UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1602

Effective Techniques for Stateless Model Checking

STAVROS ARONIS

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018

ISSN 1651-6214
ISBN 978-91-513-0160-0
urn:nbn:se:uu:diva-333541

Dissertation presented at Uppsala University to be publicly examined in ITC/2446, Lägerhyddsvägen 2, 752 37, Uppsala, Friday, 2 February 2018 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Patrice Godefroid (Microsoft Research).

Abstract

Aronis, S. 2018. Effective Techniques for Stateless Model Checking. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1602. 56 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0160-0.

Stateless model checking is a technique for testing and verifying concurrent programs, based on exploring the different ways in which operations executed by the processes of a concurrent program can be scheduled. The goal of the technique is to expose all behaviours that can be a result of scheduling non-determinism. As the number of possible schedulings is huge, however, techniques that reduce the number of schedulings that must be explored to achieve verification have been developed. Dynamic partial order reduction (DPOR) is a prominent such technique.

This dissertation presents a number of improvements to dynamic partial order reduction that significantly increase the effectiveness of stateless model checking. Central among these improvements are the Source and Optimal DPOR algorithms (and the theoretical framework behind them) and a technique that allows the observability of the interference of operations to be used in dynamic partial order reduction. Each of these techniques can exponentially decrease the number of schedulings that need to be explored to verify a concurrent program. The dissertation also presents a simple bounding technique that is compatible with DPOR algorithms and effective for finding bugs in concurrent programs, if the number of schedulings is too big to make full verification possible in a reasonable amount of time, even when the improved algorithms are used.

All improvements have been implemented in Concuerror, a tool for applying stateless model checking to Erlang programs. In order to increase the effectiveness of the tool, the interference of the high-level operations of the Erlang/OTP implementation is examined, classified and precisely characterized. Aspects of the implementation of the tool are also described. Finally, a use case is presented, showing how Concuerror was used to find bugs and verify key correctness properties in repair techniques for the CORFU chain replication protocol.

Keywords: Concurrent, Parallel, Model Checking, Partial Order Reduction, Dynamic Partial Order Reduction, DPOR, Sleep Set Blocking, Source Sets, Source DPOR, Wakeup Trees, Optimal DPOR, Observers, Verification, Bounding, Exploration Tree Bounding, Testing, Erlang, Concuerror, Protocol, Chain Replication, CORFU

Stavros Aronis, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0160-0
urn:nbn:se:uu:diva-333541 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-333541)

Αφιερώνεται στους γονείς μου, Γιώργο και Λία.

—

Dedicated to my parents, Giorgos and Lia.

Cover art: Three execution steps, from two schedulings. The first step of both schedulings is the same. The second scheduling has a different second step.

Inspired by Concuerror's logo which is in turn inspired by the tool's `--graph` output (see e.g. Fig. 6.2 on page 45).

`http://parapluu.github.io/Concuerror`

List of papers

This dissertation is based on the following papers, which are referred to in the text by their Roman numerals.

I Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction [4]
Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas

Published in the Journal of the ACM, Volume 64, Issue 4, September 2017.

Revised and extended version of "Optimal Dynamic Partial Order Reduction" [2] by the same authors, published in POPL'14.

II The Shared-Memory Interferences of Erlang/OTP Built-Ins [9]
Stavros Aronis and Konstantinos Sagonas

Published in the proceedings of the 16th *ACM SIGPLAN International Workshop on Erlang, September 2017.*

III Testing and Verifying Chain Repair Methods for CORFU Using Stateless Model Checking [7]
Stavros Aronis, Scott Lystig Fritchie, and Konstantinos Sagonas

Published in the proceedings of the 13th *International Conference on Integrated Formal Methods, September 2017.*

IV Optimal Dynamic Partial Order Reduction with Observers [8]
Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas

Submitted for publication.

Reprints were made with permission from the publishers.

Sammanfattning på Svenska

Bakgrund

Idag när flerkärniga processorer finns nästan överallt är programmering av parallella program ett väldigt intressant forskningsområde. Att utveckla korrekta parallella program är ett svårt åtagande som kräver en djup förståelse av alla sätt som operationer som exekveras av olika processer kan störa varandra. I en maskin som har delat minne mellan processorkärnor kan sådana störningar inträffa när flera processer försöker använda samma del av minnet samtidigt (ett så kallat kapplöpningsproblem). Störningarna kan också inträffa på en högre nivå när flera processer försöker komma åt samma resurs (t.ex. ett lås) men också på en så hög abstraktionsnivå som mellan nätverksanrop från flera olika datorer till en dator i ett distribuerat system.

Under exekveringen av ett parallellt program kan operationer interagera på oförutsedda sätt vilket kan leda till så kallade samtidighetsrelaterade fel (concurrency errors). Det är svårt att utreda orsaken till sådana fel eftersom de beror på en speciell schemaläggning av operationer som inte uppkommer under alla exekveringar av programmet. Det kan till och med vara så att dessa buggar försvinner när man lägger in debug-utskrifter eftersom det kan ändra schemaläggningen (vilket är orsaken till att de ibland kallas *heisenbugs*). Efter att ha försökt åtgärda sådana buggar, verifieras detta ofta genom att bara exekvera programmet flera gånger för att försöka hitta liknande problem. Denna teknik kallas stresstestning och är ofta tillräckligt bra men kan inte ge några garantier om att programmet är korrekt eftersom det alltid kan finnas schemaläggningar som ännu inte har exekverats.

Tillståndsfri modellbaserad testning (stateless model checking, SMC) är en teknik för att verifiera att ett parallellt program är korrekt genom att ta kontroll över schemaläggningen och på ett systematiskt sätt testa alla olika sätt ett program kan schemaläggas. Med denna teknik kan man bevisa att ett program är korrekt oberoende av schemaläggningen. Denna teknik kallas också systematisk parallellitetstestning (systematic concurrency testing). Metoden har en tydlig fördel gentemot stresstestning eftersom den testar alla schemaläggningar och dessutom kan förklara fel som hittas genom att rapportera den exakta schemaläggningen som orsakade felet. Det naiva sättet att prova alla schemaläggningar kan leda till en kombinatorisk explosion. Om varje process kan exekvera vid varje exekveringssteg så ökar exekveringstiden exponentiellt med längden på programmet.

Partialordningsreduktion (partial order reduction, POR) förbättrar detta skalningsproblem genom att minska antalet schemaläggningar som måste testas, samtidigt som alla möjliga beteenden som programmet kan ha fortfarande testas. POR-tekniker drar nytta av att i typiska parallella program så kan de flesta par av operationer från olika processer inte störa varandra. Det är därför tillräckligt att detektera operationer som kan störa varandra, och fokusera utforskandet på schemaläggningar av dessa operationer. Att basera sådan detektering på data som samlas in vid körning av programmet är hörnstenen till dynamisk partialordningsreduktion (dynamic partial order reduction, DPOR).

Den har avhandlingens bidrag

I den här avhandlingen presenteras flera förbättringar av DPOR som väsentligt ökar SMCs effektivitet. Den mest centrala av dessa förbättringar är algoritmerna som kallas Source-DPOR och Optimal-DPOR (och det teoretiska ramverket bakom dessa algoritmer) (Artikel I) och en teknik som gör det möjligt att använda observerbarhet av störningar mellan operationer i DPOR (Artikel IV). Båda dessa förbättringar kan ge en exponentiell minskning av antalet schemaläggningar som måste utforskas för att verifiera ett parallellt program. Den här avhandlingen presenterar också en enkel begränsningsteknik som är effektiv för testning av parallella program när antalet schemaläggningar fortfarande är för stort för att göra verifiering möjlig inom en rimlig tidsrymd, även när de nya algoritmerna används (detta diskuteras i Artikel III).

Alla förbättringar har implementerats i Concuerror som är ett verktyg för att använda SMC på program skrivna i programspråket Erlang. Erlang är relevant för industrin och använder aktörsmodellen (the actor model) för att hantera parallella program. Processer i Erlang program stör inte varandra genom att använda delat minne direkt (som är fallet med program skrivna i lågnivåspråk) utan använder istället högnivåoperationer som är inbyggda i Erlang-implementationen Erlang/OTP. Störningarna mellan sådana operationer granskas, klassificeras och karaktäriseras på ett precist sätt (Artikel II) för att öka Concuerrors effektivitet.

Erlangs modell för parallellism är centrerad runt asynkron medlandeöverföring som stöder så kallade timeouts. Språket är därför speciellt passande för design, implementation och testning av distribuerade protokoll. Avhandlingen inkluderar ett exempel som visar hur Concuerror har använts för att verifiera egenskaper av en lagningsteknik för CORFU (ett så kallat “chain replication protocol”) (Artikel III).

Finansiering

Det här arbetet har genomförts inom Linnaeus kompetenscentrum UPMARC (Uppsala Programming for Multicore Architectures Research Center) och har delvis finansierats av EU FP7 STREP projektet RELEASE (287510) och Vetenskapsrådet.

Acknowledgments

My life in Uppsala was made happier and my work in Uppsala University more productive through my interactions with a number of people who I would like to thank in this note.

At the top of this list is Kostis Sagonas, whose student I have been for ten years at the time this dissertation is published. Kostis was the first professor to teach me functional programming and the principles of programming languages, supervised my diploma thesis when I was studying in the National Technical University of Athens and was the main advisor of my PhD studies, when I followed him to Uppsala University. I cannot thank him enough for all the time he has spent working with me throughout these years.

Second is Bengt Jonsson, my second advisor in Uppsala University. The algorithmic techniques discussed in this dissertation would not have been fully developed and proven correct without his help.

Next are all of my co-authors. Parosh Aziz Abdulla made me appreciate rigorous math. Scott Lystig Fritchie enthusiastically used the tool I developed and collaborated with me to improve it. Magnus Lång helped tremendously with the theory behind the final improvement to dynamic partial order reduction techniques presented in this dissertation and I wish him all the best in the pursuit of his PhD degree.

I also had the pleasure to work with a number of other people in publications not included in this dissertation. Jonatan Cederberg was there at the beginning of the journey that lead to Optimal DPOR algorithm. Carl Leonardsson and Mohamed Faouzi Atig successfully applied the Source DPOR algorithm in their own research. I want to thank them for all their comments and feedback.

When it comes to Concuerror, the tool I expanded during my studies, special thanks need to be given to the original developers, Alkis Gotovos and Maria Christakis. As is the fate of such projects, I rewrote almost all of their code after they stopped working on the tool, but they were the first to show the value of a stateless model checking tool for Erlang. Ilias Tsitsimpis and Daniel McCain were also students who made contributions to the tool, under the supervision of Kostis and me, and have my thanks.

Regarding my working environment in Uppsala University, I had the pleasure to share an office space with Kjell Winblad, who was my first Swedish friend (and whom I also thank for his help in writing a summary of this dissertation in Swedish), Andreas Löscher, with whom I shared most of my Erlang frustrations, and David Klaftenegger, with whom I shared most of my Linux frustrations. Elias Castegren, Stephan Brandauer, and Magnus Norgren were also my office mates and I thank them for all the motivating discussions. I also enjoyed fika and long discussions with Kiko Fernandez and Albert Mingkun Yang. I wish them all success in the completion of their studies.

Most of the aforementioned people have been students or members of the Linnaeus center of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center), which generously funded my studies. UPMARC has been an extremely valuable environment for research in which I was encouraged to interact with numerous other students and senior scientists, who I all thank for their comments on my work. Participating in the UPMARC board as a PhD student representative gave me invaluable experience on how research is planned, funded and conducted successfully.

I also received funding from the EU FP7 STREP project RELEASE and am thankful for all the feedback I received from its members. Finally, I want to especially thank the people in the Erlang community, for their support, comments and feedback on all of my presentations on Concuerror.

Life in Uppsala would have been much duller without all of my friends, who deserve their own place in my list of thanks. Pavlina, Vasili, Zacharoula, Kosta K., Eirini, Alberto, Alexandra, Thomas, Esther, Alex and Kosta V. you will always be in my heart. Angelina, meeting you changed my life.

The people of Wijkmanska Blecket, Uplands Nation's student orchestra, offered me a new place to unwind and possibly pushed my graduation a tiny bit later than planned. Ilka, thank you for inviting me in. Rebecca, thank you for being my friend. Johan, Lena, Lina, Luce, and Rebeca, thank you for warming my home. Anders, Arve, Axel, Daniel, Henning, Henrik, Isaia, Justine, Malin, Oliver, Patrik, Zoë and everyone else, thank you for your company, which I have enjoyed tremendously. I must also thank my accordion teacher, Kostas Mouskoidis[1], for teaching me the skills needed to join such a fantastic group of musicians.

This list would not be complete without mentioning my life-long friends from Greece; without their support I would not have survived so far away from the sun! Nikola, Christina, Oresti and Violetta, thank you for always being there for me. Dafni and Eirini, thank you for all the time we spent together. Giorgo, Niko, Alexi, Vasili and Andrea, thank you for being my awesome mates.

I also want to thank my parents, Giorgos and Lia, for their love and support. This dissertation is dedicated to them.

Finally, Selma, your love has been one more reason for me to call Uppsala (and Sweden) my second home. Thank you for your support and motivation which helped me finish this dissertation.

[1]It seems that all my best teachers are named with some variant of "Konstantinos", indeed.

Contents

1. Overview

This dissertation describes contributions to the field of testing and verification of concurrent programs. It consists of a collection of published and submitted work (Papers I to IV), prefaced by this comprehensive summary which explains the necessary background and highlights the main results presented in the papers.

Introduction

Concurrent programming is a field of significant interest in the current, multicore age of computing. Developing correct concurrent programs is however a difficult task, requiring a deep understanding of all the ways in which operations executed by different processes interfere. Such interference can be encountered at different levels, ranging from so-called data races when processes access the shared memory of a multicore chip, races between operations requesting other shared resources (e.g., locks), and going all the way up to interference at higher-levels, e.g., between requests arriving over the network at a node of a distributed system.

During the execution of a concurrent program, interfering operations can be interleaved in unexpected ways, leading to so-called concurrency errors. Investigating such errors is hard as, due to their dependency on the scheduling of operations, they are not triggered in every execution of the program. Even worse, attempts to trace their causes can change the program's behaviour enough to make them disappear; for that reason, concurrency errors are also called *heisenbugs*. Even when such an error is identified and fixed, the absence of other similar errors is often established just by executing the presumably correct program multiple times. This approach, known as stress testing, is often good enough, but cannot *guarantee* the correctness of the program, as there may always exist schedulings leading to more errors, which have not (yet) been exercised.

Stateless model checking (SMC) [22] is a technique that can be used to *verify* a concurrent program, by taking control of the scheduling and systematically exploring all the ways in which the program's operations can be executed, thus proving that in all possible schedulings the behaviour of the program is correct. Due to this mode of operation, the technique is also known as *systematic concurrency testing*. This approach has clear advantages over stress testing, as it is exhaustive and, on top of that, any detected concurrency errors can be explained by reporting the exact scheduling that triggered them. A naïve attempt to explore all possible schedulings, however, can lead to a combinatorial explosion: if every process is considered at every execution step, the number of possible schedulings scales exponentially with respect to the total length of the program's execution [21].

Partial order reduction (POR) techniques [15, 21, 37, 42] ameliorate this problem by requiring the exploration of *only a subset* of schedulings, while provably covering all behaviours that can occur in *any* scheduling. POR techniques take advantage of the fact that, in typical concurrent programs, most pairs of operations by different processes are not interfering. As a result, a scheduling E' that can be obtained from another scheduling E by swapping the order of execution of adjacent but non-interfering (independent) execution steps will make the program behave in exactly the same way as E. Such schedulings have the same partial order of interfering operations and belong to the same equivalence class, called a *Mazurkiewicz trace* [34]. It is then sufficient for stateless model checking algorithms to explore at least one scheduling in each such equivalence class. To achieve this, algorithms using POR techniques inspect pairs of interfering operations. If it is possible to execute such operations in the reverse order, then their partial order will be different and the algorithm should also explore a scheduling from the relevant equivalence class. It is therefore enough to determine which are the interfering operations and explore additional schedulings focusing only on those. Basing such detection on data obtained at runtime is the cornerstone of *dynamic partial order reduction (DPOR)* [18].

This dissertation describes a number of improvements to the original DPOR algorithm [18] that can exponentially reduce the number of explored schedulings, increasing its effectiveness (Papers I and IV). These improvements are described in a generic way, making them applicable to several concurrency models. When it comes to Erlang programs, the use of improved DPOR techniques together with a fine-grained characterization of the interferences between the higher-level operations of the language (Paper II) have resulted in a practical verification tool, which has been shown to be effective in testing and verifying programs and protocols (Paper III). Based on these observations, this dissertation supports the following:

Thesis:

Improvements in dynamic partial order reduction techniques can significantly increase the effectiveness of stateless model checking algorithms.

Source and Optimal DPOR (Paper I)

The work that lead to this dissertation began in an attempt to increase the effectiveness of Concuerror, a stateless model checking tool for Erlang programs. Erlang is an industrially relevant programming language based on the actor model of concurrency [6]. Prior to this work, Concuerror was a prototype used for researching systematic concurrency testing and test driven development of Erlang programs. Its main achievements had been in its ability to successfully instrument and schedule Erlang programs without modifying the language's VM-based runtime environment [12] and in enabling new ways of testing concurrent programs during their development [28].

Concuerror did not originally use any POR technique and was suffering from the combinatorial explosion in the number of explored schedulings, when used for verification. It was therefore a good candidate for trying the original DPOR algorithm [18], which in this dissertation will also be referred as "classic DPOR". While implementing that algorithm in Concuerror, however, we noticed that in a significant number of cases classic DPOR performed some redundant exploration. In particular, the algorithm could initiate exploration of a scheduling, but determine at a later point that any further exploration would make the scheduling equivalent with already explored schedulings. At that point, the algorithm would abort the exploration.

Research into how this problem could be avoided identified the use of *persistent sets* [21] by the classic DPOR algorithm as one of the reasons for redundant exploration and resulted in Paper I[1] presenting a new category of sets, *source sets*, as a new theoretical foundation for POR techniques that can replace persistent sets. The paper shows that the classic DPOR algorithm can be easily modified to use source sets instead of persistent sets, leading to the *Source DPOR* algorithm, which outperforms classic DPOR. As Source DPOR could also not completely avoid redundant exploration, the same paper introduced *Optimal DPOR*, a novel algorithm that uses source sets and *wakeup trees*, a new technique complementing the use of *sleep sets* [21], to never initiate redundant exploration, therefore achieving optimal reduction.

Both Source and Optimal DPOR algorithms were experimentally tested with Concuerror on Erlang programs, but are also applicable in other models of concurrency. As an example, Nidhugg [1] is a verification tool that applied Source and later Optimal DPOR on C++/Pthread programs. Source sets and Source DPOR have since been used as a basis in a number of publications and tools [5,31], including a more in-depth comparison with persistent sets [3]. Paper I also includes proofs of the correctness and optimality of both algorithms and a comparison of the tradeoffs in the use of the Source and Optimal DPOR algorithms.

[1] These results were also presented in an earlier version of the paper, published in POPL'14 [2].

Specifying the Interferences of Erlang's High-level Built-in Operations (Paper II)

While research in the theory of DPOR was ongoing, Concuerror continued to be developed as a practical tool for testing and verifying Erlang programs. Unlike in lower level languages, where processes interfere by accessing shared memory directly or by using synchronization operations, processes in Erlang interact using higher-level operations that are built-in in the language's implementation, the Erlang/OTP system. Examples include operations for sending and receiving messages, monitoring other processes (and receiving notifications if they crash), or accessing data shared via internal databases (e.g., the Erlang Term Storage system).

POR techniques crucially depend on determining which operations interfere and, as a result, increasing the accuracy of such decisions can significantly improve their effectiveness [25]. In order to be sound, Concuerror had to start from the assumption that any two Erlang built-in operations can interfere and then carefully exclude pairs of operations that cannot. As this information was getting more and more refined, it became clear that a deeper investigation of the interference of Erlang built-in operations was warranted.

This was the motivation for Paper II, which presents the first categorization and fine-grained characterization of the interferences between the built-in operations of the Erlang/OTP implementation. These interferences can lead to observable differences in program behaviour and must therefore be considered by a testing and verification tool. The paper includes a description and treatment of implicit or asynchronous events that can interfere with such operations, such as process termination and message delivery. It is also supported by a repository of small litmus test programs that have different results based on the scheduling of their processes, each highlighting a particular interference between Erlang's built-in operations (and/or asynchronous events). Tools for Erlang (like Concuerror) can soundly focus on just the cases presented in the paper and refine their interference detection techniques appropriately. Using such precise information, Concuerror can significantly reduce the number of schedulings it needs to explore.

Applying Concuerror to Protocol Verification (Paper III)

Erlang's concurrency model revolves around asynchronous message passing, including support for timeouts. The language is therefore particularly suitable for the design, implementation and testing of distributed protocols. Wanting to test error recovery methods for CORFU [33] (a variant of the Chain Replication protocol used in distributed shared log systems [43]), an engineer at VMWare wrote an Erlang model for a CORFU system and tried Concuerror on it. Using Optimal DPOR and a simple bounding technique, the tool was able to quickly detect errors in two buggy methods but could neither find bugs nor explore all schedulings of a third (possibly correct) repair method in a reasonable amount of time.

Collaboration with this engineer lead to Paper III, which starts from the presentation of the initial model and describes a number of refinements that we applied on both the model and Concuerror's interference detection mechanism. Using the resulting refined model and optimized version of Concuerror, we achieved exhaustive testing of the third method, verifying its correctness. This case study also gave empirical proof for the usability of a simple bounding technique suitable for finding bugs (*exploration tree bounding*) and provided insight into the use of Erlang as a modeling language.

Optimal DPOR with Observers (Paper IV)

The last paper included in this dissertation contains the formal description of the improvement applied on the Optimal DPOR algorithm to achieve the verification result presented in Paper III.

In concurrent programs, it can be the case that particular operations are interfering only when executed in particular contexts. As mentioned earlier, refining the conditions under which POR algorithms consider operations as interfering has been shown to have significant impact, regardless e.g. of whether the states in which operations are executed are also taken into account or not [25]. However, in order to guarantee their soundness, POR techniques often have to be conservative, treating operations as interfering even in cases where they are not.

In Paper IV, we describe how a DPOR algorithm can decide whether operations are interfering or not using *later* operations, which we call *observers*. As an example, an algorithm can treat pairs of write operations to the same memory location or message delivery events to the same process as independent, unless there exist later read or message receiving operations, respectively. The idea that interference can be conditional had been applied before, but limited only to considering the state in which operations were executed [29]. In the paper, we describe the challenges of using observers in DPOR algorithms, give a formal description of an extension of the Optimal DPOR algorithm with observers and report on two implementations (in Concuerror and Nidhugg), demonstrating that Optimal DPOR with Observers can achieve exponentially better reduction in both shared memory and message passing programs.

Personal Contributions

As all papers included in this dissertation have been co-authored, this is an explicit note of the author's contributions to each paper.

Paper I: I contributed to the design of the Source and Optimal DPOR algorithms equally with my co-authors. I am the sole implementer of Source and Optimal DPOR in Concuerror. I performed the evaluation, highlighting the tradeoffs in the use of each algorithm.

Paper II: I am the main author of the paper. I investigated the Erlang/OTP implementation, designed the classification and wrote all the litmus programs in the test suite.

Paper III: I am the main author of the paper. I refined the models, extended Concuerror with the bounding and optimization techniques discussed in the paper, and performed the evaluation.

Paper IV: I am the main author of the paper. I designed the algorithm together with Magnus Lång, who did most of the proofs and implemented the algorithm in Nidhugg. I am the sole implementer of the algorithm in Concuerror.

Organization of this Comprehensive Summary

The contributions are organized thematically in this comprehensive summary, beginning with an introduction to concurrent programs, stateless model checking, and partial order reduction (Chapter 2).

A description of the Source and Optimal DPOR algorithms (including their background) is given next (Chapter 3), followed by a description of the extension of Optimal DPOR with observers (Chapter 4). The exploration tree bounding technique is discussed separately (Chapter 5).

The summary continues with a presentation of Erlang (including its main implementation) and Concuerror (Chapter 6), followed by a chapter describing applications of the research (Chapter 7). Last, some concluding remarks (Chapter 8) and suggested directions for future research (Chapter 9) are given.

Related Work

A separate "Related Work" chapter has not been included in this comprehensive summary, as each of the included papers discusses relevant publications.

Related work on stateless model checking and partial order reduction techniques is given in Section 1 (Introduction) and 12 (Related Work) of Paper I, and Section 8 of Paper IV. Other specification attempts and testing tools for Erlang are presented in Section 7 of Paper II. Finally, a brief discussion regarding other attempts to verify aspects of the Chain Replication protocol is given in Section 6 of Paper III.

2. Background

This chapter gives an introduction to concurrent programs, stateless model checking, and partial order reduction techniques, including dynamic partial order reduction.

2.1 Concurrent Programs

A concurrent program consists of a number of *processes*, each executing a sequential program. Each process may operate on data that is *shared* between several processes (e.g., shared memory, messages or other resources) or *private* (i.e., no other process can access/modify them). An operation executed by a process is characterized as *local* if it only affects private data, and *global* otherwise. Global operations that involve the same data can be interfering.

When executing a concurrent program, a number of *schedulers* determine when and for how long each process will execute its sequential program. In practice, schedulers correspond to software mechanisms provided by the operating system or a programming language's runtime environment. The schedulers may enforce a different order and/or duration of execution of each process each time the program is executed. This *scheduling non-determinism* can lead to different orderings between interfering global operations executed by different processes, which may in turn make those processes follow different execution paths in their programs. This can lead to *concurrency errors*, i.e., errors that appear only under particular scheduling decisions and not in every possible execution of a concurrent program.

We assume that all concurrency-related non-determinism in the programs we examine is described by the effects of scheduling. Effects from so-called relaxed accesses to shared memory are out of scope, i.e., we assume that memory accesses follow the *sequential consistency* model.

2.2 Stateless Model Checking

Model checking [14,39] is a well studied verification technique, in which an arbitrary system is described by a number of states and transitions between those states. By exploring the resulting state space, one can then check whether each reachable state satisfies some given properties; this is called a *reachability problem*. Such exploration is typically *stateful*, requiring maintenance of a representation of visited states in order to explore transitions from those states.

We can see the verification of a concurrent program as a reachability problem in model checking, by using the code and data (shared and private) of the program's processes as states and the execution of operations by the processes as transitions. If no errors (e.g., assertion violations) are reachable, then the program is correct. It is however easy to imagine that, for programs of any significant size, the number of states in the resulting system can be huge. Moreover, storing each of these individual states would require impractical amounts of memory. The goal of *stateless model checking (SMC)* [22] is to explore the described state space without explicitly storing information about each state.

2.2.1 Schedulings

The first step to enable stateless exploration of the states and transitions of a concurrent program is to take control of the scheduling of its processes. Stateless model checking tools use *cooperative scheduling*, making processes explicitly return control to a special scheduler at specific points of their execution. The points where such release of control happens are called *preemption points*, as it is only at those points that the special scheduler can preempt a process that could continue executing. This allows for more precise control compared to scheduling mechanisms provided by the operating system or a language's runtime environment. SMC tools modify the program executed by each process to insert preemption points before global operations. Local operations are executed together with a preceding global operation, as they cannot be individually affected by scheduling.

During execution under an SMC tool, the special scheduler will allow a single process at a time to execute a global operation (and any local operations following it), record information about the executed global operation and stop at the next preemption point. This is called an *execution step*. The scheduler may then allow the same process to perform more steps or choose a different one. The result of this procedure is a sequence of execution steps, called a *scheduling* of the processes (or an *execution sequence* or an *interleaving* or a *trace*). After each step, a process may not be able to continue executing (e.g., because its next global operation would be the acquisition of a lock that is held by another process). For that reason, after each execution step the scheduler needs to know which processes are *enabled*, i.e., able to continue execution. If no process is enabled, the resulting scheduling is called *maximal*.

2.2.2 Finitedness and Acyclicity

In order for schedulings to be finite, the state space corresponding to the concurrent program must be finite and acyclic. This is an assumption made by most SMC tools [18, 23, 36]. A SMC tool can use a *depth bound* to detect when a scheduling exceeds some predefined length, but techniques such as dynamic partial order reduction require that the program itself does not contain infinite schedulings. The reason is that such techniques rely on inspecting the operations that actually appear in schedulings and can therefore not take into account operations that are not executed due to a depth bound.

2.2.3 Statelessness via Determinism

If we assume that the execution of each process is deterministic, then by resetting the processes and any shared data back to their initial states, and replaying the execution steps used in a particular scheduling, we can reach any intermediate state of that scheduling. States of the program can therefore be encoded using the sequence of execution steps used to reach them. This eliminates the need to store any other state information.

In order to ensure that the execution of each process is deterministic, all other sources of non-determinism must be controlled, including inputs to the program and values returned by calls to the operating system or other programming language runtime mechanisms.

2.2.4 Soundness and Effectiveness

In order to be useful for verification, a stateless model checking algorithm needs to achieve two conflicting goals: on one hand, if a program behaviour is possible under some scheduling, then the algorithm must be able to find it (*soundness*). On the other hand, complete exploration of the state space (and therefore verification of the program) must be possible in a reasonable amount of time (*effectiveness*).

By using a controlled scheduler as described in Sect. 2.2.1, one could devise a naïve SMC algorithm which would simply try all possible scheduling choices after every preemption point. Such an algorithm would be sound, but ineffective, however, as the number of explored schedulings would be exponential with respect to the length of the execution, even when only global operations are considered as preemption points. This well-known phenomenon is often called the *state space explosion problem* [21].

2.3 Partial Order Reduction

Schedulings, as described in Sect. 2.2.1, impose a *total* order between operations, i.e., the order in which operations appear in the scheduling. When investigating the behaviour of a concurrent program, however, this total order may not be interesting, in the sense that small changes, such as swapping the execution of two adjacent steps (from different processes), may not affect the behaviour of the program.

As an example, consider a program in which two processes write at different shared memory locations. After both write operations have been completed, the state of the program is the same, regardless of the order of their execution. Exploring two schedulings whose only difference is the order of execution of these two operations would evidently be redundant.

The idea that SMC algorithms should avoid such redundant exploration is the basis of *partial order reduction (POR)* techniques [15, 21, 37, 42]. Instead of exhaustively exploring all possible scheduling choices at every step, an algorithm should focus instead on the partial order of interfering operations in a scheduling, as it is just those operations that need to be executed in a specific order to make the program's processes behave as in that particular scheduling; any other scheduling that maintains this partial order of operations will be equivalent. An SMC algorithm using POR must then ensure that it explores at least one scheduling in each such equivalence class (called a *Mazurkiewicz trace* [34]). This is sufficient for checking most interesting safety properties, including race freedom, absence of global deadlocks and absence of assertion violations [15, 21, 42].

2.3.1 Dependency Relations

In order to formally describe that two operations are interfering, POR algorithms use a *dependency* relation. This relation determines the partial order relation of interfering operations in a scheduling (also called *happens-before* relation [32]) which can be used to decide whether it is possible to reverse the order of execution of a particular pair of interfering operations. If that is the case, the pair is in a *reversible race*.

As POR algorithms work by exploring schedulings that reverse the order of such races, the precision of the dependency relation can significantly affect the achieved reduction. If, for example, all operations are assumed to be interfering, each possible scheduling will have a different partial order and no reduction will be possible. Operations should be considered as interfering only when their order of execution can affect a program's behaviour, i.e., one should be able to write a program in which executing a pair of interfering operations in a different order leads to a different result. We discuss some examples.

Read/Write Operations on Shared Memory

Two operations accessing shared memory are considered as interfering if they access the same memory location and at least one of them is a write operation. This leads to the following three pairings:

Write before Read: A write operation happens-before any later read operations at the same memory location.

Read before Write: A read operation happens-before any later write operations at the same memory location.

Write before Write A write operation happens-before any later write operations at the same memory location.

It is easy to write programs in which reversing the execution order of such pairs of operations can lead to different behaviours.

Notice that the precision of interference detection can be increased if we also consider the values used in write operations: operations that write the same value can be seen as independent, but all such operations must be ordered before a later operation that writes a different value in the location or reads it. This corresponds to the fact that operations that write the same value can be reordered without any observable result.

A common characteristic of all these orderings is that a particular pair of shared memory operations is considered ordered or not, regardless of what happens later in a scheduling. One can however argue that ordering pairs of write operations (i.e., the third case above) is interesting only when the memory location is later read. It could therefore be beneficial to treat such operations as interfering only in *some* particular extensions of the scheduling. This idea is discussed in Paper IV.

Synchronization Operations

Processes in concurrent programs often need to execute operations in a particular order, regardless of the choices of the scheduler. For that reason, concurrent systems support *synchronization* operations. A common example are operations involving locks. Once a process has acquired a lock, other processes attempting to acquire it are prevented from continuing their execution until the lock has been released. Therefore, lock acquisitions have dependencies with each other and are also dependent with lock releases.

Many other variants of synchronization operations exist, with the common feature that their execution may prevent some processes from continuing their execution. In Paper I we describe why dependencies for such variants can be trickier to handle.

Message Passing Operations

In actor programs, the sending and receiving of a particular message are dependent operations. Moreover, the delivery of a message may by itself be important, if messages can also be lost or if a process can perform some default action when no messages have arrived before some timeout.

If the order of delivery can affect which message is received, then it can alter the behaviour of a program. Even in such cases, however, if particular messages are never received, then the order of their delivery becomes irrelevant. This is an argument similar to the one made for write operations whose values are never read (in shared memory programs) and is another case examined in Paper IV.

2.4 Dynamic Partial Order Reduction

An algorithm can determine pairs of interfering operations statically, by inspecting the source code of a concurrent program. However, such analysis needs to make over-approximations in order to be sound. Aliasing of variables, for example, needs to be treated conservatively and operations that are not always executed due to the control flow of the program may also not be easy to detect accurately. In such cases, the loss of precision can make a static technique conservatively explore redundant schedulings, limiting the achievable reduction.

Dynamic Partial Order Reduction (DPOR) [18] achieves better reduction by detecting interferences between operations that are actually executed in a scheduling and planning additional schedulings *by need*. Each executed operation can be seen as an *event* in a scheduling. A DPOR algorithm can be described by the following steps:

(1) Explore some arbitrary first scheduling.
(2) In the currently explored scheduling, find pairs of events that are in a reversible race.
(3) For each such pair, check whether a different scheduling, in which the order of execution of the racing events is reversed, has already been explored or planned to be explored.
(4) If not, plan the exploration of a new scheduling that reverses the order of the racing operations. A suitable such scheduling is one that diverges from the one currently explored at the state from which the first event was executed (so that the second event can be executed before it). One or more steps of this new scheduling need to be specified.
(5) Backtrack to the latest state that describes an unexplored (diverging) scheduling (by replaying an appropriate prefix of the current scheduling), then diverge and explore a new scheduling, following any initial steps specified in step 4 and completing the scheduling arbitrarily.
(6) Repeat from step 2, until no more unexplored schedulings remain.

2.4.1 Example of Scheduling Exploration using DPOR

Let's see an example (also presented in Paper I). In Fig. 2.1, the three processes p, q, and r perform dependent (interfering) accesses to the shared variable `x`. We consider two accesses as interfering if they access the same variable and one of them is a write. Variables `y` and `z` are also shared, but since there are no write operations to them, the read accesses to them are not dependent with any other operation.

For this program, there are four Mazurkiewicz traces, each characterized by the sequence of accesses to `x` (three accesses can be ordered in six ways, but two different pairs of those orderings are equivalent since they only differ in the ordering of adjacent read operations, which are not dependent).

```
       p:             q:               r:
write x; (1) ‖ read y;      ‖ read z;
             ‖ read x; (2)  ‖ read x; (3)
```

Figure 2.1. Three processes that interfere by accessing shared memory.

Assume that the first arbitrarily explored scheduling is *p.q.q.r.r* (schedulings are denoted by the dotted sequence of scheduled process steps). A DPOR algorithm will detect that step `(1)` by *p* and step `(2)` by *q* are in a reversible race and note that it should explore a scheduling that starts with a step of *q*. The DPOR algorithm will also detect the dependency between `(1)` and `(3)` and possibly decide that it is necessary to explore schedulings that start with a step of *r*. The algorithm will then backtrack at the initial state, note that there is an unexplored scheduling diverging in the first step (starting with *q*), perform this diverging step and arbitrarily continue exploration. This procedure will continue until no more unexplored schedulings remain.

2.4.2 The Classic DPOR Algorithm

The operation of the classic DPOR algorithm [18] follows the sketch given in Sect. 2.4. Reversible races (step 2) are detected after the exploration of each execution step. New schedulings (step 4) are added by trying to schedule (at the state where the first operation was executed) a step from any process that has execution steps that happen before the second racing operation. If that is not possible (e.g., due to none of the possible processes being enabled at that state), the classic algorithm plans instead to explore schedulings starting with each enabled process. Such alternative scheduling choices are added in what is called a *backtrack set* at each state. Classic DPOR uses the following two important abstractions: (i) persistent sets, which are used to prove soundness and (ii) sleep sets, which are used to increase the effectiveness of the reduction. We take a closer look at each.

2.4.3 Persistent Sets

To prove the soundness of classic DPOR, it is shown [18] that, when backtracking, the final backtrack set at every state in the execution sequence is a *persistent set*. This is enough to guarantee the exploration of at least one scheduling in each Mazurkiewicz trace, when the explored state space is acyclic and finite. A set *P* of processes is persistent in some state if in any possible scheduling from that state, the first step that is dependent with the first step of some process in *P* is also taken by some process in *P*.

What this practically means is that, when inspecting a step p of a scheduling $E.p.w$, if the algorithm can see that by following a different scheduling $E.w'$ it would execute an operation that is interfering with p, then it must also explore a scheduling starting with an operation happening before p in w'.

The classic DPOR algorithm specifies the first step of additional schedulings in order to create backtrack sets of processes that eventually become persistent sets. In the example of Fig. 2.1, the only persistent set which contains p in the initial state is $\{p,q,r\}$. To see this, suppose that, e.g., r is not in the persistent set P, i.e., $P = \{p,q\}$. Then, the scheduling $r.r.p$ contains no step from a process in P, but its second step is dependent with the first step of p, which is in P. In a similar way, one can see that also q must be in P.

2.4.4 Sleep Sets

By just specifying the first step of new schedulings, there exists the possibility that the exploration of a scheduling does not reverse the order of execution for any pair of racing operations. In the example of Fig. 2.1, when the algorithm explores a scheduling starting with q, if it immediately continues with a step of p it will explore a scheduling that will be equivalent to the first scheduling.

A technique that can reduce the schedulings explored by a DPOR algorithm by avoiding explorations like the one just described is the use of *sleep sets* [21, 26]. Sleep sets use information from past explorations to prevent redundant future explorations. A sleep set is maintained for each prefix E of a scheduling that is currently explored, containing processes whose exploration would be redundant, because equivalent schedulings have already been explored. The algorithm then never explores steps by processes in the sleep set.

The sleep set at each prefix E is manipulated as follows: (i) after exploring schedulings that extend E with some process p, the process p is added to the sleep set at E, and (ii) when exploring executions that extend $E.p$, the sleep set at $E.p$ is initially obtained as the sleep set at E, with all processes whose next step is dependent with p removed. The result of this procedure is that in new schedulings each previously explored step needs to have some step interfering with it. In the program of Fig. 2.1, after having explored executions starting with p, the process p is added to the sleep set at the initial state, following rule (i). When initiating the exploration of executions that start with q, the process p remains in the sleep set, according to rule (ii), and it cannot be explored immediately after q, as executions that start with $q.p$ are equivalent to executions that start with $p.q$, and such executions have already been explored. The algorithm can, however, execute p after e.g., $q.q$, as the second step of q interferes with the first step of p and removes it from the sleep set.

Sleep sets are useful to guide new schedulings, but, as we will see in the next section, they are not always enough to completely avoid redundant exploration.

3. The Source and Optimal DPOR Algorithms

This chapter explains why the classic DPOR algorithm may perform redundant exploration and presents the Source and Optimal DPOR algorithms, summarizing the improvements to DPOR presented in Paper I[1].

3.1 Sleep Set Blocking

In classic DPOR, the use of persistent sets is enough to guarantee the exploration of at least one maximal scheduling in each Mazurkiewicz trace, ensuring soundness. Moreover, the use of sleep sets is sufficient to prevent the complete exploration of two different but equivalent maximal schedulings [24]. At first glance, the combination of the two techniques seems to achieve *optimal reduction*, producing an algorithm that explores exactly one scheduling in each Mazurkiewicz trace. The actual result, however, is an algorithm that *can* initiate the exploration of a scheduling equivalent to an already explored one. Such exploration will however be sooner or later blocked by the sleep sets, in the sense that all enabled processes will be in the sleep set. We call such schedulings *sleep set blocked*. When persistent sets and sleep sets are used for reduction, the exploration can include an arbitrary number of sleep set blocked schedulings.

In the example of Fig. 2.1, if the backtrack set formed at the initial state is $\{p,q,r\}$, then any schedulings that start with r will be sleep-set blocked, after having explored schedulings starting with p and q, as there is no operation that can interfere with q's read on y and take it out of the sleep set. This is clear evidence that persistent sets cannot be the basis of a DPOR algorithm that never initiates exploration of redundant schedulings.

3.2 Source Sets and Source DPOR

In Paper I, we present a fundamentally new DPOR technique, based on a new theoretical foundation for partial order reduction, in which persistent sets are replaced by a novel class of sets, called *source sets*. Source sets subsume persistent sets (i.e., any persistent set is also a source set), but are often smaller

[1] The chapter contains text from Paper I, edited to conform to the terminology used in this comprehensive summary.

than persistent sets. Moreover, source sets are provably minimal, in the sense that the set of explored processes from some state must be a source set in order to guarantee exploration of all maximal Mazurkiewicz traces.

Source sets are defined for a particular state and a set of possible continuations from that state. The set of processes S is a source set for the state after an execution sequence E and a set of sequences W such that $E.w$ is a valid execution sequence for each $w \in W$, if for all $w \in W$ there exists a scheduling $E.p.w'$ that is equivalent to $E.w$ and p is a process in S.

In the example of Fig. 2.1, the set $S = \{p,q\}$ is a source set for the initial state and the set of all maximal execution sequences, even though it does not include r. This is because any maximal scheduling starting with a step of r is equivalent to some maximal scheduling starting with the first step of q. Note that the set S is not a persistent set. Any persistent set is also a source set, but, as illustrated by this example, the converse is not true. The example also demonstrates that, if the smallest persistent set that contains a particular process contains more elements than the corresponding source set, the additional elements will always initiate sleep set blocked explorations.

As described in Sect. 2.4.2, the correctness of the classic DPOR algorithm was proven by establishing that sets of explored process steps are always persistent sets. In Paper I we prove that it is enough to show the weaker property that this set is always a source set. We thus claim that source sets are a better conceptual foundation for developing DPOR techniques.

To show the power of source sets we developed *Source DPOR* (Paper I), an algorithm based on source sets. It is derived by modifying the classic persistent-set-based DPOR algorithm [18] to generate source sets instead of persistent sets. The modification consists of a small change to a single test in the classic algorithm. The power of source sets can be observed by noting that Source DPOR achieves significantly better reduction in the number of explored schedulings than classic DPOR. In fact, Source DPOR achieves optimal reduction for a large number of the benchmarks used in Paper I.

Source sets were first presented in an earlier version of Paper I [2]. *Sufficient sets* [16] are a similar concept, described concurrently and independently but used for an entirely different purpose (bounded partial order reduction).

3.3 Wakeup Trees and Optimal DPOR

By utilizing source sets, Source DPOR explores the optimal number of executions for the program of Fig. 2.1. There are cases, however, where Source DPOR can also encounter sleep set blocked explorations.

We illustrate this by the example in Fig. 3.1 (also taken from Paper I). In this program with four processes, p,q,r and s, two operations are dependent if they access the same shared variable, i.e., `x,y` or `z`. Variables `l,m,n` and `o` are private. Each global operation has a unique label; e.g., process s has three

```
                         Initially: x = y = z = 0

      p:                  q:                 r:                     s:
  l := x; (1)  ||  y := 1; (2)  ||  m := y; (3)         ||  n := z; (5)
               ||               ||  if m = 0 then       ||  o := y; (6)
               ||               ||      z := 1; (4)     ||  if n = 1 then
               ||               ||                      ||      if o = 0 then
               ||               ||                      ||          x := 1; (7)
```

Figure 3.1. Processes whose control flow can be affected by the scheduling.

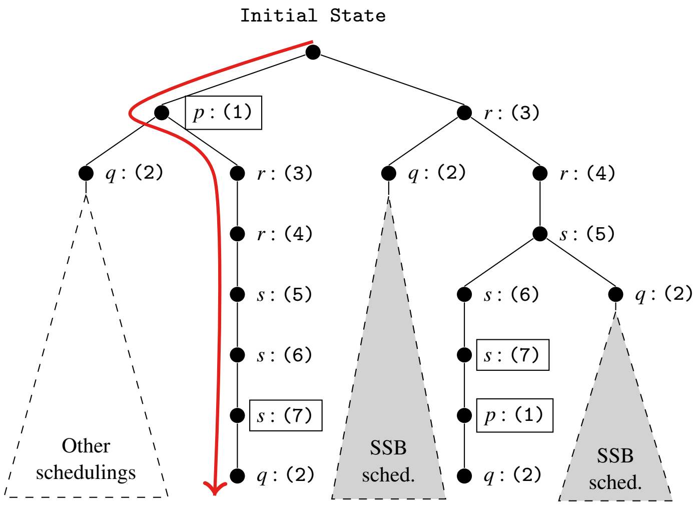

Figure 3.2. Schedulings for the program of Fig. 3.1.

such operations labeled (5), (6), and (7). Operations on private variables are assumed to be part of the previous global operation. For example, label (6) marks the read of the value of y, together with the assignment to o, and the condition check on n. If the value of n is 1, the condition check on o is also part of (6), which ends just before the assignment to x that has the label (7), if the second condition is also satisfied. Similar assumptions are made for all other local operations.

Consider a DPOR algorithm that starts the exploration with *p*. The algorithm should eventually also explore the scheduling *p.r.r.s.s.s* (marked in Fig. 3.2 with a red arrow). During this scheduling, it will detect the race between events (1) and (7). It must therefore explore some scheduling in which the race is reversed, i.e., event (7) occurs before event (1). Note that event (7) will only occur if preceded by the sequence (3)-(4)-(5)-(6) and not preceded by a step of process *q*. Thus, a scheduling that reverses this race must start with the sequence *r.r.s.s*.

When Source DPOR detects this race in *p.r.r.s.s.s*, it will add *r* to the backtrack set at the initial state in order to make it a source set. However, when exploring a scheduling starting with *r*, Source DPOR cannot 'remember' that *r* must be followed by *r.s.s* to reverse the race. It is therefore free, after executing *r*, to continue with *q*. However, after *r.q*, any further exploration is doomed to encounter sleep set blocking. To see this, note that *p* goes in the sleep set when exploring *r*, and will remain there forever in any sequence that starts with *r.q* (as explained above, *p* can be removed only by the last event of the sequence *r.r.s.s.s*). This corresponds to the left chunk labeled as "SSB sched." (Sleep Set Blocked schedulings) in Fig. 3.2.

The algorithm cannot completely ignore sleep set blocked schedulings, as it has to reverse racing operations in them to eventually find the 'correct' scheduling (shown in Fig. 3.2 between the two "SSB sched." chunks). It may however have to explore an arbitrary number of sleep set blocked schedulings; the "SSB sched." chunk on the right is reachable by a similar 'bad' scheduling of *q*.

In order to obtain an optimal DPOR algorithm, we can replace the backtrack set with a data structure called a *wakeup tree*. Wakeup trees are constructed using information from already explored schedulings, hence they do not increase the amount of exploration. They consist of so called *wakeup sequences* that guarantee the reversal of detected races, and are composed in a way that ensures that future explorations will never be sleep set blocked.

Use of wakeup trees leads to the *Optimal DPOR* algorithm. The algorithm differs from classic and Source DPOR as it performs race detection at the end of a scheduling. This happens because wakeup sequences need to contain all the events that are independent with a race, in order to guarantee soundness.

In the example, Optimal DPOR will handle the race between `(1)` and `(7)` by adding the entire wakeup sequence *r.r.s.s.s* to a wakeup tree at the initial state. When this sequence is executed, the last event will remove process *p* from the sleep set and so sleep set blocking will be avoided. Any other sequence added to this tree must also lead to an operation removing *p* from the sleep set. However, new sequences are only added when they are not 'compatible' (due to races) with any existing sequences in the tree. Such incompatibilities immediately imply that such sequences will include operations that will also clear any *future* additions to sleep sets.

In Paper I, Optimal DPOR is initially presented with the assumption that a process may only block itself, e.g., by waiting to receive a message. The handling of operations by which a process can affect the enabledness of other processes is trickier and is discussed separately.

3.4 Performance of Source and Optimal DPOR

Table 3.1 aggregates evaluation results presented in Paper I. All results correspond to verification, i.e., exploration of the entire state space of each bench-

Table 3.1. *Comparison of the classic, Source and Optimal DPOR algorithms.*

	Schedulings Explored			Time		
Benchmark	classic	source	optimal	classic	source	optimal
filesystem(14)	4	2	2	0.54s	0.36s	0.35s
filesystem(16)	64	8	8	8.13s	1.82s	1.78s
filesystem(18)	1 024	32	32	2m 11s	8.52s	8.86s
filesystem(19)	4 096	64	64	8m 33s	18.62s	19.57s
indexer(12)	78	8	8	0.74s	0.11s	0.10s
indexer(15)	341 832	4 096	4 096	56m 20s	50.24s	52.35s
readers(2)	5	4	4	0.02s	0.02s	0.02s
readers(8)	3 281	256	256	13.98s	1.31s	1.29s
readers(13)	797 162	8 192	8 192	86m 7s	1m 26s	1m 26s
dialyzer	12 436	3 600	3 600	14m 46s	5m 17s	5m 46s
gproc	14 080	8 328	8 104	3m 3s	1m 45s	1m 57s
poolboy	6 018	3 120	2 680	3m 2s	1m 28s	1m 20s
rushhour	793 375	536 118	528 984	145m 19s	101m 55s	105m 41s
lastzero(5)	241	79	64	1.08s	0.38s	0.32s
lastzero(10)	53 198	7 204	3 328	4m 47s	45.21s	27.61s
lastzero(15)	9 378 091	302 587	147 456	25h 39m 11s	55m 4s	30m 13s
example-3.1-ext(7)		373	29		2.38s	0.26s
example-3.1-ext(8)		674	33		4.70s	0.34s
example-3.1-ext(9)		1 222	37		8.79s	0.44s

mark. In all benchmarks it is evident that Source DPOR can explore an order of magnitude fewer schedulings than classic DPOR. It is even often the case that Source DPOR achieves optimal reduction. However, in cases where Source DPOR encounters a lot of sleep set blocked explorations (e.g., the lastzero benchmark), Optimal DPOR can halve the number of explored schedulings.

When Source DPOR does not encounter a lot of sleep set blocked explorations, Optimal DPOR can be slower, even when it explores fewer schedulings (e.g., the gproc and dialyzer benchmarks), due to the added complexity of maintaining wakeup trees. In our tests however, Optimal DPOR never requires more than 10% of additional time in such cases.

Notice that even in cases such as the one shown in Fig. 3.2, Source DPOR can be 'lucky' and explore the 'correct' scheduling first, encountering no sleep set blocking. When Source DPOR does encounters sleep set blocked explorations, however, Optimal DPOR can dramatically reduce the total exploration time. In Paper I, we show that on particularly hard inputs, such as an extended version of the example of Fig. 3.1 (example-3.1-ext in Table 3.1)[2], Source DPOR may explore an exponential number of additional schedulings compared to Optimal DPOR. This has also been confirmed in other work [30], showing that Source DPOR is sensitive to scheduling choices.

[2]See however Sect. 3.5.

Another observation from our tests is that memory use is practically the same between Source and Optimal DPOR (more data is given in Paper I). One can nevertheless construct programs where the size of wakeup trees grows exponentially and, consequently, the memory requirements of Optimal DPOR become considerably worse than those of Source DPOR. Each branch in a wakeup tree, however, is a prefix of some execution that needs to be explored. The size of the wakeup trees can therefore never be larger than the size of all explored executions and memory consumption becomes a problem only when any DPOR algorithm would have to explore an exponential number of schedulings.

In conclusion, we believe that, while Source DPOR is a good direct replacement of classic DPOR, Optimal DPOR is the algorithm that should be used in state-of-the-art SMC tools.

3.5 Correction for Paper I

When writing this summary, we noticed that the pseudocode given in Paper I, page 38, Fig. 10 for an extended version of the program of Fig. 3.1 in this summary (Fig. 2 in Paper I) did not exactly correspond to the program that was used to produce the results shown in Paper I, page 39, Table 1. The results were produced by a program (shown in Fig. 6.1 in this summary) in which the read of the variable `yi` (and the assignment "`l` := `yi`") by each process s_i is executed *after* the following check "`if n = 1`" that involves the local variable `n` (the two lines have essentially been swapped in the program used to produce the results in Table 1 of Paper I).

The number of explored schedulings for the program that is exactly corresponding to the pseudocode given in Paper I are:

n	1	2	3	4	5	6	7	8	9
Source DPOR	12	29	61	110	189	315	518	845	1373
Optimal DPOR	7	13	19	25	31	37	43	49	55

These results also demonstrate an exponential gap between Source and Optimal DPOR, as described in both Paper I and in Sect. 3.4 of this summary.

4. Using Observability in DPOR

This chapter describes how the use of the observability of the interference between operations can lead to better reduction in DPOR algorithms, summarizing the improvements presented in Paper IV[1].

4.1 Observability by Examples

DPOR algorithms conservatively consider operations to be interfering if their execution order may influence the result of future operations. In the previous chapter, for example, the interference of shared memory operations was determined using *data races*: two operations on the same variable were deemed as interfering if at least one of them was a write.

In the example shown on the right, the shared variable x is accessed by processes p,q and r, with r checking its value in an assertion. If interference is decided using data races then all three operations (two writes and a read) interfere with each other. As a result, each of the $3! = 6$ possible interleavings has a different partial order and therefore belongs to a different Mazurkiewicz trace that should be explored by a DPOR algorithm. In schedulings starting with r, however, the order of the execution of p and q is irrelevant (if one does not care about the final contents of the memory), as the values written by these operations will never influence the assertion. A DPOR algorithm could detect that the written values are *not observed* and treat the write operations as non-interfering.

```
         Initially: x = 0

   p:         q:          r:
x := 1 || x := 2 || assert(x < 3)
```

Taking this idea further, in the program shown on the right, N processes write on the shared variable x, and as a result there exist $N!$ schedulings. In each such scheduling, however, only the last written value will be read in the assertion, which is now executed after all processes have completed their execution.

```
          Initially: x = 0

   p1:        p2:      ...     pN:
x := 1 || x := 2 || ... || x := N

          join processes;
          assert(x > 0)
```

[1] The chapter contains text from Paper IV, edited to conform to the terminology used in this comprehensive summary.

A DPOR algorithm could consider write operations that are not subsequently observed as independent and therefore explore just N instead of $N!$ schedulings, thereby achieving an exponential reduction.

In both examples, better reduction could be obtained if the interference of write operations, which are conservatively considered as always interfering, was characterized more accurately by looking at complete executions and taking observability by 'future' operations into account.

```
Initially: r's mailbox is empty

   p:          q:            r:
r ! M1  ||  r ! M2  ||  receive x
```

This idea is also applicable in other models of concurrency. In the message passing program shown on the right, processes p and q each send a different message to the mailbox of process r using the send operator "`!`". Process r uses a receive operation to retrieve a message and store it in a (local) variable `x`. If we assume that receive operations pick and return the oldest message in the mailbox and return `null` if no message exists, send operations can interfere (the order of delivery is significant) and so can send and receive operations (an empty mailbox can yield a different value). As a result, six schedulings are again possible. However, only three schedulings need to really be explored: the receive operation interferes only with the earliest send operation and cannot be affected by a later send; moreover, if the receive operation is executed first, the order of the send operations is irrelevant.

If we instead assume that receive operations *block* if no matching message exists, only *two* schedulings need to be explored, as r can receive either $\mathtt{M}_1$ or $\mathtt{M}_2$. Again, if we generalize the example to N processes instead of just two, the behaviour is similar to the program with N writes: only N schedulings (instead of $N!$) are relevant, each determined by the first message delivered; the remaining message deliveries are not observable. Note that, in this concurrency model, we are interested in the observability of the *first* instead of the last operation in an execution sequence.

In some message-passing concurrency models (e.g., Erlang programs [6]), it is further possible to use *selective* receive operations instead, which also block when no message can be selected. Using this feature, the previous program can be generalized and rewritten so that r is explicitly picking messages in order, using pattern matching.

```
Initially: r's mailbox is empty

  p1:        p2:      ...     pN:          r:
r ! M1 || r ! M2 || ... || r ! MN || receive M1;
       ||        ||     ||        || receive M2;
       ||        ||     ||        ||     :
       ||        ||     ||        || receive MN
```

Such a program is shown on the right. Here r wants to receive the N messages in order: first $\mathtt{M}_1$, then $\mathtt{M}_2$, etc. Thus, the order of delivery of messages is irrelevant. A DPOR algorithm could take advantage of the additional information provided by the selective receive operations, notice that each such operation can pick only one specific message and therefore determine that the *N* sends are independent. A *single* scheduling is enough to explore all behaviours of the program!

This idea of *observability* can be combined with the Optimal DPOR algorithm to achieve such reductions. The intuition behind this improvement comes from the fact that operations that *observe* a value (e.g., assertions that check some value, receive statements, etc.) are the only ones that can influence the control flow and lead to erroneous or generally unexpected behaviour. At the same time, other operations (e.g., writes, sends, etc.) cannot affect program behaviour if no future operation observes their effects. In such cases, interference between those other operations can be ignored.

4.2 Optimal DPOR with Observers

In Paper IV we extend the Optimal DPOR so that it *lazily* considers interferences based on the existence of *later* operations, called *observers*. In the simplest case, operations which would normally be considered interfering are considered independent in the absence of an observer. There are two main challenges to enable this extension:

1. We need to handle the fact that interference between operations is conditional.
2. Optimal DPOR uses sleep sets to guarantee that there is no redundant exploration, but, as we explain in the paper, their use in the presence of observers is problematic. A suitable replacement must therefore be found.

To address challenge 1, we extend the wakeup sequences constructed for reversing the order of interfering operations that require an observer with a suffix that includes the observer. It is allowed for this suffix to include operations happening after the interfering operations (even in program order) as any such operations will behave identically in the new scheduling. This is because the observer is the first event in the original scheduling that could be affected by the order of the interfering operations.

To address challenge 2, we build on the intuition behind sleep sets and assert that, when the extended algorithm backtracks from a particular state, it has explored all schedulings that can start with the step that led to that state. In Optimal DPOR, sleep sets are used to perform redundancy checks before adding a wakeup sequence. When the extended algorithm needs to consider whether to add a new wakeup sequence or not, information about observers is recalculated from the operations in the sequence. The algorithm then performs an exhaustive test, ensuring that each step previously explored from any point in the execution is overtaken by some other step in the wakeup sequence under consideration.

4.3 Performance of Optimal DPOR with Observers

Results from the evaluation of Optimal DPOR with observers, as presented in Paper IV, are given in Table 4.1. As in Sect. 3.4, all results correspond to verification of the benchmarks.

Table 4.1. *Comparison of Optimal DPOR and Optimal DPOR with Observers.*

	Schedulings Explored		Time	
Benchmark	optimal	observers	optimal	observers
lock(3)	30	6	0.9s	0.9s
lock(4)	336	24	1.4s	0.9s
lock(5)	5 040	120	9s	1.3s
lock(6)	95 040	720	3m 27s	2.6s
poolboy	746	265	6.6s	4.0s
gproc	1168	784	12.7s	10s
corfu-repair	> 30 000 000	3 864 604	> 750h	52h
selective(2)	2	1	1.0s	1.0s
selective(6)	720	1	1.8s	1.0s
selective(7)	5 040	1	6.3s	1.0s
selective(8)	40 320	1	51s	1.0s

It is clear that use of observers increases the effectiveness of Optimal DPOR. In particular cases (e.g., the corfu-repair benchmark) use of observers is practically *required* to achieve verification. It is also evident (e.g., in the selective benchmark) that use of observers can even lead to an exponential reduction in the number of explored schedulings.

5. Bounding

Apart from verification, stateless model checking can also be used for testing, in which case the goal is just to find concurrency errors. The algorithms described before guarantee that if a scheduling that triggers an error exists, an equivalent scheduling will eventually be explored. Systematic algorithms, however, may need to explore an arbitrary amount of schedulings before they find a scheduling that exposes the error. When using stateless model checking for testing it may therefore be desirable to spread-out the exploration.

This can be achieved using *bounding* techniques, that impose constraints on how/when processes can be scheduled by a stateless model checking algorithm and force the algorithm to focus the exploration on schedulings that satisfy those constraints. In this way, bugs in ‘simpler’ schedulings can be detected faster than when using exhaustive exploration. Schedulings that violate the constraints can also be explored, but each exploration begins with a budget (also called a *bound*), which is spent whenever the algorithm schedules processes in a way that violates the constraints. When no budget remains, the SMC algorithm can only explore schedulings that satisfy the constraints.

The two main bounding techniques used in stateless model checking are preemption bounding and delay bounding [41].

Preemption bounding [38] limits the number of times the scheduler can preempt a process which could execute more operations, in order to run other processes. The justification is that common patterns of concurrency bugs require few operations to be scheduled in a particular order and this observation can in turn can be related to few preemptions [11]. When a process cannot run more operations (e.g., by trying to acquire a lock that is not free), any other process can be scheduled without consuming budget.

Delay bounding [17] is a more restrictive technique, forcing the scheduler to always pick the first available process from a total order (e.g., round-robin) of all processes. The bound here corresponds to the number of times a process is skipped (or delayed). Unlike preemption bounding, only a specific process can be picked for scheduling without consuming budget, even when a process is blocked; other choices would ’delay’ the next process in the order. Moreover, preempting a process (in the way described before) has a variable cost that depends on which process is scheduled instead, as this decision affects how many other processes (including the preempted one) have to be delayed.

A noteworthy point is that if a tool tries all possible ways to spend the budget without finding a bug, it can *guarantee* that no bug exists in schedulings with a cost lower than the bound.

5.1 Combining POR and Bounding

It has been shown that delay bounding is more effective than preemption bounding for finding bugs [41]. It is easy to see however, that by using just bounding a tool may explore multiple equivalent schedulings.

A combination of classic DPOR and preemption bounding has already been attempted [16]. In that work, the key idea of the proposed *Bounded DPOR* algorithm is to explore *two* schedulings in which a detected race can be reversed. The first of these schedulings diverges from the one exposing the race exactly at the step where the first operation was executed (this will possibly preempt the first operation's process and cost budget) and the second diverges at the closest step where the original process was freely scheduled (avoiding the cost, as any other process can also be freely scheduled there). This second scheduling is added conservatively, to cover cases where it is possible to reverse the race without spending budget (and save budget to be spent for reversing later races, retaining preemption bounding's guarantee). Unfortunately, in such conservative schedulings, sleep sets can no longer be used, as the racing operation is different from the one that is deferred. As a result, Bounded POR may sometimes redundantly explore maximal schedulings that are equivalent.

Bounded DPOR can be easily modified to work with source sets, as Source DPOR is very similar to classic DPOR, but the redundancy problem remains. Combining preemption or delay bounding with Optimal DPOR is significantly more difficult, as finding the minimum number of preemptions required to explore a particular wakeup sequence is an NP-complete problem [35].

5.2 Exploration Tree Bounding

In Paper III we instead showcase *exploration tree bounding*, a simple bounding technique compatible with any DPOR algorithm.

Exploration tree bounding restricts the number of times a scheduling explored by a DPOR algorithm can diverge from an already explored scheduling. In SMC tools, the first scheduling explored is usually the one chosen under preemption or delay bounding: a round-robin scheduling of the processes, without preemptions enforced by the scheduler. Exploration tree bounding limits the number of times exploration can diverge from that first scheduling, and essentially combines all the benefits of a DPOR technique (e.g., optimality), with delay bounding's effectiveness in finding bugs. Moreover, to avoid cases where an exploration would become "bound blocked", exploration tree bounding allows processes in the sleep set to be delayed without a cost.

Exploration tree bounding does not offer guarantees such as e.g., the exploration of all schedulings that satisfy some particular scheduling constraints, like preemption or delay bounding do, but it is nevertheless effective for finding bugs. In Paper III we show, e.g., a case in which use of exploration tree bounding resulted in finding an error in 57s instead of 144h.

6. Concuerror: An SMC Tool for Erlang Programs

This dissertation's contributions to stateless model checking were described from a general perspective in Chapters 3 to 5. The techniques were however also applied in practice in *Concuerror*, a SMC tool for Erlang programs. In this chapter, we describe some characteristics of Erlang together with some design and implementation aspects of the tool[1].

6.1 Erlang

Erlang is an industrially relevant programming language based on the actor model of concurrency [6]. In Erlang, actors are realized by language-level processes implemented and managed by the runtime system instead of being directly mapped to operating system threads. Each Erlang process has its own private memory area (stack, heap, and mailbox) and communicates with other processes via message passing.

Erlang is a language famous for its "shared nothing" approach to concurrency [6]. However, Erlang's main implementation, the Erlang/OTP system, comes with a large number of *built-in operations* that depend on and affect shared memory. This is not surprising, as it is impossible to write any interesting concurrent program without some interaction between processes. Even if this interaction consists of sending a message from one process to another, at the VM level this means that the send operation needs to write to some memory that is not local to the process that executes the send, namely to the recipient's mailbox. Therefore, *some* shared memory accesses do take place, even in pure message-passing concurrent Erlang programs.

Erlang/OTP also comes with a key-value store mechanism, called *Erlang Term Storage (ETS)*, that allows processes to create memory areas in which they can insert, look up, and update terms. Such areas, called ETS tables, can be explicitly declared as `public`, leading to shared access between processes. The runtime system automatically serializes accesses to these tables when this is necessary and also comes with mechanisms that guarantee atomicity of some operations (e.g., a bulk `insert`). It is however easy to see that operations on a public ETS table can give rise to interference similar to that of

[1] The chapter contains text from Papers I, II and IV, edited to conform to the terminology used in this comprehensive summary.

data races. Moreover, ETS operations can also be affected by other events in a program, such as a process crashing, as each ETS table is owned by the process that created it and its memory is reclaimed by the runtime system when this process exits if no other process has been assigned to inherit the table.

In Paper II, we systematically describe all the interferences between the built-in operations of the Erlang/OTP implementation. We also introduce a number of *events* that are considered to either be executed by processes (e.g., in the case of process termination), or by other independent entities (e.g., conceptual transit mechanisms in the case of message delivery). The reason for inclusion of events is that asynchronous message passing and mechanisms that support fault tolerance add complex ways in which processes interact, which do not always directly correspond to the execution of a built-in operation by a process (e.g., the order of delivery of messages can be different from the order of execution of send operations). Events are compatible with other built-in operations, in the sense that they can also be placed in schedulings, facilitating partial order reduction techniques.

Paper II is accompanied by a publicly available litmus test suite. Each included test is a program whose result depends on the scheduling of a pair of built-in operations or events, showing how those operations or events interfere.

6.2 Concuerror

We now turn to Concuerror [12], a stateless model checking tool for finding concurrency errors in Erlang programs or verifying their absence. The tool is publicly available at:

`http://parapluu.github.io/Concuerror/`

Concuerror is itself written in Erlang.

6.2.1 Instrumentation of Erlang Programs

Concuerror employs a source-to-source transformation that inserts preemption points in the code under execution. This instrumentation allows the tool to take control of the scheduling of the program, without having to modify the Erlang VM in any way. In the current VM, a context switch may occur at any function call. In line with other tools [22] however, Concuerror inserts preemption points only at process actions that interact with shared state (i.e., global operations).

Preemption points are implemented as `receive` statements which block the execution of a process until a suitable ‘continue’ message is sent by a separate “scheduler” process. In this way, Concuerror intervenes in a minimal way in the execution of an Erlang program.

Concuerror supports the complete Erlang language and can instrument programs of any size. Additionally, the tool uses Erlang's dynamic code loading capabilities to detect calls to any libraries a test may use, automatically instrument those libraries and reload them, without affecting other processes that use such code but are not part of the test (e.g., "system" processes of a node). Concuerror is powerful enough to be run on itself.

6.2.2 Controlled Scheduling

The ability to reach a previously encountered state by scheduling the program's processes in the same way as they were originally scheduled is a basic assumption in stateless model checking. In order to achieve that for an arbitrary Erlang program, a number of issues had to be addressed.

Non-deterministic Values

Some Erlang operations generate values in a non-deterministic way. Such values are process or ETS table identifiers, as well as other special values such as references, port identifiers, or unique integers. Such operations interfere, since they may use shared data to ensure the uniqueness of the returned values. However, values returned by such built-ins are in general unpredictable; e.g., the result of a comparison between two PIDs or two references can be different for reasons that cannot always be explained by the scheduling of the operations generating those values. As SMC tools require such values to remain identical between executions, Concuerror is recording and reusing them when needed.

Process Management

Erlang supports dynamic process creation. Concuerror's instrumentation captures calls to the `spawn` function and updates its knowledge of the available processes in the program. Moreover, Concuerror does not let processes actually terminate, but simulates all the steps happening at process termination. This allows finer control of the scheduling of the events that can happen during process termination and is also necessary in order to keep PIDs identical between schedulings.

Message Passing

Concuerror simulates the asynchronous message passing semantics of Erlang using asynchronous events, as described in Paper II.

The tool does not model/handle time related to timeout clauses. As a result, if a `receive` statement has a timeout clause, Concuerror will by default treat it as reachable. A user can however override this setting, specifying a threshold such that any such clause with a timeout value over the threshold is treated as unreachable.

Operations Outside Scope
Programs running on an Erlang node may try to interact with "system" processes that are running by default on the node. The state of such processes may however be impossible to reset between schedulings. Moreover, a process may try to execute operations which are difficult to control in the ways required by SMC, such as file modifications. Concuerror includes a whitelist for some of these operations, uses a "record and reuse" mechanism (similar to the one employed for non-deterministic values) for some others and allows users to specify code that should be trusted to behave as if it was purely functional. If the tool however detects that some executed operation returns a result different than expected, it reports the discrepancy and aborts the exploration.

6.2.3 Implementation of DPOR Algorithms

Earlier versions of the tool [27] did not include any POR techniques. Starting from Paper I, all of the code of the tool was gradually rewritten to include the several DPOR improvements described in this dissertation.

Concuerror detects interference between operations in a scheduling using hard-coded information about the interference between Erlang built-ins. It discerns races by identifying dependencies that are reversible (e.g., between the delivery of two messages to the same process) or irreversible (e.g., between a message's delivery versus its receive). It stores dependency information into a collection of vector clocks at each step of the execution (one clock per-process and an additional clock for the step itself).

In Paper II, we describe how Concuerror was used in an exhaustive mode (without any DPOR techniques enabled) to verify that the litmus tests were correct. Conversely, the tests have been used to verify that Concuerror successfully detects all the dependencies between the built-in operations and events described in the paper. This offers a minimal guarantee that the results reported by Concuerror (especially when no errors are found in a program) are valid.

6.2.4 Output

If Concuerror detects that in some explored scheduling a process has crashed or some processes are stuck in a global deadlock (i.e., they are all waiting for messages), it prints a trace of the scheduling that lead to the error in the report file it generates. The trace includes all interesting events (i.e., execution of built-in operations, message deliveries, etc.) annotated with the location at which they were called. Concuerror can also be configured to ignore some types of errors.

Alternatively, Concuerror can output an exploration graph, optionally annotated with the pairs of racing operations that justify each explored scheduling.

Such a graph, generated from a modified version of the program described in Fig. 3.1 (the corresponding Erlang program is given in Fig. 6.1) is shown in Figure 6.2. Each of the five explored schedulings had no errors (indicated by the green boxes in the bottom)[2]. Processes exiting normally are marked with green borders, racing operations that add new schedulings with red dashed arrows, and new schedulings with horizontal arrows at the state from which they diverge.

6.2.5 Usability Aspects

One of the less highlighted aspects of computer science research is its usability in the hands of non-expert users. Tools, especially, should have a clear purpose and be simple and intuitive in their use.

Regarding its purpose, Concuerror was developed in an advantageous environment, as the audience of Erlang programmers innately understands the problems of concurrent programming, particularly those involving racing messages delivered to an actor. Previous contributions in the area such as McErlang [10, 19] (a model checking tool for Erlang programs, based on a formal specification [40]) and PULSE [13] (a controlled scheduler for Erlang programs that can be used for randomized testing) have also been used to find concurrency errors.

Nevertheless, a number of features of Concuerror, such as the automatic instrumentation of programs, the graph output, and the existence of a mechanism that monitors the explored schedulings and emits tips that guide the user into a better understanding of the workings of the tool (and adjust their test and use of the tool accordingly) have been implemented to increase the tool's usability. The command-line output[3] shown in Fig. 6.3 shows examples of such features.

An attestation of the impact of these features is that the tool has been tried by independent Erlang users, leading to results such as the case study presented in Paper III and discussed in Sect. 7.2 of this summary.

[2] For this program Concuerror was configured to ignore deadlocks, as a deadlock is caused by fact that the first process (`P`) has not exited and remains blocked at the end of the execution. This deadlock is used to avoid interferences that will appear if the first process is allowed to terminate, as in that case the ETS table used to store the shared variables will be deleted and the deletion will be racing with all other operations on the table. An alternative way to avoid this behaviour is to ensure that each child process (`P1`-`P4`) has completed its accesses by having it send a message back to the first process.

[3] Notice that this is different from the report file generated by the tool.

```erlang
-module('fig3.1').

-export([test/0]).

-concuerror_options([{ignore_error, deadlock}]).

test() ->
  _P = self(),
  ets:new(table, [public, named_table]),
  ets:insert(table, {x, 0}),
  ets:insert(table, {y, 0}),
  ets:insert(table, {z, 0}),
  _P1 = spawn(fun() -> [{x, _L}] = ets:lookup(table, x) end),
  _P2 = spawn(fun() -> ets:insert(table, {y, 1}) end),
  _P3 =
    spawn(
      fun() ->
          [{y, M}] = ets:lookup(table, y),
          case M of
            1 -> ok;
            0 -> ets:insert(table, {z, 1})
          end
      end
     ),
  _P4 =
    spawn(
      fun() ->
          [{z, N}] = ets:lookup(table, z),
          %% [{y, O}] = ets:lookup(table, y), % <- This line...
          case N of
            0 -> ok;
            1 ->
              [{y, O}] = ets:lookup(table, y), % <- ... is moved here.
              case O of
                1 -> ok;
                0 -> ets:insert(table, {x, 1})
              end
          end
      end
     ),
  block().

block() -> receive after infinity -> ok end.
```

Figure 6.1. A modified version of the program shown in Fig. 3.1, written in Erlang. Shared variables are stored in a public ETS table named `table`. The modification moves the assignment to variable `O` after the first condition check and results in a program that has five instead of seven schedulings, conveniently allowing the corresponding exploration graph shown in Fig. 6.2 to fit in the next page.

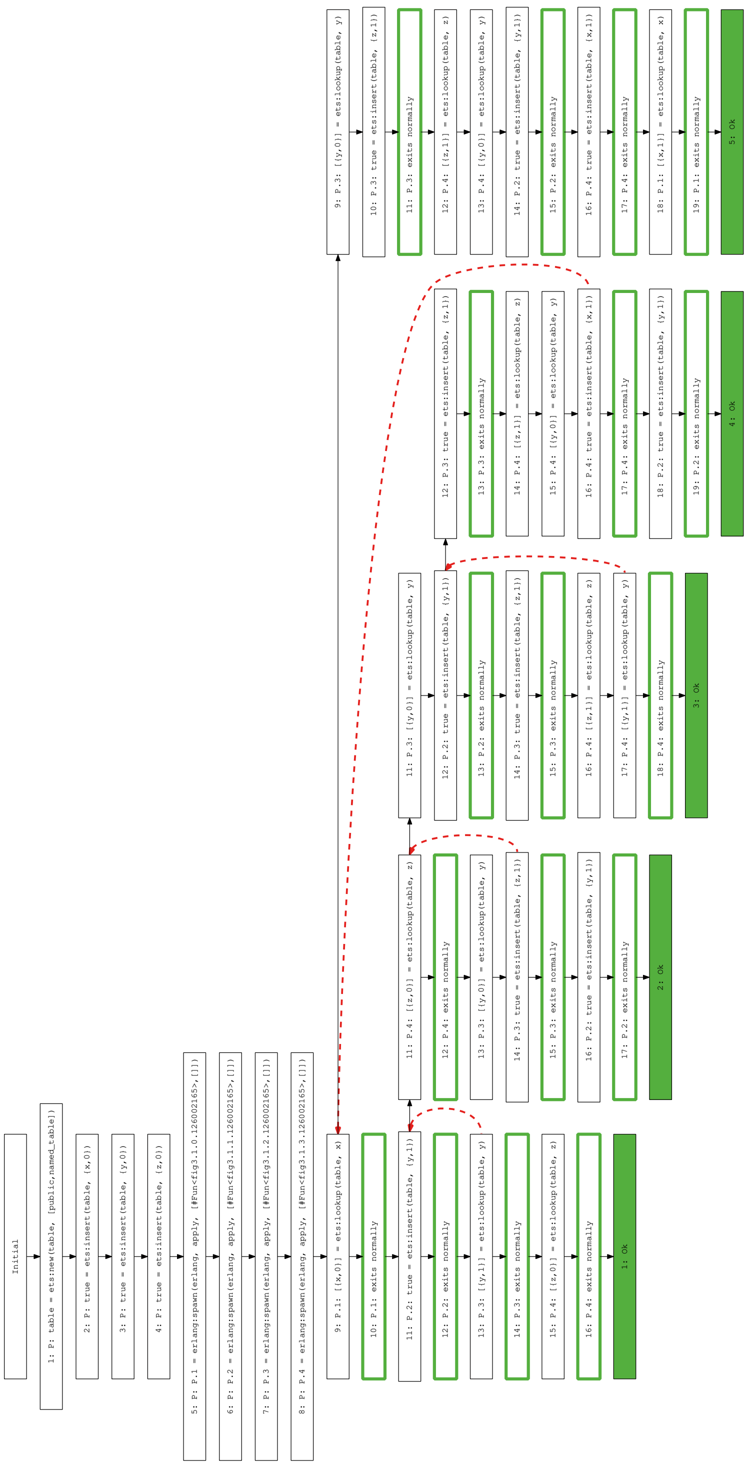
Initial
1: P: table = ets:new(table, [public,named_table])
2: P: true = ets:insert(table, {x,0})
3: P: true = ets:insert(table, {y,0})
4: P: true = ets:insert(table, {z,0})
5: P: P.1 = erlang:spawn(erlang, apply, [#Fun<fig3.1.0.126002165>,[]])
6: P: P.2 = erlang:spawn(erlang, apply, [#Fun<fig3.1.1.126002165>,[]])
7: P: P.3 = erlang:spawn(erlang, apply, [#Fun<fig3.1.2.126002165>,[]])
8: P: P.4 = erlang:spawn(erlang, apply, [#Fun<fig3.1.3.126002165>,[]])
9: P.1: [{x,0}] = ets:lookup(table, x)
10: P.1: exits normally
11: P.2: true = ets:insert(table, {y,1})
12: P.2: exits normally
13: P.3: [{y,1}] = ets:lookup(table, y)
14: P.3: exits normally
15: P.4: [{z,0}] = ets:lookup(table, z)
16: P.4: exits normally
1: Ok
11: P.4: [{z,0}] = ets:lookup(table, z)
12: P.4: exits normally
13: P.3: [{y,0}] = ets:lookup(table, y)
14: P.3: true = ets:insert(table, {z,1})
15: P.3: exits normally
16: P.2: true = ets:insert(table, {y,1})
17: P.2: exits normally
2: Ok
11: P.3: [{y,0}] = ets:lookup(table, y)
12: P.2: true = ets:insert(table, {y,1})
13: P.2: exits normally
14: P.3: true = ets:insert(table, {z,1})
15: P.3: exits normally
16: P.4: [{z,1}] = ets:lookup(table, z)
17: P.4: [{y,1}] = ets:lookup(table, y)
18: P.4: exits normally
3: Ok
12: P.3: true = ets:insert(table, {z,1})
13: P.3: exits normally
14: P.4: [{z,1}] = ets:lookup(table, z)
15: P.4: [{y,0}] = ets:lookup(table, y)
16: P.4: true = ets:insert(table, {x,1})
17: P.4: exits normally
18: P.2: true = ets:insert(table, {y,1})
19: P.2: exits normally
4: Ok
9: P.3: [{y,0}] = ets:lookup(table, y)
10: P.3: true = ets:insert(table, {z,1})
11: P.3: exits normally
12: P.4: [{z,1}] = ets:lookup(table, z)
13: P.4: [{y,0}] = ets:lookup(table, y)
14: P.2: true = ets:insert(table, {y,1})
15: P.2: exits normally
16: P.4: true = ets:insert(table, {x,1})
17: P.4: exits normally
18: P.1: [{x,1}] = ets:lookup(table, x)
19: P.1: exits normally
5: Ok

Figure 6.2. Concuerror's `--graph` output for the code in Fig. 6.1, rendered with `dot`.

```
Concuerror v0.18-hipe (cac7c44) started at 20 Nov 2017 17:56:02

Writing results in concuerror_report.txt

* Info: Automatically instrumented module io_lib
* Info: Instrumented & loaded module t_simple_reg_other
* Info: Automatically instrumented module gproc_sup
* Info: Automatically instrumented module supervisor
* Info: Automatically instrumented module gen
* Info: Automatically instrumented module proc_lib
* Info: Automatically instrumented module erlang
* Info: Automatically instrumented module application
* Info: Automatically instrumented module application_controller
* Info: Automatically instrumented module lists
* Info: Automatically instrumented module gproc
* Info: Automatically instrumented module gproc_lib
* Info: Automatically instrumented module error_logger
* Info: Automatically instrumented module gen_event
* Info: Automatically instrumented module gproc_monitor
* Info: Automatically instrumented module gproc_bcast
* Info: Automatically instrumented module gproc_pool
* Tip: A process crashed with reason '{timeout, ...}'. This may happen
 ↪  when a call to a gen_server (or similar) does not receive a reply
 ↪  within some timeout (5000ms by default). You can use e.g.
 ↪  '--after_timeout 5000' to treat after timeouts that exceed some
 ↪  threshold (here 4999ms) as 'infinity'.
* Warning: Only assertion failures are considered crashes
 ↪  ('--assertions_only').
* Tip: An abnormal exit signal was sent to a process. This is probably
 ↪  the worst thing that can happen race-wise, as any other
 ↪  side-effecting operation races with the arrival of the signal. If
 ↪  the test produces too many interleavings consider refactoring your
 ↪  code.
* Warning: Concuerror does not fully support erlang:get_stacktrace/0,
 ↪  returning an empty list instead. If you need proper support,
 ↪  notify the developers to add this feature.
* Tip: A process crashed with reason 'shutdown'. This may happen when
 ↪  a supervisor is terminating its children. You can use
 ↪  '--treat_as_normal shutdown' if this is expected behaviour.
* Info: Automatically instrumented module sys
* Info: You can see pairs of racing instructions (in the report and
 ↪  '--graph') with '--show_races true'
Done at 20 Nov 2017 17:56:11 (Exit status: ok)
  Summary: 0 errors, 784/784 interleavings explored
```

Figure 6.3. Concuerror's command-line output for the gproc benchmark of Table 4.1, showing info, tips, and warnings about automatic instrumentation, use of options, suggested refactorings, and suppression of reporting of some detected errors.

7. Applications

In Chapter 6 we described aspects of Erlang and Concuerror. In this chapter we describe two applications of the research and the tool, beyond the verification and testing of Erlang programs[1].

7.1 Informal Specification of Erlang's Implementation

Even though there is no formal specification for Erlang, prior work exists on providing formal semantics for the language [40]. That work also includes ideas such as the separation of sending and delivering operations, but the overall focus is on the formal specification of a model of the language, rather than the description of the interactions between the operations that are available in the implementation. The semantics also intentionally diverges from the actual implementation (e.g., providing only monitors and not links), in the interest of providing a design for the language that is simpler and has fewer interfering operations.

Paper II serves as an informal specification of the implementation of the language, offering a detailed exposition of the interfering operations in the Erlang runtime, beyond basic message transmission, plus an organized library of test cases. The treatment includes all the intra-node operations covered in the earlier work discussed above. Although many, if not most, of the actual interferences described in Paper II are probably well-known to seasoned Erlang programmers, we are not aware of any other document or study that tried to characterize and categorize them in the level of detail that we have done in that paper.

7.2 Verification of a Protocol

The main topic of Paper III is the use of Concuerror to test or verify the correctness of a series of methods for repairing servers that use CORFU [33], a variant of the Chain Replication [43] protocol. In this section we summarize some key aspects of that work.

[1] The chapter contains text from Papers II and III edited to conform to the terminology used in this comprehensive summary.

7.2.1 Chain Replication

Chain Replication is a leader/follower protocol [43], in which a cluster of replica servers are arranged in an ordered list of *head*, *middle*, and *tail* servers. The head server is the leader; all other servers are followers. The protocol offers linearizable read and update operations.

Clients send update operations to the head server. If the head server rejects an update operation, it sends an error back to the client. If the operation is accepted, the head server does not reply, but sends state update requests down the chain. Follower servers (if any) record the update requests to their respective local data stores and then forward the requests downstream, in the same order that they received them. After an update has been stored by the last server in the chain, the tail server sends a successful acknowledgment (ack) to the client. Thus, for a single update to e.g., a chain of length three, four messages are required: client $\rightarrow$ head, head $\rightarrow$ middle, middle $\rightarrow$ tail, and tail $\rightarrow$ client.

Clients send read operations to the tail server, which is also the conceptual linearization point for all operations. If the tail server stores a value, then all other servers upstream in the chain must already store that value or a newer one.

7.2.2 Chain Repair

The Chain Replication paper [43] describes a method to shorten a chain if a server crashes or is otherwise stopped. It also discusses how to repair a chain by reintroducing a crashed server, but omits details that an implementor must be aware of in order to maintain Chain Replication's linearizable consistency guarantee. The following steps describe a naïve repair method:

1. 'Stop' all surviving servers in the chain, e.g., $[S^a_{head}, S^b_{tail}]$,
2. Copy update history from S^b_{tail} to the server under repair S^c_{repair}, then
3. 'Restart' all servers with a chain configuration of $[S^a_{head}, S^b_{middle}, S^c_{tail}]$.

This repair method is correct, but sacrifices cluster availability, as it requires taking the servers offline.

Online repair is desirable, but it should maintain Chain Replication's implementation of linearizable read queries sent to a single chain member. A suitable repair method has been proposed and used in HibariDB [20], a system implementing Chain Replication. The repair starts with a transition from chain $[S^a_{head}, S^b_{tail}] \Rightarrow [S^a_{head}, S^b_{tail}, S^c_{repair}]$, where S^c is the crashed server. Read queries ignore the server under repair; they are sent to the tail server as usual. Updates are sent to the head server and propagate down the entire chain; replies are sent by S^c_{repair}. While this intermediate chain configuration is in place, a separate process asynchronously copies missing data from S^b_{tail} to S^c_{repair}. When all missing history items have been copied to the server under repair, servers in the chain enter read-only mode. A flush command is sent by the head server, to force all pending writes down the chain to be finalized. When the flush command reaches the repaired server, all update log histories must be equal: $S^a_{head} = S^b_{tail} = S^c_{repair}$. After an acknowledgment of the flush command (sent from the repaired server) is received by the head server, the chain can transition to $[S^a_{head}, S^b_{middle}, S^c_{tail}]$, and read-only mode can be canceled.

7.2.3 Chain Replication in CORFU

A CORFU system [33] uses Chain Replication with three changes, related to what we described so far. First, CORFU servers do not communicate with each other, so they cannot implement the original Chain Replication protocol. Instead, the replication logic is moved to the clients. Thus, for a single update to a CORFU chain of e.g., length three, six messages are now required, in three pairs between each of client $\leftrightarrow$ head, client $\leftrightarrow$ middle, and client $\leftrightarrow$ tail.

The second change is that CORFU's servers implement write-once semantics. Clients cannot overwrite a previously written value.

Third, CORFU builds upon standard Chain Replication by assigning an epoch number to each chain configuration. All clients and servers are aware of the epoch number, and all client operations include the epoch number. If a client operation contains a different epoch number, the operation is rejected by the corresponding server. A server also temporarily stops service if it receives a newer epoch number from a client. When any participant detects a newer epoch, it can retrieve the new configuration from a dedicated server that is storing cluster layout configuration info.

7.2.4 Modeling Repair Methods for CORFU in Erlang

Paper III begins by describing an approach followed by a user of Concuerror to model, test and verify chain repair methods suitable for CORFU. As CORFU servers do not communicate directly with each other, the "read-only mode + sync flush down the chain" method cannot be directly applied, as there is no central coordinator like HibariDB's head server. Moreover, a straightforward adaptation of this repair method was also found to be insufficient, as a particular race condition (explained in detail in Paper III) could lead to linearizability violations.

In order to investigate other solutions, the user modeled a number of servers and clients of CORFU using Erlang. A high-level view of the modeled CORFU system is the following: A number of stable servers (one or two suffice for the properties we want to verify) undergo a chain repair procedure to have a single additional server added to their chain. Concurrently, two other clients will try to write two different values to the same key, while a third client will try to read the key twice. A coordinator process collects information from each client and the repair process after it has completed its execution and checks a number of assertions.

On this initial model, three repair methods were tested, differing in where the recovered server is placed in the chain: the head, the tail or an intermediate position. Concuerror was run in two modes:

1. Using exploration tree bounding (with a bound of at most 4) in order to detect bugs, and
2. Without bounding the exploration, i.e., using the tool for verification.

The results of this investigation were the following:

1. Concuerror detected bugs in the first and second methods fairly quickly.
2. In the first method, exploration tree bounding was crucial for finding a bug in a reasonable amount of time (57s instead of 144h).
3. After running for more than 750h, Concuerror could *neither* find bugs *nor* verify the third method.

7.2.5 Optimizing Concuerror and Refining the Model

Paper III continues by describing an optimization of the tool and two refinements of the model. The motivation behind these changes was to further reduce the number of schedulings explored by Concuerror and achieve full verification for the third repair method.

The optimization is an early version of the observers technique described in Paper IV: when determining which other messages are racing with a message's delivery, Concuerror was extended in order to take into account patterns of the corresponding `receive` statement. Since the model required a coordinator process, which expected `done` messages that all clients sent upon completion, the use of this optimization together with appropriate `receive` patterns in the coordinator was expected to significantly reduce the number of explored schedulings.

The two refinements were:

1. a change in the behaviour of the reader client, adding a condition to the execution of the second read operation, with the goal to reduce the read requests sent in non-interesting cases and
2. a simplification in the modeling of the layout server, using an ETS table instead of a process, in order to take advantage of the commutativity of read operations, which were the majority of requests towards that server.

7.2.6 Verifying a Repair Method for a CORFU Cluster

After applying the optimization and the two refinements, Concuerror managed to verify that the third repair method had no bugs in 48 hours, after exploring 3 931 413 schedulings. The effect of each change was not evaluated on its own on this method, since the required time would be significantly larger (e.g., when not using the observers optimization, the conservatively interfering `done` messages sent back to the coordinator would conceptually lead to the exploration of $4! = 24$ times as many schedulings). However, we evaluated the changes separately on the first (buggy) repair method without using a bound, and found that the optimization reduces the time required to find the first bug to 5h 30m and the refinement of the reader even more so: a bug is found in 6m 20s. When used together, these improvements lead to the discovery of a bug in just 19 seconds (only 212 schedulings are explored). The layout server refinement was not so effective on its own; it also slightly increased the number of schedulings when combined with the reader refinement. With all three changes, the schedulings were shorter (no back and forth communication with an extra server) and thus a bug was found slightly faster (in 18 instead of 19 seconds), even though slightly more schedulings (289 instead of 212) were explored. A full table of these results is given in Paper III.

Overall, Paper III shows that by using Concuerror on a fairly straightforward model written in Erlang, it is possible to find bugs in two repair methods for a CORFU system, some more quickly detectable after applying exploration tree bounding. By optimizing Concuerror and using two refinements of the model, it is also possible to verify the correctness of a third method in a reasonable amount of time.

8. Conclusion

Reduction in the number of explored schedulings makes stateless model checking more effective for finding bugs and enables verification of more concurrent programs.

In this dissertation, we began by describing a number of improvements to the original dynamic partial order reduction algorithm used in stateless model checking. Source sets and wakeup trees were introduced to describe the Source and Optimal DPOR algorithms. The algorithms were presented in a generic form, applicable to several concurrency models. Each of these algorithms can achieve exponential reduction in the number of explored schedulings over the original algorithm. Optimal DPOR was then extended to support observer operations achieving even further reduction. When stateless model checking is used for testing, bounding techniques can lead to faster bug detection. We described a simple bounding technique that can be combined with any DPOR algorithm, including Optimal DPOR, increasing its effectiveness in finding bugs.

Next, we presented Concuerror, a public, open-source implementation of all described improvements into a stateless model checking tool for Erlang programs. Concuerror combines the new techniques with a fine-grained characterization of the interferences between the higher-level operations of the Erlang language. This characterization, which also functions as an informal specification of the Erlang/OTP implementation, allows precise detection of races. The result is tool that is effective in testing and verifying concurrent Erlang programs. To show that, we presented how Concuerror was used to investigate, find bugs and verify repair techniques for CORFU, a variant of the Chain Replication protocol.

Based on these statements, we believe that this dissertation successfully supports the following:

Thesis:

Improvements in dynamic partial order reduction techniques can significantly increase the effectiveness of stateless model checking algorithms.

9. Directions for Future Work

The work presented in this dissertation can be extended in several promising directions.

Extending Optimal DPOR with observer operations was a first step towards a DPOR algorithm that can *focus* on specific operations. This improvement can be further refined to focus exploration around schedulings that involve particular operations in a program, e.g., assertions, possibly specified by the user. This can dramatically reduce the number of explored schedulings by filtering those that cannot lead to error discovery.

Source and Optimal DPOR were described as sequential algorithms, in the same spirit as the original DPOR algorithm. Indeed, one can also see that, at the time of publishing of this dissertation, Concuerror and Nidhugg explore one scheduling at a time. Parallelizing the algorithms is a natural next step to further increase the effectiveness of the tools by reducing their total execution time on multicore or distributed systems.

When it comes to Erlang, the investigation of interference between operations and events was a sufficient first step to show Concuerror's correctness and increase its effectiveness by limiting the pairs of operations it considers as interfering. However, Concuerror often needs to inspect all possible pairs of operations, increasing the complexity of its implementation of the algorithms (Nidhugg does not suffer from this problem). The conditions of interference of Erlang operations can be further specified, e.g., with an index of memory locations affected by each operation, and this specification can be used to reduce the complexity of Concuerror's interference detection.

Finally, having predictable execution time is a basic usability requirement for tools. Unfortunately, limited experimentation showed that providing accurate estimations for the total number schedulings and exploration time when using the new DPOR algorithms is difficult. Use of exploration tree bounding helps, leading to roughly an order of magnitude increase in the number of schedulings explored for each increment of the bound, but how close or far that number is from the total number of Mazurkiewicz traces is usually a puzzle. Giving accurate online estimates for total exploration time would be a significant step towards further increasing the usability of DPOR tools.

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. In *Proc. TACAS '15, 21th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proc. POPL '14, 41th ACM Symp. on Principles of Programming Languages*, pages 373–384. ACM, 2014.

[3] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Comparing source sets and persistent sets for partial order reduction. In *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pages 516–536. Springer, 2017.

[4] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *Journal of the ACM*, 64(4):25:1–25:49, August 2017.

[5] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In *Proc. CAV 2017, 29th Int. Conf. on Computer Aided Verification*, pages 526–543. Springer, 2017.

[6] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[7] Stavros Aronis, Scott Lystig Fritchie, and Konstantinos Sagonas. Testing and verifying chain repair methods for CORFU using stateless model checking. In *Proc. IFM 2017, 13th Int. Conf. on Integrated Formal Methods*, pages 227–242. Springer, 2017.

[8] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. Submitted for publication.

[9] Stavros Aronis and Konstantinos Sagonas. The shared-memory interferences of Erlang/OTP built-ins. In *Proc. Erlang 2017, 16th ACM SIGPLAN Int. Workshop on Erlang*, pages 43–54. ACM, 2017.

[10] Clara Benac Earle and Lars-Åke Fredlund. Recent improvements to the McErlang model checker. In *Proc. Erlang '09, 8th ACM SIGPLAN Int. Workshop on Erlang*, pages 93–100. ACM, 2009.

[11] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. ASPLOS XV, 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 167–178. ACM, 2010.

[12] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. Systematic testing for detecting concurrency errors in Erlang programs. In *Proc. ICST 2013, 6th IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 154–163. IEEE Computer Society, 2013.

[13] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proc. ICFP '09, 14th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 149–160. ACM, 2009.

[14] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logics specification: A practical approach. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 117–126. ACM, 1983.

[15] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, November 1999.

[16] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. In *Proc. OOPSLA '13, 2013 ACM SIGPLAN Int. Conf. on Object Oriented Programming Systems Languages & Applications*, pages 833–848. ACM, 2013.

[17] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Proc. POPL '11, 38th ACM Symp. on Principles of Programming Languages*, pages 411–422. ACM, 2011.

[18] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. POPL '05, 32th ACM Symp. on Principles of Programming Languages*, pages 110–121. ACM, 2005.

[19] Lars-Åke Fredlund and Hans Svensson. McErlang: A model checker for a distributed functional programming language. In *Proc. ICFP '07, 12th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 125–136. ACM, 2007.

[20] Scott Lystig Fritchie. Chain replication in theory and in practice. In *Proc. Erlang '10, 9th ACM SIGPLAN Int. Workshop on Erlang*, pages 33–44. ACM, 2010.

[21] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* PhD thesis, University of Liège, 1996. Also, volume 1032 of LNCS, Springer.

[22] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. POPL '97, 24th ACM Symp. on Principles of Programming Languages*, pages 174–186. ACM, 1997.

[23] Patrice Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, March 2005.

[24] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, November 1995.

[25] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. CAV 93, 5th Int. Conf. on Computer Aided Verification*, volume 697 of *LNCS*, pages 438–449. Springer, 1993.

[26] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. CAV 1991, Computer Aided Verification*, volume 575 of *LNCS*, pages 332–342. Springer, 1991.

[27] Alkis Gotovos. Dynamic systematic testing of concurrent Erlang programs. Master's thesis, School of Electrical and Computer Engineering, National Technical University of Athens, December 2011. `http://artemis.cslab.ntua.gr/Dienst/UI/1.0/Display/artemis.ntua.ece/DT2011-0081`.

[28] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. Test-driven development of concurrent programs using Concuerror. In *Proc. Erlang '11, 10th ACM SIGPLAN Int. Workshop on Erlang*, pages 51–61. ACM, 2011.

[29] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, July 1992.

[30] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2, POPL 2018, January 2018.

[31] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the Linux kernel's hierarchical read-copy-update (Tree RCU). In *Proc. SPIN 2017, 24th Int. SPIN Symposium on Model Checking of Software*. ACM, 2017.

[32] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[33] Dahlia Malkhi, Mahesh Balakrishnan, John D. Davis, Vijayan Prabhakaran, and Ted Wobber. From Paxos to CORFU: A flash-speed shared log. *SIGOPS Oper. Syst. Rev.*, 46(1):47–51, February 2012.

[34] Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 279–324. Springer, 1987.

[35] Madanlal Musuvathi and Shaz Qadeer. Partial-order reduction for context-bounded state exploration. Technical report, Microsoft Research, 2007.

[36] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerald Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. OSDI '08, 8th Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX, 2008.

[37] Doron Peled. All from one, one for all, on model-checking using representatives. In *Proc. CAV 93, 5th Int. Conf. on Computer Aided Verification*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.

[38] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Proc. TACAS '05, 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2005.

[39] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Programming 1982, Int. Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[40] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A unified semantics for future Erlang. In *Proc. Erlang '10, 9th ACM SIGPLAN Int. Workshop on Erlang*, pages 23–32. ACM, 2010.

[41] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing - Special Issue on PPOPP 2014*, 2(4):23:1–23:37, March 2016.

[42] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.

[43] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. OSDI '04, 6th Symposium on Operating Systems Design & Implementation*, pages 91–104. USENIX, 2004.