land m \in T \land \lambda(m) \neq \tau \text{ (model move)}. 
\end{cases}
\]

This cost function mainly penalizes log moves, which as a consequence causes an optimal alignment to minimize the number of log moves and thus maximize the number of synchronous moves. The \( \varepsilon \) cost for model moves further filters optimal alignments to only include shortest paths through the model that maximize synchronous moves. One should note that the combined total of all \( \varepsilon \)-cost moves may accumulate to a value greater than 1. This can influence the resulting alignment if we face the choice of taking a log move versus taking more than \( \frac{1}{\varepsilon} \) model moves. We assume that in practice, \( \varepsilon \) is set to a considerably low value such that a non-infinite sequence of model moves is preferred over taking an additional log move.

An advantage of the max-sync cost function over the standard cost function is that observed behaviour is not sacrificed for shorter paths through the model (as Figure 4.2 illustrates). A disadvantage is that in order to maximize the number
Chapter 4. Conformance Checking for Specific Costs

of synchronous moves, it may be possible that many model moves are required, especially in case the model contains cycles. Consider for example the Petri net model from Figure 4.4. We are always able to synchronize all observed behaviour by executing the loop. By simply taking the reset transition one can traverse the model again to find a particular event, and report the steps to reach it as model moves. Therefore, we introduce the notion of milestone activities.

4.3.1 Milestones

To mitigate the problem with undesired alignments, we define milestones as labels that may not be chosen in model moves. Note that we do not assign an infinite cost to log moves with milestone actions.

**Definition 4.11: Milestones**

Given a Petri net \( N = (P, T, F, \Sigma_T, \lambda, m_0, m_F) \) and a cost function \( c \), milestones are labels \( Y \subseteq \Sigma \), such that a new cost function \( c' \) is formed, which sets the cost for all model moves with a milestone label to infinity, i.e.

\[
\forall (\ell, m) \in (\Sigma_{\Rightarrow} \times T_{\Rightarrow}) : c'(\ell, m) = \begin{cases} 
\infty & \text{if } \ell = \Rightarrow \land m \in T \land \lambda(m) \in Y \\
c(\ell, m) & \text{otherwise.}
\end{cases}
\]

Consider the example model in Figure 4.4 and assume we compute an alignment with the \( c_{\text{sync}} \) cost function. If we choose the reset action to be the only milestone, then the reset is only scheduled if it synchronizes with a log move. Therefore, an optimal alignment includes exactly as many reset actions in the model as there are in the log. Note that this is not necessarily the case if we employ a different cost function.

An interesting consequence of having milestones in the model, is that not every
alignment is a successful one. If we assign the start action in Figure 4.4 as a milestone, and align it with a log trace that does not contain a start event, then the alignment cost would be infinite and therefore we call the alignment failed.

### 4.3.2 Relating the model and event log

We consider scenarios in which particular knowledge about the model and/or event log is present. We may exploit such instances to simplify the conformance checking procedure.

Given a Petri net model $\mathcal{N}$ and an event log $L \in \mathcal{B}(\Sigma^*)$, we distinguish four cases based on the languages that they describe. By distinguishing the relative granularities of $\mathcal{N}$ and $L$ we define four cases of alignment problems as follows.

**C1:** $\forall \sigma_1 \in L : (\exists \sigma_2 \in \mathcal{L}(\mathcal{N}) : \sigma_1 = \sigma_2)$; all log traces correspond to paths in the model. Then, every log trace can be mapped onto the model by only using synchronous and silent moves, which is optimal for both $c_{st}$ and $c_{sync}$.

**C2:** $\forall \sigma_1 \in L : (\exists \sigma_2 \in \mathcal{L}(\mathcal{N}) : \sigma_1 \sqsubseteq \sigma_2)$; all log traces correspond to subsequences of paths in the model. Then, every log trace can be mapped onto the model without using any log moves. The example model from Figure 4.2 for $\sigma = \langle B \rangle$ is such an instance.

**C3:** $\forall \sigma_1 \in L : (\exists \sigma_2 \in \mathcal{L}(\mathcal{N}) : \sigma_2 \sqsubseteq \sigma_1)$; for every log trace there is a path in the model that forms a subsequence of the log trace. Then, every log trace can be mapped onto the model without using any model move.

**C4:** None of the above properties hold. All move types may be necessary for alignments. We regard this as a standard scenario. Depending on the use case, either $c_{st}$ or $c_{sync}$ could be preferred.

We note that each of these cases could also be considered on a trace level, where e.g. we could have an event log that strictly contains log traces of the form C1 and C3. Aside from C4, we consider cases C2 and C3 as common instances in practice, as logging software often causes either too many or too little events to be logged or in case the model is over/underspecified. Discrepancies then show whether the model is of the right granularity. That is, assuming that the model is correct of course, since the model itself may also contain incorrect behaviour. We note that it is also possible to hide certain activities in the model or log before alignment. This is however not trivial, especially if there are (slight) deviations in the log such that the alignment problem does not fit C2 or C3 exactly any more.

When considering instances that exactly fit case C2 or C3, we are able to construct
alignments by respectively removing all log or model moves from the product of the model and log. We define the cost functions $c_{\text{add}}$ and $c_{\text{rem}}$ to be variants of $c_{\text{st}}$ such that model and log moves respectively have a cost of $\infty$, i.e.:

$$
c_{\text{add}}(\ell, m) = \begin{cases} 
\varepsilon & \ell = \gg \land m \in T \land \lambda(m) = \tau \text{ (silent move)} \\
0 & \ell \in \Sigma \land m \in T \land \lambda(m) = \ell \text{ (synchronous move)} \\
1 & \ell \in \Sigma \land m = \gg \text{ (log move)} \\
\infty & \ell = \gg \land m \in T \land \lambda(m) \neq \tau \text{ (model move)} 
\end{cases}
$$

$$
c_{\text{rem}}(\ell, m) = \begin{cases} 
\varepsilon & \ell = \gg \land m \in T \land \lambda(m) = \tau \text{ (silent move)} \\
0 & \ell \in \Sigma \land m \in T \land \lambda(m) = \ell \text{ (synchronous move)} \\
\infty & \ell \in \Sigma \land m = \gg \text{ (log move)} \\
1 & \ell = \gg \land m \in T \land \lambda(m) \neq \tau \text{ (model move)} 
\end{cases}
$$

Note that the resulting alignments for $c_{\text{add}}$ and $c_{\text{rem}}$ will still be successful for cases C3 and C2 respectively. The cost functions may then be used to guide the alignment creation process, by omitting model or log moves completely from the synchronized product.

### 4.4 Preprocessing the reference model

When constructing an alignment under the $c_{\text{sync}}$ cost function, we can disregard the cost for model moves to a certain extent. The goal is to find a path through the model that maximizes the number of synchronous moves. We can achieve this by searching for a subsequence in the log trace that is also included in the language of the model. By computing the transitive closure of the model’s marking graph (note that we do not compute the synchronous product), we find all paths and subsequences of paths through the model. We extend the structure from [BvZvdA+18] to include milestones, which we call a **Milestone Transitive Closure Graph**, or MTCG. Given a log trace, we can use dynamic programming to search for the maximum-length subsequence that can be replayed in the MTCG. From this subsequence we can construct a path through the marking graph and obtain an optimal alignment.

#### 4.4.1 Milestone transitive closure graph

We construct an MTCG as described in Definition 4.12. Here, $\tau$-edges are added for every edge in the marking graph, except for milestone edges (which cannot be skipped), such that, in case there are no milestones, every marking is reachable via
4.4. Preprocessing the reference model

\( \tau \)-steps from the initial marking set. After determinization, for every path \( P \) in the original marking graph the MTCG contains a path \( P' \) such that \( \lambda(P') = \lambda(P) \).

In case there are no milestones, the MTCG also contains every path \( P' \) such that \( \lambda(P') \sqsubseteq \lambda(P) \) holds, that is, any subsequence of a path’s labels can be formed in the MTCG.

**Definition 4.12: Milestone Transitive Closure Graph**

Given a marking graph \( \mathcal{M} = (Q, \Sigma, \delta, q_0, Q_F) \) and a set of milestones \( Y \), we first construct an extended marking graph \( \mathcal{M}' = (Q, \Sigma, \delta', q_0, Q_F) \), with

\[
\delta' = \delta \cup \{ (\text{src}(e), \tau, \text{tgt}(e)) \mid e \in \delta \land \lambda(e) \notin Y \land \lambda(e) \neq \tau \}.
\]

A *Milestone Transitive Closure Graph (MTCG)* \( \mathcal{M} = (Q, \Sigma, \Delta, Q_0, Q_F) \) is defined as the result of determinizing \( \mathcal{M}' \) (by using a standard determinization algorithm [Sud88]) such that:

- \( Q \subseteq 2^Q \),
- \( \Sigma = \Sigma_T \setminus \{ \tau \} \),
- \( \Delta \subseteq (Q \times \Sigma \times Q) \),
- \( Q_0 \subseteq Q \), and
- \( Q_F = \{ F \mid F \in Q \land q_F \in F \} \).

Given an MTCG \( \mathcal{M} = (Q, \Sigma, \Delta, Q_0, Q_F) \), for an edge \( e \in \Delta \) we also use the notation \( \text{src}(e) \) and \( \text{tgt}(e) \) to respectively refer to the source and target marking sets in the MTCG. Given an MTCG \( \mathcal{M} \), paths over \( \mathcal{M} \) are defined analogously to paths over marking graphs (see Definition 4.3) and we use \( \text{Paths}(\mathcal{M}) \) and \( \mathcal{L}(\mathcal{M}) \) to respectively denote the set of all paths in \( \mathcal{M} \) and the language of \( \mathcal{M} \).

In Figure 4.5, we give an example of an MTCG with milestones \( B \) and \( D \). Let us first consider the example without milestones. Because the marking graph consists of a single strongly connected component, every marking becomes reachable from every other marking. Therefore, if we compute its MTCG without any milestones, we obtain a single state with a self-loop for each label. Trivially, all paths and subsequences of paths are preserved in this structure, however, this is rather useless in practice.

If we add the milestones \( B \) and \( D \) to the model, we obtain a more interesting MTCG. An observation is that the initial state is no longer an accepting one, as we can only reach the \( p_2 \) marking by firing a \( B \) and a \( D \) action. Other events can still be skipped, however, hence we can directly take a \( B \) action from the initial state to \( Q_2 \), but taking an \( E \) action is not allowed. As a result, we ensure that paths through the MTCG may not skip milestone actions, and are forced to take these actions if there is no alternative path to reach a final state (i.e. \( Q_4 \)).
4.4.2 Searching for optimal subsequences

Given a log trace $\sigma$, we use the MTCG of a model to obtain a subsequence of the log trace $\hat{\sigma} \sqsubseteq \sigma$, such that $\hat{\sigma}$ fully synchronizes with the model. From the construction of the MTCG we have the following property.

**Lemma 4.1: Paths in an MTCG synchronize with the model**

Given a Petri net model $N$, its marking graph $M$, and its corresponding MTCG $M$. If $\rho$ is a path in $M$, then there exists a path $\rho'$ in $N$ such that $\lambda(\rho) \sqsubseteq \lambda(\rho')$. Thus, $\rho \in Paths(M) \Rightarrow \exists \rho' \in Paths(N) \land \lambda(\rho) \sqsubseteq \lambda(\rho')$.

**Proof.** This follows from the construction of an MTCG. An MTCG $M$ is formed from a marking graph $M$ by adding $\tau$ actions to existing edges. Therefore, the only additional behaviour that $M$ has compared to $M$ is to replace visible actions by $\tau$ actions in its paths. Hence, any path in $M$ forms a subsequence of a path in $M$, i.e. $\forall \rho \in \text{Paths}(M), \exists \rho' \in \text{Paths}(M) : \lambda(\rho) \sqsubseteq \lambda(\rho')$. \qed
4.4. Preprocessing the reference model

We use the result of Lemma 4.1 to search for a subsequence of a log trace that can fully synchronize with the model. For instance in the example of Figure 4.6 (without milestones), consider a log trace \( \sigma = \langle F, D, E, B \rangle \). Note that \( \sigma \) does not correspond to a path through the model. The \( F \) event can be fired from \( Q_0 \), after which \( M \) is in the state \( Q_{10} \). From this state, it is not possible to perform any other event from log trace. A better choice is to skip the \( F \) event (which results in a log move) and form the subsequence \( \langle D, E \rangle \), as highlighted\(^2\). We call the maximum-length subsequence \( \hat{\sigma} \) from a log trace a maximum fitting subsequence if \( \hat{\sigma} \) also forms a path through the MTCG, as defined in Definition 4.13.

**Definition 4.13: Maximum fitting subsequence**

Given a sequence (log trace) \( \sigma \in \Sigma^* \) and an MTCG \( M = (Q, \Sigma, \Delta, Q_0, Q_F) \), then \( \hat{\sigma} \subseteq \sigma \) is a maximum fitting subsequence if and only if \( \hat{\sigma} \in L(M) \land \forall \hat{\sigma}' \subseteq \sigma : \hat{\sigma}' \in L(M) \Rightarrow |\hat{\sigma}| \geq |\hat{\sigma}'| \).

Given a log trace \( \sigma \) and an MTCG \( M = (Q, \Sigma, \Delta, Q_0, Q_F) \), we construct a maximum fitting subsequence \( \hat{\sigma} \) by using dynamic programming to search for a sub-

\(^2\) Note that after performing the \( D \) action in the MTCG, in the Petri net we have not yet made the choice to fire either an \( A \) or a \( B \) transition; we implicitly make the decision to fire the \( B \) transition after choosing the \( E \) event.
sequence of $\sigma$ that forms a maximum-length path in $M$. A straightforward implementation of this is by a 2D array ($|\sigma| \times |Q|$) of sequences. We then iterate over each element $\sigma_i$ from the log trace and track the paths in $M$ that are formed by choosing to include $\sigma_i$ or deciding to skip it (in case it is not a milestone). Subsequently we return the largest sequence that ends in a final state.

Once we have found the maximum fitting subsequence $\hat{\sigma}$ for a given model and log trace, we still have to determine which model moves should be applied to form a path through the original model. This is achieved by using the MTCG and $M$, by traversing backwards through $\hat{\sigma}$ as we show in Algorithm 13.

We assume a maximum fitting subsequence $\hat{\sigma} \neq \langle \rangle$ (otherwise, there are no synchronous moves and an optimal alignment would simply consist of a shortest path through the model). We first construct a Maximal Fitting Path (MFP) from the subsequence $\hat{\sigma}$ (line 2). In the example from Figure 4.6 with $\hat{\sigma} = \langle D, E \rangle$ (see also Figure 4.7 for an illustration of the path construction process), this is $\text{MFP} = \langle (Q_0, D, Q_8), (Q_8, E, Q_9) \rangle$. Then in line 3, a backward search procedure ($\text{BWD}$) is called to search for a path $P$ in the marking graph from an $E$-edge to the final marking $[p_7]$.

The $\text{BWD}$ procedure takes a target marking $m$, label $a$ and search space $S$ as arguments. A sequence $W$ is maintained to process unvisited markings from $S$ and a mapping $F : Q \rightarrow (\hat{\delta} \cup \{\perp\})$ is used for reconstructing the path. Starting from the target marking $m$ (which is $W_0$), the procedure searches for edges $e$ directing towards $m$ in line 16-18 such that $\text{src}(e)$ is in $S$ and not already visited. For every such edge $e$, its source is appended to $W$ (to be considered in a future iteration) and $\text{src}(e)$ is mapped to $e$ for later path reconstruction.

The following iterations of the for loop in line 16-18 consider a predecessor $W_i$ of $m$ and search for edges directing to $W_i$. In this way, the search space is traversed backwards in a breadth-first manner, resulting in shortest paths to $m$.

In line 12-15 the $\text{BWD}$ procedure checks whether there is an edge $m' \xrightarrow{a'} W_i$ for some $m'$ (or an edge $q_0 \xrightarrow{a'} W_i$ for arbitrary $a'$ in case $a = \perp$) and if so, constructs a path towards $m$ in line 14 which is then returned. In the example, the path $\langle ([p_3, p_5], E, [p_5, p_6]), ([p_5, p_6], G, [p_7]) \rangle$ will be returned for the first $\text{BWD}$ call.

After the first $\text{BWD}$ call, the main function iterates backwards over all remaining edges from $\text{MFP}$ (line 4-5) to create paths between $\hat{\sigma}_i$ and $\hat{\sigma}_{i+1}$, which are inserted in the path before $P$. Finally, in line 6 a path from the initial marking $[p_0]$ towards the first label $\hat{\sigma}_0$ is inserted before $P$ to complete the path (here the label is set to $\perp$ to search for $[p_0]$ in the $\text{BWD}$ call).
4.4. Preprocessing the reference model

Figure 4.7: Path construction using Algorithm 13 on the example from Figure 4.6 for a maximum fitting subsequence \( \hat{\sigma} = \langle D, E \rangle \subset \langle F, D, E, B \rangle \). Markings in the grey region are not part of the path. The resulting alignment \( \gamma \) is shown right.

**Algorithm 13:** Path construction from a maximum fitting subsequence \( \hat{\sigma} \)

```java
function PC \( M = (Q, \Sigma, \Delta, Q_0, q_F), M = (Q, \Sigma, \hat{\sigma}, q_0, q_F), \hat{\sigma} = (\hat{\sigma}_0, \hat{\sigma}_1, \ldots, \hat{\sigma}_n) \)
// construct path Maximal Fitting Path (MFP) on \( M \) such that \( \lambda(MFP) = \hat{\sigma} 
MFP := \langle (Q, \hat{\sigma}_0, S), (S, \hat{\sigma}_1, S'), \ldots, (S'', \hat{\sigma}_n, S''') \rangle \) s.t. \( \forall 0 \leq i < n : \) MFP\(_i \in \Delta \)
\( P := \text{BWD}(M, q_F, \hat{\sigma}_n, \text{tgt}(MFP_n)) // \) path \( \hat{\sigma}_n \) to \( q_F \) on \( M \)
for \( i := n - 1 ; i \geq 0 ; i := i - 1 \) do // add paths from \( \hat{\sigma}_i \) to \( \hat{\sigma}_{i+1} \)
    \( P := \text{BWD}(M, \text{src}(P_i), \hat{\sigma}_i, \text{tgt}(MFP_i)) \cdot P \)
return \( \text{BWD}(M, \text{src}(P_0), \perp, Q_0) \cdot P // \) add path from \( q_0 \) to \( \hat{\sigma}_0 \)

\( W := \langle m \rangle // \) sequence of unvisited markings in the backward search
\( \forall m \in S : F[m] := \perp // \) mapping from markings to edges; \( F : Q \rightarrow (\delta \cup \perp) \)
i := 0
while \( i < |W| \) do // continue for all markings in \( W \)
    if \( \exists m' \in Q, a' \in \Sigma : (m', a', W_i) \in \delta \land (a' = a \lor (a = \perp \land m' = q_0)) \) then
        \( P := \langle (m', a', W_i) \rangle // \) found path from \( a \) (or initial marking)
        while \( \text{tgt}(P_{|P|-1}) \neq m \) do \( P := P \cdot F[\text{tgt}(P_{|P|-1})] \)
        return \( P // \) shortest path from \( a \) (or \( q_0 \)) to \( m \)
forall \( e \in \delta : \text{src}(e) \in (S \setminus W) \land \text{tgt}(e) = W_i \) do
    \( W := W \cup \{ \text{src}(e) \} // \) add predecessor markings of \( m \) to \( W \)
    \( F[\text{src}(e)] := e // \) direct the source markings towards \( m \)
i := i + 1
return {} // no path from \( a \) (or \( q_0 \)) is found (should never occur)
```

In the example, we first compute the path \( \langle [p_3, p_5], E, [p_5, p_6], G, [p_7] \rangle \) in line 3, then after line 4-5 we insert the path \( \langle [p_2, p_3], D, [p_3, p_5] \rangle \). In line 6 we insert the path from the initial state \( q_0 = [p_0], \langle [p_0], B, [p_2, p_3] \rangle \), to create the complete minimal-length path \( P \) in the marking graph such that \( \hat{\sigma} \subseteq \lambda(P) \), i.e.

\[
P = \langle [p_0], B, [p_2, p_3], [p_2, p_3], D, [p_3, p_5], [p_3, p_5], E, [p_5, p_6], [p_5, p_6], G, [p_7] \rangle.
\]
Alignment construction. The alignment can be reconstructed by marking all events in the maximum fitting subsequence as synchronous moves, by marking the remaining labels in the log trace as log moves, and inserting the model and silent moves (as computed by Algorithm 13) at the appropriate places. Milestones cannot be taken as model moves because all milestone edges are explicitly present in the MTCG. Therefore, a milestone action is only taken if it is part of the maximum fitting trace.

Note that the MTCG algorithm does not exactly compute an alignment for the cost function \( c_{\text{sync}} \). The backwards BFS does ensure a shortest path through the model from the initial to the final marking while synchronizing with the maximum fitting subsequence. However, there might exist another maximum fitting subsequence that leads to an alternative path through the model with a lower total cost (i.e. fewer model moves). This can be repaired by computing the alignments for all maximum fitting subsequences. If we consider a cost function \( c_{\text{sync}}' \), where silent- and model moves have cost 0, then the MTCG does compute an optimal alignment.

If the marking graph contains cycles, the corresponding markings get contracted to a single state in the MTCG with a self-loop for each activity in the cycle (as we showed in Figure 4.5), which may be prevented by adding milestones. Also, the MTCG may contain exponentially more states than there are markings in the marking graph. As a result, we hypothesize that the MTCG algorithm is applicable for small-sized models in combination with many log traces, but it may be ineffective for large models.

4.5 Symbolic alignment computation

In the previous section, we introduced a data structure and algorithm that may be used to compute many alignments for the same model efficiently. Here, we introduce a symbolic algorithm that can compute alignments for large synchronous products.

We first discuss the main idea of our symbolic alignment algorithm and an improvement to the base version. Then, the detailed algorithm is explained, an example of its usage is given and we show its correctness.

4.5.1 General idea of the symbolic algorithm

For now, we consider a variant of the standard cost function to simplify the algorithm explanation, where silent moves have a cost of 0 (instead of \( \varepsilon \), we later show how the algorithm can be extended to compute alignments for the complete cost
4.5. Symbolic alignment computation

The idea of the algorithm is to split up the Petri net transitions of the *synchronous product* into two groups, a group that only consists of 0-cost moves (in the simplified standard cost function: \( \tau \)-moves and synchronous moves) and 1-cost moves (in the cost function: model and log moves). We call these groups respectively \( T_0 \) and \( T_1 \). We present an algorithm that is only applicable for uniform cost functions, i.e., cost functions that only assign a cost of 0 or 1 to the different moves. We will later show extensions to suit more generalized variants.

A decision diagram is used to symbolically store a set of markings. Moreover, it is possible to apply transitions on such sets to compute a new set that contains all successors for every marking in the original set.

With the \( T_0 \) and \( T_1 \) groups, we can compute a shortest path from the initial marking to the final marking (in the marking graph of the synchronous product) by first considering all reachable markings when we repeatedly apply \( T_0 \) transitions. We thus create a transitive closure with \( T_0 \) operations, which we denote as \( T_0^\ast \). After applying \( T_0^\ast \) on the initial marking we obtain the set \( S_1 \) of all markings that are reachable by only applying 0-cost steps. Then, on the set \( S_1 \) we apply a *single* \( T_1 \) step, for which we obtain the set \( S_2 \). On \( S_2 \) we again apply \( T_0^\ast \) to obtain \( S_3 \), which is the set of markings that can be reached from the initial states with at most 1 cost.

This process is repeated until the smallest \( n \) such that \( S_n \) contains the final marking. Once this is reached, a path can be constructed from the final marking to the initial marking by reverting the symbolic operations. Note that this path is also a shortest path (with minimal cost) from the initial marking to the final marking, where the cost equals the number of \( T_1 \) transitions taken. Also note that we assume that the final marking is reachable via finitely many steps from the initial marking, with the consequence that each set \( S_i \) contains a finite number of markings.

### 4.5.2 Bidirectional search

Instead of just searching from the initial marking to the final marking, we can also search from the final marking to the initial marking by taking the transitions in the opposite direction. When the forward and backward search have a non-empty intersection, a shortest path can be constructed. A shortest path is formed by selecting a marking from the non-empty intersection and reverting the symbolic operations from this marking to get back to the initial and final marking.

The illustration from Figure 4.8 depicts the procedure. The intuition for searching in both directions is that the total number of visited states (which may relate to the number of decision nodes) is reduced this way. In our preliminary experiments we
found that it is more time and memory efficient to search bidirectionally compared to only searching in the forward direction.

Instead of switching search directions after each iteration, preliminary experiments showed that it is more beneficial to continue searching on the set that has the smallest number of nodes in the decision diagrams. The algorithm first performs a $T_0^*$ step in both directions, after which it applies a $T_1$ transition followed by a $T_0^*$ application on the set with the smallest number of nodes. It then again performs a $T_1$-$T_0^*$ step on the set with the smallest number of nodes until it terminates. In general, the number of nodes in the decision diagram correlates with the time required for performing a symbolic operation.

4.5.3 Detailed algorithm

Here we discuss Algorithm 14. We first discuss the auxiliary functions DoTrans, which computes the successors after applying either a $T_0$ or $T_1$ transition, and $T0Closure$, which computes the transitive closure for $T_0$ transitions.

Given a set of current states Cur, the DoTrans function attempts to fire all transitions in $T_X$ (which is either $T_0$ or $T_1$) to form a set of successor or predecessor states N, which is returned. The for loop in lines 3 to 5 iterates over all transitions in $T_X$ and depending on the current direction (dir) computes the successors or predecessors from Cur by attempting to fire the transition t on all markings in Cur. The set of all successors is then stored in N and the union of all successors is returned.
Algorithm 14: Symbolic alignment algorithm

1 function $DoTrans\ (Cur \subseteq B(P), \ T_X \in \{T_0, T_1\}, \ dir \in \{FWD, BWD\})$
2 \hspace{1em} N := \emptyset
3 \hspace{1em} forall \ t \in T_X \ do
4 \hspace{2em} if \ dir = FWD \ then \ N := N \cup Succ(Cur, t)
5 \hspace{2em} else \ N := N \cup Prev(Cur, t)
6 \hspace{1em} return \ N

7 function $T0Closure\ (Cur \subseteq B(P), \ T_0, \ V_{FWD}, \ V_{BWD}, \ dir \in \{FWD, BWD\})$
8 \hspace{1em} N := Cur
9 \hspace{1em} while \ N \neq \emptyset \land V_{FWD} \cap V_{BWD} = \emptyset \ do \ // \ continue \ until \ we \ find \ no \ new \ states
10 \hspace{2em} N := DoTrans(Cur, T_0, dir) \setminus V_{dir} \ // \ T_0 \ transition \ (only \ successors)
11 \hspace{2em} V_{dir} := V_{dir} \cup N
12 \hspace{2em} Cur := N
13 \hspace{1em} return \ V_{dir}

14 function $SymAlign\ (\mathcal{N} = (P, T, F, \Sigma_I, \lambda, m_0, m_F), \ T_0 \subseteq T, \ T_1 \subseteq T)$
15 \hspace{1em} V_{FWD} := \{m_0\} \ // \ forward \ visited \ set
16 \hspace{1em} V_{BWD} := \{m_F\} \ // \ backward \ visited \ set
17 \hspace{1em} V_{FWD} := T0Closure(V_{FWD}, T_0, V_{FWD}, V_{BWD}, FWD)
18 \hspace{1em} if \ V_{FWD} \cap V_{BWD} \neq \emptyset \ then \ return \ trace \ // \ 0-cost \ alignment
19 \hspace{1em} V_{BWD} := T0Closure(V_{BWD}, T_0, V_{FWD}, V_{BWD}, BWD)
20 \hspace{1em} do
21 \hspace{2em} if \ BDD\_Nodes(V_{FWD}) < BDD\_Nodes(V_{BWD}) \ then \ dir := FWD
22 \hspace{2em} else \ dir := BWD \ // \ dir \in \{FWD, BWD\}
23 \hspace{2em} N := DoTrans(V_{dir}, T_1, dir) \setminus V_{dir} \ // \ T_1 \ transition \ (only \ successors)
24 \hspace{2em} V_{dir} := V_{dir} \cup N \ // \ add \ successors \ to \ V_{FWD} \ or \ V_{BWD}
25 \hspace{2em} V_{dir} := T0Closure(N, T_0, V_{FWD}, V_{BWD}, dir)
26 \hspace{2em} if \ V_{FWD} \cap V_{BWD} \neq \emptyset \ then \ return \ trace
27 \hspace{1em} while \ N \neq \emptyset \ // \ continue \ until \ we \ find \ no \ new \ states
28 \hspace{1em} return no-trace \ // \ m_F \ is \ unreachable \ from \ m_0
The $T_0$Closure function repeatedly calls the DoTrans function to compute $T_0^*$, i.e. the transitive closure of applying $T_0$ transitions. The set of successors $N$ is computed in line 10 and used as the current set in the following iteration. Note that already visited states are removed (with the \$V_{dir}$ operation) and newly found states are added to $V_{dir}$. The function returns when either there are no new successors ($N = \emptyset$) or when a shortest path is found ($V_{FWD} \cap V_{BWD} \neq \emptyset$).

The main function, SymAlign works as follows. The forward and backward visited sets are initialized in lines 15 and 16. From the initial state, the $T0$Closure is called to perform a forward $T_0^*$ application. The set $V_{FWD}$ now contains all states reachable from the initial state via $T_0$ transitions.

It may be possible that an optimal alignment exists with a total cost of 0, then the final state should be included in $V_{FWD}$ and hence $V_{FWD} \cap V_{BWD} \neq \emptyset$. If this is the case, then a trace can be reconstructed and returned (line 18). See Section 4.5.5 for more information on the trace reconstruction. If this is not the case, the $T0$Closure is called from the final state in the backwards direction (line 19).

In lines 20 to 27 the algorithm continuously performs a $T_1$ transition followed by a $T_0^*$ application until it either found and returned a shortest path or found no new states with a $T_1$ transition.

In lines 21 to 22 the algorithm decides on which direction to search. The number of decision diagram nodes for $V_{FWD}$ and $V_{BWD}$ are compared (with the BDD_Nodes function) and the one with the fewest number of nodes is chosen for the current iteration.

The $T_1$ successors or predecessors are computed for all states in $V_{dir}$ in line 23 (note that $N$ only contains new states). The visited set is updated and a $T_0^*$ application is performed. If there is a non-empty intersection between the forward and backward search, a shortest path is found and returned (line 26). Otherwise, the algorithm continues with a next iteration.

If there is no path from the initial to the final state, the entire state space gets explored (assuming this is finite) and no new states can be found, after which the loop ends and the function returns no-trace. Note that the two while-loops always terminate on finite state spaces, either when a path to the final marking is found, or when no new successors are found after trying to fire transitions from $T_0$ or $T_1$.

**Example 4.3: A run of the algorithm**

Figure 4.9 depicts the marking graph for the Petri net of Figure 4.1 and the numbers in the nodes denote the exploration order by Algorithm 14. The initial and final states are respectively $m_0$ and $m_{37}$. Algorithm 14 starts by first computing $T_0^*$ from the initial state. It finds $m_4$ via the synchronous A
4.5. Symbolic alignment computation

Figure 4.9: Marking graph for the example of Figure 4.1. The numbers in the nodes denote the order of exploring the marking graph by Algorithm 4. Green nodes denote a forward search, red nodes denotes a backward search, and yellow nodes denote the intersection between the forward and backward search.

transition. Then, no more 0-cost moves are possible. The same procedure is then started from the final state. No 0-cost transitions can be taken to \( m_{37} \). In this example, for simplicity, we assume that the number of markings equals the number of decision nodes.

Then, since the backward set contains fewer markings (decision nodes), a \( T_1 \) transition is taken towards \( m_{37} \) to obtain states \( m_{35} \) and \( m_{36} \). Then \( T_0^* \) is computed. States \( m_{28} \) and \( m_{34} \) are found in the first iteration, then \( m_{23} \) is encountered, then \( m_{13} \), and finally \( m_3 \). After these \( T_0 \) iterations, no other 0-cost transition can be taken from the backward set.

Now, the forward set contains fewer decision nodes. A \( T_1 \) transition is taken to obtain the states \( m_1, m_2, m_8, \) and \( m_9 \). After applying \( T_0^* \), we obtain \( m_{18} \). The forward set again contains fewer decision nodes, thus we take a \( T_1 \) transition from the forward set. We find states \( m_3, m_5, m_{12}, m_{13}, m_{14}, m_{15}, m_{23}, \) and \( m_{24} \).

Since we now have a non-empty intersection between the forward and back-
ward search, we found a shortest path and can compute the corresponding alignment. By taking the path \( m_9 \xrightarrow{A} m_4 \xrightarrow{B} m_8 \xrightarrow{C} m_{18} \xrightarrow{D} m_{23} \xrightarrow{E} m_{28} \xrightarrow{F} m_{35} \xrightarrow{G} m_{37} \), we find the optimal alignment \( \gamma \) (which is equal to the one given in Figure 4.1).

### 4.5.4 Correctness

To show that Algorithm 14 is correct, we prove that the algorithm always returns a trace if there is a path from the initial to the final marking (Theorem 4.1), and that a returned trace forms an optimal alignment (Theorem 4.2).

#### Theorem 4.1: Completeness

Given a synchronous product Petri net \( N = (P, T, F, \Sigma, \lambda, m_0, m_F) \) and a uniform cost function \( c \), if there is a finite path from \( m_0 \) to \( m_F \), then Algorithm 14 returns a trace for \( N \) and \( c \).

**Proof.** Assume that there is a path from \( m_0 \) to \( m_F \) and assume that the algorithm does not return a trace. We consider two cases: (1) The algorithm terminates and does not return a trace, and (2) the algorithm does not terminate.

(1) The algorithm terminates, thus it has finished exploring the state space. Note that \( T_0 \cup T_1 \) forms the set of all transitions and that the algorithm performs a \( T_1 \) and \( T_0^* \) application in each step (aside from the first one). Therefore, both the forward and backward search have visited the respective forward and backward reachable state spaces. Since there is a path to the final state, there must be a non-empty intersection between these searches and a trace must be reported.

(2) The algorithm does not terminate. Since there is a finite path of length \( L \) from \( m_0 \) to \( m_F \), the forward or backward search must find this path after at most \( 2L + 1 \) steps. This consists of two times the initial step, \( L - 1 \) times a forward (or backward) step and \( L \) times a backward (or forward) step. Since all paths with a length of \( L \) must be included in this search, the path from the initial to the final state must have been detected.

#### Theorem 4.2: Soundness

Given a synchronous product Petri net \( N \) and a uniform cost function \( c \), if Algorithm 14 returns a trace, it is optimal for \( N \) and \( c \).
4.5. Symbolic alignment computation

Proof. Assume that the total cost for an optimal trace is \( K \). Note that an optimal trace in our setting minimizes the number of 1-cost moves. By considering all possible states reachable via \( T_0 \) transitions both before and after a single \( T_1 \) application we consider every state reachable with a maximum cost of 1. Thus after \( K \) iterations of applying a 1-cost move (note that the directional changes do not affect the result) we must also detect the states that form a shortest path, which is encountered before any longer path.

From the proofs of Theorem 4.1 and Theorem 4.2, we can also conclude that Algorithm 14 needs at most \( 2 + K \) (2 initial \( T_0^* \) applications and \( K \) times \( T_1 \cdot T_0^* \)) steps to return an optimal alignment of cost \( K \). Note that the algorithm avoids traversing cycles infinitely often as they only contain a finite number of markings. However, termination is not guaranteed for infinitely sized marking graphs.

Adaptation for more general cost functions. Currently, Algorithm 14 is only able to compute alignments for uniform cost functions. We can however extend the algorithm to also handle \( \varepsilon \) and \( \infty \) cost values. We can simply disregard \( \infty \)-cost transitions, as these may not be taken anyway. Hence, we can allow for milestone events. Implementing a cost function that combines both 0 cost, \( \varepsilon \) cost and 1 cost edges requires more work. We can achieve this by including a transition group \( T_\varepsilon \) that contains all \( \varepsilon \) cost transitions. We will assume, however, that for optimal alignments the total of all \( \varepsilon \) costs does not become larger than 1. This allows us to perform \( T_\varepsilon \) transitions until that cost reaches 1. Furthermore, we can perform countably infinite \( T_0 \) transitions before the total cost reaches \( \varepsilon \). We can employ a similar mechanism as we currently do in Algorithm 14.

We start by computing \( T_0^* \) from the initial state. Then we perform a single \( T_\varepsilon \) step, followed by another \( T_0^* \) operation. We continue this process until no new states can be found and then perform a single \( T_1 \) step. If we now repeat the entire process, we effectively compute a \( T_0^* \) before and after every \( T_\varepsilon \) step, and perform a \( T_\varepsilon^* \) before and after every \( T_1 \) step. As a result, we explore the state space in minimal total cost and hence find a shortest path to the final marking.

The algorithm cannot be easily extended if the cost function contains e.g. the combination of 1 and 2 cost moves. In this case, we might be required to explicitly track the current total cost for reaching each marking – which is certainly doable, but may be computationally expensive.

4.5.5 Implementation

We discuss some details on how the algorithm is implemented.
Decision diagrams. We implemented the decision diagrams using the multicore BDD package Sylvan [vDvdP15, vD16]. Instead of using BDDs for the implementation, we used the LDD variant. Unless the Petri net is 1-safe, meaning that each place can contain at most one token (e.g., see Figure 4.3), it is difficult to decide how many boolean variables are required for representing a place. An LDD is extended on-the-fly and is better suited for model checking [vDvdP15].

For variable reordering, we experimented with various algorithms that have been shown to perform well in practice for symbolic reachability [MvdP16]. However, none of the variable reordering algorithms we tested improved the overall performance. We suspect this is due to the fact that we split up the transitions into two groups and already order the transitions to some degree.

We note that the current approach may also be combined by employing saturation [CMS06] to cluster transitions together. We may for instance combine multiple $T_0$ operations together, such that with one $T_0'$ operation we compute each of $T_0, T_0^2, \ldots, T_0^n$ (i.e. $n$ applications of $T_0$). Future work will have to show how this strategy compares to our current approach.

Bidirectional search. We experimented with various alternatives for choosing to search forward or backward. We only observed a slight performance improvement from choosing the smallest decision diagram versus alternating in each step. However, when compared to a strictly forward search, we found that the (combined) LDD size of the bidirectional search is often significantly smaller than the one for the strictly forward search. Its performance is also significantly better.

Trace construction. When a non-empty intersection is found, a trace is returned. This is realized by storing the LDD after each operation during the algorithm, which we call levels. We show how a trace is reconstructed from the forward search. From the non-empty intersection, a single state $s$ is selected and the transition relation is applied in the backward direction to obtain a set $S$ of predecessor states. We intersect $S$ with the previous level from the forward search and pick a state $s'$. The same procedure is applied until the initial state is retrieved. The trace is combined with the backward reconstruction (also starting from state $s$).

The trace reconstruction is efficient in practice since the transitions are only applied on single states (and thus also small LDDs). Note that it is also possible to obtain all shortest paths by selecting the entire intersection as a starting point and by iteratively computing all possible predecessors\(^3\).

\(^3\)In the case for returning all shortest paths, Algorithm 14 should be updated to only return a trace after all $T_0^*$ procedures have finished.
4.6 Experiments

For the experiments, there are a number of aspects to consider, i.e.:

- The algorithm to evaluate. We consider our MTCG algorithm (with and without milestones), our symbolic algorithm, and two state-of-the-art algorithms: the A* algorithm from [Adr14] and the incremental algorithm from Van Dongen [vD18].

- The domain of interest for evaluation. We consider diagnostics (the structure of the alignments) and the time to compute alignments.

- The scale of the problem. We may have large or small models and a large or small number of log traces to align.

- The origin of the problem instances. We may have real or generated models in combination with real or generated event logs.

- Various other characteristics. We may e.g. consider the length of the log traces, the amount of noise (how much the log traces deviate from the model), structural properties of the model, how log traces relate to the model, how many milestones are present, and on what platform the experiments are performed.

It is infeasible to provide an extensive analysis on each possible combination. Instead, we studied the following combinations to evaluate our approach:

- In Section 4.6.1, we compare the alignment structure for the $c_{st}$ and $c_{sync}$ cost functions (and to some extent the $c_{add}$ and $c_{rem}$ cost functions), using generated models and single-trace event logs with varying properties (generated using the PTandLogGenerator [JD16] from the RapidProM [vdABvZ17] platform). We also analyse the impact of having milestones.

- In Section 4.6.2, we compare the performance of the symbolic algorithm with that of A* and the incremental algorithm, using 8 threads. We use generated models and single-trace event logs with varying properties and study how certain characteristics influence the result.

- In Section 4.6.3, we study the performance of the MTCG algorithm on industrial (small-scale) models (obtained from [FFK+11]) that are accompanied by many generated log traces\(^4\). We compare our approach with the incremental algorithm and investigate how the performance is affected by the inclusion of milestones.

\(^4\) We consider multi-threaded experiments not as useful in this scenario, as the log traces can e.g. be divided over the different threads such that the alignments are computed independently.
All of our experiments have been performed on the same architecture. All experiments were performed on an Intel® Core™ i7-4710MQ processor with 2.50GHz and 7.4GiB memory. For all experiments, we used a timeout of 60 seconds. Our implementation, experiments, and results are available online\textsuperscript{5}.

The A* algorithm [Adr14] computes a shortest path to the final marking by using a heuristic function to guide the search. To achieve this, one generally uses Integer Linear Programs (ILPs) based on the Petri net marking equation (i.e. an equation that uses the flow relation of a Petri net to underestimate the cost of reaching a given marking). The A* algorithm is generally considered to be more favourable to classical shortest path techniques (e.g. [Dij59]) as the searchable state space may be pruned effectively.

The incremental algorithm [vD18] is a recent improvement on A* by using an extend version of the marking equation to provide a more accurate underestimation of the remaining cost, and by minimizing the total number of ILPs to be solved.

4.6.1 Alignment differences for generated models

Model generation. Using the PTandLogGenerator [JD16] we generated Petri net models with process operators and additional features set to their defaults; where the respective probabilities for sequence, XOR, parallel, loop, OR are set to 45%, 20%, 20%, 10%, and 5%. The additional features for the occurrence of silent and duplicate activities, and long-term dependencies were all set to 20%.

To examine how alignments scale (i.e. the distribution of move types) we ranged the average number of activities from 25, 50, and 75, resulting in respectively 110, 271, and 370 transitions on average (due to duplicate and silent transitions). For these settings, we generated 30 models (thus 90 in total) and generated a single log trace per model. For this log trace we added 10%, 30%, 50%, and 70% noise in three different ways (thus 12 noisy singleton logs are created); by (1) adding, removing and swapping events (resembling case C4), (2), by only adding events (resembling case C3), and (3) by only removing events (resembling case C2). In total there are 1,080 noisy singleton logs. We first consider noise of type (1).

\textsuperscript{5} The implementation and experimental results for the symbolic algorithm are available at https://github.com/utwente-fmt/SymbolicAlign-ACSD18. The corresponding models and event logs used in the experiments for the symbolic algorithm are located at https://data.4tu.nl/repository/uuid:a6709ee4-2aa3-49a3-92db-247e8b5bf340. The implementation for the MTCG algorithm is available in the MaxSyncAlignments package from the ProM 6.8 toolset and the results for the experiments are available at https://github.com/utwente-fmt/MaxSync-BPM2018. The corresponding models and event logs used in the experiments are available at https://data.4tu.nl/repository/uuid:5f168a76-cc26-42d6-a67d-48be9e978309.
4.6. Experiments

Table 4.1: Average number of move types per alignment. Comparison between alignments generated using the $c_{st}$ and $c_{sync}$ cost functions. The numbers show averages, e.g. the value of 2.3 in the top-left corner denotes the average number of log moves for all computed alignments for which 10% noise is added, using the $c_{st}$ cost function.

<table>
<thead>
<tr>
<th>Moves</th>
<th>Noise added (add, remove, swap)</th>
<th>Number of activities</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10% 30% 50% 70%</td>
<td>25 50 75</td>
<td></td>
</tr>
<tr>
<td>Log</td>
<td>$c_{st}$ $c_{sync}$ $c_{st}$ $c_{sync}$</td>
<td>$c_{st}$ $c_{sync}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.3 1.3 6.5 9.4</td>
<td>4.7 3.2 8.9 4.6</td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td>$c_{st}$ $c_{sync}$ $c_{st}$ $c_{sync}$</td>
<td>$c_{st}$ $c_{sync}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0 15.7 4.6 30.9</td>
<td>3.3 10.7 5.6 39.1</td>
<td></td>
</tr>
<tr>
<td>Silent</td>
<td>$c_{st}$ $c_{sync}$ $c_{st}$ $c_{sync}$</td>
<td>$c_{st}$ $c_{sync}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17.3 24.4 14.7 30.4</td>
<td>13.8 15.4 23.2 27.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Average number of move types per alignment. Comparison between alignments generated using the $c_{st}$ and $c_{sync}$ cost functions for alignment problems, where noise only consists of adding (Add) or removing (Rem) events. The cost functions $c_{add}$ and $c_{rem}$ are variations on $c_{st}$ such that model and log moves respectively have a cost of $\infty$.

<table>
<thead>
<tr>
<th>Moves</th>
<th>Log events added (Add)</th>
<th>Log events removed (Rem)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10% 30% 50%</td>
<td>10% 30% 50%</td>
</tr>
<tr>
<td>Log</td>
<td>$c_{st}$ $c_{sync}$ $c_{add}$</td>
<td>$c_{st}$ $c_{sync}$ $c_{rem}$</td>
</tr>
<tr>
<td></td>
<td>3.1 2.0 3.1 7.5</td>
<td>0.3 0.0 0.0 1.0</td>
</tr>
<tr>
<td>Model</td>
<td>$c_{st}$ $c_{sync}$ $c_{add}$</td>
<td>$c_{st}$ $c_{sync}$ $c_{rem}$</td>
</tr>
<tr>
<td></td>
<td>0.0 13.1 0.0 0.1</td>
<td>3.0 3.3 3.3 6.3</td>
</tr>
<tr>
<td>Silent</td>
<td>$c_{st}$ $c_{sync}$ $c_{add}$</td>
<td>$c_{st}$ $c_{sync}$ $c_{rem}$</td>
</tr>
<tr>
<td></td>
<td>29.4 30.5 29.4 26.5</td>
<td>30.3 30.7 30.7 21.0</td>
</tr>
</tbody>
</table>

Alignment differences for $c_{st}$ and $c_{sync}$. In Table 4.1 we compare the resulting alignments for the different cost functions. When comparing the overall results of $c_{st}$ and $c_{sync}$ (rightmost column), we observe that $c_{sync}$ uses about 43% fewer log moves, which are added as synchronous moves. However in doing so, more than six times as many model moves are required.

When looking at an increase in the amount of noise, the relative difference between the number of log moves remains the same, while this difference in model moves slightly drops. When increasing the number of activities from 25 to 75, We observe an increase in the number of model moves for $c_{sync}$ from 3.2 times to 9.3 times as many compared to $c_{st}$. This only results in a few more synchronized moves than an alignment for a $c_{st}$ cost function returns. The difference between log moves from $c_{sync}$ and $c_{st}$ stays relatively the same for increasing activities.

We conclude that, while $c_{sync}$ does result in more synchronized moves than $c_{st}$, the number of additional model moves increases when the amount of noise, or the number of activities grows. As a result, the alignments resulting from $c_{sync}$ and $c_{st}$ become more diverse for larger models.
Table 4.3: Average number of move types per alignment. Comparison between alignments for the $c_{\text{sync}}$ cost function on models that have either 2 or 5 randomly chosen milestones. Failed denotes the percentage of failed alignments.

<table>
<thead>
<tr>
<th>Noise added (add, remove, swap)</th>
<th>Number of activities</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10% 30% 50% 70%</td>
<td></td>
</tr>
<tr>
<td>Milestones</td>
<td>2 5 2 5 2 5 2 5</td>
<td></td>
</tr>
<tr>
<td>Failed</td>
<td>2% 6% 4% 14% 8% 22% 12% 27%</td>
<td>4% 20% 10% 16% 4% 14% 6% 17%</td>
</tr>
<tr>
<td>Log</td>
<td>1.5 1.6 2.7 3.1 3.6 4.6 4.2 5.2</td>
<td>2.6 2.9 2.9 3.6 2.7 2.7 2.7 3.1</td>
</tr>
<tr>
<td>Model</td>
<td>2.2 2.0 4.2 3.2 5.4 3.9 5.8 4.3</td>
<td>2.8 2.0 4.3 3.6 6.2 4.7 4.0 3.1</td>
</tr>
<tr>
<td>Sync</td>
<td>19.6 19.6 15.2 14.0 11.9 10.8 9.4 9.1</td>
<td>10.7 10.8 15.5 14.7 18.7 16.5 13.9 13.4</td>
</tr>
<tr>
<td>Silent</td>
<td>12.4 12.0 11.5 10.6 10.8 9.6 10.1 9.4</td>
<td>8.5 8.2 11.1 10.0 18.5 16.3 11.2 10.5</td>
</tr>
</tbody>
</table>

Alignment problems that only add or remove events. In Table 4.2 we compare the resulting alignments for adding or removing events. When inspecting the Add case, we find that, in contrast to $c_{\text{sync}}$, $c_{\text{st}}$ already avoids model moves for the most part, which is as we expected. Moreover, there are only small differences between alignments from $c_{\text{st}}$ and $c_{\text{add}}$. For $c_{\text{sync}}$, many model moves may be chosen to increase the number of synchronous moves. These additional synchronous moves are arguably not part of the ‘desired’ alignment since they require a large detour through the model.

When removing events from the log trace, the $c_{\text{st}}$ cost function is only partly able to describe the removal of events as it still chooses log moves. The $c_{\text{sync}}$ cost function does not take any log moves as this maximizes the number of synchronous moves, making it equal to $c_{\text{rem}}$. When comparing $c_{\text{st}}$ and $c_{\text{sync}}$, we argue that for the Add case, the $c_{\text{st}}$ cost function better represents a ‘correct’ alignment and for the Rem case $c_{\text{sync}}$ is better suited. For all cases, the effects become larger when the amount of noise is increased.

Influence of milestones on alignments. The results from Table 4.3 show how milestones affect alignments, and can be compared with the results from Table 4.1 for 0 milestones. An interesting result is that even though we use the $c_{\text{sync}}$ cost function, the number of model moves remains relatively low for both 2 and 5 milestones. For the alignments with milestones we observe that the number of activities does not significantly affect the ratio of failed alignments, even though a larger model should make it easier to ‘avoid’ a milestone. We do observe a rather clear correlation between the increasing amount of noise and the ratio of failed alignments. A possible explanation for this is that more noise in the log increases the probability that a milestone activity is removed. As expected, having more milestones results in a larger alignment failure ratio.

When comparing the alignments resulting from the 2 and 5 milestone versions, we
discover that these are remarkably similar. The largest difference is in the number of model moves, which is slightly lower for 5 milestones, which makes sense as we put a more severe restriction on which model moves can be taken. In general, only a few milestones are sufficient for mitigating the downsides of $c_{\text{sync}}$, as large detours through the model are now likely avoided.

4.6.2 Symbolic algorithm applied on generated models

We now study the performance of the symbolic algorithm compared to the A* and incremental algorithms. We extend the set of models from Section 4.6.1 to include variations on the model that we discuss as follows.

**Model generation.** We generated Petri net models with specific characteristics, which we explain as follows. We consider the standard (STD) setting [JD16] as a basis for the models. Here we have set the probabilities for sequence, XOR, parallel, loop, and OR operators to 45%, 20%, 20%, 10%, and 5% respectively. The additional features for the occurrence of silent and duplicate activities, and long-term dependencies were all set to 20% (the same as in Section 4.6.1).

We also consider variants, where sequence operators occur 25% instead of 45% and there is a 20% increase in probability for one of the other operators (the total must be 100%), which gives us the following settings:

- the probability for XOR: 40% (+XOR),
- the probability for parallel: 40% (+PAR),
- the probability for loop: 30% (+LOOP),
- the probability for OR: 25% (+OR).

We also consider another variant (ALT) with sequence, parallel, XOR, loop, and OR respectively set to 46%, 35%, 19%, 0%, 0%, to resemble standard models without loops [KIWW11, vZBvD17].

For the standard (STD) setting, we also consider variations to the additional features (silent, duplicate, long-term), which are normally all set to 20%. For each of these parameters, we consider a variant with one feature set to 0% (−SIL, −DUP, −LONG) and a variant with one feature set to 50% (+SIL, +DUP, +LONG). We only use these variants for standard process operators and use the standard additional features setting when we change one of the process operator settings, such that only one aspect is changed.
Table 4.4: Experimental results for the various types of experiments. From top to bottom it shows the total number of experiments per category, the number of time-outs that occurred for respectively the incremental (Incr) and the symbolic algorithm (Sym), the number of times that Incr was faster than the symbolic algorithm and vice versa, and finally the relative time improvement of Sym.

<table>
<thead>
<tr>
<th>Activities</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>Additional features</th>
<th>-DUP</th>
<th>-LONG</th>
<th>-SIL</th>
<th>+DUP</th>
<th>+LONG</th>
<th>+SIL</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiments</td>
<td>1,440</td>
<td>1,440</td>
<td>1,440</td>
<td>1,440</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>2,160</td>
</tr>
<tr>
<td>Incr time-out</td>
<td>88</td>
<td>433</td>
<td>576</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>2,160</td>
</tr>
<tr>
<td>Sym time-out</td>
<td>2</td>
<td>78</td>
<td>255</td>
<td>2</td>
<td>78</td>
<td>255</td>
<td>2</td>
<td>78</td>
<td>255</td>
<td>2</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>Incr &lt; Sym</td>
<td>1,174</td>
<td>833</td>
<td>718</td>
<td>232</td>
<td>239</td>
<td>222</td>
<td>157</td>
<td>235</td>
<td>179</td>
<td>1,461</td>
</tr>
<tr>
<td></td>
<td>Sym &lt; Incr</td>
<td>265</td>
<td>544</td>
<td>501</td>
<td>102</td>
<td>112</td>
<td>113</td>
<td>170</td>
<td>107</td>
<td>121</td>
<td>585</td>
</tr>
<tr>
<td>Incr / Sym</td>
<td>4.73</td>
<td>2.40</td>
<td>1.47</td>
<td>1.81</td>
<td>3.59</td>
<td>2.05</td>
<td>2.37</td>
<td>2.00</td>
<td>1.58</td>
<td>1.70</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process operators</th>
<th>+LOOP</th>
<th>+OR</th>
<th>+PAR</th>
<th>+XOR</th>
<th>ALT</th>
<th>STD</th>
<th>Noise</th>
<th>10%</th>
<th>30%</th>
<th>50%</th>
<th>70%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiments</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>360</td>
<td>2,520</td>
<td>1,080</td>
<td>1,080</td>
<td>1,080</td>
<td>1,080</td>
<td></td>
</tr>
<tr>
<td>Incr time-out</td>
<td>142</td>
<td>96</td>
<td>92</td>
<td>43</td>
<td>12</td>
<td>712</td>
<td>140</td>
<td>239</td>
<td>343</td>
<td>335</td>
<td></td>
</tr>
<tr>
<td>Sym time-out</td>
<td>23</td>
<td>28</td>
<td>35</td>
<td>12</td>
<td>32</td>
<td>205</td>
<td>58</td>
<td>74</td>
<td>98</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Incr &lt; Sym</td>
<td>162</td>
<td>227</td>
<td>236</td>
<td>283</td>
<td>333</td>
<td>1,484</td>
<td>843</td>
<td>684</td>
<td>605</td>
<td>593</td>
</tr>
<tr>
<td></td>
<td>Sym &lt; Incr</td>
<td>175</td>
<td>109</td>
<td>93</td>
<td>69</td>
<td>20</td>
<td>844</td>
<td>195</td>
<td>334</td>
<td>390</td>
<td>391</td>
</tr>
<tr>
<td>Incr / Sym</td>
<td>2.99</td>
<td>1.67</td>
<td>1.76</td>
<td>1.69</td>
<td>0.41</td>
<td>2.01</td>
<td>1.31</td>
<td>1.91</td>
<td>2.11</td>
<td>2.01</td>
<td></td>
</tr>
</tbody>
</table>

With the above we obtain $3 \cdot (6 + 6) = 36$ different settings. For each setting, we generate four models and generate a single random log trace per model. In the four log traces we add 10%, 30%, 50% and 70% noise by adding, removing, and swapping events. We now have $36 \cdot 4 = 144$ different models and log traces and we repeat this procedure 30 times to obtain $144 \cdot 30 = 4,320$ different experiments.\(^6\)

**Experimental setup.** We compute alignments using the simplified standard cost function (i.e. the standard cost function with the cost for silent moves set to 0). For each of the 4,320 experiments, we execute the A* and incremental algorithm with 8 cores and set a time-out of 60 seconds, and do the same for our symbolic algorithm. The times include the trace generation process to form the alignment. When summarizing the results, we average the times and in case of a time-out use 60 seconds for the time. We found that the MTCG algorithm is in general unsuitable for alignment problems of this form, as the models become too large for the MTCG precomputation process.

\(^6\)We could have used the same model in multiple experiments by for instance generating multiple log traces, but by using different experiments we eliminate the randomness of model and log generation as much as possible.
General results. The results of the experiments are depicted in Figure 4.10 and Table 4.4 (in the table we have omitted the results for A*). Overall, the average time that the incremental algorithm requires for computing an alignment is 87% more than that of the symbolic algorithm. Interestingly, in 2,725 experiments Incr outperformed the symbolic algorithm and the symbolic algorithm outperformed Incr only 1,310 times (in 284 cases both algorithms did not compute an alignment within 60 seconds). It should also be noted that Incr had a time-out in 1,097 cases and the symbolic algorithm had 335 time-outs. We can conclude the following.

- The symbolic algorithm is significantly faster in computing alignments when it comes to the average time.
- Incr was able to compute more experiments faster than the symbolic algorithm than the other way around.
- If the incremental algorithm faced a time-out, in 74% of the cases the sym-
bolic algorithm was able to compute an alignment, and in 15% of the cases \textsc{Incr} was able to compute an alignment when the \textsc{Sym} timed out.

In the results for our symbolic algorithm, we observed that 63% of the time is spent on the search procedure, and the remaining 37% of the time is used for setting up the model checker (i.e. parsing the Petri net) and the trace construction. With more optimizations we think it is possible to reduce the trace construction time.

\textbf{Results for specific cases.} The number of activities in the Petri net corresponds to the size of the model. We can see that when the number of activities increases, all algorithms take on average more time to compute alignments. Interestingly, the symbolic algorithm seems to perform relatively better for the smaller class of models than for the larger models. We argue that this is mainly due to the large influence of the 60 second time-out penalties in the 25 activity cases. We can observe that, when both \textsc{Incr} and \textsc{Sym} compute an alignment within the timeout, the incremental algorithm is more often faster than the symbolic algorithm than the other way around, but for larger models the difference becomes smaller.

From the results, we observe large differences when changing the probabilities for various process operators. The symbolic algorithm performs relatively well for the \texttt{+LOOP}, \texttt{+OR}, \texttt{+PAR} and STD cases. One similarity between these instances is that they all significantly increase the size of the marking graph. Just as with the increase in activities, we consider the symbolic algorithm better suited for larger state spaces that have a regular structure. This is supported by the observed performance increase. An interesting result is the ALT case, where the incremental algorithm significantly outperforms the other algorithms. Since these models contain no loops (and hence the 'bad' performance for \textsc{Incr} on \texttt{+LOOP}, compared to the other two algorithms) we argue that \textsc{Incr} is especially well-suited for models without (many) loops.

For the additional features we see fairly similar results for each case, implying that these changes have little to no effect on the \textit{relative} difference between the algorithms.

For the increase in noise we can see that the performance of \textsc{Sym} stays relatively the same while the performance of the other algorithms drop.

\subsection{MTCG algorithm on models with many traces}

We now consider smaller models that we use to align many log traces. For our experiments, we selected 9 instances from the 735 industrial business process Petri
4.6. Experiments

net models from financial services, telecommunications and other domains, obtained from the data sets presented in Fahland et al. [FFK+11].

**Experimental setup.** For our model selection, we picked the 9 most interesting cases, e.g. the models with the largest Petri net models, largest marking graphs, largest MTCG graph. For each model, we generated a set of 10, 100, and 1,000 log traces and added 30% noise by adding, removing, and swapping events. We also investigate the effects of having 0, 2, or 5 milestones. We compare the performance of the MTCG algorithm with the incremental algorithm from [vD18] using a single thread (as mentioned earlier, the performance between A* and Incr is similar in this scenario). Note that in our experiments, we only consider the $c_{sync}$ cost function, as the MTCG algorithm cannot use the $c_{st}$ cost function. In preliminary experiments we found that our symbolic algorithm is unsuitable for alignment problems of this form (i.e. small models with many traces), since its setup time becomes the dominating factor.

**Results.** The results are presented in Table 4.5. For the incremental algorithm we clearly see a computation time that is proportional to the log size. However, for the MTCG algorithm this is not the case. The computation time for aligning 10 log traces is practically the same when 1,000 log traces are aligned. This shows us that after computing the MTCG as a precomputation step (which takes up most of the time), actually aligning log traces takes very little time.

However, for the models b, c, and e, the MTCG is not able to compute alignments within 60 seconds, even for 10 log traces. Here, the size of the MTCG becomes too large. For each of the models for which MTCG timed out, the algorithm reported having explored more than 200,000 different sets of markings (a set of marking forms a single state in the MTCG). This results from models that express concurrency or branching behaviour. Every allowed order for executing events in the model is tracked in the MTCG by storing all corresponding intermediate states (marking sets) that may be visited.

In particular for model g, we observe that the incremental algorithm already takes quite some time to compute alignments for 10 log traces, and times out for 1,000 traces. The MTCG algorithm is for this model significantly faster in computing alignments. An interesting aspect of this model is that, while the number of places and transitions is high, there is almost no branching in the process. The model has a few branching points, followed by long sequential processes. Arguably, this is a difficult situation for an on-the-fly algorithm such as the incremental algorithm or

---

7 In some cases we can even observe that MTCG performs faster for 100 traces than for 10 traces. However, this is caused by the randomness that is inherent to program execution.
Table 4.5: Comparison of computation times for aligning small models with a large number of log traces using our MTCG algorithm and the incremental algorithm from [vD18]. All times are in milliseconds. For every log trace we applied 30% noise by adding, removing and swapping events. Here, \( M \) is the model name and \(|P|\) and \(|T|\) denote the number of places and transitions in the model, respectively. The number of log traces is given by \(|\text{Log}|\) and \(|\text{Fail}|\) is the number of failed alignments. A TO indicates that the algorithm timed out (i.e. requires more than 60 seconds to compute).

| M     | \(|P|\) | \(|T|\) | \(|\text{Log}|\) | 0 milestones |  | 2 milestones |  | 5 milestones |  |
|-------|--------|--------|----------------|--------------|----------------|----------------|----------------|--------------|----------------|
|       |        |        |                | MTCG | Incr | MTCG | Incr | MTCG | Incr |
| a 35  | 26     |        |                | 10   | 723  | 288  | 2   | 465  | 346  |
| b 13  | 9      |        |                | 10   | TO   | 23   | 2   | 764  | 17   |
| c 23  | 17     |        |                | 10   | TO   | 102  | 1   | TO   | 113  |
| d 40  | 26     |        |                | 10   | 310  | 82   | 0   | 127  | 60   |
| e 25  | 17     |        |                | 10   | TO   | 281  | 1   | TO   | 232  |
| f 59  | 43     |        |                | 10   | 602  | 260  | 0   | 174  | 274  |
| g 123 | 81     |        |                | 10   | 576  | 2980 | 0   | 611  | 3387 |
| h 38  | 18     |        |                | 10   | 260  | 74   | 4   | 122  | 37   |
| i 24  | 18     |        |                | 10   | 103  | 62   | 3   | 15   | 55   |

A*, as it may have to explore a large subprocess before realizing that a different choice at the start of the process would lead to a better alignment. Here, the MTCG remains reasonably small and for alignment computations we do not have to decide which path to take.

Another interesting result is that the computation time decreases for both algorithms when we increase the number of milestones. The model becomes more
4.7 MTCG structure for diagnostics

Once alignments have been computed for a model and a set of log traces, we aim to obtain actionable insights from these alignments. For a single log trace, we can simply map the alignment on the Petri net or marking graph. However, this approach is not applicable for multiple alignments on a single model. Instead, two common ways of visualizing alignments are to (1) only depict the alignments,
1 × \langle A[0], B[5], D[15] \rangle \\
1 × \langle A[0], C[10], D[15] \rangle \\
1 × \langle A[0], B[5], C[10], D[15] \rangle

Figure 4.12: An event log with timestamps for each event (boxed) displayed on a Petri net (left) and its corresponding MTCG (right). While the MTCG visualization only shows the observed events, the Petri net version needs to create timestamps for the generated model moves.

e.g. colour-coded for more convenient inspection, and (2) to map the alignments on the model such that the count or ratio of synchronized versus model moves is depicted on the model [CvDSW18].

In Figure 4.11 (left), we show how two sets of log traces are aligned on a Petri net model. We count the number of times that a particular transition is taken as a synchronous move (e.g. the arc denoting 20 from p_0 to A) and the number of times that a transition is skipped in the log, forming a model move (e.g. the arc from p_0 to p_1). This approach provides insights in the observed behaviour with respect to the model, e.g. each transition is fired an equal number of times and skipped in one third of the cases. In Figure 4.11 (right), we show the number of times in which a transition is taken as a synchronous move. Additionally, we show how many traces end at each MTCG location. Model moves are not represented in the MTCG depiction, but can be derived. We are able to better depict the differences with the MTCG visualization compared to the alternative approach.
4.7. MTCG structure for diagnostics

In Figure 4.12 we show how an event log with timing information is aligned to a Petri net model. Here, the timestamps are given for each event (e.g. B[5]), which are integrated into the model. In the MTCG we can clearly see how much time each event takes. This is also mostly true for the Petri net visualization. However, whenever an event is skipped in the model, we need to generate a model move along with a timestamp. For the first log trace, we could have constructed the following alignment:

\[
\gamma = \begin{array}{c|c|c|c}
t_0 & t_2 & t_1 & t_3 \\
\end{array}
\]

where C[2.5] is a generated model action, with a timestamp that is the average of the time for the two neighbouring events (i.e. \( \frac{0+5}{2} \)). This approach may seem logical, but produces useless information. We could also have chosen 0 or 5. Moreover, if we would have swapped B and C in the alignment, we would end up with a different timestamp for C, which could have been 15.

**Drawbacks of mapping alignments on the model.** There are some drawbacks to this approach. As we show in the picture, the results for two distinct event logs result in the same diagnostic information. Observe that a model move is a generated event. Hence, when applicable, all accompanying data for this event (e.g. the timestamp denoting when the event occurred) is generated as well, while in practice the event did not take place at all. By displaying model moves we are effectively trying to ‘fill the gap’ between two synchronous moves, while in fact we observed that there is no gap to fill.

**Depicting only synchronous moves.** A useful property of the MTCG is that it allows us to represent all synchronous moves in an alignment, without having to generate artificial model moves to ‘fill in the gaps’. Consider the rightmost illustrations from Figure 4.11. Here we only depict which moves in the model can be synchronized with the log traces. In the top image, we clearly see how the trace \( \langle A, C \rangle \) is depicted. And, since we do not create artificial moves, the bottom illustration only depicts the 20 \( \langle A, B, C \rangle \) traces (the \( \langle \rangle \) traces remain in the initial state). Of course, here it is still possible to find contrived examples in which two distinct event logs result in the same diagnostics, but the MTCG structure does mitigate that effect to some extent. The drawback is that for larger models it may become infeasible to represent alignments on a MTCG, simply because the graph becomes too big.
4.8 Related work

We refer to [vdA16] for an overview of different process mining techniques. One of the earliest works in conformance checking was from Cook and Wolf [CW99]. They compared log traces with paths generated from the model.

**Token-based replay** [RvdA08] is one technique to check for conformance. The idea is to ‘replay’ the event logs by trying to fire the corresponding transitions, while keeping track of possible missing and remaining tokens in the model. However, this technique does not provide a path through the model. When traces in the event log deviate a lot, the Petri net may get flooded with tokens and does not provide good insights any more. However, token-based replay is faster to compute when compared to alignments, and it has been implemented in commercial tools like Celonis.

Alignments were introduced [vdAAvD12, Adr14] to overcome some limitations of the token-based replay technique. Alignments formulate conformance checking as an optimization problem, i.e. minimizing the alignment cost-function. Since its introduction, alignments have quickly become the standard technique for conformance checking along with the A* algorithm for computing alignments [Adr14, vZBvD18].

Alignments were introduced [vdAAvD12, AvDvD11, Adr14] to overcome the limitations from the token-based replay technique. Alignments formulate conformance checking as an optimization problem, i.e. minimizing the alignment cost-function. Since its introduction, alignments have quickly become the standard technique for conformance checking. The A* shortest path algorithm [HNR68] in combination with Integer Linear Programming (ILP) to prune the search space (i.e. the synchronous product of the model and the log trace) is considered to be the standard in computing alignments [Adr14]. The incremental algorithm [vD18] improves on A* by considerably reducing the number of linear programs required for guiding the search.

For larger models, techniques have also been developed to decompose the Petri net in smaller subprocesses [vdA13, vdAKRV15, MCvdA14, PVV10]. For instance, fragments that have a single-entry and single-exit node (SESE) represent an isolated part of the model [MCvdA14]. This way, localizing conformance problems becomes easier in large models. It would be interesting to combine the MTCG algorithm or even the symbolic algorithm with such decomposed models. Another approach is to combine the different log traces in a trie or directed acyclic automaton [RCD+17]. While we have not studied this in depth, we argue that this approach may combine well with the MTCG alignment procedure, as it allows us to compute multiple alignments at once.
A sub-field of alignments is to compute a *prefix-alignment* for an incomplete log trace. This is useful for analysing processes in real-time instead of *a posteriori*. Several techniques exist for computing prefix-alignments [Adr14, vZBH+17]. The MTCG approach that we introduced could also be suitable for computing prefix-alignments.

A different solution to deal with large instances is to no longer guarantee optimality. Several techniques exist [vDCCT17, TC16, BC17] that can very efficiently compute ‘good’, but not necessarily optimal, alignments for large instances. For instance, Burattin and Carmona [BC17] introduced a technique similar to the MTCG approach, in which the marking graph is extended with additional edges to allow for deviations. However, it cannot guarantee optimality as a *single* successor marking is chosen per event, while instead we consider all possible successors and can, therefore, better adapt for future events.

In a more general setting, conformance checking is related to finding a *longest common subsequence*, computing a *diff*, or computing minimal *edit distances*. Here, the problem is translated to searching for a string $B$ from a regular language $L$ such that the edit distance of $B$ and an input word $\alpha$ is minimal [Wag74].

There are a number of existing algorithms for computing the shortest path symbolically. Symbolic versions of Dijkstra’s algorithm, Bellman-Ford, and A* exist [Saw04, EK09]. However one of the main problems with the related approaches is the overhead for bookkeeping or updating the shortest path info. For instance, the symbolic A* implementation [EK09] tracks a $g \times h$ matrix, where each cell represents the set of states that have current cost $g$ from the initial state and heuristic cost $h$ to the final state. Our symbolic approach is designed in such a way that no additional cost information needs to be stored directly for the decision diagrams, with the downside that it is only suitable for uniform cost functions.

### 4.9 Conclusion

In this chapter, we looked into the computation of alignments and explicitly focused on the associated cost function. We introduced a max-sync cost function that instead of minimizing discrepancies between a log trace and the model, maximizes the number of synchronous events. We empirically evaluated the differences with the standard cost function, compared the alignment computation times. The max-sync cost function also lead to the MTCG algorithm for computing alignments.

We observed that in general, a considerable amount of model moves may be required to add a few additional synchronous moves, when comparing max-sync
with the standard cost function. Milestones, i.e. unskippable actions in the model, address this problem. Milestones may be used to refine the model or guide the alignment computation. For instance, we observed that combining the max-sync cost function with a few milestones prevents alignments to take long deviations through the model. We consider milestones to be valuable in general as they allow a user to filter traces and steer alignment computations however he or she pleases.

To cope with large models, we have designed and implemented a symbolic algorithm for computing shortest paths on uniform cost functions. From the experimental results we conclude that our symbolic algorithm is competitive with the state-of-the-art for computing alignments. We found that the symbolic technique is more robust in computing alignments when compared to other algorithms. We further observe that the symbolic approach is especially favourable in computing alignments for models with large state spaces. However, it is not as performant for smaller instances.

On smaller-scale industrial models with many log traces, we showed that the MTCG algorithm, which uses a preprocessing step on the model, is quite different to the state-of-the-art techniques for computing alignments. While it does not scale well for large models, it is able to handle many log traces with ease.

We conclude that the max-sync cost function and the inclusion of milestones is complementary to the standard approach, as it provides an alternative view that may be preferred in some contexts. We further show that we are able to address (to some extent) two major limitations for computing alignments. Namely, we are able to efficiently compute alignments for many log traces with the MTCG algorithm, and we are able to quickly compute alignments for large models using the symbolic algorithm.

4.10 Future work

There are several directions for future work. The most interesting direction would likely be to further study and improve the symbolic algorithm. We already mentioned a few points of improvement, e.g. improving the single-threaded and multicore scalability, the inclusion of saturation, and extending it for more general cost functions. It may also be interesting to implement and compare with several symbolic shortest path approaches from the literature, especially if we consider the applicability for a more general cost function.

Another direction for future work is to extend the applicability for the MTCG algorithm. When e.g. combined with a decomposition strategy, we may be able to compute alignments for significantly larger models. We may also consider ap-
4.10. Future work

approaches to reduce the number of visible labels in the model, by removing ones that are not interesting and thereby reducing the size of the model. This way, diagnostic information can be represented more clearly.
In Chapter 4 we showed how shortest paths are used in the context of conformance checking. We presented new algorithms to compute the shortest paths more efficiently for specific constraints on the alignment cost function.

In this chapter we study the applicability of shortest paths in the context of parametric timed automata (PTAs). A PTA is an extension of a timed automaton (TA), which itself is an extension of a finite-state automaton by allowing the user to reason about temporal aspects via internal clocks. A timed automaton is a commonly used formalism for modelling real-time systems, e.g. a message passing scheme. The minimal-time reachability problem is a well-studied problem for TAs and can be solved with a shortest path computation. However, the problem is only scarcely studied for PTAs, which extend TAs by allowing parametric constant values in timing constraints. Such a formalism is for instance useful when reasoning about unknown delays in a timed system. Using existing techniques, a user can synthesize the parameter constraints that allow the system to reach a specified goal location.

We focus on synthesizing parameters such that not only the goal location is reached, but we also address the following questions: what is the minimal time to reach the goal location? and for which parameter values can we achieve this? We analyse the problem and present a semi-algorithm that solves it. We also provide a more efficient variant of the algorithm and show that the algorithms can
Figure 5.1: Example timed automaton.

also be used to minimize and maximize a specific parameter value while ensuring that a target location is reached. Finally, we analyse the maximal-time synthesis problem, and show why this problem cannot be solved efficiently.

The main content of this chapter is based on the work from [ABPvdP19], which is joint work with Étienne André, Laure Petrucci, and Jaco van de Pol. The published work solely focuses on the minimal-time and minimal-parameter problems. We extend that work by also considering their maximized variants, and we also provide more details and examples compared to the paper.

5.1 Introduction

Timed Automata (TA) [AD94] extend finite state automata with clocks, to model real-time systems. These clocks can be used to constrain transitions between two locations with a guard, e.g. the transition can only be taken if at least 5 time units have passed. Furthermore, aside from taking transitions, it is possible to wait some time at a location. This waiting time can also be constrained by an invariant associated with the location. Multiple clocks can coexist, but all clocks advance with the same speed, and clocks may also be reset when taking a transition (written as $x := 0$ for clock $x$).

Timed automata allow for reasoning about temporal properties of the designed system. In addition to reachability problems, it is possible to compute the minimal or maximal time required to reach a specific goal location for TAs. Such a result is valuable in practice, as it provides boundaries on the response time of a system or it may for instance indicate when a component failure may occur. We present an example of a timed automaton and analyse the optimal-time reachability in Example 5.1.
5.1. Introduction

Example 5.1: Optimal-time reachability in a timed automaton

In Figure 5.1 we show an example of a timed automaton with two clocks, $x$ and $y$. The automaton describes a person travelling from his/her home to a destination, and the person can either walk or take the bus to get there. Both clocks are initially set to 0, and increase at the same rate. The walk action from the home location can be taken when $9 \leq x \leq 12$, thus from 9 time units until 12 time units we can take the action to arrive at the bus stop (the time range models that the person may have to wait because of traffic). The clock $x$ is then reset to 0 ($y$ is not reset).

At the bus stop, the person can continue walking to the destination, which can be taken after 7 time units\(^a\). If the person waits at the bus stop until $y = 10$, he/she can also get on the bus (after which $y$ is reset). Then it takes 3 time units to get from the In bus location to the destination.

For this system we assume that we reach the destination and can calculate the minimal and maximal time to reach the destination. The minimal time is 13, since it takes at least 9 time units to reach the bus stop, the bus ‘leaves’ at 10 time units, and it takes 3 time units to reach the destination with the bus. The maximal time is 19, which results from waiting 12 time units to reach the bus stop, then missing the bus and walking another 7 minutes to the destination.

Such results may be useful in providing boundaries on the system. For instance, we can ensure that the person will reach the destination between 13 and 19 time units.

\(^a\) It would be more realistic to model an intermediate state, e.g. called Walking, which can be reached in 0 time units from the bus stop, and the person has to wait 7 time units in the Walking state until he/she can move to the destination. However, for simplicity we omit such intermediate states.

It may not always be possible to describe a real-time system with a TA. There are often uncertainties in the timing constraints, for instance how long it takes between sending and receiving a message, or how much time it takes to walk to the bus stop in Example 5.1. We may also be interested in using such uncertainties as design parameters, and optimize them to improve the throughput of a system, as we will show in Example 5.2. Such uncertainties can however be modelled using a parametric timed automaton (PTA) [AHV93]. A PTA adds parameters, or unknown constants, to the TA formalism. By examining the reachability of a goal location, the parameters get constrained and we can observe which parameter valuations preserve the reachability of the goal location.

The process of obtaining these parameter valuations, also called parameter synthesis, is definitely useful for analysing reachability properties of a system. However,
Figure 5.2: Train delay scheduling problem: Alice (depicted in dotted red), located at A, wants to go to location D. Bob (depicted in dashed blue), located at B, wants to go to location A. By setting the train delays $D_1$ and $D_2$ for train 1 and 2, make sure that both Alice and Bob reach their target location in minimum total time, i.e. the target state in the complete system is reached when Alice is at location D and Bob is at location A.

this technique does disregard timing aspects to some extent. Given the parameter constraints, it is no longer possible to give clear boundaries on the time to reach the goal, as this may depend on the parameter valuations (e.g. consider $x = p_1$ with clock $x$ and parameter $p_1$). We focus on the parameter synthesis problem, while reaching the goal location in minimal time, as demonstrated in Example 5.2.

**Example 5.2: Train delay scheduling**

Consider the example PTA in Figure 5.2, which depicts a train network consisting of two trains. Both trains share locations B and D (the station platforms) while locations A', B', C', D', B'', and D'' represent a train travelling (tracks). The travel time for train 1 between any two stations is 100, and 55 for train 2. Train 1 stops at stations A, B, C, and D, for time $D_1$ (and train 2 stops for $D_2$ time units at B and D). Here, the train delays $D_1$ and $D_2$ are parameters and $x_1$ and $x_2$ are clocks. Both clocks start at 0 and are reset after every transition. We assume that the trains use different tracks, and changing trains at the platform of a station can be done in negligible time.

Alice is starting her journey from station A and would like to go to station D. Bob is located at B and wants to go to A. Train 1 and/or 2 can
be used to travel if both the train and the person are at the same location. Initially, both Alice and Bob wait for a train, since the initial positions of train 1 and 2 are respectively C’ and D’.

We would like to set the train delays $D_1$ and $D_2$ in such a way that the total time for Alice and Bob to reach their target location is minimal. The optimal solution is $D_1 = 25 \land D_2 = 15$, which leads to a total time of 405 units\(^a\). Note that this is neither optimal for Alice, as she would be able to reach D in time 355 by setting $D_1 = 0$ and $D_2 = 5$, but then Bob travels for 600 time units. Alternatively, $D_1 = 10 \land D_2 = 0$ is an optimal solution for Bob and reaches A in time 210, but now Alice requires 440 time units to reach her destination.

\(^a\) Alice waits for train 1 to reach A at time 225, then she hops on and exits the train on time 350 at B. There she can immediately take train 2 and reach D at time 405. Bob waits for train 2 to reach B at time 55 and takes this train. At time 125 he reaches D and can immediately hop on train 1. Bob reaches A at time 225.

Instead of a single parameter valuation, such as the one we have in Example 5.2, the time to reach a goal location may be an interval, describing the lower- and upper-bound on the time. This can be achieved in the example by changing the travel time from train 1 to be between 95 and 105, by guarding the outgoing transitions from locations A’, B’, C’ and D’ with $95 \leq x_1 \leq 105$ (instead of $x_1 = 100$).

Throughout this chapter we focus on optimizing the global time to reach a target location, meaning that we look at the minimal and maximal total time passed in the system, which may differ from the clock values as the clocks can be reset. We are also interested in optimizing a single parameter, i.e. minimizing or maximizing a given parameter such that the target location is still reachable.

With reachability of a target location, we refer to the existence of a run in the automaton that ends in a target location. Note that for reachability we omit runs that visit a target location multiple times, i.e. we only focus on the first time that a target location is reached.

We address the following problems:

- **optimal-time reachability**: synthesizing a single parameter valuation for which the goal location can be reached in either minimal (lower-bound) or maximal (upper-bound) time,\(^1\)

- **optimal-time synthesis**: synthesizing all parameter valuations such that the time to reach the goal location is minimized or maximized, and

\(^1\) For the timed automaton that is formed by the PTA and the parameter valuation, there must exist a run that reaches the target location in optimal time, and there may exist runs that reach the target location in a suboptimal time and runs that do not reach the target at all.
optimal-parameter synthesis: synthesizing all parameter valuations such that a particular parameter is minimized (or maximized) and the goal location can still be reached (a solution to this problem can also address the optimal-time synthesis problem by adding a parameter to equal the final clock value).

Note that the above problems cannot be solved if the target location is not reachable, in which case an empty or unsatisfiable result is returned. We provide semi-algorithms (algorithms that may not terminate) to solve the problems. Most algorithms have been implemented and we empirically compare them with a set of benchmark experiments for PTAs, obtained from [And18a]. Interestingly, compared to standard reachability and synthesis, minimal-time reachability and synthesis is in general computed faster as fewer states have to be considered in the exploration.

Since reachability and synthesis is in general undecidable for PTAs [AHV93], we are interested in the decidability of optimal-time reachability and synthesis for simpler automata. Hence, we look at the computability and tractability of the problems for PTAs and L/U-PTAs (PTAs for which each parameter only appears as a lower or upper bound).

5.2 Preliminaries

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative real numbers. We denote sets of comparison operators $\prec \in \{<, \leq, =, \geq, >\}$, $\preceq \in \{<, \leq\}$, and $\succeq \in \{>, \geq\}$. We first provide preliminaries on clocks, parameters, valuations, constraints, and guards. Then we provide the definitions and semantics of TAs and PTAs, and finally we give preliminaries on reachability aspects. Given that this chapter builds upon existing work on PTAs, we attempt to remain consistent (in syntax and semantics) with related work in the field [HRSV02, JLR15, And18b].

Clocks, parameters, and valuations. We assume a set $\mathbb{X} = \{x_1, \ldots, x_{|\mathbb{X}|}\}$ of clocks, i.e. real-valued non-negative variables that evolve at the same rate. A clock valuation is a function that assigns a value to each clock variable, that is $v_{\mathbb{X}} : \mathbb{X} \rightarrow \mathbb{R}_{\geq 0}$. We write $\mathbf{0}$ for the clock valuation that assigns $0$ to all clocks. Given a constant $d \in \mathbb{R}_{\geq 0}$, the expression $v_{\mathbb{X}} + d$ denotes the valuation s.t. $(v_{\mathbb{X}} + d)(x) = v_{\mathbb{X}}(x) + d$, for all $x \in \mathbb{X}$. Given $R \subseteq \mathbb{X}$, we define the reset of a valuation $v_{\mathbb{X}}$, denoted by $[v_{\mathbb{X}}]_R$, as follows: $[v_{\mathbb{X}}]_R(x) = 0$ if $x \in R$, and $[v_{\mathbb{X}}]_R(x) = v_{\mathbb{X}}(x)$ otherwise.

We assume a set $\mathbb{P} = \{p_1, \ldots, p_{|\mathbb{P}|}\}$ of parameters, i.e. unknown real-valued non-negative constants. A parameter valuation $v_{\mathbb{P}}$ is a function $v_{\mathbb{P}} : \mathbb{P} \rightarrow \mathbb{R}_{\geq 0}$. Given a clock valuation $v_{\mathbb{X}}$ and a parameter valuation $v_{\mathbb{P}}$, we use the notation $v_{\mathbb{X}} | v_{\mathbb{P}}$
to denote the valuation over $X \cup P$ such that $\forall x \in X : v_X|_P(x) = v_X(x)$ and $\forall p \in P : v_X|_P(p) = v_P(p)$.

We denote the set of all clock valuations and parameter valuations by $V_X$ and $V_P$, respectively.

### 5.2.1 Constraints

We define constraints and guards, operations on constraints, and show how the constraints relate with valuations. A linear term $lt$ over $X \cup P$ is of the form $\sum_{1 \leq i \leq |X|} \alpha_i x_i + \sum_{1 \leq j \leq |P|} \beta_j p_j + d$, with $x_i \in X$, $p_j \in P$, and $\alpha_i, \beta_j, d \in \mathbb{Z}$. We denote a parametric linear term over $P$ by $plt$, which is a linear term without clocks of the form $\sum_{1 \leq j \leq |P|} \beta_j p_j + d$, with $p_j \in P$, and $\beta_j, d \in \mathbb{Z}$.

**Constraints and guards.** A constraint $C$ (i.e. a convex polyhedron) over $X \cup P$ is a conjunction of inequalities of the form $lt \gg 0$. We do not consider diagonal inequalities (i.e. inequalities of the form $x_i - x_j \gg lt$). A guard $G$ is a constraint over $X \cup P$ defined by a conjunction of inequalities of the form $x \gg plt$, with $x \in X$. We write $G^t$ for a non-parametric guard over $X$, which is a conjunction of inequalities of the form $x \gg d$, with $x \in X$, and $d \in \mathbb{N}$.

**Constraint satisfiability.** Given a parameter valuation $v_P$, $v_P(C)$ denotes the constraint over $X$ obtained by replacing each parameter $p$ in $C$ with $v_P(p)$. Likewise, given a clock valuation $v_X$, then $v_X(v_P(C))$ is the expression obtained by replacing each clock $x$ in $v_P(C)$ with $v_X(x)$. We say that $v_P$ satisfies $C$, denoted by $v_P \models C$, if the set of clock valuations satisfying $v_P(C)$ is non-empty. Similarly, $C$ is satisfiable if and only if $\exists v_X \in V_X$ and $\exists v_P \in V_P$ such that $v_X|_P \models C$, meaning that $v_X(v_P(C))$ evaluates to true. We use the $\bot$ symbol to denote that a constraint $C$ has no valuation and is unsatisfiable (i.e. $v_X|_P \not\models C$).

**Example 5.3: Guards and valuations**

Suppose we have a set of clocks $X = \{x_1, x_2\}$ and one parameter $P = \{p_1\}$, then we could define the guard $G = x_1 < 5 \land x_1 \geq 10 \land x_2 > p_1$. In this case there is no clock valuation for $x_1$ such that $v_X|_P \models G$ holds, hence $G$ is unsatisfiable.

**Time elapsing and clock reset.** We define the time elapsing of a constraint $C$, denoted by $C\uparrow$, as the constraint over $X$ and $P$ obtained from $C$ by delaying
all clocks by an arbitrary amount of time:

\[ v'_{\mathcal{X}}|_{v_{\mathcal{P}}} = C^{\prime} \text{ iff } \exists v_{\mathcal{X}} \in \mathcal{V}_{\mathcal{X}}, \exists d \in \mathbb{R}_{\geq 0} \text{ s.t. } v'_{\mathcal{X}}|_{v_{\mathcal{P}}} = C \land v'_{\mathcal{X}} = v_{\mathcal{X}} + d. \]

Given \( R \subseteq \mathcal{X} \), we define the reset of \( C \), denoted by \([C]_R\), as the constraint obtained from \( C \) by resetting the clocks in \( R \) and keeping the other clocks unchanged, as follows:

\[ v'_{\mathcal{X}}|_{v_{\mathcal{P}}} = [C]_R \text{ iff } \exists v_{\mathcal{X}} \in \mathcal{V}_{\mathcal{X}} \text{ s.t. } v_{\mathcal{X}}|_{v_{\mathcal{P}}} = C \land v'_{\mathcal{X}} = [v_{\mathcal{X}}]_R. \]

**Projections.** Given a subset \( \mathbb{P}' \subseteq \mathbb{P} \) of parameters, we denote the projection of \( C \) onto \( \mathbb{P}' \) by \( C_{\downarrow \mathbb{P}'} \), i.e. obtained by eliminating the clock variables and the parameters in \( \mathbb{P} \setminus \mathbb{P}' \) (e.g. using Fourier-Motzkin [Sch99]). Therefore, \( C_{\downarrow \mathbb{P}} \) denotes the elimination of the clock variables only, i.e. the projection onto \( \mathbb{P} \). Similarly, we can make a projection on a subset \( \mathcal{X}' \subseteq \mathcal{X} \) of clocks, denoted by \( C_{\downarrow \mathcal{X}'} \), where all parameters \( \mathbb{P} \) and the clock variables from \( \mathcal{X} \setminus \mathcal{X}' \) are eliminated.

**Constraints to valuations.** Given a constraint \( C \), we can obtain a (possibly infinite-sized) set of parameter valuations \( V_{\mathbb{P}} \subseteq \mathcal{V}_{\mathbb{P}} \) by projecting \( C \) on \( \mathbb{P} \), i.e. \( V_{\mathbb{P}} = \{ v_{\mathbb{P}} | v_{\mathbb{P}} \models C_{\downarrow \mathbb{P}} \} \). We have that \( V_{\mathbb{P}} \neq \emptyset \) if and only if \( C \) is satisfiable.

### 5.2.2 Timed automata

We define a timed automaton, and its operational semantics as follows.

**Definition 5.1: Timed automaton [AD94]**

A timed automaton is a tuple \( \mathcal{A}^t = (\Sigma, L, \ell_0, \mathcal{X}, \mathcal{I}, E) \), where:
- \( \Sigma \) is a finite set of actions,
- \( L \) is a finite set of locations,
- \( \ell_0 \in L \) is the initial location,
- \( \mathcal{X} \) is a finite set of clocks,
- \( \mathcal{I} \) is the invariant, that assigns a non-parametric guard \( \mathcal{I}(\ell) \) to every \( \ell \in L \), and
- \( E \) is a finite set of edges \( e = (\ell, \mathcal{G}^t, a, R, \ell') \) where \( \ell, \ell' \in L \) are the source and target locations, \( a \in \Sigma \) is an action, \( R \subseteq \mathcal{X} \) is a set of clocks to be reset, and \( \mathcal{G}^t \) is a non-parametric guard.
5.2. Preliminaries

**Definition 5.2: Semantics of a TA**

Given a TA $\mathcal{A}^t = (\Sigma, L, \ell_0, X, \mathcal{I}, E)$, the semantics of $\mathcal{A}^t$ is given by the (infinite-state) timed transition system (TTS) $(S, s_0, \rightarrow)$, with:

- $S = \{ (\ell, v_X) \in L \times \mathbb{R}_{\geq 0}^{|X|} \mid v_X \models \mathcal{I}(\ell) \}$,
- $s_0 = (\ell_0, 0) \in S$ (hence $0 \models \mathcal{I}(\ell_0)$), and
- $\rightarrow$ consists of the discrete and (continuous) delay transition relations:
  1. Given an edge $e = (\ell, \mathcal{G}, a, R, \ell') \in E$, and $(\ell, v_X), (\ell', v'_X) \in S$ such that $v_X \models \mathcal{G}$ and $v'_X = [v_X]_R$, then $(\ell, v_X) \xrightarrow{e} (\ell', v'_X)$ is a discrete transition.
  2. Given $d \in \mathbb{R}_{\geq 0}$ such that $\forall d' \in [0, d] : (\ell, v_X + d') \in S$, then $(\ell, v_X) \xrightarrow{d} (\ell, v_X + d)$ is a delay transition.

Moreover we write $(\ell, v_X) \xrightarrow{(d,e)} (\ell', v'_X)$ for a combination of a delay and discrete transition if $\exists v''_X : (\ell, v_X) \xrightarrow{d} (\ell, v''_X) \xrightarrow{e} (\ell', v'_X)$.

**Run of a TA.** Given a TA $\mathcal{A}^t$ with concrete semantics $(S, s_0, \rightarrow)$, we refer to the states of $S$ as the *concrete states* of $\mathcal{A}^t$. A concrete run $\rho$ of $\mathcal{A}^t$ is a possibly infinite alternating sequence of concrete states of $\mathcal{A}^t$, and pairs of edges and delays, starting from the initial state $s_0$ of the form $s_0 \xrightarrow{d_0,e_0} s_1 \xrightarrow{d_1,e_1} \cdots$, with $i \in \mathbb{N}$, and $d_i \in \mathbb{R}_{\geq 0}$, $e_i \in E$, and $(s_i, e_i, s_{i+1}) \in \rightarrow$. The *duration* of a finite run $\rho = s_0 \xrightarrow{d_0,e_0} s_1 \xrightarrow{d_1,e_1} \cdots \xrightarrow{d_{i-1},e_{i-1}} s_i$, is given by $\text{duration}(\rho) = \sum_{0 \leq j \leq i-1} d_j$.

### 5.2.3 Parametric timed automata

Parametric timed automata (PTA) extend timed automata with parameters within guards and invariants in place of integer constants.

**Definition 5.3: Parametric timed automaton [AHV93]**

A parametric timed automaton is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, X, \mathcal{P}, \mathcal{I}, E)$, where:

- $\Sigma$ is a finite set of actions,
- $L$ is a finite set of locations,
- $\ell_0 \in L$ is the initial location,
- $X$ is a finite set of clocks,
- $\mathcal{P}$ is a finite set of parameters,
- $\mathcal{I}$ is the invariant, assigning a guard $\mathcal{I}(\ell)$ over $X \cup \mathcal{P}$ to every location $\ell \in L$,
- $E$ is a finite set of edges $e = (\ell, \mathcal{G}, a, R, \ell')$ where $\ell, \ell' \in L$ are the
PTA to TA. Given a parameter valuation \( v_P \) and PTA \( A \), with \( v_P(A) \) we denote the non-parametric structure where all occurrences of a parameter \( p \in P \) has been replaced by \( v_P(p) \). Any structure \( v_P(A) \) is thus a timed automaton. By rescaling the constants (multiplying all constants in \( v_P(A) \) by their least common denominator), we obtain an equivalent (integer-valued) TA, as defined in [AD94].

L/U-PTAs. A subclass of a PTA, called L/U-PTA, restricts the guards such that parameters are only bounded from below, or above.

**Definition 5.4: L/U-PTA [HRSV02]**

An L/U-PTA is a PTA where the set of parameters is partitioned into lower-bound parameters and upper-bound parameters, i.e. parameters that appear in guards and invariants in inequalities of the form \( p \bowtie x \), and of the form \( p \triangleright x \) respectively.

5.2.4 Symbolic semantics of PTAs

We recall the symbolic semantics of PTAs (see e.g. [HRSV02, ACFE09]).

**Symbolic semantics.** A symbolic state is a pair \( (\ell, C) \) where \( \ell \in L \) is a location, and \( C \) its associated parametric zone (represented by a constraint over clocks and parameters).

**Definition 5.5: Symbolic semantics of a PTA**

Given a PTA \( A = (\Sigma, L, \ell_0, X, P, I, E) \), the symbolic semantics of \( A \) is defined by the labelled transition system called the parametric zone graph \( \mathcal{PZG} = (S, s_0, T) \), with

- \( S = \{((\ell, C) | C \text{ is satisfiable } \wedge \forall v_X, v_P : v_X\mid v_P \neq I(\ell) \Rightarrow v_X\mid v_P \neq C\}, \)
  i.e. for each symbolic state \( (\ell, C) \), every constraint \( C \) is at least as strict as \( I(\ell) \),
  - \( s_0 = (\ell_0, (\wedge_{1 \leq i \leq |X|} x_i = 0)^\mathcal{G} \wedge I(\ell_0)) \in S \), and
  - \( ((\ell, C), e, (\ell', C')) \in T \) if \( (\ell, C), (\ell', C') \in S \), \( e = (\ell, G, a, R, \ell') \in E \), and \( C' = (((C \land G)|_R \land I(\ell')))^\mathcal{G} \wedge I(\ell') \) such that \( C' \) is satisfiable.
That is, in the parametric zone graph, nodes are symbolic states, and arcs are labelled by the *edges* of the original PTA. Given $s = (\ell, C)$, $s' = (\ell', C')$, and $(s, e, s') \in T$, we write $\text{Succ}(s, e) = (\ell', C')$. By extension, we write $\text{Succ}(s)$ for $\bigcup_{e \in E} \{ \text{Succ}(s, e) \}$. Given $(s, e, s') \in T$, we also write $s \xRightarrow{e} s'$.

Note that, by definition, the constraint of a successor state may only add *more* constraints in addition to those of the source state, which we state in Lemma 5.1.

**Lemma 5.1: Successor constraint inclusion [HRSV02, JLR15]**

Let $(\ell', C') \in \text{Succ}((\ell, C))$. Then $C' \Downarrow_{p} \subseteq C \Downarrow_{p}$.

For example, if $p > 5$ holds in state $s$, then for every state in $\text{Succ}(s)$ (and in fact every reachable symbolic state from $s$) we also have $p > 5$. This property will become valuable in the upcoming algorithms.

A *symbolic run* $\rho$ of $A$ is a possibly infinite alternating sequence of symbolic states and edges of $A$, starting from the initial state $s_0$ of the form $s_0 \xRightarrow{e_0} s_1 \xRightarrow{e_1} \cdots$, with $i \in \mathbb{N}$, and $e_i \in E$, and $(s_i, e_i, s_{i+1}) \in T$.

**Connecting concrete and symbolic semantics.** The following results connect the concrete and the symbolic semantics.

**Lemma 5.2: Concrete run from a symbolic run [HRSV02]**

Let $A$ be a PTA, let $\rho$ be a symbolic run of $A$ reaching $(\ell, C)$, and let $v_X$ and $v_P$ be a clock and parameter valuation such that $v_X|v_P \models C$. Then, for the TA $v_P(A)$ there exists a concrete run that reaches $(\ell, v_X)$.

**Lemma 5.3: Symbolic run from a concrete run [HRSV02]**

Let $A$ be a PTA, $v_P$ a parameter valuation, and $v_X$ a clock valuation. Let $\rho$ be a concrete run of $v_P(A)$ reaching $(\ell, v_X)$, then there exists an equivalent symbolic run in $A$ reaching $(\ell, C)$, with $v_X|v_P \models C$.

5.2.5 Reachability synthesis

**Reachability.** We define reachability as the first time that a state or location has been visited in a run. Given a concrete state $s = (\ell, v_X)$ of the timed automaton $A^1$, we say that $s$ is reachable if $s$ is the last state in a run of $A^1$ for which $s$ is
not visited earlier in the run. By extension, we say that \( \ell \) is reachable; and by extension again, given a set \( T \) of locations, we say that \( T \) is reachable if there exists a location \( \ell \in T \) such that \( \ell \) is reachable in \( A^t \). The set of all finite runs of \( A^t \) that reach \( T \) (for the first time) is denoted by \( \text{Reach}(A^t, T) \).

**Optimality with comparison operators.** As the minimal and maximal time or value may not be an integer, but also the smallest value less than or larger than an integer\(^2\), we define a *comparative value* as follows.

**Definition 5.6: Comparative value**

A comparative value is either a pair in \( \mathbb{R}_{\geq 0} \times \{<,=,>\} \), or \( \infty \) (note that \( \infty \notin \mathbb{R}_{\geq 0} \)). We denote a comparative value pair \((d, \diamond)\) by \( d \diamond \). The comparison operators function as follows: \( d < < \infty \), \( d = < \infty \), \( d > < \infty \), and \( d_1^< < d_2^< \) iff either \( d_1 < d_2 \) or in case \( d_1 = d_2 \), we have that \( d_1^< < d_1^= \) and \( d_1^= < d_1^> \) (\( d_1^< < d_1^> \) follows by transitivity).

**Optimal-time reachability.** Given a TA \( A^t \) and a set of locations \( T \), the minimal-time reachability of \( T \) in \( A^t \) is defined as \( \text{MinTime} TA(A^t, T) \). This is the infimum duration \( d \in \mathbb{R}_{\geq 0} \) over all (possibly infinite) runs of \( A^t \) reaching \( T \), i.e. \( \inf \{ \text{duration}(\rho) \mid \rho \in \text{Reach}(A^t, T) \} \), combined with the relational operator = in case there exists a run \( \rho \) in \( A^t \) with \( \text{duration}(\rho) = d \), or > if no such explicit run exists, to form a comparative value (\( d = \) or \( d > \)). If \( \text{Reach}(A^t, T) = \emptyset \), then \( \text{MinTime} TA(A^t, T) = \infty \).\(^3\)

\( \text{MaxTime} TA \) is defined analogously as the maximum (supremum) duration over all runs of \( A^t \) reaching \( T \), combined with the operator = or < to indicate whether or not an explicit run exists that reaches \( T \) in time \( d \). In case \( \text{Reach}(A^t, T) = \emptyset \), then \( \text{MaxTime} TA(A^t, T) = 0^< \).\(^3\)

By extension, given a PTA \( A \), with \( \text{MinTime}(A, T) \) we denote the minimal time reachability of \( T \) over all (possibly infinitely many) valuations, or formally,

\[
\text{MinTime}(A, T) = \inf \{ \text{MinTime} TA(v_p(A), T) \mid v_p \in \mathbb{V}_p \}.
\]

As we will be interested in synthesizing the valuations leading to the minimal time, we define \( \text{MinTimeSynth}(A, T) \) as all parameter valuations that reach \( T \) in

\(^2\)Consider a TA with a transition guarded by \( x > 1 \) from \( \ell_0 \) to \( \ell_1 \), then the minimal duration of runs reaching \( \ell_1 \) is not 1, but slightly more.

\(^3\) Since the comparative values \( 0^< \) and \( \infty \) cannot be represented as parameter or clock valuations, we use these values to represent that the target location is unreachable.
minimal time:

\[
\text{MinTimeSynth}(A, T) = \{v_{P} \mid \text{MinTimeTA}(v_{P}(A), T) = \text{MinTime}(A, T)\}.
\]

We define \(\text{MaxTime}\) and \(\text{MaxTimeSynth}\) analogously, for which we maximize the
time (take the supremum) and obtain the corresponding parameter valuations,
i.e.:

\[
\text{MaxTime}(A, T) = \sup\{\text{MaxTimeTA}(v_{P}(A), T) \mid v_{P} \in V_{P}\}, \text{ and}
\]

\[
\text{MaxTimeSynth}(A, T) = \{v_{P} \mid \text{MaxTimeTA}(v_{P}(A), T) = \text{MaxTime}(A, T)\}.
\]

**Optimal-time reachability and synthesis problems.** We define the follow-
ing computation problems regarding optimal-time reachability and synthesis for
PTAs.

**Optimal-time reachability problem:**

**INPUT:** A PTA \(A\), a subset \(T \subseteq L\) of its locations.

**PROBLEM:** Compute \(\text{MinTime}(A, T)\) (or \(\text{MaxTime}(A, T)\)), i.e. the minimal
(maximal) time for which \(T\) is reachable in \(v_{P}(A)\) for any valuation \(v_{P} \in V_{P}\).

**Optimal-time synthesis problem:**

**INPUT:** A PTA \(A\), a subset \(T \subseteq L\) of its locations.

**PROBLEM:** Compute \(\text{MinTimeSynth}(A, T)\) (or \(\text{MaxTimeSynth}(A, T)\)), i.e. the set of all parameter valuations \(V_{P} \subseteq V_{P}\) such that \(T\) is reachable in minimal
(maximal) time in \(v_{P}(A)\), for each \(v_{P} \in V_{P}\).

**Optimal parameter reachability.** We will also be interested in minimizing
(and maximizing) the valuation of a given parameter \(p \in P\) (without any notion
of time), while reaching a given location, which we define as follows:

\[
\text{MinParam}(A, T, p) = \inf\{v_{P}(p) \mid \text{Reach}(v_{P}(A), T) \neq \emptyset\}.
\]

Similarly, we will be interested in synthesizing all valuations that minimize a given
parameter \(p\), while reaching \(T\), i.e.:

\[
\text{MinParamSynth}(A, T, p) = \{v_{P} \mid v_{P}(p) = \text{MinParam}(A, T, p)\}.
\]

\(\text{MaxParam}\) and \(\text{MaxParamSynth}\) are defined analogously.
Optimal-parameter reachability and synthesis problems. We define the following computation problems regarding optimal-parameter reachability and synthesis for PTAs.

**Optimal-parameter reachability problem:**
**INPUT:** A PTA $\mathcal{A}$, a parameter $p \in \mathbb{P}$, a subset $T \subseteq L$ of the locations of $\mathcal{A}$.
**PROBLEM:** Compute $\text{MinParam}(\mathcal{A}, T, p)$ (or $\text{MaxParam}(\mathcal{A}, T, p)$), i.e. the minimal (maximal) valuation for $p$ for which $T$ is reachable in $v_P(\mathcal{A})$.

**Optimal-parameter synthesis problem:**
**INPUT:** A PTA $\mathcal{A}$, a parameter $p \in \mathbb{P}$, a subset $T \subseteq L$ of the locations of $\mathcal{A}$.
**PROBLEM:** Compute $\text{MinParamSynth}(\mathcal{A}, T, p)$ (or $\text{MaxParamSynth}(\mathcal{A}, T, p)$), i.e. the set of all parameter valuations $V_P \subseteq V_P$ such that $T$ is reachable for a minimal (maximal) valuation of $p$ in $v_P(\mathcal{A})$, for each $v_p \in V_P$.

We will see in Proposition 5.4 that the optimal-parameter problems can also give answers to the minimal- and maximal-time reachability and synthesis problems.

5.2.6 Computing the optimum

**Optimum of a constraint.** Given a constraint $C$ and $p \in \mathbb{P}$, with $\text{GetMin}(C, p)$ we denote the minimum of $p$ as a comparative value. Technically, $\text{GetMin}$ can be implemented using polyhedral operations as follows: $C_{\downarrow \{p\}}$ is computed as a projection, then the infimum is extracted; and finally an operator in $\{=, >\}$ is inferred, depending on whether the lower-bound of $C_{\downarrow \{p\}}$ is exact, we either use a closed or an open constraint. As $C_{\downarrow \{p\}}$ results in a constraint of the form $\bigwedge_{i=1}^n p \bowtie_i d_i$, with $\bowtie_i \in \{<, \leq, =, \geq, >\}$ and $d_i \in \mathbb{R}$, we consider the following cases for each conjunct:

$$\text{MinConjunct}(p \bowtie_i d_i) = \begin{cases} 0^= & \text{if } d_i > 0 \land \bowtie_i \in \{<, \leq\} \\ 0^- & \text{or } d_i = 0 \land \bowtie_i \in \{\leq\} \\ 0^+ & \text{or } d_i < 0 \land \bowtie_i \in \{\geq, >\} \\ d_i^- & \text{if } d_i \geq 0 \land \bowtie_i \in \{=, \geq\} \\ d_i^+ & \text{if } d_i \geq 0 \land \bowtie_i \in \{>, \} \\ \infty & \text{otherwise (e.g. if } d_i < 0 \land \bowtie_i \in \{<, \leq, =\}). \end{cases}$$

The maximal value of all conjuncts then forms the result, i.e. given that $C_{\downarrow \{p\}}$ results in $\bigwedge_{i=1}^n c_i$ such that each $c_i$ is a conjunct of the form $p \bowtie_i d_i$, then $\text{GetMin}(C, p) = \max\{\text{MinConjunct}(c_1), \ldots, \text{MinConjunct}(c_n)\}$. If $C_{\downarrow \{p\}} = \bot$, then $\text{GetMin}(C, p) = \infty$. Since the comparative values $0^<$ and $\infty$ cannot be represented as parameter or clock valuations, we exploit these values for stating that the constraint is unsatisfiable in the definitions of $\text{GetMin}$ and $\text{GetMax}$. We extend
GetMin to accommodate clocks, thus given a clock \( x \in \mathbb{X} \), \( \text{GetMin}(C, x) \) returns the minimal clock value that \( x \) can take, while conforming to \( C \) (via computing \( C \downarrow \{x\} \)).

We define \( \text{GetMax} \) analogously for parameters and clocks, hence we extract the supremum value from the projection and combine it with the appropriate operator from \{<,=\}, which we show for a single conjunct as follows:

\[
\text{MaxConjunct}(p \triangleright_i d_i) = \begin{cases} 
  d_i^= & \text{if } d_i \geq 0 \land \triangleright_i \in \{\leq,=\} \\
  d_i^- & \text{if } d_i \geq 0 \land \triangleright_i \in \{<\} \\
  \infty & \text{if } \triangleright_i \in \{\geq,>\} \\
  0^< & \text{otherwise} \text{ (if } d_i < 0 \land \triangleright_i \in \{<,\leq,=\} \text{).}
\end{cases}
\]

The minimal value of all conjuncts then forms the result, i.e. given that \( C \downarrow \{p\} \) results in \( \bigwedge_{i=1}^n c_i \) such that each \( c_i \) is a conjunct of the form \( p \triangleright_i d_i \), then \( \text{GetMax}(C, p) = \min\{\text{MaxConjunct}(c_1), \ldots, \text{MaxConjunct}(c_n)\} \). If \( C \downarrow \{p\} = \bot \), then \( \text{GetMax}(C, p) = 0^< \).

\section*{5.3 Related work}

For related work, we distinguish between decidability results for TAs and PTAs, optimal-time reachability algorithms for TAs, and related work on reachability and synthesis for PTAs.

\subsection*{5.3.1 Decidability}

Given a timed automata \( \mathcal{A}^t \), and a set of locations \( T \), the reachability problem for TAs is to decide whether a target location \( \ell \in T \) is reachable from the initial state, i.e. deciding whether \( \text{Reach}(\mathcal{A}^t, T) \neq \emptyset \). This has famously been shown to be PSPACE-complete by Alur and Dill [AD94].

\textbf{Optimal-time reachability for TAs.} The earliest work on optimal-time reachability was by Courcoubetis and Yannakis [CY92], who first addressed the problem of computing lower and upper bounds in timed automata. They showed that the minimal- and maximal-time reachability problems are both PSPACE-complete. We note that optimal-time reachability has also been proven to be PSPACE-complete for weighted/priced timed automata [BBBR07], which extend TAs with weights on both locations and edges. For more decidability results on timed automata, we would like to refer the reader to a survey on the topic by Waez et al. [WDR13].
Undecidability for PTAs. The reachability problem is undecidable for PTAs [AHV93], which implies that every problem from Section 5.2.5 is undecidable as well, since solving any of the stated problems would also give an answer to the reachability problem.

The problem remains undecidable with various restrictions on a PTA, e.g. a PTA that only allows for a single clock [Mil00], or a PTA with a single rational-valued or integer-valued parameter [Mil00, BBL15]. However, the problem becomes decidable when the number of clocks is limited and the parameters are restricted as well (to e.g. the integer domain) [AHV93, BO14, BBL15, AM15]. See the survey by André [And18c] for an overview on decidability results for PTAs.

Decidability for L/U-PTAs. The reachability and synthesis problems are decidable for the subclass of L/U-PTAs [BL09]. A consequence is that the minimal- and maximal-parameter reachability problems are decidable for L/U-PTAs, since the minimum (and maximum) parameter values can be extracted from the synthesis results. We show in Proposition 5.2 that the minimal- and maximal-time reachability problems are also decidable for L/U-PTAs.

5.3.2 Optimal-time reachability for timed automata

There are various solutions to the minimal-time reachability problem for TAs, while maximal-time variant has been studied to a lesser extent. In [AM99], the authors proposed a backward fix-point computation to find the minimal time. A downside of this approach is that it might explore more states than necessary. Niebert et al. [NTY00] propose three algorithms to improve on the earlier work. Zhang et al. [ZNL16b, ZNL16a] present distributed algorithms for the minimal-time reachability problem, which are based on swarm verification, i.e. spawning multiple randomized search instances. These randomized searches provide an approximation of the optimal time, which is then used to prune the search space for computing the optimal result.

The maximal-time reachability problem is considered in [ARF17], which is more complex due to the fact that (1) the upper-bound time is not necessarily monotonically increasing during a run (as opposed to the lower-bound time, which we show in Proposition 5.10), and (2) cycles have to be considered (for minimal-time reachability a path does never traverse the same state more than once [NTY00]). Minimal-time reachability has also been studied for priced timed automata in various works [LBB+01, ALP04, BLR05].
5.3. Related work

5.3.3 Reachability and synthesis for PTAs

We note that while there is plentiful work on optimal-time reachability in timed automata, the algorithms do not necessarily translate well to a parametric domain. The obvious reason is due to the added complexity from parameters, a consequence is that successor computations become significantly (often by an order of magnitude) more expensive. This is because the difference bounded matrix (DBM) representation, which is an efficient data structure for TAs, cannot be used for PTAs and no efficient alternative yet exists [And18b]. Hune et al. [HRSV02] proposed a parametric extension for DBMs, but this structure still requires the use of a polyhedra library.

Parameter synthesis. As mentioned in Section 5.3.1, reachability and synthesis are undecidable for PTAs. While this is the case, the problem may still be partly solved with a semi-algorithm, i.e., a procedure that may not terminate, but if it does then the result is sound and complete. In Algorithm 15 we present the semi-algorithm from [JLR15] for performing reachability synthesis.

The EFSynth algorithm traverses the parametric zone graph of the PTA from the initial symbolic state. It continues to explore successor states that (1) are not in the \( W \) set (the set of to-be-explored states) and (2) are not in the \( P \) set (the set of explored states). Once a target location is reached (line 9), the constraint for the state, projected on the parameters, is stored in \( K \) (successors of the target state do not have to be considered as a result of Lemma 5.1). When the \( W \) set is empty, the algorithm returns all stored constraints. For correctness, we refer to Jovanović et al. [JLR15] (see also Proposition 5.1).

**Proposition 5.1: Correctness of Algorithm 15 [JLR15]**

Assume that EFSynth\((A, T)\) terminates with result \( K \), then \( v_P \models K \) iff \( T \) is reachable in \( v_P(K) \). In other words, if the algorithm terminates, then \( K \) contains exactly the set of parameter valuations s.t. \( T \) is reachable.

Note that depending on the implementation of \( W \), the algorithm can explore the parametric zone graph in different orders. The algorithm explores the graph in a BFS order if \( W \) is implemented as a FIFO list, with the first element being selected, and it searches in a DFS order if \( W \) is implemented as a stack.

Merging states. There are two main approaches used in practice to improve the performance of the symbolic state space computation. These techniques are
Algorithm 15: Semi-algorithm for synthesis for PTA.

input : A PTA $A$ with symbolic initial state $s_0 = (\ell_0, C_0)$, a set of target locations $T$.

output: Constraint $K$ over the parameters.

1 function $EFSynth(A, T)$
   // Initialization
   2 $W := \{s_0\}$ // waiting set
   3 $P := \emptyset$ // passed set
   4 $K := \bot$ // current optimum valuations
   // Main loop
   5 while $W \neq \emptyset$ do
      6 $s = (\ell, C) \in W$ // Pick a state from $W$
      7 $W := W \setminus \{s\}$
      8 $P := P \cup \{s\}$
      // target state: store constraint and do not explore successors
      9 if $\ell \in T$ then $K := K \vee C\downarrow_P$
      // otherwise explore successors
     10 else
       11 for each $s' \in Succ(s)$ do
          // add to waiting set if not seen before
          12 if $s' \notin W \land s' \notin P$ then $W := W \cup \{s'\}$
     13 return $K$

compatible with the $EFSynth$ algorithm, as well as the algorithms that we will propose later in the chapter.

- **State inclusion** [DT98]: Given two symbolic states $s_1 = (\ell_1, C_1)$ and $s_2 = (\ell_2, C_2)$ with $\ell_1 = \ell_2$, we say that $s_1$ is included in $s_2$ if all parameter valuations for $s_1$ are also contained in $s_2$, e.g. assume that $C_1$ is $p > 5$ and $C_2$ is $p > 2$. We have that $C_1$ is included in $C_2$ and may thus conclude that $s_1$ can be ignored. This check can be performed efficiently in the successor computation ($Succ$) to remove included states. Note that it does not alter correctness for minimal- and maximal-time (or parameter) reachability synthesis.

- **Convex state merging** [AFS13]: Two states $s_1 = (\ell_1, C_1)$ and $s_2 = (\ell_2, C_2)$ can be merged if $\ell_1 = \ell_2$ and $C_1 \cup C_2$ is a convex polyhedron. The resulting state $(\ell_1, C_1 \cup C_2)$ replaces $s_1$ and $s_2$ and is an over-approximation of both states. As a result, reachable locations, minimality, maximality, and executable actions are preserved. However, the technique is computationally
intensive in practice and should therefore not be applied abundantly.

**Reachability preservation.** The *Inverse Method* (IM) is a semi-algorithm that, given a PTA $A$ and a parameter valuation $v_p$, synthesises all other valuations such that the same *traces* are preserved. By taking the path directed by $v_p$, the constraints in the PTA are used to synthesize other valuations [ACFE09, AM15].

A different approach that combines IM and EFSynth, called *Parametric Reachability Preservation* (PRP), synthesises valuations such that the *reachability* of the target location is preserved [ALNS15]. This technique starts with the IM algorithm until a target location is reached, then it continues by performing EF SYNTH to explore the remainder of the state space and returns the associated constraints from all reached target locations.

**Behavioural cartography.** Given a bounded parameter domain, *behavioural cartography* is a technique to compute the set of valuations that reach a target state. The idea is to split up the parameter domain into integer points, and using either IM or PRP for a given point to compute a larger set of parameter constraints such that the (un)reachability of the target location is preserved. This process is repeated for remaining points that are not yet covered by the computed constraints [AF10, ALNS15]. An advantage is that this approach can provide meaningful results even if it does not terminate, and it can also be executed in a distributed setting [ACN15].

**Optimal-time parameter synthesis.** Regarding the minimal- and maximal-time problems for PTAs, Luthmann et al. [LGS+19] studied them in the context of generating test cases. Their method was constructed at the same time and independent of ours. It adds an additional clock and parameters for tracking the total time (which we will also propose), then it computes all parameter valuations that reach the target locations (and also considers all different untimed runs), and finally derives the parameter valuations from the result that lead to the minimum and maximum times. It could be implemented as an application of Algorithm 15 and intersecting the resulting constraint $K$ with the parameter of interest to obtain its minimal and maximal value.

While the approach by Luthmann et al. [LGS+19] is useful for simultaneously extracting both the minimal- and maximal-time, in our work we focus on the separate problems of extracting either the minimal time or the maximal time. In terms of performance, the technique from [LGS+19] requires at least as much time as the standard parameter synthesis algorithm (i.e. Algorithm 15). We will show in Section 5.7 that our approach requires significantly less time to compute
an answer. Aside from the work from Luthmann et al., we are unaware of other studies that addressed the optimal-time and parameter reachability and synthesis problems.

5.4 Computability and intractability

We provide theoretical results on the decidability for the research problems presented in Section 5.2.5, for PTAs and L/U-PTAs.

5.4.1 Optimal-time reachability/synthesis for L/U-PTAs

Optimal-time reachability for L/U-PTAs. The following result is a consequence of a monotonicity property of L/U-PTAs [HRSV02]. We can safely replace parameters with some constants in order to compute the solution to the optimal-time reachability problems, which reduces to the optimal-time reachability in a TA, which is PSPACE-complete [CY92].

**Proposition 5.2: Optimal-time reachability for L/U-PTAs**

The minimal- and maximal-time reachability problems for L/U-PTAs are PSPACE-complete.

We omit the proof for the maximal-time case as it is analogous to the one given for the minimal-time reachability problem.

**Proof.** We show that the minimal-time reachability problem for PTAs reduces to the minimal-time reachability problem for TAs.

Let $\mathcal{A}$ be an L/U-PTA. Let $v_{p_{0,\infty}}$ denote the valuation assigning every lower-bound parameter (resp. upper-bound parameter) in the guards of $\mathcal{A}$ to 0 (resp. $\infty$). Let $\mathcal{A}_{0,\infty} = v_{p_{0,\infty}}(\mathcal{A})$ denote the structure obtained as follows: any occurrence of a lower-bound parameter is replaced with 0, and any occurrence of a conjunct $x \prec p$ (where $p$ is necessarily an upper-bound parameter) is deleted, i.e. replaced with $true$. Note that $x \prec \infty$ is always satisfiable, and therefore equivalent to $true$.

We show that the minimal-time reachability problem for the L/U-PTA $\mathcal{A}$ is equivalent to the minimal-time reachability problem for the TA $\mathcal{A}_{0,\infty}$.

$\Rightarrow$ Let $d$ be the solution to the minimal-time reachability problem for $\mathcal{A}$, i.e. $MinTime(\mathcal{A}, T)$. Then we must show that $T$ is reachable in $d$ time units in $\mathcal{A}_{0,\infty}$.
Recall that \( \text{MinTime}(A, T) = \inf \{ \text{MinTime} TA(v_p(A), T) \mid \forall v_p \in V_p \} \). Let \( v_p \) be a valuation\(^4\) for which the minimal time is obtained. Let \( \rho \) be a run of \( v_p(A) \) for which this minimal time is obtained. We note the following monotonicity result for L/U-PTAs. Basically, any run of a valuation is also a run of a “larger” valuation (i.e. smaller lower-bound parameters and larger upper-bound parameters).

**Lemma 5.4: Monotonicity for L/U-PTAs [HRSV02]**

Let \( A \) be an L/U-PTA and \( v_p \) be a parameter valuation. Let \( v'_p \) be a valuation such that for each upper-bound parameter \( p^+ \), \( v'_p(p^+) \geq v_p(p^+) \) and for each lower-bound parameter \( p^- \), \( v'_p(p^-) \leq v_p(p^-) \). Then any run of \( v_p(A) \) is a run of \( v'_p(A) \).

Therefore, \( \rho \) is a run of \( A_{0,\infty} \) and \( T \) is reachable in \( d \) time units in \( A_{0,\infty} \).

\[\iff\]

Let \( d \) be the solution of the minimal-time reachability problem for \( A_{0,\infty} \), i.e. \( \text{MinTime} TA(A_{0,\infty}, T) \), we must show that there exists a parameter valuation \( v_p \) such that \( T \) is reachable in \( d \) time units in \( v_p(A) \).

Let \( \rho \) be a run of \( A_{0,\infty} \) for which \( T \) is reachable in \( d \) time units. The result could follow immediately from Lemma 5.4—if only assigning 0 and \( \infty \) to parameters was a proper parameter valuation. From [HRSV02, BL09], if a location is reachable in the TA obtained by valuating lower-bound parameters with 0 and upper-bound parameters with \( \infty \), then there exists a sufficiently large constant \( C \) such that this run exists in \( v_p(A) \) such that \( v_p \) assigns 0 to lower-bound and \( C \) to upper-bound parameters. Here, we can trivially pick \( d \), as any clock constraint \( x \leq d \) will be satisfied for a run of duration \( d \). Let \( v_p \) assign 0 to lower-bound and \( d \) to upper-bound parameters. Then, \( \rho \) is a run of \( v_p(A) \). Therefore, \( T \) is reachable in \( d \) time units in \( v_p(A) \), which concludes the proof.

The result finally follows from the fact that minimal- and maximal-time reachability problem for TAs is PSPACE-complete [CY92].

**Intractability of optimal-time L/U-PTA synthesis.** Computing the optimal time for which a location is reached (Proposition 5.2) does not mean that we are able to exactly compute all valuations for which this location is reachable in minimal or maximal time. In fact, we show that it is not possible in a formalism for which the emptiness of the intersection is decidable—which notably rules out that the solution cannot be represented as a finite union of polyhedra.

\(^4\)This valuation is not necessarily unique.
Figure 5.3: Intractability of minimal-time reachability synthesis for L/U-PTAs. The yellow dotted state depicts an urgent location and \( \ell'_f \) is a target location.

Proposition 5.3: Intractability of optimal-time synthesis for L/U-PTA

The solution to the minimal- and maximal-time reachability synthesis problem for L/U-PTAs cannot be represented in a formalism for which the emptiness of the intersection is decidable.

We prove the proposition for the minimal-time synthesis problem. A similar approach can be used for proving the maximal-time case.

Proof. We assume that the solution to the minimal–time reachability synthesis problem for L/U-PTAs can be represented in a formalism for which the emptiness of the intersection is decidable and show that this leads to a contradiction.

Assume an arbitrary PTA \( \mathcal{A} \) with initial location \( \ell_0 \) and target location \( \ell_f \). We add a new clock \( x \) that is not used in \( \mathcal{A} \) (and never resets) and add a new upper-bound parameter \( p^u \). We augment \( \mathcal{A} \) as follows: add a new initial location \( \ell'_0 \), and a transition guarded with \( x = 0 \) from \( \ell'_0 \) to \( \ell_0 \). Add a transition guarded by \( x = 2 \land x < p^u \) from \( \ell'_0 \) to a new location \( \ell'_f \). Add a transition guarded by \( x \leq 1 \) from \( \ell_f \) to \( \ell'_f \). Make \( \ell_f \) urgent\(^5\). The construction is given in Figure 5.3.

We change the PTA \( \mathcal{A} \) into an L/U-PTA as follows: for any parameter \( p' \), we replace any guard of the form \( x \ll p', x \gg p' \), and \( x = p' \) with \( x \ll p'^u, x \gg p'^l \), and \( p'^l \leq x \leq p'^u \), respectively. The obtained PTA \( \mathcal{A}' \) made of the parameters set \( \{p'^l, p'^u \mid p' \in \mathbb{P}\} \cup \{p^u\} \) is an L/U-PTA.

Clearly, \( \ell'_f \) is reachable in \( \mathcal{A}' \) in time 2 by taking the transition from \( \ell'_0 \) to \( \ell'_f \), for any valuation \( v_\mathbb{P} \) such that \( v_\mathbb{P}(p^u) > 2 \). In addition, it is reachable in \( \mathcal{A}' \) in time \( \leq 1 \) for all valuations of \( p^u \) iff there exists a parameter valuation for which \( \ell_f \) is reachable in \( \mathcal{A} \) in \( \leq 1 \) time unit.

\(^5\)An urgent location is a location where time cannot elapse (depicted in dotted yellow in our figures, and which can be encoded using an extra clock).
Let $K$ be the solution for the synthesis problem in $A'$ for $T = \{\ell_f\}$. Then, there exists a parameter valuation reaching $\ell_f$ in $A$ in time $\leq 1$ iff the intersection of $K$ with $p^n < 2 \land \bigwedge_i p^n_i = p^n_i$ is non-empty. But since reachability emptiness is undecidable for PTAs over bounded time (typically in $\leq 1$ time unit) [ALM18, Theorem 17], this leads to a contradiction. Therefore, $K$ cannot be represented in a formalism for which the emptiness of the intersection is decidable. 

5.4.2 Optimal-parameter reachability and synthesis

For the full class of PTAs, we will see that these problems are clearly out of reach: if it was possible to compute the solution to the optimal-parameter reachability and synthesis problems, then it would be possible to answer the reachability problem—which is undecidable in most settings [And18c].

We first show that an algorithm for the optimal-parameter synthesis problems can be used to solve the optimal-time synthesis problems, i.e. the optimal-parameter synthesis problems are harder than the optimal-time problems.

**Proposition 5.4: Optimal-time from optimal-parameter synthesis**

An algorithm that solves the minimal-parameter (or maximal-parameter) synthesis problem can be used to solve the minimal-time (maximal-time) synthesis problem by extending the PTA.

**Proof.** Assume we are given an arbitrary PTA $A$, a set of target locations $T$, and a global clock $x_{\text{global}}$ that never resets. We construct the PTA $A'$ from $A$ by adding a new parameter $p_{\text{global}}$, and for every edge $(\ell, G, a, R, \ell')$ in $A$ such that $\ell' \in T$, we replace $G$ by $G \land x_{\text{global}} = p_{\text{global}}$. Note that when a target location from $T$ is reached (for the first time), we have that $x_{\text{global}} = p_{\text{global}}$, hence by minimizing $p_{\text{global}}$ we also minimize $x_{\text{global}}$. Thus, by solving $\text{MinParamSynth}(A', T, p_{\text{global}})$, we effectively solve $\text{MinTimeSynth}(A, T)$. The same holds for the maximal-time problem, where solving $\text{MaxParamSynth}(A', T, p_{\text{global}})$ provides a solution for $\text{MaxTimeSynth}(A, T)$. 

Intractability of optimal-parameter reachability for PTAs. The following result states that synthesis of the optimal-value of the parameter is intractable for PTAs.
Proposition 5.5: Intractability of optimal-parameter reachability and synthesis for PTAs

The solution to the minimal- and maximal-parameter reachability and synthesis for PTAs cannot be computed in general.

We provide the proof for the minimal-parameter reachability and synthesis problems. The proof for the maximal-parameter problems are omitted, but a similar construction could be used.

Proof. We assume that the solution to the minimal-parameter reachability for PTAs can always be computed and show that this leads to a contradiction.

Assume an arbitrary PTA $A$ with an initial location $\ell_0$ and a given target location $\ell_f$. Add a new clock $x$ and a new parameter $p$ not used in $A$. Augment $A$ as follows: add a new initial location $\ell'_0$, and a transition guarded with $x = 0$ from $\ell'_0$ to $\ell_0$. Add an unguarded transition from $\ell_f$ to a new location $\ell_f'$ resetting $x$, and then a transition guarded by $x = 0 \land x = p$ from $\ell_f'$ to a new location $\ell_f''$. Add an unguarded transition from $\ell_0$ to $\ell_f'$ guarded with $x = 1 \land x = p$. Let $A'$ denote this augmented PTA. The construction is given in Figure 5.4.

Clearly, $\ell_f'$ is reachable in $A'$ if $p = 1$. In addition, it is reachable in $A'$ for $p = 0$ iff there exists a parameter valuation for which $\ell_f$ is reachable in $A$. Now, assume the solution to the minimal-parameter reachability for $A'$ and $p$ can be computed. Let $K$ denote this solution (which will be $p = 0$ or $p = 1$ depending on whether $\ell_f$ is reachable in $A$). Then, there exists a parameter valuation reaching $\ell_f$ in $A$ iff $K$ is equal to $p = 0$. But since reachability emptiness is undecidable for PTAs [AHV93], this leads to a contradiction. Hence, $K$ cannot be computed in general.

Since minimal-parameter reachability is intractable, we can imply that the minimal-parameter synthesis problem for PTAs is intractable as well. \qed
5.5. Optimal parameter and time synthesis

**Intractability of the synthesis for L/U-PTAs.** The following result states that optimal-parameter synthesis is intractable for L/U-PTAs. Notably, this rules out the possibility to represent the result using a finite union of polyhedra.

<table>
<thead>
<tr>
<th>Proposition 5.6: Intractability of optimal-parameter synthesis for L/U-PTAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>The solution to the minimal- and maximal-parameter synthesis problems for L/U-PTAs cannot always be represented in a formalism for which the emptiness of the intersection is decidable.</td>
</tr>
</tbody>
</table>

*Proof.* From Propositions 5.3 and 5.4.

The optimal-parameter reachability problems remain open for L/U-PTAs.

Despite these negative results, we will define procedures that address not only the class of L/U-PTAs, but in fact the class of full PTAs. Of course, these procedures are not guaranteed to terminate. When the procedures do terminate, they produce correct and complete results.

### 5.5 Optimal parameter and time synthesis

We present a semi-algorithm that, if it terminates, provides a solution to the minimal-parameter reachability synthesis problem. The algorithm is based on the EFSynth algorithm from Algorithm 15. We prove it correct and show how it can be modified to also solve the maximal-parameter problem, the optimal-time problem, and also the reachability variants.

#### 5.5.1 Algorithm

We give MinParamSynth$(A, T, p)$ in Algorithm 16. It maintains a set $W$ of waiting (to-be-explored) symbolic states, a set $P$ of passed states, a current optimum $Opt$ and the associated optimal valuations $K$.

While $W$ is not empty, a state is picked in line 7. If it is a target state ($\ell \in T$) then the projection of its constraint onto $p$ is computed, and the minimum is inferred (line 11). If that projection improves the known optimum, then the associated parameter valuations $K$ are replaced by the one obtained from the current state (i.e. the projection of $C$ onto $P$, intersected with $p = s_{opt}$).
Technicalities 5.1: Implementation of comparative values

In case \( s_{opt} \) is of the form \( c^> \) with \( c \in \mathbb{R}_{\geq 0} \), then the intersection of \( C \) with the linear term \( p = c \) would result in \( \bot \), as the exact value \( c \) is not part of the constraint, i.e. \( (p > c \land p = c) = \bot \). In an implementation we intersect \( C \) with \( p = s_{opt} + \varepsilon \), for some \( \varepsilon > 0 \), to obtain a non-empty constraint \( K' \). The exact value for \( \varepsilon \) is not important, since \( K' \) contains the minimal value for \( p \). The algorithm may thus terminate with a lower- and upper-bound for \( p \) in its final constraint \( K \). The final parameter constraints can be obtained by intersecting \( K \) with the lower-bound value for \( p \) (which is the smallest value greater than \( s_{opt} \) that the implementation can represent).

Otherwise, if \( s_{opt} \) is equal to the known optimum (line 15), then we add (using disjunction) the associated valuations. Note that if \( s_{opt} > Opt \), the state is ignored since successor states can only further restrict the constraint.

If the picked state is not a target state, then we compute and process its successors. We check in line 19 whether the successor is not already part of the waiting set nor part of the passed set. Then, as an optimization, we compute \( \text{GetMin}(C', p) \) and make sure that this value does not exceed the current optimum. If this value does exceed \( Opt \), then we can safely discard \( s' \) as neither it nor its successors, can improve the end result. Otherwise we add \( s' \) to the waiting set and process it later.

Example 5.4: A run of Algorithm 16

Consider the PTA \( A \) in Figure 5.5, and run \( \text{MinParamSynth}(A, \{\ell_3\}, p_1) \). We describe the process as follows.

1. The initial state is \( s_1 = (\ell_1, x \geq 0) \) (and trivial constraints \( p_i \geq 0 \)).
2. \( s_1 \) is not a target state, thus its successors are computed. Its successors \( s_2 = (\ell_3, x \geq 2 \land p_1 > 2) \) and \( s_3 = (\ell_2, x \geq 0 \land p_2 > 1) \) are added to \( W \) (\( Opt = \infty \), so the condition on line 20 is trivially true).
3. In the second iteration, pick \( s_2 \) from \( W \): it is a target, and therefore \( \text{GetMin}(C_2, p_1) \) is computed, which gives \( 2^> \). Since \( 2^> < \infty \), we found a new minimum, and \( K \) becomes \( (C_2 \land p_1 = s_{opt}) \downarrow \mathcal{P} \), i.e. \( p_1 = 2^> \).
4. Pick \( s_3 \) from \( W \): it is not a target, therefore we compute its successors \( s_4 = (\ell_3, x \geq 2 \land p_1 = 2 \land 1 < p_2 < 2) \) and \( s_5 = (\ell_3, x \geq 2 \land p_1 = p_3 = 2 \land p_2 > 1) \). Both states get added to \( W \).
5. Pick \( s_4 \): it is a target, with \( \text{GetMin}(C_4, p_1) = 2^= \). Since we have \( 2^= < 2^> \), we found a new minimum, and \( K \) is replaced with the constraint \( (C_4 \land p_1 = s_{opt}) \downarrow \mathcal{P} \), i.e. \( p_1 = 2^= \land 1 < p_2 < 2 \).
6. Pick \( s_5 \): it is a target, with \( \text{GetMin}(C_5, p_1) = 2^= \). As we have \( 2^= = \)

**Input**: A PTA \( \mathcal{A} \) with symbolic initial state \( s_0 = (\ell_0, C_0) \), a set of target locations \( T \), a parameter \( p \).

**Output**: Constraint \( K \) over the parameters, which solves \( \text{MinParamSynth}(\mathcal{A}, T, p) \).

```plaintext
function MinParamSynth(\mathcal{A}, T, p)
    // Initialization
    W := \{s_0\} // waiting set
    P := \emptyset // passed set
    Opt := \infty // current optimum
    K := \bot // current optimum valuations
    // Main loop
    while W \neq \emptyset do
        s = (\ell, C) \in W // pick a state from W
        W := W \setminus \{s\}
        P := P \cup \{s\}
        if \ell \in T then // \( s \) is a target state
            s_{opt} := GetMin(C, p) // compute local optimum
            if s_{opt} < Opt then // the optimum is strictly better
                Opt := s_{opt} // we found a new best optimum: replace it
                K := (C \land p = s_{opt}) \downarrow_P // replace the found valuations
            else if s_{opt} = Opt then // the optimum is equal
                K := K \lor (C \land p = s_{opt}) \downarrow_P // add the found valuations
            else // otherwise explore successors
                for each \( s' = (\ell', C') \in \text{Succ}(s) \) do
                    if \( s' \notin W \land s' \notin P \) then
                        // also ignore successors that cannot improve the result
                        if GetMin(C', p) \leq Opt then W := W \cup \{s'\}
        return K
```

2\(^{2} \), we found an equally good minimum, and \( K \) is improved with \( (C_5 \land p_1 = s_{opt}) \downarrow_P \), giving a new \( K \) equal to \( p_1 = 2^2 \land 1 < p_2 < 2 \vee (p_1 = p_3 = 2^2 \land p_2 > 1) \).

7. As \( W = \emptyset \), the algorithm terminates and \( K \) is returned.
5.5.2 Correctness

We prove Algorithm 16 correct by proving that it is both sound and complete.

**Proposition 5.7: Soundness**

Assume that $\text{MinParamSynth}(A, T, p)$ terminates with result $K$ and let $v_p \models K$. Then $v_p \models \text{MinParamSynth}(A, T, p)$.

*Proof.* Recall that $\text{MinParamSynth}(A, T, p) = \{v_p \mid v_p(p) = \text{MinParam}(A, T, p)\}$. Let us first show that $\text{Reach}(v_p(A), T) \neq \emptyset$, i.e. that $T$ is reachable in $v_p(A)$. From Algorithm 16 (lines 14 and 16), $K$ is only made of the projection onto $\mathbb{P}$ of constraints associated with target symbolic states (i.e. such that $\ell \in T$). Therefore, from Lemma 5.2 there exists an equivalent concrete run reaching $T$ in $v_p(A)$, which gives that $\text{Reach}(v_p(A), T) \neq \emptyset$.

Let us now show that $v_p(p) = \text{MinParam}(A, T, p)$. First, notice that the entire parametric zone graph of $A$ is explored by Algorithm 16, except when branches are cut (i.e. the successors are not explored), which occurs at two places:

- When a target state is encountered (line 10). Here, the constraint $C$ is added to $K$, or ignored completely if it cannot be part of the optimum.
- When the successor states are processed (line 20). Here, we (also) discard the state completely if it cannot improve the result.

From Lemma 5.1 we can imply that the unexplored symbolic states do not add any valuation to the known valuation in $K$. In addition, as Algorithm 16 iteratively searches for the minimal $Opt$, then

1. $Opt$ is eventually the minimum of $p$, and

Figure 5.5: A PTA for Example 5.4 and Example 5.6.
5.5. Optimal parameter and time synthesis 193

Therefore, \( \forall v_p \in V_p \) such that \( v_p \models K \), we have \( v_p(p) = \text{MinParam}(A,T,p) \) and \( v_p \models \text{MinParamSynth}(A,T,p) \).

**Proposition 5.8: Completeness**

Assume that \( \text{MinParamSynth}(A,T,p) \) terminates with result \( K \). Let \( v_p \models \text{MinParamSynth}(A,T,p) \). Then \( v_p \models K \).

**Proof.** We use a reasoning dual to Proposition 5.7. By definition of \( \text{MinParam} \), \( v_p \) is the valuation such that \( v_p(p) \) is the smallest value of \( p \) for which \( T \) is reachable (i.e. \( p = \text{Opt} \)). Since \( \exists \ell \in T \) reachable in \( v_p(A) \), from Lemma 5.3, there exists an equivalent symbolic run in \( A \) reaching \( (\ell,C) \), with \( v_p \models (C \land p = \text{Opt}) \downarrow P \).

In addition, consider the way the minimum is managed in Algorithm 16 together with the fact that the unexplored states do not bring any interesting valuation (Lemma 5.1). Hence, the constraint for the symbolic state \( (\ell,C) \) is kept by Algorithm 16, either at line 14 or line 16, and no further symbolic state will replace it. Thus, \( (C \land p = \text{Opt}) \downarrow P \subseteq K \), and therefore \( v_p \models K \).

**Theorem 5.1: Correctness**

Assume \( \text{MinParamSynth}(A,T,p) \) terminates with result \( K \). Let \( v_p \) be a parameter valuation. Then \( v_p \models K \) iff \( v_p \models \text{MinParamSynth}(A,T,p) \).

**Proof.** From Propositions 5.7 and 5.8.

5.5.3 A subclass for which the solution can be computed

Algorithm 16 is a semi-algorithm; if it terminates with result \( K \), then \( K \) is a solution for the \( \text{MinParamSynth} \) problem. We showed its correctness in Theorem 5.1. Furthermore, the minimum is tracked and updated whenever a target state is reached. We show that synthesis can effectively be achieved for PTAs with a single clock, which is a decidable subclass [Mil00].

**Proposition 5.9: Synthesis for one-clock PTAs**

The solution to the minimal-parameter reachability synthesis can be computed for 1-clock PTAs using a finite union of polyhedra.
Proof. Let us prove termination of Algorithm 16. André and Markey [AM15] showed that the parametric zone graph of a 1-clock PTA is finite. By computing successors of symbolic states, Algorithm 16 clearly explores (a sub-part of) the parametric zone graph of $A$. In addition, no symbolic state is explored twice, thanks to the $P$ set. Therefore, Algorithm 16 terminates for 1-clock PTAs and returns a finite union of polyhedra (from the way $K$ is synthesized). The correctness follows from Theorem 5.1.

5.5.4 Maximal-parameter and reachability problems

Maximal-parameter synthesis. With a slight modification to Algorithm 16, it can provide a solution to the maximal-parameter synthesis problem. We replace the initial assignment of $Opt$ to $Opt := 0^<$ in line 4, replace the occurrences of GetMin by GetMax, the comparison $s_{opt} < Opt$ by $s_{opt} > Opt$ at line 12, and the comparison on line 20 by $GetMax(C', p) \geq Opt$. The locations of the stated modifications are marked yellow in the algorithm.

Optimal-parameter reachability. When concerned with the optimal-parameter reachability problems, the synthesis algorithms already provide a solution. With performance in mind, there is a subtle change we can make in the algorithm, namely to replace the comparison from line 20 to $GetMin(C', p) < Opt$ (or $GetMax(C', p) > Opt$ when maximizing a parameter value) as we are only interested in successor states that may improve the current optimum parameter value, hence we can ignore valuations leading to equal optimums$^6$. Furthermore, lines 15 and 16 may be removed as we only have to provide a single valuation. Thus, we only update $K$ in line 14 whenever a better optimum is found.

Preserving correctness. For the maximal-parameter synthesis and the variants concerning reachability, correctness is preserved. The proof for the maximal-parameter synthesis version follows from the proof of Theorem 5.1. One can observe that the modifications to the algorithm preserve reachability and maximize $p$ instead of minimizing it.

For reachability, it is sufficient to show that the result $K$ contains a parameter valuation that minimizes (or maximizes) $p$, which is clearly the case as the algorithm

$^6$ We remark that in case the optimal value for $p$ is $\infty$ (or 0 in case $p$ is maximized), we must ensure that $K$ contains at least one valuation, which can be realized by adding a simple emptiness check of $K$ to line 20, i.e. $K = \bot$. Thus, it is not strictly correct to ignore all valuations with equal optimums.
modifications preserve that the result contains at least one parameter valuation such that $p$ is minimized (maximized) and $T$ is reached.

### 5.5.5 Optimal-time reachability and synthesis

For optimal-time reachability and synthesis, we assume that the PTA contains a global clock $x_{global}$ that is never reset. In case this is not true, we extend the PTA by simply adding a ‘dummy’ clock $x_{global}$ without any associated guards or invariants for it.

For any of the solutions below, we modify the return value at line 21 of Algorithm 16 from $K$ to $(Opt, K)$, since we will also be interested in the optimal time value.

**Solution by extending the PTA.** From Proposition 5.4 we have that the PTA could be extended with a parameter $p_{global}$ to solve the minimal- and maximal-time synthesis and reachability problems. The downsides of this approach are that (1) an additional parameter $p_{global}$ is added to the PTA, which affects performance, and (2) since $p_{global}$ is only constrained when the target state is reached (i.e. $x_{global} = p_{global}$), there is generally little performance to be gained from cutting branches.

**Monotonicity and optimal time.** Instead of optimizing a given parameter, Algorithm 16 could also be modified to minimize or maximize a clock directly. However, before we discuss these modifications, we state the following monotonicity property on the minimal (or lower-bound) time of a symbolic state.

**Proposition 5.10: Monotonic minimal-time for symbolic states**

<table>
<thead>
<tr>
<th>Proposition 5.10: Monotonic minimal-time for symbolic states</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a PTA $A$ with symbolic states $s_1, s_2 \in S$ with $s_1 = (\ell_1, C_1)$ and $s_2 = (\ell_2, C_2)$, such that $s_2 \in \text{Succ}(s_1)$ and let $x_{global}$ be a clock that is not reset on the arc from $s_1$ to $s_2$. Then $\text{GetMin}(C_1, x_{global}) \leq \text{GetMin}(C_2, x_{global})$, i.e. the minimal time increases monotonically.</td>
</tr>
</tbody>
</table>

**Proof.** By definition of a PTA’s symbolic semantics (Definition 5.5) we have that $C_2 = \left(\left(\left([C_1 \wedge G]\right)_R \wedge \mathcal{I}(\ell_2)\right)^\triangleright \wedge \mathcal{I}(\ell_2)\right)$. Since $x_{global}$ is never reset and can therefore only increase in a symbolic transition, we maintain that $x_{global}$ in $C_2$ is at least bounded from below by the minimal value for $x_{global}$ in $C_1$, i.e. $x_{global} \geq \text{GetMin}(C_1, x_{global})$. Therefore we conclude that the inequality
GetMin\((C_1, x_{global})\) ≤ GetMin\((C_2, x_{global})\) must be preserved in symbolic transitions.

Technicalities 5.2: Proposition 5.10 is sufficiently maintained for state inclusion and state merging

Given two states \(s_1 = (\ell_1, C_1)\) and \(s_2 = (\ell_2, C_2)\) with \(\ell_1 = \ell_2\). In state merging, \(s_1\) and \(s_2\) may be combined together in state \(s = (\ell_1, C_1 \cup C_2)\) if \(C_1 \cup C_2\) forms a convex polyhedron (note that state inclusion is a strictly weaker property). Now, assume that \(s_3 = (\ell_3, C_3)\) is a predecessor state of \(s_2\), with GetMin\((C_2, x_{global})\) = GetMin\((C_3, x_{global})\). We may now have a situation where GetMin\((C_1 \cup C_2, x_{global})\) = GetMin\((C_1, x_{global})\) < GetMin\((C_2, x_{global})\). Here, Proposition 5.10 does not hold, since GetMin\((C_1 \cup C_2, x_{global})\) < GetMin\((C_3, x_{global})\). However, we do preserve the property that there exists a path that reaches state \(s\) such that monotonicity is preserved for all states on this path. For Algorithm 16 (and for its upcoming improved version, Algorithm 17) this weaker property is sufficient to preserve correctness.

The same reasoning can be used for minimal- and maximal-parameter monotonicity (cf. Lemma 5.1).

For the maximal (upper-bound) time, we have the following result.

**Proposition 5.11:** Maximal-time is not monotonic in general

The monotonicity property from Proposition 5.10 is not preserved for maximal-time, i.e. we may have GetMax\((C_1, x_{global})\) > GetMax\((C_2, x_{global})\) and, given \(s_3 \in S\) with \(s_3 = (\ell_3, C_3)\) and \(s_3 \in \text{Succ}(s_2)\) such that \(x_{global}\) is not reset, we may also have GetMax\((C_2, x_{global})\) < GetMax\((C_3, x_{global})\).

**Proof.** We show that both situations are possible and follow from Definition 5.5. Consider GetMax\((C_1, x_{global})\) > GetMax\((C_2, x_{global})\). We could simply construct an invariant on the location of \(s_2\) that provides a stricter upper-bound on the time. For instance, let the transitions between \(s_1\) and \(s_2\) be unguarded and assume that the location invariant for \(s_1\) is \(x_{global} < 10\), and that the location invariant for \(s_2\) is \(x_{global} < 5\). This way, we can achieve that GetMax\((C_1, x_{global})\) > GetMax\((C_2, x_{global})\).

Consider now that GetMax\((C_2, x_{global})\) < GetMax\((C_3, x_{global})\). By continuing the concrete example, assume an unguarded transition from \(s_2\) to \(s_3\) and set the location invariant for \(s_3\) to \(x_{global} < 15\) (see e.g. Figure 5.6 and Example 5.5). This situation could be explained by the mechanism of time elapsing, i.e. all clocks are
5.5. Optimal parameter and time synthesis

incremented by some value \( d \in \mathbb{R}_{\geq 0} \) in the arc from \( s_2 \) to \( s_3 \). Note that the location invariant of \( s_2 \) on the upper-bound for \( x_{\text{global}} \) is not preserved any more. \( \square \)

Minimial-time synthesis and reachability. We modify Algorithm 16 by simply replacing all occurrences of the parameter \( p \) by the global clock \( x_{\text{global}} \), also when adding the constraint to \( K \), i.e. \( p = s_{\text{opt}} \) becomes \( x_{\text{global}} = s_{\text{opt}} \). The correctness for this algorithm does not hold trivially, as Lemma 5.1 is only applicable to parameters. However, from Proposition 5.10 we have that the minimal value for \( x_{\text{global}} \) can only increase for successor states, which causes that the same results as Lemma 5.1 can be implied and thus correctness is preserved. The corresponding reachability problem can be addressed in the same way as we propose for optimal-parameter reachability. While the solution is the same as achieved by extending the PTA to include a \( p_{\text{global}} \) parameter, this approach avoids explicitly adding an additional parameter to the PTA.

Maximal-time synthesis and reachability. Because of the negative result from Proposition 5.11, a straightforward modification, i.e. by combining the modifications to address the maximal-parameter and minimal-time problems, is not sufficient for maintaining correctness. Since the maximum time of a symbolic state may both increase and decrease throughout a run, we can no longer ignore successor states with a worse maximal time than the current optimum. Hence, at line 20 in the algorithm, we replace the entire line by \( W := W \cup \{s'\} \). That is, we no longer discard successors of non-target states in the algorithm. When a target state is reached we do not have to consider its successor states, since the problem is stated for the first time a target state is reached.

As the modified algorithm now explores the entire \( \mathcal{PZG} \), except for successors of target states, we now consider all symbolic states that may improve or extend the optimal result. Thus, we maintain that the modified algorithm is correct for solving the maximal-time synthesis problem. Its corresponding reachability problem, however, cannot be improved using the modifications that we proposed earlier. This is because of the result from Proposition 5.11, where the maximal time to reach a successor state may be decrease or increase compared to the maximal time for the current state, hence we cannot cut branches. Therefore, we rely on the synthesis result to perform maximal-time reachability.

Note that we do not encounter the same problems as we faced here when we solve the maximal-time synthesis problem via the maximal-parameter synthesis.

\( ^7 \) Notably, it is possible to allow for the ‘absolute’ maximal time to end in a target state, i.e. by allowing multiple visits to a target state, by still considering successor states of a target state and thus exploring the entire \( \mathcal{PZG} \).
approach. The reason is that the parameter, $p_{\text{global}}$, is conforming to the property from Lemma 5.1. In practice, however, this implies that there is basically no performance to be gained, as comparisons between $x_{\text{global}}$ and $p_{\text{global}}$ are only made on arcs toward target states.

**Example 5.5: Maximal-time synthesis**

Consider the PTA $A$ in Figure 5.6, and run $\text{MaxTimeSynth}(A, \{\ell_4\})$, i.e. the modified version of Algorithm 16 for maximal-time synthesis (since $x$ is never reset, we use it as $x_{\text{global}}$). We describe the process as follows.

1. The initial state is $s_1 = (\ell_1, 0 \leq x \leq 10 \land p \geq 0)$.
2. Since $s_1$ is not a target state, we compute the successors of $s_1$, which are $s_2 = (\ell_4, x = p \land 0 \leq p \leq 10)$ and $s_3 = (\ell_2, 0 \leq x \leq 5 \land p \geq 0)$.
3. In the second iteration, pick $s_2$ from $W$: it is a target, and therefore $\text{GetMax}(C_2, x)$ is computed, which gives $x = 10^\leq$. Since $10^\leq > 0^<$, we found a new maximum, and $K$ becomes $(C_2 \land x = 10)^\downarrow_P$, i.e. $p = 10^\leq$.
4. Now we pick $s_3$ from $W$, which is not a target, therefore we compute its successor $s_4 = (\ell_3, p \leq x \leq 15 \land 0 \leq p \leq 5)$.
5. Pick $s_4$, which is also not a target state and compute its successor $s_5 = (\ell_4, x = 15 \land 0 \leq p \leq 5)^\leq$.
6. Finally, we pick $s_5$ from $W$, which is a target state. Therefore, we compute $\text{GetMax}(C_5, x)$. This gives $x = 15^\leq$. Since $15^\leq > 10^\leq$, we found a new maximum, and $K$ becomes $(C_5 \land x = 15)^\downarrow_P$, i.e. $0^\leq \leq p \leq 5^\leq$.
7. As $W = \emptyset$, the algorithm terminates and $(\text{Opt}, K)$, i.e. $(15^\leq, 0^\leq \leq p \leq 5^\leq)$, is returned.

Note that for the path with the symbolic states $s_1, s_3,$ and $s_4$, the maximal value for $x$ changes from $10^\leq, 5^\leq,$ and $15^\leq$, respectively. Hence, this is an example for which monotonicity is not preserved for maximal-time (c.f. Proposition 5.11).
5.6 Improved optimal-parameter and minimal-time synthesis

We present a semi-algorithm Algorithm 17 that exploits the monotonicity of the optimal-parameter and minimal-time problems. We present the algorithm as a solution to the minimal-parameter problem and show how it can be extended to address the other problems. The idea is to order the to-be-explored states such that the most promising states for improving the optimal result are processed first, which allows us to terminate the algorithm earlier.

5.6.1 Algorithm

We give ImprovedMinParamSynth($A, T, p$) in Algorithm 17. The main difference with Algorithm 16 is that we maintain a priority queue $Q$ of waiting symbolic states (instead of the waiting set $W$) and order these by the minimal parameter value\(^8\). Thus, the priority queue consists of pairs of the form $(s_{opt}, s)$, such that $s_{opt}$ represents the optimal parameter value for $s = (\ell, C)$, i.e., $s_{opt} = \text{GetMin}(C, p)$. $Q$ is then sorted on the $s_{opt}$ values of each pair, such that the pair with the lowest $s_{opt}$ value is the first pair to be popped from $Q$.

For the initial state we have $p \geq 0$, so this becomes $0^\ell$ in the comparative value representation (line 2). We further maintain a set $P$ of passed states, a current optimum $Opt$ (initially $\infty$), and the associated optimal valuations $K$.

While $Q$ is not empty, the state $s$ with the lowest associated minimal value $s_{opt}$ is popped from the head of the queue (line 7). At line 9 we see the benefit of maintaining the to-be-explored states in a priority queue. If $s_{opt}$ is larger than $Opt$, then we know that this is also true for all remaining states in $Q$, as well as any reachable state from $Q$ and $s$ (Lemma 5.1). Thus, when $s_{opt} > Opt$, we can ignore all remaining states (including $s$) and return the current result.

Otherwise, if $s$ is a target state (line 10), we have $s_{opt} \leq Opt$. We assume that for any target state $s$ we have that $s_{opt} \notin Opt$ (we will guarantee this property when

---

\(^a\) Note that since we are concerned with maximal-time reachability, we do not allow clocks to advance in target states.
Algorithm 17: Improved semi-algorithm for solving MinParamSynth.

**input**: A PTA \( A \) with symbolic initial state \( s_0 = (\ell_0, C_0) \), a set of target locations \( T \), a parameter \( p \).

**output**: Constraint \( K \) over the parameters, which solves \( \text{MinParamSynth}(A, T, p) \).

1 function ImprovedMinParamSynth \((A, T, p)\)

// Initialization
2 \( Q := \{(0^-, s_0)\} \) // priority queue ordered by optimum value for \( p \)
3 \( P := \emptyset \) // passed set
4 \( \text{Opt} := \infty \) // current optimum
5 \( K := \bot \) // current optimum valuations

// Main loop
6 while \( Q \neq \emptyset \) do
7 \((s_{\text{opt}}, s = (\ell, C)) := Q.\text{Pop}()\) // take head of queue and remove it
8 \( P := P \cup \{s\} \)
9 if \( s_{\text{opt}} > \text{Opt} \) then return \( K \) // early termination
10 else if \( \ell \in T \) then // \( s \) is a target state and assume \( s_{\text{opt}} = \text{Opt} \)
11 \( K := K \lor (C \land p = s_{\text{opt}})_{\downarrow}\text{P} \) // valuations s.t. \( s_{\text{opt}} = \text{Opt} \)
12 else // otherwise explore successors
13 for each \( s' = (\ell', C') \in \text{Succ}(s) \) do
14 // only consider state if not seen before
15 if \( s' \notin Q \land s' \notin P \) then
16 \( s'_{\text{opt}} := \text{GetMin}(C', p) \) // compute optimum for \( C' \)
17 if \( s'_{\text{opt}} \leq \text{Opt} \) then // only add if not exceeding \( \text{Opt} \)
18 // check for a better optimum to reach the target
19 if \( \ell' \in T \land s'_{\text{opt}} < \text{Opt} \) then \( \text{Opt} := s'_{\text{opt}} \)
20 \( Q.\text{Insert}((s'_{\text{opt}}, s')) \) // add to the priority queue
21 return \( K \)

inserting states in the queue). Thus we conclude that \( s_{\text{opt}} = \text{Opt} \) holds. We add the parameter valuations to \( K \) in line 11, but we first intersect the constraint with \( p = s_{\text{opt}} \) to ensure that \( p \) equals the minimal result from \( C \).

If \( s \) is not a target state, then we compute and process its successors in line 13-18. We ignore states that have been visited before (line 14), and compute the minimal parameter value for \( s' \) in line 15. We then compare \( s'_{\text{opt}} \) with \( \text{Opt} \) in line 16. All successor states for which \( s'_{\text{opt}} \) exceeds \( \text{Opt} \) are ignored, as they cannot improve the result.
At line 17, if \( s' \) is a target state and \( s'_\text{opt} < \text{Opt} \), then we found a better optimum. We update \( \text{Opt} \) here (before even ‘exploring’ the state \( s' \)) since this allows us to maintain the property from which we rely on in line 10. Finally, the successor state is inserted in the priority queue in line 18.

Regarding the implementation of Algorithm 17, we would like to refer to Technicalities 5.1, as the same principles hold.

### Example 5.6: A run of Algorithm 17

Consider the PTA \( \mathcal{A} \) in Figure 5.5, and run the algorithm \( \text{ImprovedMinParamSynth}(\mathcal{A}, \{\ell_3\}, p_1) \). We describe the process as follows.

1. The initial state is \( s_1 = (\ell_1, x \geq 0) \) (and trivial constraints \( p_i \geq 0 \)).
2. \( s_1 \) is not a target state, thus its successors are computed. The first successor, \( s_2 = (\ell_3, x \geq 2 \land p_1 > 2) \) is a target state and we compute \( \text{GetMin}(C_2, p_1) = 2^\succ \). Since \( 2^\succ < \infty \), we update \( \text{Opt} \) to equal \( 2^\succ \). Then we insert \((2^\succ, s_2)\) in \( Q \).
3. The next successor from \( s_1 \) is \( s_3 = (\ell_2, x \geq 0 \land p_2 > 1) \). It is not a target state, and its minimal value for \( p_1 \) is \( 0^\succ \), so the state gets inserted in \( Q \) (before \( s_2 \)).
4. In the second iteration, we take \((0^\succ, s_3)\) from \( Q \): it is not a target and therefore we compute its successors. We find \( s_4 = (\ell_3, x \geq 2 \land p_1 = 2 \land 1 < p_2 < 2) \). This is a target state and the minimal value for \( p_1 \) is \( 2^\succ \). We update \( \text{Opt} \) because \( 2^\succ < 2^\succ \) and insert \((2^\succ, s_4)\) in \( Q \).
5. The other successor from \( s_3 \) is \( s_5 = (\ell_3, x \geq 2 \land p_1 = p_3 = 2 \land p_2 > 1) \), which is also a target with \( \text{GetMin}(C_5, p_1) = \text{Opt} \). We insert \((2^\succ, s_5)\) in \( Q \).
6. We now have \( Q = \langle (2^\succ, s_4), (2^\succ, s_5), (2^\succ, s_2) \rangle \) and take \( s_4 \) from \( Q \). It is a target, so we add its constraint to \( K \) in line 11, i.e. \( K = (C_4 \land p_1 = s_{\text{opt}})_{\downarrow P} \), which equals \( p_1 = 2^\succ \land 1 < p_2 < 2 \).
7. In the next iteration we take \((2^\succ, s_5)\) from \( Q \): it is a target, with \( 2^\succ = \text{Opt} \). We therefore add its constraint to \( K \), giving a new \( K \) equal to \((p_1 = 2^\succ \land 1 < p_2 < 2) \lor (p_1 = p_3 = 2^\succ \land p_2 > 1) \).
8. In the following iteration we take \((2^\succ, s_2)\) from \( Q \) and observe at line 9 that \( 2^\succ > \text{Opt} = 2^\succ \). The algorithm returns \( K \) and terminates.

While in this example, Algorithm 17 we actually do not improve in performance compared to Algorithm 16. However, the example does illustrate the algorithm’s inner workings.
5.6.2 Correctness

Algorithm 17 is a semi-algorithm; if it terminates with result \( K \), then \( K \) is a solution for the \textit{MinParamSynth} problem. We show that it is correct by relating it with the algorithm from Algorithm 16.

The main difference in Algorithm 17 is that the algorithm may terminate early (line 9). At this point there may still be ‘unexplored’ states in the priority queue \( Q \). However, when adding a state to the priority queue in line 18, we guarantee for the added state \( s' = (\ell', C') \) that

- If \( s' \) is a target state (i.e. \( \ell' \in T \)), then we assert \( \text{GetMin}(C', p) = \text{Opt} \) from line 17.
- If \( s' \) is not a target state, then it is only added to \( Q \) if \( \text{GetMin}(C', p) \leq \text{Opt} \).

Therefore, throughout the algorithm we have for any target state \( s' = (\ell', C') \) in \( Q \) that \( \text{GetMin}(C', p) \geq \text{Opt} \). We also have that there is at least one a target state \( s'' = (\ell'', C'') \) in \( Q \) with \( \text{GetMin}(C'', p) = \text{Opt} \). Thus, when we are processing a state \( s \) that has a worse minimal parameter value compared to \( \text{Opt} \), we can conclude that \( s \) cannot provide us with any interesting valuation. As a result, we can ignore all states for which \( s_{opt} > \text{Opt} \) and terminate the algorithm early.

5.6.3 Maximal-parameter, minimal-time and reachability problems

Maximal-parameter synthesis. We can use Algorithm 17 to solve the maximal-parameter synthesis problem. Similar to the modifications from Section 5.5.4, we update the locations in the algorithm that are marked yellow. The priority queue \( Q \) should be ordered such that the maximal value is at the head of the queue (instead of the minimal one), and it is initialized with \( Q := \{ (\infty, s_0) \} \). The optimum, \( \text{Opt} \) is initialized to \( 0^< \), \( \text{GetMin} \) is replaced by \( \text{GetMax} \), and the occurrences of \( >, \leq, < \) are respectively replaced by \( <, \geq, > \).

Optimal-parameter reachability. Since we never completely replace \( K \), the reachability version is simply realized by modifying line 11 to return \( (C \land p = s_{opt}) \downarrow_P \). The same modification holds for the maximal-parameter version.

Correctness. The correctness of both variant problems relies on the correctness of the \textit{ImprovedMinParamSynth} algorithm. Reachability remains preserved, and for the maximal-parameter problems the algorithm is updated accordingly to now return the maximal result.
Minimal-time synthesis and reachability. By relying on the result of Proposition 5.10 we can use Algorithm 17 to solve the minimal-time synthesis problem. Here we (again) assume that there is a global clock $x_{\text{global}}$ that is never reset in the PTA. We then modify the algorithm similar to the process described in Section 5.5.5. Minimal-time reachability can be achieved in the same way as described for optimal-parameter reachability. Because monotonicity is not preserved for maximal-time (Proposition 5.11), we cannot apply Algorithm 17 for the maximal-time reachability or synthesis problems.

5.7 Experiments

We implemented a number of our algorithms and variants in the IMITATOR tool [AFKS12] and compared their performance with the standard (non-minimization) EFSynth parameter synthesis algorithm from [JLR15]. For the experiments, we are interested in analysing the performance (in terms of computation time) of each algorithm, and comparing that with the performance of standard synthesis.

5.7.1 Experimental setup

Benchmark models. We collected PTA models and properties from the IMITATOR benchmarks library [And18a] which contains numerous benchmark models from scientific and industrial domains. We selected all models with reachability properties and extended these to include: (1) a new clock variable that represents the global time $x_{\text{global}}$, i.e. a clock that does not reset, and (2) a new parameter $p_{\text{global}}$ along with the linear term $x_{\text{global}} = p_{\text{global}}$ for every transition that targets a goal location, to ensure that when minimizing $p_{\text{global}}$ we effectively minimize $x_{\text{global}}$. In total we have 68 models, and for every experiment we used the extended model that includes both the global time clock $x_{\text{global}}$ and the corresponding parameter $p_{\text{global}}$.

Implementation. We use the existing implementation in IMITATOR of the EFSynth algorithm (Algorithm 15). And we obtained the implementation of Algorithm 16 from Étienne André. Note that this implementation only allows for minimal- and maximal-parameter synthesis, as well as optimal-parameter reachability. The minimal- and maximal-time synthesis versions of Algorithm 16 have not been implemented. The implementation of Algorithm 16 is such that the state space is explored in a breadth-first search manner. We implemented Algorithm 17
to solve the minimal-time synthesis and reachability problems based on a priority queue. We have not implemented any alternatives.

Subsumption. As mentioned in Section 5.3.3 we can use both state inclusion and state merging to reduce the number of states that we have to visit in the algorithm, while maintaining correctness (see Technicalities 5.2).

State inclusion is a relatively inexpensive computational task and preliminary results showed that it caused the algorithm to perform equally fast or faster than without the check. Checking for merging is however a computationally expensive procedure and thus should not be performed for every newly found state. For all BFS-based algorithms (the EFSynth algorithm from Algorithm 15 and the algorithm from Algorithm 16) we merge at every BFS layer.

For the improved algorithm (Algorithm 17), we empirically studied various merging heuristics and found that merging every ten iterations of the algorithm yielded the best results. We assume that both the inclusion and merging state space reductions are used in all experiments, unless otherwise mentioned.

Run configurations. For the experiments we used the following configurations:

- **MTReach**: Minimal-time reachability (with Algorithm 17)
- **MTSynth**: Minimal-time synthesis (with Algorithm 17),
- **MTSynth-noRed**: Minimal time synthesis, without reductions (with Algorithm 17),
- **MPReach**: Minimal-parameter reachability (of $p_{global}$, with Algorithm 16), and
- **MPSynth**: Minimal-parameter synthesis (of $p_{global}$, with Algorithm 16), and
- **EFSynth**: Classical reachability synthesis (with Algorithm 15).

Thus, the minimal-parameter configurations are implementations of the base algorithm from Algorithm 16, and the minimal-time configurations have been implemented with the improved algorithm from Algorithm 17.

Experimental setup. We performed all our experiments on an Intel® Core™ i7-4710MQ processor with 2.50GHz and 7.4GiB memory, using a single thread. The six run configurations were executed on each benchmark model, with a timeout of 3600 seconds. All models, results, and information on how to reproduce the results are available on https://github.com/utwente-fmt/OptTime-TACAS19.
Figure 5.7: Scatterplot comparisons of different algorithm configurations. The marks on the red dashed line did not finish computing within the allowed time (3600s).

The results of our experiments are displayed in Figure 5.7. We discuss each graph individually.

**MTSynth vs EFSynth.** We observe that for most of the models MTSynth clearly outperforms EFSynth. This is to be expected since all states that take more than the minimal time can be ignored. Note that the experiments that appear on a vertical line between $0.1s < x < 1s$ are a scaled-up variant of the same model, indicating that this scaling does not affect minimal-time synthesis. Finally, the
model plotted at (1346, 52) does not heavily modify the clocks. This means that there is little to be gained from using a priority queue to order the states on minimal time. As a consequence, MTSynth has to explore most of the state space while continuously having to extract the time constraints, making it inefficient.

**MPSynth vs EFSynth.** We can see that MPSynth performs more similar to EFSynth than MTSynth, which is to be expected since the constraint on \( p_{global} \) only takes effect when the goal location has been reached, so there is not much pruning. Still, MPSynth significantly outperforms EFSynth. This is also because fewer states have to be explored to guarantee optimality (once a parameter value exceeds the minimal value, all its successors can be ignored).

**MTSynth vs MPSynth.** Here, we find that MTSynth outperforms MPSynth, similar to the comparison with EFSynth. The results also show a second scalable model (for which EFSynth timed out) around (0.003, 10) and we see that MPSynth is able to solve the ‘bad performing model’ for MTSynth as quickly as EFSynth. Still, we can conclude that the minimal-time synthesis problem is in general more efficiently solved with the MTSynth algorithm than with MPSynth.

**MTSynth vs MTSynth-noRed.** Here we can see the advantage of using the inclusion and merging reductions to reduce the search space. For most models there is a non-existent to slight improvement, but for others it makes a large difference. While there is some computational overhead in performing these reductions, we conclude that this overhead is not significant enough to outweigh their benefits.

**MTReach vs MTSynth.** With MTReach we expect faster execution times as the algorithm terminates once a parameter valuation is found. The experiments show that this is indeed the case (mostly visible from the timeout line). However, we also observe that for quite a few models the difference is not as significant, implying that synthesis results can often be quickly obtained once a single minimal-time valuation is found.

**MPReach vs MPSynth.** Here we also expect MPReach to be faster than its synthesis variant. While it does quickly solve six instances for which MPSynth timed out, other than that there is no real performance gain. We also argue here that synthesis is obtained quickly when a minimal parameter bound is found. Of course we are effectively computing a minimal global time, so results may change when a different parameter is minimized (or maximized).
5.8 Conclusion

We have designed and implemented several algorithms to solve the optimal-time and optimal-parameter synthesis and reachability problems for PTAs. From our experiments we observed in general that minimal-time synthesis, computed with our improved algorithm, is in fact faster to compute compared to standard synthesis. We further show that minimal-parameter synthesis is also more efficient, and that existing search space reductions apply well to our algorithms.

Aside from the performance improvement, we deem optimal-parameter and optimal-time synthesis to be useful in practice. It allows for evaluating which parameter valuations guarantee that e.g. the goal is reached in minimal time. We consider it particularly valuable when reasoning about real-time systems.

On the theoretical side, we provided some results on L/U-PTAs. While finding the minimal valuation of a given lower-bound parameter is trivial (the answer is 0 iff the target location is reachable), finding the minimum of an upper-bound parameter boils down to reachability-synthesis for U-PTAs, a problem that remains open in general (it is only solvable for integer-valued parameters [BL09]), as well as to shrinking timed automata [SBM14], but with 0-coefficients in the shrinking vector—not allowed in [SBM14].

5.9 Future work

There are several directions for future work. We consider the following two topics to be especially interesting, considering their practical usefulness.

Parallelism. One direction is to improve performance by exploiting parallelism. Parallel random search or swarming-based solutions could significantly speed up the computation process, as demonstrated for timed automata [ZNL16b, ZNL16a]. However, for PTAs we may not be able to gain significant improvements by using a similar approach. The main performance bottleneck for PTA synthesis seems to be the successor computation process. Therefore we deem it more beneficial to investigate in parallel solutions for that process.

It is also possible, while not completely trivial, to modify Algorithms 16 and 17 to run in parallel. This can be achieved via a master-worker scheme, in which a master thread maintains the waiting set (or priority queue) of to-be-explored states. A worker can then request a state from this set, explore the state and compute its successors, and returning the successor states back to the master thread. This way, different states can be processed at the same time, without causing (much)
redundant work (there would be far more redundancy if we use the approach to launch swarmed instances of the search). Note that the optimal parameter or time value should be maintained carefully, by only concluding optimality once all threads are in agreement.

**Minimal upper-bound time.** Another interesting research direction is to minimize the *upper-bound* time to reach the target, e.g. for minimizing the worst-case response-time in real-time systems. We can define the minimal upper-bound time as follows. \( \text{MinUBTime}(A, T) = \inf \{ \text{MaxTimeTA}(v_P(A), T) \mid \forall v_P \in V_P \} \). We found that this problem is at least as difficult as the maximal-time problem. The issue with computing a solution to this problem is that the maximal-time is not monotonic (Proposition 5.11), so we basically cannot give any guarantees during the algorithm on when optimality is reached. We also suspect that state merging (and possibly also state inclusion) cannot be used, as it may be advantageous to revisit a location if this allows for restricting the maximal-time.
Conclusion

In this dissertation, we made several improvements in the area of formal methods, in particular for algorithmic approaches that relate to checking models. We studied the subject using a bottom-up approach, i.e. we mainly focused on the core techniques that are used in the field and exploited domain knowledge to improve their performance, in a sequential or parallel setting. All algorithms are based on graph exploration, sometimes using symbolic representations.

6.1 Contributions

Our contributions consist of context-driven performance improvements for Strongly Connected Component (SCC) decomposition and shortest path algorithms, and applications of both graph search techniques to solve new problems. We summarize our contributions, per chapter, as follows.

6.1.1 Multi-core on-the-fly SCC decomposition

In Chapter 2 we focused on improving the performance of on-the-fly SCC decomposition. We developed a multi-core algorithm that scales effectively on many-core systems. In its construction, we additionally developed an iterable concurrent union-find structure that may be used in other applications.

Iterable concurrent union-find structure. The union-find, or disjoint-set, data structure [GF64] is a structure that allows for tracking sets of elements.
Initially, it assumes all elements to be partitioned in distinct sets and two sets are combined by updating a single pointer. A concurrent version of union-find exists [AW91] that allow multiple threads to concurrently unite sets.

We presented a useful extension to the union-find data structure [BLvdP16]. In combination with tracking disjoint sets in a concurrent union-find structure, we may also efficiently iterate over the list of distinct elements in each set. Additionally, individual elements may be removed from the iterable list, while remaining present in the original set. This allows us to merge different sets of states together, while concurrently iterating over the elements in a set and e.g. marking elements as being fully processed. We showed that the structure is correct, efficient, and scalable in practice. That said, various extensions and variants can be considered to further enhance the structure.

**Efficient multi-core on-the-fly SCC algorithm.** There are several algorithms for decomposing a graph into SCCs, such as Tarjan’s algorithm [Tar72]. There are also a number of approaches that operate in parallel, e.g. [Orz04, HRO13, SRM14, JLH18]. However, these techniques all rely on full prior knowledge of the graph and cannot therefore not be applied in an on-the-fly setting, i.e. where we are given an initial state and must compute successor states to explore the graph. Existing techniques that can be applied on-the-fly [Low16, RDKP15] only scale in the presence of small SCCs. Therefore, in case the graph contains large SCCs, these techniques do not improve in performance when multiple cores are used.

We presented a new quasi-linear multi-core on-the-fly SCC algorithm [BLvdP16]. The algorithm communicates partial SCCs using our new iterable union-find structure. This way, multiple workers aid each other in decomposing an SCC, causing the algorithm to not fall victim to the limitations of related work. As a result, the algorithm manages to scale in the presence of both small and large SCCs. Our algorithm is therefore the first parallel DFS-based SCC decomposition algorithm that scales in the presence of large SCCs. Experiments demonstrated a significant speedup over the current best known approaches, on a variety of graph instances.

### 6.1.2 Model checking generalized automata

In Chapter 3 we considered SCC decomposition in the domain of model checking. We showed that our algorithm from Chapter 2 can be applied to outperform the state-of-the-art techniques. Additionally, we explored how more general automata could be model checked by providing techniques that achieve this. Based on empirical studies we discuss their benefits and drawbacks.
Efficient SCC-based multi-core model checking. In a multi-core setting, the best known approach for explicit-state Linear-time Temporal Logic (LTL) model checking is a multi-core version of nested depth-first search, called CNDFS \cite{ELPvdP12}. In addition, the technique has a lower memory overhead compared to SCC-based algorithms. However, a downside of CNDFS (and NDFS-based techniques in general) is that it may only be applied on State-based Büchi Automata (SBAs) and cannot be used to check more general automata.

We showed that the SCC decomposition algorithm from Chapter 2 is well-suited for multi-core on-the-fly LTL model checking and demonstrated that it performs comparable to the CNDFS algorithm \cite{BvdP16}. In fact, we showed that SBAs may be checked faster with our approach, compared to CNDFS, for large state spaces that do not contain counterexamples. Since these instances take the longest time to model check, one may consider our algorithm to be favourable. Additionally, our technique can also be applied on more generalized automata, i.e. Transition-based Generalized Büchi Automata (TGBAs), which can result in smaller automata and therefore may result in a smaller state space to model check. However, in our empirical study we found no performance improvement compared to our SBA checking variant, which may be different for more complex LTL formulas.

Checking complex LTL formulas with TGRAs. In order to perform explicit-state LTL model checking, the LTL formula is translated to an $\omega$-automaton. The standard approach is to translate the formula to an SBA, which accepts runs that visit an accepting state infinitely often. We may also translate an LTL formula to a TGBA, which accepts runs that visits a set of edges infinitely often. A TGBA has the advantage that it may contain fewer states than an SBA, which in turn may lead to a smaller cross-product to model check.

Yet another direction is to translate LTL to Transition-based Generalized Rabin Automata (TGRAs). These automata may be more concise (and more deterministic) than TGBAs. The acceptance condition of a TGRA is more involved and consists of a disjunction of Transition-based Generalized Rabin Pairs (TGRPs). Each TGRP contains a set of transitions that may only be visited finitely often (Fin transitions) and a set of transitions that must be visited infinitely often. However, the problem of checking TGRAs directly has only been studied to a limited extent \cite{Wij16, CGK13}, making it unclear whether the state space reduction from TGRAs outweighs the added complexity of checking these automata.

We studied whether it is more advantageous to directly model check (smaller-sized) TGRAs instead of using simpler automata. We presented an algorithm for directly checking TGRAs in parallel, by avoiding Fin transitions \cite{BDLvdP17}. We extended our union-find structure to realize this, by tracking an additional list of states that are reached via Fin transitions. During the state space exploration, these Fin
transitions are avoided and stored in the list. This way, detected cycles can never include Fin transitions. In practice, however, we found that this technique does not improve over alternative approaches. The results do provide insights in how the different features of TGRAs impact performance. For example, we encountered many instances for which the combined product consists of a large proportion of Fin transitions. We also found that there are often significant differences between the generated TGRAs from the three LTL translation tools that were used.

**Fin-less automata to more efficiently check TGRAs.** Despite the negative result of checking TGRAs directly, its pursuit has lead to an alternative approach. We introduced Fin-less automata, which can be constructed from TGRAs, and showed that the emptiness check for a Fin-less automaton can be performed efficiently [BDLvdP19]. The acceptance condition for a Fin-less automaton consists of a disjunction of edge-sets that must be visited infinitely often. In contrast to checking TGRAs, checking Fin-less automata is only slightly more involved when compared to TGBAs. Experiments showed that emptiness checks for Fin-less automata and TGBAs perform comparably. The acceptance condition for Fin-less automata extends TGBA acceptance, while not noticeably impacting performance. Therefore, we regard it an interesting type of automaton to consider in the future, as the size of Fin-less automata may be smaller than TGBAs.

### 6.1.3 Efficient alignment computation

In Chapter 4 we studied the shortest path problem in the context of computing alignments. In alignments we are concerned with comparing observed and modelled behaviour by using a cost function to penalize events that cannot be matched. The shortest path on the synchronized product then forms an optimal alignment. We consider the problem from two directions. By exploiting characteristic choices for the cost function, we compute alignments via an algorithm based on symbolic reachability. We also consider an alternative cost function and show how this leads to a new data structure and algorithm.

**Symbolically computing alignments for uniform cost functions.** We focus on the problem of computing an alignment for a large model and a single log trace. This reduces to a shortest path search on the state space, which is of large scale, of the synchronized product of the model and log trace. Existing techniques (e.g. [Adr14, vD18]) apply heuristics to direct the shortest path search, and therefore prevent exploring many irrelevant states. However, it remains a time-consuming task to align traces with large models.
6.1. Contributions

We have designed and implemented a new algorithm for computing alignments on uniform cost functions [BvdPvdA18]. In contrast to related work, we traverse the state space via symbolic reachability, i.e. instead of traversing single states, we store sets of states (via decision diagrams) and compute the successors for every state in a set at once. Symbolic reachability tends to perform well in situations where the state space is structured, which is often the case for models in the context of conformance checking (here, models are typically represented by a concise Petri net format). In order to compute shortest paths (and alignments), we split up the transitions of the model in a group of 0-cost transitions and a group of 1-cost transitions. The idea is then to compute all reachable markings by applying 0-cost transitions, followed by a single 1-cost transition, and iterate this process until the final marking is reached. From the experimental results we conclude that our symbolic algorithm is competitive with the state-of-the-art for computing alignments. The approach is especially favourable in computing alignments for models with large state spaces. In addition to these promising results, there is still plenty of room for further improvements and enhancements (e.g. to support more general cost functions).

Transitive closure graph for max-sync alignments with milestones. We may also consider alignment problems for which the model is of a smaller scale, but the model is combined with many log traces. While the task of aligning a single log trace onto the model may be performed quickly, it becomes a time-consuming process if the number of log traces accumulate.

We introduced a max-sync cost function that instead of minimizing discrepancies between a log trace and the model, maximizes the number of synchronous events [BvZvdA+18]. We showed that this cost function is complementary to the standard one. In addition, we consider milestone activities that may not be skipped in alignments, which can be used to guide the computation of alignments and refine the model. We constructed an algorithm that exploits the max-sync cost function, and constructs a Milestone Transitive Closure Graph (MTCG) from the model (not the synchronous product). This precomputed MTCG structure is then used to efficiently align many log traces. In practice, we show good performance for smaller-scale industrial models, but the technique cannot be applied for large models due to the size of the MTCG. Additionally, we show that the MTCG structure may be used effectively for representing diagnostic information on alignments.
6.1.4 Optimal-time and parameter synthesis for PTAs

In Chapter 5 we studied new problems for Parametric Timed Automata (PTAs), which extend timed automata with unknown constant values, or parameters. There has been prior work on parameter synthesis for PTAs [JLR13], which is an undecidable problem. Therefore, we rely on a semi-algorithm, i.e. an algorithm that may not terminate, but if it does, then the result is sound and complete. It may be useful to synthesize parameters to gain an understanding on the best- and worst-case behaviour, e.g. the maximal value of a parameter or the minimal total time to reach a location. However, these problems have not been studied in detail for PTAs.

Optimal-time and parameter synthesis for PTAs. Optimal-time (and parameter) synthesis remains undecidable for the full class of PTAs, but we proved decidability for a subclass of PTAs. Additionally, we have designed and implemented semi-algorithms to solve the optimal-time and optimal-parameter synthesis and reachability problems for the full class of PTAs [ABPvdP19]. Instead of performing full parameter synthesis, we track the minimal (or maximal) value of a parameter (or the total time) and guide the search accordingly to obtain all parameter valuations that lead to this result. Except for maximal-time synthesis, we can transform the problem to a shortest path search, in which which the state space exploration is guided by the local optimum value and pruned to avoid states that cannot improve the result. In our experiments, we observed in general that minimal-time synthesis, with our improved algorithm, is in fact faster to compute compared to standard synthesis. We further show that minimal-parameter synthesis is also more efficient, and that existing search space reductions apply well to our algorithms.

6.2 Reflection

We take a step back from our individual contributions and reflect on our research from a broader perspective.

First of all, we have not enforced a restriction to a single area of research, nor did we focus on a single overarching research problem. Instead, our work was based on the subject of graph traversal algorithms and (a subset of) applications in verification. In a general sense, by being introduced to an unfamiliar area of research, we may extend the perspective on known subjects, which may lead to new insights. Moreover, novel ideas may be formed in the new subject as a result of having a fresh perspective.
When comparing the different fields to each other, we made a few notably subjective, but nonetheless interesting observations.

- For high-performance algorithms and software, one cannot simply ignore parallelism any more as this may result in significant performance improvements. We should therefore consider utilizing the full potential of the nowadays readily available multi-core architectures.

- Efficient graph traversal is a central theme in both conformance checking and model checking. While there is quite some overlap in the underlying techniques that are used, related work from both fields suggest that these topics are mostly studied independently from each other.

- In model checking, the main limitation tends to be regarded as the state space explosion. An arguably much more neglected aspect is the modelling process and how one asserts that the model is an accurate representation of reality. In contrast, in process mining (and conformance checking) the model itself is often considered to be untrustworthy and not taken for granted.

- Efficient graph traversal generally tends to converge towards exploring as many states per second in model checking and conformance checking. For more complex systems (e.g., PTAs), however, we found that computing successor states is far more time consuming compared to the other fields, making it of importance to reduce redundant work as much as possible.

### 6.3 Future work

Given our contributions and gained insights in the fields that we studied, we consider the following directions and ideas to be interesting for future research (note that we provide additional concrete directions for future work in each of the main chapters).

- The iterable union-find data structure that we used for multi-core SCC decomposition tracks ‘worker bits’, i.e. a single bit for every worker thread that uses the structure. We showed that this technique works efficiently in practice by using atomic 64-bit operations. Notably, such a structure should be avoided for algorithms that use many thousands of worker threads in parallel. However, tracking worker bits and iterating over elements in the set may be useful for various other (practically oriented) parallel algorithms. One may for instance consider multi-core versions of e.g. Kruskal’s minimum-spanning tree algorithm [Kru56], (undirected) connected component algorithms [LCD+17], or SAT/SMT solving [LSB07].
• While we consider various ways to improve on our SCC decomposition algorithm in Section 2.10, there is likely little to be gained in practice. However, an improvement that prevents contention in the cyclic list may lead to significant improvements. We showed that related work (note that this is still an active area of research [HRO13, SRM14, JLH18]) is mostly focused on combining techniques together to achieve the best performance. A more promising direction to improve on our and existing work would be to think of radically different approaches. Alternatively, with a practical application in mind, one could consider exploiting specific structural properties of the target application (e.g. to what extent is the real-world phenomenon preserved?) to significantly improve performance. Alternatively, one may consider improving distributed or GPU algorithms for SCC decomposition [BCvdP11, WKB16].

• For model checking, while there are various improvements possible, we consider scalability on multi-core systems to be a solved problem. It becomes therefore far more interesting to combine our and existing scalable algorithms with state space reduction techniques. In Section 3.9 we consider partial order reduction to be a particularly interesting direction, considering the reductions achieved by related work [LW14, NWBvdP16].

• Instead of combining techniques to further improve model checking performance in general, it may be more interesting to analyze structural properties of the combined product of a model and an automaton [Pel04], and exploit these aspects to design tailor-made algorithms. Alternatively, one may be able to construct a selection procedure that, based on either the model or automaton separately, selects an appropriate algorithm to run.

• For the construction of alignments, we argue that quite some improvements are possible (see Section 4.10), e.g. saturation [CMS06]. In particular, we consider the symbolic algorithm to be a promising alternative compared to other techniques. Further research could investigate whether there are still large performance gains to be made.

• For PTAs, we presented some interesting directions in Section 5.9. A particularly interesting problem is to minimize the upper-bound time to reach a target location, which relates to the worst-case execution time for timed automata [ARF17]. A solution may be used to provide strict time boundaries on complex real-time systems. However, the problem may be considerably more difficult than the ones that we studied, since the upper-bound time is not strictly monotonically increasing for successive states.

• For each topic that we studied, we evaluated our approaches on specific benchmark suites (e.g. [Pel07, KGH+15, FFK+11, And18a]). We argue that
constructing and studying these benchmark instances is a particularly valuable topic of research. As we showed with our algorithms, it is highly beneficial to exploit the context of the problem domain for designing and improving algorithms.

Additionally, one could consider combining aspects from different fields. For instance, it may be useful in conformance checking to include uncertainties (e.g. in the form of parameters) to reason about missing aspects in the model or log trace.

As a final thought, we would like to encourage the reader to actively explore a variety of topics. Aside from being beneficial in general to be aware of a broader area of research, we argue that great ideas can be formed by combining various domains together.
Bibliography


[BBKS13] Tomáš Babiak, Frantisek Blahoudek, Mojmír Kretínský, and Jan Strejček. Effective translation of LTL to deterministic rabin automata: Beyond the (f, g)-fragment. In *Proceedings of the 11th International Symposium on Automated Technology for Verifica-


[ELL01] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In SPIN, volume


[HRO13] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (SCC) in small-


ACM Symposium on Principles of Distributed Computing, PODC 2016, pages 75–82. ACM, 2016. 31


[KMSZ18] Jan Kretínský, Tobias Meggendorfer, Salomon Sickert, and Christopher Ziegler. Rabinizer 4: From LTL to your favourite deterministic


shop on Parallel and Distributed Methods in verification, PDMC 2011, volume 72 of EPTCS, pages 13–28, 2011. 82


Samenvatting

Wij bestuderen gerichte grafen en met name twee klassieke problemen op grafen; de decompositie van een graaf in Sterk Samenhangende Componenten (SSCs) en het Kortste Pad probleem. In het bijzonder ontwikkelen wij nieuwe graaf algoritmes voor het controleren van modellen, oftewel technieken waarmee een gebruiker een systeem kan analyseren en zodanig kan verifiëren of bepaalde eigenschappen gelden. Onze bijdragen verbeteren de snelheid van bestaande technieken voor ‘model-checken’ en ‘conformiteit-checken’. Daarnaast hebben wij de horizonnen van deze onderzoeksgebieden verbreed door de problemen vanuit een ander perspectief te bekijken.

Wij hebben een algoritme ontwikkeld voor de decompositie van SSCs die efficiënt gebruik maakt van meerdere processors. Hiervoor hebben wij een itereerbare disjuncte-vereniging datastructuur bedacht die wellicht in andere toepassingen gebruikt kan worden.

We hebben vervolgens ons SSC decompositie algoritme toegepast voor model-checken en laten daarbij zien dat het sneller werkt dan bestaande technieken. Ook hebben wij onderzocht hoe generieke automaten direct gebruikt kunnen worden voor model-checken, door hier algoritmes voor te ontwikkelen.

We bestuderen het kortste pad probleem in de context van conformiteit-checken, met name voor het berekenen van een alignment. Door te profiteren van karakteristieke keuzes voor de kostenfunctie kunnen we een alignment berekenen met behulp van symbolische bereikbaarheid.

Tot slot hebben we ook nieuwe problemen bestudeerd voor geparameteriseerde getimede automaten, die getimede automaten uitbreiden met onbekende waarden, of parameters. Wij hebben algoritmes ontwikkeld voor het synthetiseren van de parameters voor het beste- of slechtste scenario. Wij kunnen bijvoorbeeld alle
mogelijke waarden voor de parameters berekenen waarvoor een eindlocatie bereikt wordt in minimale- of maximale totale tijd.
Titles in the IPA Dissertation Series since 2016


S.J.C. Joosten. Verification of Interconnects. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. Fixpoint Logic, Games, and Relations of Consequence. Faculty of Mathematics and Computer Science, TU/e. 2016-03


P.M. Heck. Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. Physical Security Analysis of Embedded Devices. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. Algorithms for Curved Schematization. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. Sylvan: Multi-core Decision Diagrams. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. Run-time resource management for component-based systems. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. Modeling the Dynamics of Requirements Process Improvement. Faculty of Mathematics and Computer Science, TU/e. 2016-12


J.N. van Rijn. Massively Collaborative Machine Learning. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. Efficient Immutable Collections. Faculty of Science, UvA. 2017-01

Model Checking. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02


A. Krasnova. Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT). Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. Data Structures for Analyzing Geometric Data. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities. Faculty of Science, UvA. 2017-07

W. Lueks. Security and Privacy via Cryptography – Having your cake and eating it too. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. Engineering the Dynamic Semantics of Domain Specific Languages. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. Geographic Graph Construction and Visualization. Faculty of Mathematics and Computer Science, TU/e. 2017-11


S. Darabi. Verification of Program Parallelization. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. Coequations and Eilenberg-type Correspondences. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broștean. Active Model Learning for the Analysis of Network Protocols. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. From Concurrent State Machines to Reliable Multi-threaded Java Code. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. Mixed Inductive-Coinductive Reasoning Types, Programs and Logic. Faculty of Science,
Mathematics and Computer Science, RU. 2018-06


N. Bezirgiannis. Abstract Behavioral Specification: unifying modeling and programming. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. Trajectory Analysis: Bridging Algorithms and Visualization. Faculty of Mathematics and Computer Science, TU/e. 2018-09


L. Swartjes. Model-based design of baggage handling systems. Faculty of Mechanical Engineering, TU/e. 2018-12


R. Kumar. Truth or Dare: Quantitative security analysis using attack trees. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15


M. Mehr. Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems. Faculty of Mathematics and Computer Science, TU/e. 2018-17


P.A. Inostroza Valdera. Structuring Languages as Object-Oriented Libraries. Faculty of Science, UvA. 2018-19


A. Serrano Mena. Type Error Customization for Embedded Domain-
Specific Languages. Faculty of Science, UU. 2018-21

S.M.J. de Putter. Verification of Concurrent Systems in a Model-Driven Engineering Workflow. Faculty of Mathematics and Computer Science, TU/e. 2019-01


Ö. Babur. Model Analytics and Management. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. Practical General Top-down Parsers. Faculty of Science, UvA. 2019-04


J. Moerman. Nominal Techniques and Black Box Testing for Automata Learning. Faculty of Science, Mathematics and Computer Science, RU. 2019-06