Improved Verification Techniques for Concurrent Systems

by

Viktor Toman

October, 2021

A thesis submitted to the
Graduate School
of the
Institute of Science and Technology Austria
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

IST AUSTRIA
Institute of Science and Technology

The thesis of Viktor Toman, titled *Improved Verification Techniques for Concurrent Systems*, is approved by:

Supervisor: Krishnendu Chatterjee, IST Austria, Klosterneuburg, Austria

Signature: ______________________________

Committee Member: Thomas Anton Henzinger, IST Austria, Klosterneuburg, Austria

Signature: ______________________________

Committee Member: Parosh Aziz Abdulla, Uppsala University, Uppsala, Sweden

Signature: ______________________________

Committee Member: Andreas Pavlogiannis, Aarhus University, Aarhus, Denmark

Signature: ______________________________

Defense Chair: Tamás Hausel, IST Austria, Klosterneuburg, Austria

Signature: ______________________________

Signed page is on file

IST Austria Thesis, ISSN: 2663-337X

I hereby declare that this thesis is my own work and that it does not contain other people's work without this being so stated; this thesis does not contain my previous work without this being stated, and the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

I certify that any republication of materials presented in this thesis has been approved by the relevant publishers and co-authors.

Signature: ______________________________

Viktor Toman
October, 2021

Signed page is on file

Abstract

The design and verification of concurrent systems remains an open challenge due to the non-determinism that arises from the inter-process communication. In particular, concurrent programs are notoriously difficult both to be written correctly and to be analyzed formally, as complex thread interaction has to be accounted for. The difficulties are further exacerbated when concurrent programs get executed on modern-day hardware, which contains various buffering and caching mechanisms for efficiency reasons. This causes further subtle non-determinism, which can often produce very unintuitive behavior of the concurrent programs.

Model checking is at the forefront of tackling the verification problem, where the task is to decide, given as input a concurrent system and a desired property, whether the system satisfies the property. The inherent state-space explosion problem in model checking of concurrent systems causes naive explicit methods not to scale, thus more inventive methods are required. One such method is *stateless model checking (SMC)*, which explores in memory-efficient manner the program executions rather than the states of the program. State-of-the-art SMC is typically coupled with *partial order reduction (POR)* techniques, which argue that certain executions provably produce identical system behavior, thus limiting the amount of executions one needs to explore in order to cover all possible behaviors. Another method to tackle the state-space explosion is *symbolic model checking*, where the considered techniques operate on a succinct implicit representation of the input system rather than explicitly accessing the system.

In this thesis we present new techniques for verification of concurrent systems. We present several novel POR methods for SMC of concurrent programs under various models of semantics, some of which account for write-buffering mechanisms. Additionally, we present novel algorithms for symbolic model checking of finite-state concurrent systems, where the desired property of the systems is to ensure a formally defined notion of fairness.

Acknowledgements

First and foremost, I want to thank Krish, the best advisor I could wish for. Early on, he saw the potential in me that I did not see myself, and he played the lead role in how I developed as a computer scientist. I am thankful for all those meetings where just a few minutes of his insight had my head exploding. The level of discussion that at first seemed impossible to even follow, really did become a (mostly) comfortable routine by the end of my PhD. Thank you for everything, Krish.

My closest collaborator, and the de facto co-advisor in the latter part of my PhD, is Andreas. My big thanks goes just as well to him, this thesis would not have been possible without him. I admire the clarity and calmness with which he could discuss and explain vaguely specified ideas and problems. At the same time, whenever I was describing my fuzzy thoughts and feeling that I am explaining myself so poorly, Andreas still somehow always understood and immediately started refining the ideas. It was such a pleasure to work with him.

I further want to thank all the other IST scientists with which I discussed my research ideas. This is mainly (though not only) the groups of Krish, Tom and Christoph. I specifically want to thank Christoph for our collaboration early on, this played a key role in me obtaining my Google internships. Christoph is also (unknowingly) responsible for wiping out my imposter syndrome, which happened the day he praised me at the end of our rotation project.

I also thank the other PhD collaborators that I have not mentioned so far. My thanks go to Jan, Tom, Pranav, Veronika, Simin and Monika, for the collaboration on strategies and fairness. Further I thank the students that I could be an advisor of during their internships and rotations – Tushar, Buj, Shreya, Pratyush and Konstantin. I really enjoyed working with every one of you. Additionally, I want to thank the awesome IST staff, they made sure that my scientific endeavours were always free of administrative stress.

While my work done at Google is not a part of this thesis, the experience I gained during my internships helped me greatly to progress in my PhD. I thank all the great Google folks I worked with. In particular, I want to thank Christian and Sarah so, so much, for believing in me and giving me the first (i.e., the most important) opportunity. I feel incredibly lucky that every single "boss" I have had so far (including already my undergraduate advisors) was such an enjoyable person to work with. I hope that this (very pleasant) trend shall continue!

I am grateful for the grants that I was a part of during my PhD, namely, the EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant No. 665385, the Austrian Science Fund (FWF) NFN Grant S11402-N23 (RiSE/SHiNE), the Vienna Science and Technology Fund (WWTF) Project ICT15-003, and the ERC CoG 863818.

I thank all the amazing friends I have made at IST and in Vienna. I would rather not name all of you, lest I forget someone, you know who you are. Thank you for making these five years unforgettable and full of joy. Finally, I thank my family for their constant love and support, not just during this PhD, but during my whole life.

About the Author

Viktor Toman obtained a Bc and Mgr in computer science at Masaryk University, Czech Republic, before joining the computer science PhD program at IST Austria in September 2016. His PhD program is in the field of formal methods, his advisor is Krishnendu Chatterjee. His main PhD research topic is verification of concurrent systems, where his works have been published in four conference papers, two in CAV and two in OOPSLA. He further worked on efficient representation of strategies, producing two more conference papers, one in TACAS and one in QEST. Additionally, during his PhD program Viktor completed three summer internships at Google Research, where in 2018 and 2019 he worked on machine learning in theorem proving, and in 2020 he worked on machine learning in music.

List of Collaborators and Publications

Below publications were obtained during the PhD program and are presented in this thesis.

- Krishnendu Chatterjee, Monika Henzinger, Veronika Loitzenbauer, Simin Oraee, and Viktor Toman. Symbolic algorithms for graphs and Markov decision processes with fairness objectives. In *Computer Aided Verification (CAV)*, 2018
- Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. Value-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019
- Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless model checking under a reads-value-from equivalence. In *Computer Aided Verification (CAV)*, 2021
- Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. The reads-from equivalence for the TSO and PSO memory models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), 2021

Below further publications were also obtained during the PhD program and they are not presented in this thesis.

- Tomáš Brázdil, Krishnendu Chatterjee, Jan Křetínský, and Viktor Toman. Strategy representation by decision trees in reactive synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018
- Pranav Ashok, Tomáš Brázdil, Krishnendu Chatterjee, Jan Křetínský, Christoph H. Lampert, and Viktor Toman. Strategy representation by decision trees with linear classifiers. In *Quantitative Evaluation of Systems (QEST)*, 2019

Each publication has all the collaborators listed as co-authors. For each publication, the authors are listed in alphabetical order, as is customary in theoretical computer science.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

MDP Markov decision process. xii–xiv, 1, 3, 5–7, 109–114, 118, 122–124, 126–137, 139, 140

MEC maximal end-component. xii, xiii, 5–7, 109, 110, 112–114, 122–130, 132–134, 136, 140

POR partial order reduction. vii, xi, 2–4, 6, 7, 9, 10, 12–19, 139, 140

PSO partial store order. xii, xiii, 2, 5–7, 9–16, 71–100, 102–106, 139, 140

RF reads-from. 4, 7, 12, 13, 15, 16, 47, 71, 73, 95, 102–105, 140

RVF reads-value-from. xii, 6, 7, 47–52, 55, 57, 61, 63–66, 68, 139

SC sequential consistency. xi, xiii, 2, 4–7, 9–12, 14–18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 47, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 71–73, 91, 93–95, 97, 102–106, 139, 140

SCC strongly connected component. 5, 6, 109, 110, 112–132, 134, 136, 140

SMC stateless model checking. vii, xii, xiii, 2–7, 9–11, 15–19, 29, 39, 47, 48, 50, 52, 55–57, 64, 65, 71, 73, 93, 95–98, 102, 104–106, 139, 140

SMT satisfiability modulo theories. 4, 5, 48

TSO total store order. xii, xiii, 2, 5–7, 9–16, 71–106, 139, 140

VC value-centric. xii, 6, 17–22, 29, 30, 32, 35, 37–39, 43, 139

CHAPTER 1

Introduction

Model checking. The fundamental problem of *model checking* asks, given a model and a specification, whether the model satisfies the specification [CGP99a]. Model-checking algorithms are rigorous methods for formal verification of diverse software and hardware systems. To reason about an input system, a model checker aims to cover all possible system behaviors that can arise due to non-determinism, randomness, or outside environment. There are numerous successful and well-established model-checking tools, including VeriSoft [God97, God05], CHESS [MM07], BLAST [BHJM07], SLAM [BR02], SPIN [Hol97], PRISM [KNP11], and DiVinE [BBC$^+$06].

Concurrent finite-state systems. Two fundamental types of transition structures to model finite-state systems are *graphs* and *Markov decision processes (MDPs)*. A Graph can efficiently capture the non-determinism of the underlying system, whereas an MDP captures both the non-determinism and the randomness present in its system. Thus graphs and MDPs can precisely model various classes of software and hardware systems. In particular, *concurrent systems* are systems that inherently exhibit non-determinism due to the interaction between different modules (often called processes, or threads) present in one system, and formal analysis of concurrent systems has been a subject of extensive research [Pet62, CL73, Lip75, CES86, LR09, FM09, FK12]. Concurrent finite-state systems are naturally represented as graphs, where vertices capture possible states of the system (typically given as variable valuations of all the modules), while edges (or transitions) denote atomic actions performed by some module. However, when graphs and MDPs are used to model realistic systems (including concurrent systems), they suffer from the well-known *state-space explosion* problem, which is a combinatorial blow-up on the number of states (i.e., vertices) due to the inter-play between modules and variables of the modelled system. Thus graphs and MDPs are central models used in model checking, with the challenge that for realistic systems they are often way too big for naive explicit exploration [CGP99a, BK08].

Concurrent programs. Model checking of concurrent programs is one of the key challenges in formal methods. Inter-process communication in concurrent programs incurs non-determinism in the program behavior, which is resolved by a scheduler. However, as the programmer has no control over the scheduler, program correctness has to be guaranteed under all possible schedulers, i.e., the scheduler is adversarial to the program and can generate erroneous behavior if one can arise out of scheduling decisions. On the other hand, during program testing, the adversarial nature of the scheduler is to hide erroneous runs, making bugs extremely difficult to reproduce by testing alone (called Heisenbugs [MQB$^+$08]). These facts make it very hard for concurrent programs both to be written correctly, and to be analyzed formally, as both the programmer and the model checker need to account for all possible communication patterns among the threads. As a result, systematic state-space exploration by model checking is an important approach for verification of concurrent programs [CGP99a, AQR$^+$04, God05, MQ07, AKT13, BBH$^+$13].

Semantics of concurrent programs. Traditional verification has focused on concurrent programs adhering to the semantics of sequential consistency (SC) [Lam79]. Under SC, the read and write operations of a program are considered atomic. This typically does not precisely reflect the program behavior when executed on real-world hardware, which makes use of various buffering and caching mechanisms. Descriptions of semantics that model such mechanisms are broadly called *relaxed memory models*, and programs operating under relaxed-memory semantics can exhibit additional behavior as compared to SC. This makes it exceptionally hard to reason about correctness since, besides scheduling subtleties, the formal reasoning needs to additionally account for the non-determinism induced by the buffering and caching mechanisms. Two of the most standard operational relaxed memory models in the literature are *total store order (TSO)* and *partial store order (PSO)* [SI94, AG96, OSS09, SSO$^+$10, Alg10, ACU17].

On the operational level, both TSO and PSO introduce subtle mechanisms via which write operations become visible to the shared memory and thus to the whole system. Under TSO, every thread is equipped with its own buffer. Then, every write to a shared variable is pushed into the buffer, and thus initially remains hidden from the other threads. The buffer is flushed non-deterministically to the shared memory, at which point the writes become visible to the other threads. The semantics under PSO are even more involved, as now every thread has one buffer *per shared variable*, and non-determinism now governs not only when a thread flushes its buffers, but also which buffers are flushed. Consequently, the great challenge in verification under TSO and PSO is to systematically, yet efficiently, explore all such extra behaviors of the system, i.e., account for the additional non-determinism that comes from the buffers.

Stateless model checking. Model checkers typically store a large number of global states, and hence due to state-space explosion they struggle to handle realistic concurrent programs. The standard solution that is adopted to battle this problem on concurrent programs is *stateless model checking (SMC)* [God96]. SMC methods typically explore traces rather than states of the analyzed program, and only have to store a small number of traces at any time. In such techniques, model checking is achieved by a controllable scheduler, which drives the program execution based on the desired interaction between the threads. The depth-first nature of the search enables it to be both systematic and memory-efficient. SMC techniques have been employed successfully in numerous well-known model checkers [God97, God05, MM07, AAA$^+$15, Hua15, DL15, KLSV17].

Partial order reduction. While SMC deals with the state-space issue, one key challenge that remains is to efficiently explore the exponential number of interleavings, which results from the non-deterministic inter-process communication. There exist various techniques for reducing the number of explored interleavings, such as depth bounding and context bounding [MQ07, LR09]. Notably though, one of the most well-studied techniques is *partial order reduction (POR)* [Pel93, God96, CGMP99]. The main principle of POR is that two interleavings can be regarded as equal if they agree on the order of conflicting (i.e., dependent) events. In other words, POR considers certain pairs of program executions (i.e., traces) to be equivalent, and the theoretical foundation of POR is an equivalence relation induced on the trace space, known as the *happens-before* (or the *Mazurkiewicz*) equivalence [Maz87]. POR algorithms explore at least one trace from each equivalence class and thus guarantee a complete coverage of all behaviors that can occur in any interleaving, while exploring only a subset of the trace space. For the most interesting properties that arise in formal verification, such as safety, race freedom, absence of global deadlocks, and absence of assertion violations, POR-based algorithms make sound reports of correctness [God96].

An on-the-fly version of POR is called *dynamic partial order reduction* (often abbreviated as DPOR) [FG05]. Dynamic POR records conflicts that actually occur during the execution of traces, and thus is able to infer independence more frequently than static POR, which typically relies on over-approximations of conflicting events. Similar to static POR, dynamic-POR-based algorithms guarantee the exploration of at least one trace in each class of the happens-before partitioning. Further, the great advantage of dynamic POR is that it handles indirect memory accesses precisely without introducing spurious interleavings. As a result, state-of-the-art SMC methods to tackle verification of concurrent programs are all coupled with dynamic POR techniques.

Symbolic model checking. An approach to tackle the state-space explosion which is orthogonal to SMC is called *symbolic model checking*. Symbolic model-checking algorithms operate on a succinct implicit representation of the input system rather than explicitly accessing the system, thus sidestepping the state-space problem.

In contrast to traditional *explicit* algorithms that operate on the explicit representation of the system (i.e., a graph or an MDP), *symbolic* algorithms only use a set of predefined operations to manipulate the system and they never explicitly access the system [CGP99b]. As a result, the explicit representation of the system does not have to be constructed and stored at all, and only the implicit representation (which is typically exponentially smaller) is constructed and manipulated. Thus symbolic algorithms are scalable, whereas explicit algorithms do not scale as it is computationally too expensive to even explicitly construct the system.

Symbolic algorithms for the analysis of graphs and MDPs are at the heart of many state-of-the-art model-checking tools, such as SPIN [Hol97], NuSMV [CCGR00] for graphs, and PRISM [KNP11], LiQuor [CB06], STORM [DJKV17] for MDPs.

1.1 Previous Literature – POR

To deal with the exponential number of interleavings faced by the early model checking [God97], several reduction techniques have been proposed, such as depth bounding, context bounding [Pel93, MQ07, LR09, CKK$^+$17, BMTZ21], unfoldings [McM95, KSH12, RSSK15], and importantly also POR. As many interleavings (i.e., traces) induce the same program behavior, POR partitions the trace space into equivalence classes and attempts to sample a few representative traces from each class. The initial POR techniques deem interleavings as equivalent based on the way that conflicting memory accesses are ordered, also known as the happens-before (or the Mazurkiewicz) equivalence [Maz87]. Originally, several static POR methods, based on persistent-set [Val91, God96, CGMP99] and sleep-set techniques [God97], have been studied. The first dynamic POR technique was proposed in [FG05], where the Mazurkiewicz equivalence is constructed dynamically, as all memory accesses are known, and thus this approach does not suffer from the imprecision of earlier approaches based on static information. After [FG05], several variants and improvements have been proposed [SA06, SA07, WYKG08, KWG09, LKMA10, TKL$^+$12, SKH12]. In particular, in [AAJS14], source sets and wakeup trees were developed to make the Mazurkiewicz dynamic-POR technique exploration-optimal, i.e., each class of the Mazurkiewicz partitioning is explored exactly once. The underlying computational problems of [AAJS14] were further studied in [NRS$^+$18]. Notably, while in this thesis we focus on concurrent programs with shared memory, techniques for POR have also been considered for message-passing concurrency [KP92, GHP95, God96].

Beyond the Mazurkiewicz equivalence. Consider an SMC algorithm that utilizes a dynamic POR technique to explore classes of some trace-space partitioning. The performance of the SMC algorithm is generally a product of two factors: (a) the size of the underlying partitioning that is explored, and (b) the total time spent in exploring each class of the partitioning. Typically, the task of visiting a partitioning class requires solving a consistency-checking problem, where the algorithm checks whether a semantic abstraction, used to represent some partitioning class, has a consistent concrete interleaving that witnesses the class. For this reason, the search for efficient SMC is reduced to the search of coarse partitionings for which the consistency problem is tractable. The idea of searching for coarse trace partitionings was initially proposed in [GP93], and it has become a very active research direction in recent years in the context of SMC coupled with dynamic POR.

In [AJLS18], the Mazurkiewicz partitioning was reduced by ignoring the order of conflicting write events that are not observed, while retaining polynomial-time consistency checking. The work of [AAdlB$^+$17] utilizes context-sensitivity to achieve partitioning reduction as compared to Mazurkiewicz. The approach in [Hua15] considers a very coarse partitioning based on maximal causal models, where the consistency checking is solved using satisfiability modulo theories (SMT) solvers, and the work was later improved with static analysis techniques [HH17]. Similarly to [Hua15] using SMT solvers, the SMC work of [DL15] utilizes propositional-logic solvers (commonly known as SAT solvers) for consistency checking. Orthogonally to other works, [KRV19a] proposes an efficient method of handling locks in SMC, thereby achieving coarse partitionings on concurrent programs utilizing lock accesses.

A new direction of SMC has recently been developed where the techniques consider the *reads-from (RF)* equivalence to partition the trace space. The key principle is to classify traces as equivalent based on whether the read accesses observe the same write accesses. The idea was initially explored in [CCP$^+$17] for a subset of concurrent programs with acyclic communication topologies. The work of [CCP$^+$17] uses polynomial time for consistency checking, and it does not handle concurrent programs with arbitrary communication topologies[1], since the corresponding consistency-checking problem in the general setting is NP-hard [GK97], while it was recently shown to be even $\mathrm{W}[1]$-hard [MPV20]. Nevertheless, efficient SMC based on the RF equivalence handling concurrent programs with all communication topologies was recently developed in [AAJ$^+$19, KRV19b]. The work of [AAJ$^+$19] sidesteps the consistency-checking hardness by proposing a consistency-checking approach that is polynomial-time when the number of threads is bounded by a constant, and further the approach is very efficient in practice. Similarly, the work of [KRV19b] utilizes a simple consistency-checking technique which, while exponential-time in the worst case, is also efficient in practice.

Dynamic POR for relaxed memory models. While the above-mentioned SMC techniques mostly consider concurrent programs operating under the SC memory model, it is very important to also study programs under more complex memory models that more accurately reflect the

[1]More precisely, [CCP$^+$17] handles programs with arbitrary topologies by considering an equivalence that is a mixture of RF and Mazurkiewicz, and hence admits polynomial-time consistency checking.

behavior made possible by present-day hardware. Hence the SMC literature has taken up the challenge of model checking concurrent programs under relaxed memory. Extensions to SMC for TSO and PSO have been considered by [ZKW15] using shadow threads to model memory buffers, as well as by [AAA$^+$15] using chronological traces to represent the Shasha–Snir notion of trace under relaxed memory [SS88]. Chronological/Shasha–Snir traces are the generalization of Mazurkiewicz traces to TSO and PSO. The work of [DL15] utilizing SAT solvers also handles TSO in addition to SC, and the work of [HH16] extends the SMT-using maximal-causal-model approach of [Hua15] to handle TSO and PSO. Further extensions have also been made to other memory models, namely by [AAJN18] for the release-acquire fragment of C++11, [KLSV17, KRV19b] for the RC11 model [LVK$^+$17], and [KV20] for the IMM model [PLV19]. Notably, the approaches in [KRV19b] and [KV20] introduce a general SMC interface that can potentially handle a large variety of relaxed memory models, provided that a consistency-checking technique is externally supplied for the corresponding desired memory model.

1.2 Previous Literature – Fairness

One of the very basic specifications (i.e., desired properties) that arise in verification of reactive systems is the strong fairness (also known as the *Streett*) objective [CGP99a, MP96, AH04]. Given different types of requests and corresponding grants, the Streett objective requires that for each type, if the request event happens infinitely often, then the corresponding grant event must also happen infinitely often. After safety, reachability, and liveness, the Streett condition is one of the most standard properties that arise in the analysis of reactive systems. Moreover, all ω-regular objectives can be described by Streett objectives, e.g., linear-temporal-logic formulas and non-deterministic ω-automata can be translated to deterministic Streett automata [Saf88], and efficient translation has been an active research area [CGK13, EK14, KK14]. Thus Streett objectives are a canonical class of verification objectives.

Problems. The algorithmic model-checking problem of graphs and MDPs with Streett objectives is a core problem in verification. For graphs the requirement is that there is a trajectory (i.e., an infinite path) that belongs to the set of paths described by the Streett objective. For MDPs the satisfaction requires that there is a policy to resolve the non-determinism such that the Streett objective is ensured almost-surely (i.e., with probability 1). The basic approach to tackle model checking of graphs with Streett objectives utilizes repeated computation of the strongly connected component (SCC) decomposition of graphs. Similarly, for MDPs the core subproblem of Streett model checking is the maximal end-component (MEC) decomposition of MDPs. Thus efficient solutions to the SCC and MEC computation are necessary for efficient model-checking algorithms with Streett objectives.

Explicit algorithms. The traditional model-checking studies consider *explicit* algorithms that operate on the explicit representation of the system [CE81]. Several explicit algorithms have been introduced for graphs with Streett objectives [HT96, CHL15], and for MDPs with Streett objectives [CDHL16]. Further, various explicit algorithms for MEC decomposition of MDPs have been recently proposed [CH11, CH12, CH14]. For SCC decomposition, classical linear-time explicit algorithms are well-known [Tar72, Dij76, Sha81].

Symbolic algorithms. In contrast to the explicit algorithms, *implicit* (or *symbolic*) algorithms only use a set of predefined operations to access the system [CGP99b]. Symbolic algorithms for MDP with liveness (i.e., Büchi) objectives have been proposed in [CHJS13]. Further, a symbolic algorithm for SCC decomposition that requires $O(n)$ symbolic steps, for graphs with n vertices, was proposed in [GPP08], and later shown to be optimal in [CDHL18]. In contrast, the current best-known symbolic algorithm for MEC decomposition requires $O(n^2)$ symbolic steps.

Basic symbolic model-checking algorithms for Streett objectives can be obtained as follows. For an input graph, first the SCC decomposition is computed, and then given an SCC, (a) if for every request type that is present in the SCC the corresponding grant type is also present in the SCC, then the SCC is identified as "good", (b) else vertices of each request type that has no corresponding grant type in the SCC are removed, and the algorithm recursively proceeds on the remaining graph. Finally, reachability to good SCCs is computed. For MDPs a similar approach is followed, but the SCC computation has to be replaced by MEC computation. Utilizing the above-mentioned best-known algorithms for SCC and MEC computation, the bounds for basic symbolic Streett model-checking algorithms are $O(n \cdot \min(n, k))$ for graphs and $O(n^2 \cdot \min(n, k))$ for MDPs, where k is the number of types of request-grant pairs.

1.3 Contributions

In this thesis we present our novel contributions towards verification of concurrent systems. In particular, we present several novel works on verification of concurrent programs via POR methods, and a novel work on symbolic algorithms for model checking of graphs and MDPs. In POR, we present new dynamic POR methods, two for the memory model of SC, and one for the memory models of TSO and PSO. Each our POR method is underpinned by a newly proposed equivalence, which aims to be *coarser* than the equivalences of other state-of-the-art POR methods, i.e., we aim to (soundly) consider more program traces equivalent, while still being able to efficiently explore the resulting trace space via SMC. Additionally, in symbolic model checking, we present new algorithms for fairness objectives that have superior complexity bounds as compared to the previously known algorithms.

The rest of this thesis is organized as follows.

Chapter 2: We introduce the basic concepts and definitions that we utilize in the later chapters to describe our POR works.

Chapter 3: We present a novel *value-centric (VC)* trace equivalence for concurrent programs under the SC memory model. The VC equivalence is coarser than the baseline Mazurkiewicz equivalence, and crucially at the same time, the consistency checking problem corresponding to the VC equivalence admits a polynomial solution, and we present such a solution. Furthermore, we develop an SMC approach coupled with dynamic POR that utilizes the VC equivalence to verify concurrent programs operating under SC. We implement the techniques and experimentally evaluate the SMC approach with other state-of-the-art SMC methods, demonstrating the advantages of the VC equivalence in SMC as compared to the other methods. This chapter provides the full technical report of the work [CPT19].

Chapter 4: We consider the SC memory model, and we generalize the VC equivalence of Chapter 3 into a new trace equivalence, called the *reads-value-from (RVF)* equivalence. The RVF equivalence is coarser than other state-of-the-art SMC equivalences, including the VC equivalence, and the corresponding consistency checking problem for RVF is NP-hard. Nevertheless, we propose a new solution for the RVF consistency checking, which is polynomial when the number of threads and variables is bounded by a constant, and we further present practical heuristics that make our consistency-checking approach efficient in practice. Moreover, we introduce an SMC technique coupled with dynamic POR underpinned by the RVF equivalence. We implement all the methods, and we experimentally evaluate the RVF-based SMC approach with other state-of-the-art SMC approaches, highlighting the desirable properties of utilizing RVF in SMC. This chapter provides the full technical report of the work [ACP$^+$21].

Chapter 5: We utilize the well-established *reads-from (RF)* equivalence, and we develop SMC with dynamic POR for programs under the relaxed memory models TSO and PSO. In particular, we consider the recent reads-from SMC approach on SC [AAJ$^+$19], and we extend it to TSO and PSO. In our extension, for the consistency checking problem, the core algorithmic subproblem of SMC, we present novel algorithms for TSO and PSO that improve upon naive extensions of the solutions for SC presented in [AAJ$^+$19]. Specifically, given an execution of n events, k threads and d variables, for TSO we improve the bound from $O(n^{2\cdot k})$ to $O(k \cdot n^{k+1})$, and for PSO we improve the bound from $O(n^{k\cdot(d+1)})$ to $O(k \cdot n^{k+1} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d}))$. We implement all the algorithms and experimentally evaluate both the consistency algorithms and the SMC approach, both in TSO and in PSO, against state-of-the-art alternatives. This chapter provides the full technical report of the work [BCG$^+$21].

Chapter 6: We present new symbolic algorithms for model checking of graphs and MDPs with fairness objectives, and a new symbolic algorithm for maximal end-component (MEC) decomposition of MDPs. Our symbolic algorithms improve in complexity upon state of the art, and thus we establish new complexity bounds for the respective algorithmic problems. As an example, for the fairness model checking problem of MDPs, the best previously-known solution has the bound on the number of symbolic steps $O(n^2 \cdot \min(n, k))$, where n and k is the number of vertices and fairness pairs, respectively. However, our new symbolic algorithm has the symbolic-steps bound of $O(n\sqrt{m \log n})$ (where m is the number of edges), which yields a significant improvement for MDPs with few edges and/or many fairness pairs. We implement the algorithms and evaluate them on large-scale benchmarks, showcasing the improvement of our algorithms in terms of both the runtime and the number of performed symbolic steps. This chapter provides the full technical report of the work [CHL$^+$18].

Chapter 7: We summarize the contributions of this thesis and we conclude by suggesting directions for future research of concurrent-systems verification.

CHAPTER 2

Preliminaries for the POR Techniques

In this chapter we define the main concepts that are used in the presentation of our partial order reduction (POR) works in Chapter 3, Chapter 4 and Chapter 5.

2.1 General Notation.

Given a natural number $i \geq 1$, we let $[i]$ be the set $\{1, 2, \ldots, i\}$. Given a map $f \colon X \to Y$, we let $\mathsf{dom}(f) = X$ denote the domain of f. We represent maps f as sets of tuples $\{(x, f(x))\}_x$. Given two maps f_1, f_2 over the same domain X, we write $f_1 = f_2$ if for every $x \in X$ we have $f_1(x) = f_2(x)$. Given a set $X' \subset X$, we denote by $f|X'$ the restriction of f to X'. A binary relation $\sim$ on a set X is an *equivalence* iff $\sim$ is reflexive, symmetric and transitive. Given an equivalence $\sim$ on a set X, we denote by $X/\sim$ the *quotient set* of X under $\sim$, i.e., the set of all equivalence classes of X under $\sim$.

2.2 Concurrent Model

Here we describe the computational model of concurrent programs with shared memory. We first describe concurrent programs under the sequential consistency (SC) memory model, following a standard exposition of stateless model checking (SMC), similarly to [FG05, AAJS14, CCP+17, AAJ+19, KRV19b, CPT19]. Afterwards we describe concurrent programs under the relaxed memory models total store order (TSO) and partial store order (PSO), following the exposition similar to [AAA+15, HH16].

2.2.1 Concurrent Program under the SC Memory Model

We consider a concurrent program $\mathscr{P} = \{\mathsf{thr}_i\}_{i=1}^k$ of k deterministic threads. The threads communicate over a shared memory $\mathcal{G}$ of global variables with a finite value domain $\mathcal{D}$. Threads execute *events* of the following types.

1. A *write event* w writes a value $v \in \mathcal{D}$ to a global variable $x \in \mathcal{G}$.

2. A *read event* r reads the value $v \in \mathcal{D}$ of a global variable $x \in \mathcal{G}$.

Additionally, threads can execute local events which do not access global variables and thus are not modeled explicitly.

Semantics under SC. The semantics of $\mathscr{P}$ are defined by means of a transition system over a state space of global states. A global state consists of (i) a memory function that maps every global variable to a value, and (ii) a local state for each thread, which contains the values of the local variables and the program counter of the thread. We consider the standard setting of sequential consistency, and refer to [FG05] for formal details. As usual in SMC works, $\mathscr{P}$ is execution-bounded, which means that the state space is finite and acyclic.

2.2.2 Concurrent Program under the TSO and PSO Memory Models

Similarly to the SC case, we consider a concurrent program $\mathscr{P} = \{\mathsf{thr}_i\}_{i=1}^k$ of k threads, communicating over a shared memory $\mathcal{G}$ of global variables with a finite value domain $\mathcal{D}$. In TSO, each thread additionally owns a *store buffer*, which is a first-in-first-out queue for storing updates of variables to the shared memory. In PSO, each thread is equipped with a store buffer for each global variable, rather than a single buffer for all global variables.

Threads execute *events* of the following types.

1. A *buffer-write event* wB enqueues into the local store buffer an update that wants to write a value $v \in \mathcal{D}$ to a global variable $x \in \mathcal{G}$.
2. A *read event* r reads the value $v \in \mathcal{D}$ of a global variable $x \in \mathcal{G}$. The value v is the value of the most recent local buffer-write event, if one still exists in the buffer, otherwise v is the value of x in the shared memory.

Additionally, whenever a store buffer of some thread is nonempty, the respective thread can execute the following.

3. A *memory-write event* wM that dequeues the oldest update from the local buffer and performs the corresponding write-update on the shared memory (in PSO any nonempty store buffer of the thread can deque its oldest update).

Threads can also flush their local buffers into the memory using fences.

4. A *fence event* fnc blocks the corresponding thread until its store buffer is empty (resp., in PSO, until all buffers of the thread are empty).

Finally, threads can execute local events that are not modeled explicitly, as usual. We refer to all non-memory-write events as *thread events*. Following the typical setting of SMC [FG05, AAJS14, AAA+15, CCP+17], each thread of the program $\mathscr{P}$ is deterministic, and further $\mathscr{P}$ is bounded, meaning that all executions of $\mathscr{P}$ are finite and the number of events of $\mathscr{P}$'s longest execution is a parameter of the input.

Semantics under TSO and PSO. Similarly to the SC case, the semantics of $\mathscr{P}$ under TSO resp. PSO are defined by means of a transition system over a state space of global states. However, here a global state consists of (i) a memory function that maps every global variable to a value, (ii) a local state for each thread, which contains the values of the local variables of the thread, and (iii) a local state for each store buffer, which captures the contents of the queue. We consider the standard setting with the TSO/PSO memory model, and refer to [AAA+15] for formal details. Analogously to the SC case, the state space of $\mathscr{P}$ is finite and acyclic, as is usual in SMC.

2.2.3 Definitions regarding Concurrent Programs

Here we define concepts around concurrent programs, which we operate with when presenting our methods.

Given an event e, we denote by $\mathsf{thr}(e)$ its thread and by $\mathsf{var}(e)$ its global variable. We denote by $\mathcal{E}$ the set of all events, by $\mathcal{R}$ the set of all read events, and by $\mathcal{W}$ the set of all write events. In TSO/PSO, given a buffer-write event $wB \in \mathcal{W}^B$ and its corresponding memory-write $wM \in \mathcal{W}^M$, we let $\boldsymbol{w} = (wB, wM)$ be the two-phase write event, we denote $\mathsf{thr}(\boldsymbol{w}) = \mathsf{thr}(wB) = \mathsf{thr}(wM)$ and $\mathsf{var}(\boldsymbol{w}) = \mathsf{var}(wB) = \mathsf{var}(wM)$, and $\mathcal{W}$ then represents the set of all such two-phase write events. Further, we denote by $\mathcal{W}^B$ the set of buffer-write events, by $\mathcal{W}^M$ the set of memory-write events, and by $\mathcal{F}$ the set of fence events. Given two events $e_1, e_2 \in \mathcal{E}$, we say that they *conflict*, denoted $e_1 \bowtie e_2$, if they access the same global variable and at least one of them is a write event in SC (resp., a memory-write event in TSO/PSO).

Event sets. Given a set of events $X \subseteq \mathcal{E}$, we write $\mathcal{R}(X) = X \cap \mathcal{R}$ for the set of read events of X, and $\mathcal{W}(X) = X \cap \mathcal{W}$ for the set of write events of X. In TSO/PSO we similarly write $\mathcal{W}^B(X) = X \cap \mathcal{W}^B$ and $\mathcal{W}^M(X) = X \cap \mathcal{W}^M$ for the buffer-write and memory-write events of X, respectively, and here $\mathcal{W}(X) = (X \times X) \cap \mathcal{W}$ represents the set of two-phase write events in X. In TSO/PSO we only consider event sets X where $wB \in X$ iff $wM \in X$ for each $(wB, wM) \in \mathcal{W}$. We also denote by $\mathcal{L}(X) = X \setminus \mathcal{W}^M(X)$ the thread events (i.e., the non-memory-write events) of X. Finally, in all memory models, given a set of events $X \subseteq \mathcal{E}$ and a thread thr, we denote by X_{thr} and $X_{\neq \mathsf{thr}}$ the events of thr, and the events of all other threads in X, respectively.

Sequences and Traces. Given a sequence of events $\tau = e_1, \dots, e_j$, we denote by $\mathcal{E}(\tau)$ the set of events that appear in τ. We denote $\mathcal{R}(\tau) = \mathcal{R}(\mathcal{E}(\tau))$ and $\mathcal{W}(\tau) = \mathcal{W}(\mathcal{E}(\tau))$, in TSO/PSO we further denote $\mathcal{W}^B(\tau) = \mathcal{W}^B(\mathcal{E}(\tau))$ and $\mathcal{W}^M(\tau) = \mathcal{W}^M(\mathcal{E}(\tau))$. Finally we denote by ϵ an empty sequence.

Given a sequence τ and two events $e_1, e_2 \in \mathcal{E}(\tau)$, we write $e_1 <_\tau e_2$ when e_1 appears before e_2 in τ, and $e_1 \leq_\tau e_2$ to denote that $e_1 <_\tau e_2$ or $e_1 = e_2$. Given a sequence τ and a set of events A, we denote by $\tau | A$ the *projection* of τ on A, which is the unique subsequence of τ that contains all events of $A \cap \mathcal{E}(\tau)$, and only those events. Given a sequence τ and a thread thr, let τ_{thr} be the subsequence of τ with events of thr, i.e., $\tau | \mathcal{E}(\tau)_{\mathsf{thr}}$. Given a sequence τ and an event $e \in \mathcal{E}(\tau)$, we denote by $\mathsf{pre}_\tau(e)$ the prefix up until and including e, formally $\tau | \{e' \in \mathcal{E}(\tau) \,|\, e' \leq_\tau e\}$. Given two sequences τ_1 and τ_2, we denote by $\tau_1 \circ \tau_2$ the sequence that results in appending τ_2 after τ_1.

A (concrete, concurrent) *trace* is a sequence of events σ that corresponds to a concrete valid execution of $\mathscr{P}$ under the respective memory model. We let $\mathsf{enabled}(\sigma)$ be the set of enabled events after σ is executed, and call σ *maximal* if $\mathsf{enabled}(\sigma) = \emptyset$. A concrete *local trace* ρ is a sequence of thread events of the same thread. As $\mathscr{P}$ is bounded, all executions of $\mathscr{P}$ are finite and the length of the longest execution in $\mathscr{P}$ is a parameter of the input.

Reads-from functions. Given an event set $X \subseteq \mathcal{E}$, a *reads-from (RF) function* over X is a function that maps each read event of X to some write event of X accessing the same global variable. Formally, $\mathsf{RF}\colon \mathcal{R}(X) \to \mathcal{W}(X)$, where $\mathsf{var}(r) = \mathsf{var}(\mathsf{RF}(r))$ for all $r \in \mathcal{R}(X)$. In TSO/PSO, given a buffer-write event wB (resp. a memory-write event wM), we write $\mathsf{RF}(r) = (wB, \_)$ (resp. $\mathsf{RF}(r) = (\_, wM)$) to denote that $\mathsf{RF}(r)$ is a two-phase write for which wB (resp. wM) is the corresponding buffer-write (resp. memory-write) event.

Given a sequence of events τ, we define the *reads-from (RF) function* of τ, denoted $\mathsf{RF}_\tau\colon \mathcal{R}(\tau) \to \mathcal{W}(\tau)$, as follows.

RF_τ in SC: Given a read event $r \in \mathcal{R}(\tau)$, we have that $\mathsf{RF}_\tau(r)$ is the latest write (of any thread) conflicting with r and occurring before r in τ, i.e., (i) $\mathsf{RF}_\tau(r) \bowtie r$, (ii) $\mathsf{RF}_\tau(r) <_\tau r$, and (iii) for each $\overline{w} \in \mathcal{W}(\tau)$ such that $\overline{w} \bowtie r$ and $\overline{w} <_\tau r$, we have $\overline{w} \leq_\tau \mathsf{RF}_\tau(r)$.

RF_τ in TSO/PSO: Given a read event $r \in \mathcal{R}(\tau)$, consider the set Upd of enqueued conflicting updates in the same thread that have not yet been dequeued, i.e., $\mathsf{Upd} = \{(wB, wM) \in (\mathcal{W}(\tau))_{\mathsf{thr}(r)} \mid wM \bowtie r,\ wB <_\tau r <_\tau wM\}$. Then, $\mathsf{RF}_\tau(r) = (wB', wM')$, where one of the two cases happens:

- $\mathsf{Upd} \neq \emptyset$, and $(wB', wM') \in \mathsf{Upd}$ is the latest in τ, i.e., for each $(wB'', wM'') \in \mathsf{Upd}$ we have $wB'' \leq_\tau wB'$.
- $\mathsf{Upd} = \emptyset$, and $wM' \in \mathcal{W}^M(\tau)$, $wM' \bowtie r$, $wM' <_\tau r$ is the latest memory-write (of any thread) conflicting with r and occurring before r in τ, i.e., for each $wM'' \in \mathcal{W}^M(\tau)$ such that $wM'' \bowtie r$ and $wM'' <_\tau r$, we have $wM'' \leq_\tau wM'$.

Notice how relaxed memory comes into play in the above definition, as $\mathsf{RF}_\tau(r)$ does not record which of the two above cases actually happened.

We say that r reads-from $\mathsf{RF}_\tau(r)$ in τ. For simplicity, we assume that $\mathscr{P}$ has an initial salient write event on each variable.

Value functions. Given a trace σ, we define the *value function* of σ, denoted $\mathsf{val}_\sigma\colon \mathcal{E}(\sigma) \to \mathcal{D}$, such that $\mathsf{val}_\sigma(e)$ is the value of the global variable $\mathsf{var}(e)$ after the prefix of σ up to and including e has been executed. Intuitively, $\mathsf{val}_\sigma(e)$ captures the value that a read (resp. write) event e shall read (resp. write) in σ. The value function val_σ is well-defined as σ is a valid trace and the threads of $\mathscr{P}$ are deterministic.

Root thread and Side functions. For certain concepts used in our POR approaches, we distinguish a single thread thr_1 as the *root thread* of $\mathscr{P}$, and refer to the remaining threads $\mathsf{thr}_2, \dots, \mathsf{thr}_k$ as *leaf threads*.

One such concept is a *side function*. Given a sequence of events τ, the side functions of τ is a function $S_\tau\colon \mathcal{R}(\tau)_{\mathsf{thr}_1} \to [2]$ such that $S_\tau(r) = 1$ if $\mathsf{thr}(\mathsf{RF}_\tau(r)) = \mathsf{thr}_1$ and $S_\tau(r) = 2$ otherwise. In other words, a side function is defined for the read events of the root thread, and assigns 1 (resp., 2) to each read event if it reads-from a write of the root thread (resp., of some leaf thread) in the sequence.

Trace spaces and partitionings. Given a concurrent program $\mathscr{P}$ and a memory model $\mathcal{M} \in \{\mathsf{SC}, \mathsf{TSO}, \mathsf{PSO}\}$, we write $\mathcal{T}_\mathcal{M}^{all}$ (resp., $\mathcal{T}_\mathcal{M}^{max}$) for the set of all traces (resp., the set of all maximal traces) of the program $\mathscr{P}$ under the respective memory model. We sometimes omit the subscript and simply write $\mathcal{T}^{all}$ resp. $\mathcal{T}^{max}$, when the memory model is clear from the context.

We call two traces σ_1 and σ_2 *reads-from equivalent* (RF equivalent) if $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$ and $\mathsf{RF}_{\sigma_1} = \mathsf{RF}_{\sigma_2}$. The corresponding *RF equivalence* $\sim_{RF}$ partitions the trace space into equivalence classes $\mathcal{T}^{max}_{\mathcal{M}} / \sim_{RF}$ and we call this the *RF partitioning*. Traces in the same class of the RF partitioning visit the same set of local states in each thread, and thus the RF partitioning is a sound partitioning for local state reachability [AAJ+19, CCP+17, KRV19b].

2.3 Partial Orders

Here we present relevant notation around partial orders, which are a central object in our POR works.

Partial orders. Given a set of events $X \subseteq \mathcal{E}$, a *(strict) partial order* P over X is an irreflexive, antisymmetric and transitive relation over X (i.e., $<_P \subseteq X \times X$). Given two events $e_1, e_2 \in X$, we write $e_1 \leq_P e_2$ to denote that $e_1 <_P e_2$ or $e_1 = e_2$. Two distinct events $e_1, e_2 \in X$ are *unordered* by P, denoted $e_1 \parallel_P e_2$, if neither $e_1 <_P e_2$ nor $e_2 <_P e_1$, and *ordered* (denoted $e_1 \not\parallel_P e_2$) otherwise. Given a set $Y \subseteq X$, we denote by $P|Y$ the *projection* of P on the set Y, where for every pair of events $e_1, e_2 \in Y$, we have that $e_1 <_{P|Y} e_2$ iff $e_1 <_P e_2$. Given two partial orders P and Q over a common set X, we say that Q *refines* P, denoted by $Q \sqsubseteq P$, if for every pair of events $e_1, e_2 \in X$, if $e_1 <_P e_2$ then $e_1 <_Q e_2$. A *linearization* of P is a total order that refines P.

Lower sets. Given a pair (X, P), where X is a set of events and P is a partial order over X, a *lower set* of (X, P) is a set $Y \subseteq X$ such that for every event $e_1 \in Y$ and event $e_2 \in X$ with $e_2 \leq_P e_1$, we have $e_2 \in Y$.

The program order PO. The *program order* PO of $\mathscr{P}$ is a partial order $<_{\mathsf{PO}} \subseteq \mathcal{E} \times \mathcal{E}$ that defines a fixed order between some pairs of events of the same thread, reflecting the semantics of $\mathscr{P}$ under the respective memory model.

In TSO/PSO, for each thread thr, the program order PO satisfies the following conditions.

- $wB <_{\mathsf{PO}} wM$ for each $(wB, wM) \in \mathcal{W}_{\mathsf{thr}}$.
- $wB <_{\mathsf{PO}} \mathrm{fnc}$ iff $wM <_{\mathsf{PO}} \mathrm{fnc}$ for each $(wB, wM) \in \mathcal{W}_{\mathsf{thr}}$ and fence event $\mathrm{fnc} \in \mathcal{F}_{\mathsf{thr}}$.
- $wB_1 <_{\mathsf{PO}} wB_2$ iff $wM_1 <_{\mathsf{PO}} wM_2$ for each $(wB_i, wM_i) \in \mathcal{W}_{\mathsf{thr}}, i \in \{1, 2\}$
(in PSO, this condition is enforced only when $\mathsf{var}((wB_1, wM_1)) = \mathsf{var}((wB_2, wM_2))$).

For simplicity of presentation, we assume a static set of threads for a given concurrent program. However, all our POR approaches straightforwardly handle dynamic thread creating, by further including in the program order PO the orderings naturally induced by thread-spawn and thread-join events.[1]

A set of events $X \subseteq \mathcal{E}$ is *proper* if the following hold.

- X is a lower set of $(\mathcal{E}, \mathsf{PO})$.

[1]In all experiments in our POR works, the considered benchmarks spawn threads dynamically.

- Based on the considered memory model, one of the following holds.
  - In SC, for each thread thr, the events X_{thr} are totally ordered in PO (i.e., for each distinct $e_1, e_2 \in X_{\mathsf{thr}}$ we have $e_1 \not\parallel_{\mathsf{PO}} e_2$).
  - In TSO, for each thread thr, its thread events $\mathcal{L}(X)_{\mathsf{thr}}$ are totally ordered in PO, and its memory-write events $\mathcal{W}^M(X)_{\mathsf{thr}}$ are totally ordered in PO.
  - In PSO, for each thread thr, (i) its thread events $\mathcal{L}(X)_{\mathsf{thr}}$ are totally ordered in PO, and (ii) for each variable the memory-write events of this thread and variable are totally ordered in PO.

A sequence τ is *well-formed* if (i) its set of events $\mathcal{E}(\tau)$ is proper, and (ii) τ respects the program order (formally, $\tau \sqsubseteq \mathsf{PO}|\mathcal{E}(\tau)$). Every trace σ of $\mathscr{P}$ is well-formed, as it corresponds to a concrete valid execution of $\mathscr{P}$. Each event of $\mathscr{P}$ is then uniquely identified by its PO predecessors, and by the values its PO predecessor reads have read.

2.3.1 Partial-Order Concepts for the SC Memory Model

Here we present definitions of several additional objects revolving around partial orders, which are used in our works analyzing concurrent programs under the SC memory model.

Visible writes. Given a partial order P over a set X, and a read event $r \in \mathcal{R}(X)$, the set of *visible writes* of r is defined as

$$\begin{aligned}\mathsf{VisibleW}_P(r) = \{\, w \in \mathcal{W}(X) :\ & \text{(i) } r \bowtie w \text{ and (ii) } r \not<_P w \text{ and (iii) for each} \\ & w' \in \mathcal{W}(X) \text{ with } r \bowtie w', \text{ if } w <_P w' \text{ then } w' \not<_P r \,\}\end{aligned}$$

i.e., the set of write events w conflicting with r that are not “hidden” to r by P.

Maximal and minimal writes. Consider a partial order P over a set X. An event $e \in X$ is *maximal* (resp., *minimal*) in P if there is no $e' \in X$ such that $e <_P e'$ (resp., $e' <_P e$). Given a read event r, the set of *maximal writes* $\mathsf{MaxW}_P(r)$ (resp., *minimal writes* $\mathsf{MinW}_P(r)$) of r contains the write events that are maximal (resp., minimal) elements in $P|\mathsf{VisibleW}_P(r)$.

Width and Mazurkiewicz width. Let P be a partial order over a set X. The *width* $\mathsf{width}(P)$ of P is the length of its longest antichain, i.e., it is the smallest integer i such that for every set $Y \subseteq X$ of size $i+1$, there exists a pair $e_1, e_2 \in Y$ with $e_1 \not\parallel_P e_2$.

The *Mazurkiewicz width* $\mathsf{Mwidth}(P)$ of P is the smallest integer i such that the following holds. For every set $Y \subseteq X$ of size $i+1$ such that every pair of distinct events $e_1, e_2 \in Y$ is conflicting ($e_1 \bowtie e_2$), there exists a pair $e_1, e_2 \in Y$ with $e_1 \not\parallel_P e_2$. Intuitively, $\mathsf{Mwidth}(P)$ is similar to $\mathsf{width}(P)$, with the difference that, in the first case, we focus on events that are conflicting as opposed to any events.

Happens-before partial orders. A trace σ induces a *happens-before* partial order $\hookrightarrow_\sigma \subseteq \mathcal{E}(\sigma) \times \mathcal{E}(\sigma)$, which is the weakest partial order such that (i) it refines the program order (i.e., $\hookrightarrow_\sigma \sqsubseteq \mathsf{PO}|\mathcal{E}(\sigma)$), and (ii) for each pair of conflicting events ($e_1, e_2 \in \mathcal{E}(\sigma)$ with $e_1 \bowtie e_2$) such that $e_1 <_\sigma e_2$, we have that $e_1 \hookrightarrow_\sigma e_2$. In other words, $\hookrightarrow_\sigma$ retains the program order and the orderings of conflicting events in σ (hence $\mathsf{Mwidth}(\hookrightarrow_\sigma) = 1$).

Causally-happens-before partial orders. A trace σ induces a *causally-happens-before* partial order $\mapsto_\sigma \subseteq \mathcal{E}(\sigma) \times \mathcal{E}(\sigma)$, which is the weakest partial order such that (i) it refines the

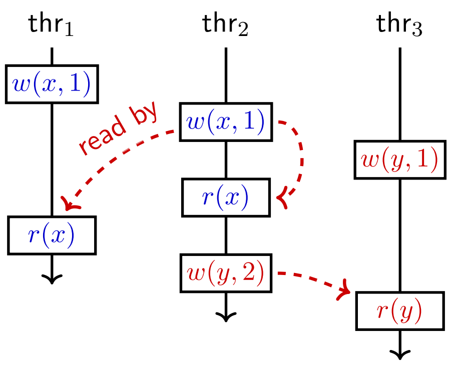


Figure 2.1: A trace and its causal orderings.

program order (i.e., $\mapsto_\sigma \sqsubseteq \mathsf{PO}|\mathcal{E}(\sigma)$), and (ii) for every read event $r \in \mathcal{R}(\sigma)$, its reads-from $\mathsf{RF}_\sigma(r)$ is ordered before it (i.e., $\mathsf{RF}_\sigma(r) \mapsto_\sigma r$). Intuitively, $\mapsto_\sigma$ contains the causal orderings in σ, i.e., it captures the flow of write events into read events in σ together with the program order.

Fig. 2.1 presents a trace σ and its causal orderings. The displayed events $\mathcal{E}(\sigma)$ are vertically ordered as they appear in σ. The solid black edges represent the program order PO. The dashed red edges represent the reads-from function RF_σ. The transitive closure of all the edges then gives us the causally-happens-before partial order $\mapsto_\sigma$.

2.4 Problems and Complexity Parameters

Here we describe the two core algorithmic problems studied in our POR works.

A. Execution consistency verification. One of the most basic problems for a given memory model is the verification of the consistency of program executions with respect to the given model [CS20]. The input is a set of thread executions, where each execution performs operations accessing the shared memory. The task is to verify whether the thread executions can be interleaved to a concurrent execution, which has the property that every read observes a specific value written by some write [GK97]. The problem is of foundational importance to concurrency, and has been studied heavily under SC [CYH+09, CL02, HCC+12].

Problem variants. The input can often be enhanced with additional objects, imposing further constraints on the acceptable solution. As an example, the input is often enhanced with a *reads-from (RF) map*, which further specifies for each read access the write access that the former should read-from. Under SC, the corresponding problem $\mathrm{VSC\text{-}rf}$ was shown to be NP-hard in the landmark work of [GK97], while it was recently shown $\mathrm{W}[1]$-hard [MPV20]. The problem lies at the heart of many verification tasks in concurrency, such as dynamic analyses [SES+12, KMV17, Pav19, MPV20, RGB20, MPV21], linearizability and transactional consistency [HW90, BE19], as well as SMC [AAJ+19, CCP+17, KRV19b]. In Chapter 5 we study the problem of verifying execution consistency with an RF function. In Chapter 3 and Chapter 4 we study the consistency verification problem where, instead of an RF function, the input is annotated with more involved objects.

Executions under relaxed memory. The natural extension of verifying execution consistency is from the SC memory model to relaxed memory models such as TSO and PSO. We denote by $\mathrm{VTSO\text{-}rf}$ (resp., $\mathrm{VPSO\text{-}rf}$) the problem of verifying execution consistency with an RF map under TSO (resp., PSO). Given the importance of $\mathrm{VSC\text{-}rf}$ for SC, and the success in establishing both upper and lower bounds, the complexity of $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$ is a very natural question and of equal importance. The verification problem is known to be NP-hard for most memory models [FMSS15], including TSO and PSO, however, no other bounds are known. Some heuristics have been developed for $\mathrm{VTSO\text{-}rf}$ [MH06, ZBEE19], while other works study TSO executions that are also sequentially consistent [BMM11, BDM13]. In Chapter 5 we study the problems $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$, while in Chapter 3 and Chapter 4 we study variants of the verification problem under the SC memory model.

B. The local-state reachability. The task here is detecting erroneous local states of threads, e.g., whether a thread ever encounters an assertion violation. The underlying algorithmic problem is that of discovering every possible local state of every thread of $\mathscr{P}$, and checking whether a bug occurs in each local state. The most standard solution to this problem is *stateless model checking (SMC)* [God96]. In SMC, the focal object for this task is the trace, and algorithms solve the problem by exploring different maximal traces of the trace space $\mathcal{T}^{max}$. Methods that couple SMC with *dynamic partial order reduction (DPOR)* techniques use an equivalence E to partition the trace space into equivalence classes, and explore the partitioning $\mathcal{T}^{max}/E$ instead of the whole space $\mathcal{T}^{max}$.

Complexity parameters. Given an equivalence E over $\mathcal{T}^{max}$, the efficiency of an algorithm that explores the partitioning $\mathcal{T}^{max}/E$ is typically a product of two factors $O(\alpha \cdot \beta)$. The first factor α is the size of the partitioning itself, i.e., $\alpha = |\mathcal{T}^{max}/E|$, which is typically exponentially large. As we construct coarser equivalences E, α decreases. The second factor β captures the amortized time on each explored class, and can be either polynomial (i.e., efficient) or exponential. There is a tradeoff between α and β: typically, for coarser equivalences E the algorithms spend more time to explore each class, and hence α is decreased at the cost of increasing β. Hence, the challenge is to make α as small as possible without increasing β much.

In Chapter 5 we consider SMC with the well-established RF equivalence, under the TSO and PSO memory models. In Chapter 3 and Chapter 4 we consider SMC under the SC memory model, using novel trace equivalences.

CHAPTER 3

The Value-Centric Equivalence for the SC Memory Model

In this chapter we present a new trace equivalence, called the *value-centric (VC)* equivalence, and we show that it has two appealing features. First, the VC equivalence is always at least *as coarse as* the happens-before (i.e., the Mazurkiewicz) equivalence, and it can be even exponentially coarser. Second, despite its coarseness properties, the consistency checking problem of VC (i.e., the problem of deciding whether an abstract description of a VC equivalence class has a concrete trace realizing it) can be solved in polynomial time, which we demonstrate in this chapter. As a result, in stateless model checking (SMC) the VC partitioning is efficiently explorable by dynamic partial order reduction (POR) when the number of threads is bounded. We present an algorithm called *value-centric dynamic partial order reduction* (VC-DPOR), which explores the underlying partitioning while spending polynomial time per equivalence class. Finally, we perform an experimental evaluation of VC-DPOR on various benchmarks, and compare it against other state-of-the-art approaches. Our results show that the VC equivalence typically induces a significant reduction in the size of the underlying partitioning, which leads to a considerable reduction in the running time for exploring the whole partitioning.

Motivating Example. Consider the simple program given in Fig. 3.1, which consists of two threads communicating over a global variable x. We have two types of events: thr$_1$ writes to x the value 1, whereas thr$_2$ first writes to x the value 2, then it writes to x the value 1, and finally it reads the value of x to its local variable. When we analyze this program, it becomes apparent that a model-checking algorithm can benefit if it takes into account the values written by the write events. Indeed, denote by w_i^j the j-th write event of thread i, and

Thread thr$_1$:	Thread thr$_2$:
1. $w(x,1)$	1. $w(x,2)$
	2. $w(x,1)$
	3. $r(x)$

Figure 3.1: A toy program with two threads.

by r the unique read event. There exist 4 Mazurkiewicz orderings.

$$\sigma_1 : \ w_1^1 w_2^1 w_2^2 r \qquad \sigma_2 : \ w_2^1 w_1^1 w_2^2 r \qquad \sigma_3 : \ w_2^1 w_2^2 w_1^1 r \qquad \sigma_4 : \ w_2^1 w_2^2 r w_1^1$$

Hence, any algorithm that uses the Mazurkiewicz equivalence for exploring the trace space of the above program will have to explore at least 4 traces. Moreover, any sound algorithm that is insensitive to values will, in general, explore at least two traces (e.g., σ_1, and σ_3), since the value read by r can, in principle, be different in both cases. On the other hand, it is clear that examining a single trace suffices for visiting all the local states of all threads. Although minimal, the above example illustrates the advantage that SMC algorithms can gain by being sensitive to the values used by the events during an execution.

Challenges. The above example illustrates that a value-sensitive partitioning can be coarse. The challenge that arises naturally is to produce a value-sensitive partitioning that (a) is provably *always* coarser than Mazurkiewicz trace equivalence, and (b) is *efficiently* explorable (i.e., the time required for each class of the partitioning is small/polynomial). In this chapter we address this challenge.

Our contributions. The main contribution of this chapter is a new equivalence sensitive to values, called the *value-centric (VC)* equivalence ($\sim_{VC}$). Intuitively, the VC equivalence distinguishes (arbitrarily) a thread of the program, called the *root*, from the other threads, called the *leaves*. The coarsening of the $\sim_{VC}$ partitioning is achieved by relaxing the happens-before orderings between events that belong to the root and leaf threads. Given two traces σ_1 and σ_2 which have the same happens-before ordering on the events of leaf threads, $\sim_{VC}$ deems σ_1 and σ_2 equivalent by using a combination of (i) the *values* and (ii) the *causally-happens-before* orderings on pairs of events between the root and the leaves.

Properties of $\sim_{VC}$. We discuss two key properties of the VC equivalence $\sim_{VC}$.

1. *Soundness.* The VC equivalence is sound for reporting correctness of local-state properties. In particular, if $\sigma_1 \sim_{VC} \sigma_2$, then the same local states are guaranteed to be visited in both executions. Thus, in order to report local-state-specific properties (e.g., absence of assertion violations), it is sound to explore a single representative from each class of the underlying partitioning. Global-state properties can be encoded as local properties by using a thread to monitor the global state. Due to this fact, recent works on dynamic POR focus on local-state properties only [Hua15, HH17, AJLS18, CCP$^+$17].

2. *Exponentially coarser than happens-before.* The VC equivalence is always at least as coarse as the happens-before (or the Mazurkiewicz) equivalence, i.e., if two traces are Mazurkiewicz-equivalent, then they are also $\sim_{VC}$-equivalent. This implies that the underlying $\sim_{VC}$ partitioning is never larger than the Mazurkiewicz partitioning. In addition, we show that there exist programs for which the $\sim_{VC}$ partitioning is exponentially smaller, thereby getting a significant reduction in one of the two factors that affect the efficiency of SMC algorithms. Interestingly, this reduction is achieved even if there are *no concurrent writes* in the program.

Value-centric DPOR. We develop an efficient SMC algorithm utilizing the dynamic POR approach that explores the $\sim_{VC}$ partitioning. We call the SMC algorithm VC-DPOR. The algorithm is guaranteed to visit every class of the $\sim_{VC}$ partitioning, and for a constant number of threads, the time spent in each class is polynomial. Hence, VC-DPOR explores efficiently the VC partitioning without relying on NP oracles. For example, in the program of Fig. 3.1, $\mathrm{VC\text{-}DPOR}$ explores only one trace.

Experimental results. Finally, we make a prototype implementation of $\mathrm{VC\text{-}DPOR}$ and evaluate it on various classes of concurrency benchmarks. We use our implementation to assess (i) the coarseness of the $\sim_{VC}$ partitioning in practice, and (ii) the efficiency of $\mathrm{VC\text{-}DPOR}$ to explore such partitionings. To this end, we compare these two metrics with existing state-of-the-art SMC algorithms, namely, Source-DPOR [AAJS14], Optimal-DPOR [AAJS14], Optimal-DPOR with observers [AJLS18], as well as $\mathrm{DC\text{-}DPOR}$ [CCP+17]. Our results show a significant reduction in the size of the partitioning compared to the partitionings explored by existing techniques, which also typically leads to smaller running times.

3.1 Value-Centric Equivalence

In this section we introduce our new equivalence between traces, called the *value-centric (VC)* equivalence, and we prove some of its properties. We start with the Mazurkiewicz equivalence (also called the happens-before equivalence), which has been used by SMC algorithms in the literature.

The Mazurkiewicz equivalence. Two traces $\sigma_1, \sigma_2 \in \mathcal{T}^{all}$ are called *Mazurkiewicz-equivalent* (sometimes referred to as *happens-before-equivalent*), written $\sigma_1 \sim_{\mathsf{Maz}} \sigma_2$, if the following hold.

1. $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$, i.e., they consist of the same set of events.
2. $\hookrightarrow_{\sigma_1} = \hookrightarrow_{\sigma_2}$, i.e., their happens-before partial orders are equal.

The value-centric equivalence. Two traces $\sigma_1, \sigma_2 \in \mathcal{T}^{all}$ are called *value-centric-equivalent*, written $\sigma_1 \sim_{VC} \sigma_2$, if the following hold.

1. $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$, $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$ and $S_{\sigma_1} = S_{\sigma_2}$, i.e., they consist of the same set of events, and their value functions and side functions are equal.
2. $\mapsto_{\sigma_1}|\mathcal{R} = \mapsto_{\sigma_2}|\mathcal{R}$, i.e., their causally-happens-before partial orders $\mapsto_{\sigma_1}$ and $\mapsto_{\sigma_2}$ agree on the read events.
3. $\hookrightarrow_{\sigma_1}|\mathcal{E}_{\neq \mathsf{thr}_1} = \hookrightarrow_{\sigma_2}|\mathcal{E}_{\neq \mathsf{thr}_1}$, i.e., their happens-before partial orders $\hookrightarrow_{\sigma_1}$ and $\hookrightarrow_{\sigma_2}$ agree on the events of the leaf threads. (i.e., the events of all threads except the root thread thr_1).

Remark 3.1 (Soundness)**.** Since every thread of $\mathscr{P}$ is deterministic, for any two traces $\sigma_1, \sigma_2 \in \mathcal{T}^{all}$ such that $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$ and $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$, the local states of each thread after executing σ_1 and σ_2 agree. It follows that any algorithm that explores every class of the partitioning $\mathcal{T}^{max}/VC$ provably discovers every reachable local state of every thread, and thus VC is a sound equivalence for local-state reachability.

Exponential coarseness. Here we provide two toy examples which illustrate different cases where the VC equivalence can be exponentially coarser than the Mazurkiewicz equivalence Maz, i.e., $\mathcal{T}^{all}/\mathsf{Maz}$ can have exponentially more classes than $\mathcal{T}^{all}/VC$.

Thread$_1$	Thread$_2$
1. $w(x, 0)$	1. $r(x)$
2. $w(x, 0)$	2. $r(x)$
$\dots$ $\dots$	$\dots$ $\dots$
n. $w(x, 0)$	n. $r(x)$

(a) Many operations on one variable.

Thread$_1$	Thread$_2$
1. $w(x_1, 0)$	1. $r(x_1)$
2. $w(x_1, 0)$	2. $r(x_2)$
$\dots$ $\dots$	$\dots$ $\dots$
$2n-1$. $w(x_n, 0)$	n. $r(x_n)$
$2n$. $w(x_n, 0)$	

(b) Few operations on many variables.

Figure 3.2: Programs where $\sim_{VC}$ is exponentially coarser than Mazurkiewicz equivalence.

Many operations on one variable. First, consider the program shown in Fig. 3.2a which consists of two threads thr$_1$ and thr$_2$, with thr$_1$ being the root thread. This program has a single global variable x, and the threads perform operations on x repeatedly. We assume a salient write event $w(x, 0)$ that writes the initial value of x. Consider any two traces σ_1, σ_2 that consist of the $i \geq 0$ first $w(x)$ events of thr$_1$ and $j \geq 0$ first $r(x)$ events of thr$_2$ (hence $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$). Since each $w(x)$ writes the same value, we have $\mathsf{val}_{\sigma_1}(r) = \mathsf{val}_{\sigma_2}(r)$ for every read event r in thr$_2$. Moreover, since the root thread thr$_1$ has no read events, we trivially have $S_{\sigma_1} = S_{\sigma_2}$. Since all read events are on thread thr$_2$, we have $\mapsto_{\sigma_1}|\mathcal{R} = \mapsto_{\sigma_2}|\mathcal{R} = \mathsf{PO}|\mathcal{R}(\sigma_1)$. Finally, since we only have one leaf thread, $\hookrightarrow_{\sigma_1}|\mathcal{E}_{\neq \mathsf{thr}_1} = \hookrightarrow_{\sigma_2}|\mathcal{E}_{\neq \mathsf{thr}_1} = \mathsf{PO}|\mathcal{E}_{\neq \mathsf{thr}_1}(\sigma_1)$. We conclude that $\sigma_1 \sim_{VC} \sigma_2$, and thus given $i \geq 0$ and $j \geq 0$ there exists a single class of $\sim_{VC}$ that contains the first i and first j events of thr$_1$ and thr$_2$, respectively. Thus $|\mathcal{T}^{all}/VC| = O(n^2)$. On the other hand, given the first $i \geq 0$ and $j \geq 0$ events of threads thr$_1$ and thr$_2$, respectively, there exist $\frac{(i+j)!}{i! \cdot j!} = \binom{i+j}{i}$ different ways to order them without violating the thread order. Observe that every such reordering induces a different happens-before relation. Using Stirling's approximation, we obtain

$$|\mathcal{T}^{all}/\mathsf{Maz}| \geq \frac{(2 \cdot n)!}{(n!)^2} \simeq \frac{\sqrt{2 \cdot \pi \cdot 2 \cdot n} \cdot (2 \cdot n/e)^{2 \cdot n}}{\left(\sqrt{2 \cdot \pi \cdot n} \cdot (n/e)^n\right)^2} = \Omega\left(\frac{4^n}{\sqrt{n}}\right)$$

Few operations on many variables. Now consider the example program shown in Fig. 3.2b which consists of two threads thr$_1$ and thr$_2$, with thr$_1$ being the root thread. We assume a salient write event $w(x_i, 0)$ that writes the initial value of x_i. Consider any two traces σ_1, σ_2 that consist of the $i \geq 0$ first $w(x)$ events of thr$_1$ and $j \geq 0$ first $r(x)$ events of thr$_2$ (hence $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$). Since each $w(x_i, 0)$ writes the same value, we have $\mathsf{val}_{\sigma_1}(r) = \mathsf{val}_{\sigma_2}(r)$ for every read event r in thr$_2$. Moreover, since the root thread thr$_1$ has no read events, we trivially have $S_{\sigma_1} = S_{\sigma_2}$. Since all read events are on thread thr$_2$, we have $\mapsto_{\sigma_1}|\mathcal{R} = \mapsto_{\sigma_2}|\mathcal{R} = \mathsf{PO}|\mathcal{R}(\sigma_1)$. Finally, since we only have one leaf thread, $\hookrightarrow_{\sigma_1}|\mathcal{E}_{\neq \mathsf{thr}_1} = \hookrightarrow_{\sigma_2}|\mathcal{E}_{\neq \mathsf{thr}_1} = \mathsf{PO}|\mathcal{E}_{\neq \mathsf{thr}_1}(\sigma_1)$. We conclude that $\sigma_1 \sim_{VC} \sigma_2$, and thus given $i \geq 0$ and $j \geq 0$ there exists a single class of $\sim_{VC}$ that contains the first i and first j events of thr$_1$ and thr$_2$, respectively. Thus $|\mathcal{T}^{all}/VC| = O(n^2)$. On the other hand, given the first i read events of thr$_2$ and $2 \cdot i$ write events of thr$_1$, there exist at least 2^i different observation functions that map each read event r to one of the two write events that r observes. Hence $|\mathcal{T}^{all}/\mathsf{Maz}| = \Omega(2^n)$.

Theorem 3.1. *The VC equivalence is sound for local-state reachability. Also, VC is at least as coarse as the Mazurkiewicz equivalence, and there exist programs where VC is exponentially coarser than the Mazurkiewicz equivalence.*

Proof. The fact that VC is sound follows from Remark 3.1. Here we prove that that VC is at least as coarse as Maz. Then Fig. 3.2 presents two examples where VC can, in fact, be exponentially coarser.

Consider two traces $\sigma_1, \sigma_2 \in \mathcal{T}^{all}$ such that $\sigma_1 \not\sim_{VC} \sigma_2$. If $\mathcal{E}(\sigma_1) \neq \mathcal{E}(\sigma_2)$ then $\sigma_1 \not\sim_{\mathsf{Maz}} \sigma_2$. Else, if $\mathsf{val}_{\sigma_1} \neq \mathsf{val}_{\sigma_2}$ or $S_{\sigma_1} \neq S_{\sigma_2}$, we have that $\hookrightarrow_{\sigma_1} \neq \hookrightarrow_{\sigma_2}$. Else, if $\mapsto_{\sigma_1} | \mathcal{R} \neq \mapsto_{\sigma_2} | \mathcal{R}$, then there exists a read event such that $\mathsf{RF}_{\sigma_1}(r) \neq \mathsf{RF}_{\sigma_2}(r)$, which implies that $\hookrightarrow_{\sigma_1} \neq \hookrightarrow_{\sigma_2}$. Finally, if $\hookrightarrow_{\sigma_1} | \mathcal{E}_{\neq \mathsf{thr}_1} \neq \hookrightarrow_{\sigma_2} | \mathcal{E}_{\neq \mathsf{thr}_1}$ then trivially $\hookrightarrow_{\sigma_1} \neq \hookrightarrow_{\sigma_2}$. Hence, in all cases we obtain $\sigma_1 \not\sim_{\mathsf{Maz}} \sigma_2$. The desired result follows. $\square$

3.2 Verifying Annotated Partial Orders

In this section we develop the core algorithmic concepts that will be used in the enumerative exploration of the VC trace partitioning. We introduce *annotated partial orders*, which are traditional partial orders over events, with additional constraints. We formulate the question of the verification of an annotated partial order $\mathcal{P}$, which asks for a witness trace σ that linearizes $\mathcal{P}$ and satisfies the constraints. We develop the notion of *closure* of annotated partial orders, and show that (i) an annotated partial order is realizable if and only if its closure exists, and (ii) deciding whether the closure exists can be done efficiently. This leads to an efficient procedure for the verification of annotated partial orders.

3.2.1 Annotated Partial Orders

Here we introduce the notion of annotated partial orders, which is a central concept of our work. We build some definitions and notation, and provide some intuition around them.

Annotated Partial Orders. An *annotated partial order* is a tuple $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ where the following hold.

1. X_1, X_2 are sets of events such that $X_1 \cap X_2 = \emptyset$.
2. P is a partial order over the set $X = X_1 \cup X_2$.
3. $\mathsf{val}\colon X \to \mathcal{D}$ is a value function.
4. $S\colon \mathcal{R}(X_1) \to [2]$ is a side function.
5. $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$ is a good-writes function such that $w \in \mathsf{GoodW}(r)$ only if (i) $r \bowtie w$ and (ii) $\mathsf{val}(r) = \mathsf{val}(w)$ and (iii) if $r \in X_1$ then $w \in X_{S(r)}$.
6. $\mathsf{width}(P|X_1) = \mathsf{Mwidth}(P|X_2) = 1$.

We let the bad-writes function be $\mathsf{BadW}(r) = \{w \in \mathcal{W}(X) \setminus \mathsf{GoodW}(r) \mid r \bowtie w\}$. Further, given an event $e \in X$, we let $\mathcal{I}_{\mathcal{P}}(e) = i$ such that $e \in X_i$.

We call an annotated partial order $\mathcal{P}$ *consistent* if for every thread thr, we have that $\tau_{\mathsf{thr}} = \mathsf{PO}|(X \cap \mathcal{E}_{\mathsf{thr}})$ is a local trace of thread thr that occurs if every event e of τ_{thr} reads/writes the value $\mathsf{val}(e)$. Hereinafter we only consider consistent annotated partial orders.

Verification of annotated partial orders. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$. A trace σ is a *witness* of $\mathcal{P}$ if (i) $\sigma \sqsubseteq P$ and (ii) for every read event $r \in \mathcal{R}(X_1 \cup X_2)$ we have that $\mathsf{RF}_\sigma(r) \in \mathsf{GoodW}(r)$. In words, σ is a linearization of the partial order P with the additional constraint that the reads-from function of σ must agree with the good-writes function GoodW of $\mathcal{P}$. We call $\mathcal{P}$ *realizable* if it has a witness. The associated problem of verifying annotated partial orders takes as input an annotated partial order $\mathcal{P}$ and asks whether $\mathcal{P}$ is realizable.

Remark 3.2 (Verification to valid traces.)**.** If σ is a witness of some consistent annotated partial order $\mathcal{P}$, then σ is a valid concrete trace of $\mathscr{P}$. This holds because of the following observations.

1. Since σ is a witness of $\mathcal{P}$, we have $\mathsf{RF}_\sigma(r) \in \mathsf{GoodW}(r)$ for every read event $r \in \mathcal{R}(\sigma)$.
2. Due to the previous item and the consistency of $\mathcal{P}$, for every thread thr we have that $\tau_{\mathsf{thr}} = \mathsf{PO}|(X \cap \mathcal{E}_{\mathsf{thr}})$ is a valid local trace of thr.

Intuition. An annotated partial order $\mathcal{P}$ contains a partial order P over a set $X = X_1 \cup X_2$ of events and the value of each event of X. Intuitively, the consistency of $\mathcal{P}$ states that we obtain the set of events X if we execute each thread and force every read event in this execution to observe the value of a write event according to the good-writes function. In the next section, our $\mathrm{VC\text{-}DPOR}$ algorithm uses annotated partial orders to represent different classes of the VC equivalence in order to guide the trace-space exploration. The set X_1 (resp., X_2) will contain the events of the root thread (resp., leaf threads). We will see that if $\mathrm{VC\text{-}DPOR}$ constructs two annotated partial orders $\mathcal{P}'$ and $\mathcal{P}''$ during the exploration, then any two witnesses σ' and σ'' of $\mathcal{P}'$ and $\mathcal{P}''$, respectively, will satisfy that $\sigma' \not\sim_{VC} \sigma''$, and hence $\mathcal{P}'$ and $\mathcal{P}''$ represent different classes of the VC trace partitioning.

3.2.2 Closure of annotated partial orders

We now turn our attention to closed annotated partial orders and closure, which will provide us with a way of solving the verification problem.

Closed annotated partial orders. Let us consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ and let $X = X_1 \cup X_2$. We say that $\mathcal{P}$ is *closed* if the following conditions hold for every read event $r \in \mathcal{R}(X)$.

1. There exists a write event $w \in \mathsf{GoodW}(r) \cap \mathsf{MinW}_P(r)$ such that $w <_P r$.
2. $\mathsf{MaxW}_P(r) \cap \mathsf{GoodW}(r) \neq \emptyset$.
3. For every write event $w' \in \mathsf{BadW}(r) \cap \mathsf{MinW}_P(r)$ such that $w' <_P r$ there exists a write event $w \in \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_P(r)$ such that $w' <_P w$.

Our motivation behind this definition becomes clear from the following lemma, which states that closed annotated partial orders are realizable.

Lemma 3.1. *If $\mathcal{P}$ is closed then it is realizable and a witness can be constructed in $O(\mathsf{poly}(n))$ time.*

Proof. Let $\mathcal{P} = (X_1, X_2, \mathsf{val}, P, S, \mathsf{GoodW})$, and we construct a witness σ of P as follows.

1. Create a partial order Q as follows.
   a) For every pair of events e_1, e_2 with $e_1 <_P e_2$, we have $e_1 <_Q e_2$.
   b) For every pair of events e_1, e_2 with $e_i \in X_i$ for each $i \in [2]$, if $e_2 \not<_P e_1$ then $e_1 <_Q e_2$.
2. Create σ by linearizing Q arbitrarily.

It is easy to see that since $\mathsf{width}(P|X_1) = 1$, Q is indeed a partial order and thus σ is well defined. In addition, the above process takes $O(\mathsf{poly}(n))$ time. We now argue that σ is indeed a witness trace. It is clear that $Q \sqsubseteq P$ and thus σ is a linearization of P. It remains to argue that for every read event $r \in \mathcal{R}(X)$, we have that $\mathsf{RF}_\sigma \in \mathsf{GoodW}(r)$. We distinguish between the following cases.

1. $r \in X_1$. Let $w = \mathsf{RF}_\sigma(r)$, and observe that $w <_P r$. Assume towards contradiction that $w \in \mathsf{BadW}(r)$. If $S(r) = 1$, by Item 3 of closed annotated partial orders we have that there exists a write event $w' \in X_1$ such that $w <_P w'$. Since $S(r) = 1$, we have $w' <_{\mathsf{PO}} r$ thus $w' <_P r$ and $w \notin \mathsf{VisibleW}_P(r)$, a contradiction. Otherwise, $S(r) = 2$, and by Item 1 of closed annotated partial orders, there exists a write event $w' \in \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_P(r) \cap X_2$ such that $w' <_P r$. Observe that in this case $w = w'$, a contradiction.

2. $r \in X_2$. Let $w = \mathsf{RF}_\sigma(r)$, and observe that $w \in \mathsf{MaxW}_P(r)$. Assume towards contradiction that $w \in \mathsf{BadW}(r)$. By Item 3 of closed annotated partial orders, we have that $w \not<_P r$. In this case $w \in X_1$, and since $\mathsf{RF}_\sigma(r) = w$, there exists no $w' \in X_2 \cap \mathsf{GoodW}_P(r) \cap \mathsf{VisibleW}_P(r)$. It followed that $|\mathsf{MaxW}_P(r)| = 1$, and by Item 2 of closed annotated partial orders we have that $w \in \mathsf{GoodW}(r)$, a contradiction.

The desired result follows. $\square$

We now introduce the notion of *closure*. Consider an annotated partial order $\mathcal{P}$ that is not closed. Intuitively, the closure of $\mathcal{P}$ strengthens $\mathcal{P}$ by introducing the smallest set of event orderings such that the resulting annotated partial order $\mathcal{Q}$ is closed. The intuition behind the closure is the following: whenever a rule forces some ordering, any trace that witnesses the realizability of $\mathcal{P}$ also linearizes $\mathcal{Q}$. In some cases this operation results to cyclic orderings, and thus the closure does not exist. We also show that obtaining the closure or deciding that it does not exist can be done in polynomial time. Thus, in combination with Lemma 3.1, we obtain an efficient algorithm for deciding whether $\mathcal{P}$ is realizable, by deciding whether it has a closure.

Closure of annotated partial orders. Let us consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$. An annotated partial order $\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ is a *closure* of $\mathcal{P}$ if (i) $Q \sqsubseteq P$, (ii) $\mathcal{Q}$ is closed, and (iii) for any partial order $K \neq Q$ with $Q \sqsubseteq K \sqsubseteq P$, we have that the annotated partial order $(X_1, X_2, K, \mathsf{val}, S, \mathsf{GoodW})$ is not closed. As the following lemma states, $\mathcal{P}$ can have at most one closure.

Lemma 3.2. *There exists at most one weakest partial order Q such that $Q \sqsubseteq P$ and $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ is closed.*

Proof. Assume towards contradiction otherwise, and let Q_1, Q_2 be two weakest partial orders (i.e., $Q_i \not\sqsubseteq Q_{3-i}$ for each $i \in [2]$) with the stated properties. Let $Q = Q_1 \cap Q_2$, thus $Q_1, Q_2 \sqsubseteq Q$, and we argue that $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ is closed. Let $X = X_1 \cup X_2$ and consider any read event $r \in \mathcal{R}(X)$, and we show that each of closure conditions holds for r.

1. Assume that for some $i \in [2]$ there exists a write event $w_i \in \mathsf{GoodW}(r) \cap \mathsf{MinW}_{Q_i}(r) \cap X_{\mathcal{I}_\mathcal{P}(r)}$. Since $Q_i \sqsubseteq Q$, we have that $w_i \in \mathsf{MinW}_Q(r)$ and thus Item 1 of closure is satisfied. Otherwise, for each $i \in [2]$ there exists a write event $w_i \in \mathsf{GoodW}(r) \cap \mathsf{MinW}_{Q_i}(r) \cap X_{3-\mathcal{I}_\mathcal{P}(r)}$ such that $w_i <_{Q_i} r$. Since $\mathsf{Mwidth}(P|X_{3-\mathcal{I}_\mathcal{P}(r)}) = 1$, we have that $w_i <_Q w_{3-i}$ for some $i \in [2]$, and thus $w_i <_Q r$. Finally, since $Q_i \sqsubseteq Q$ we have $w_i \in \mathsf{VisibleW}_Q(r)$ and thus $w_i \in \mathsf{MinW}_Q(r)$.

2. Assume that for some $i \in [2]$ there exists a write event $w_i \in \mathsf{GoodW}(r) \cap \mathsf{MaxW}_{Q_i}(r) \cap X_{\mathcal{I}_\mathcal{P}}(r)$. Since $Q_i \sqsubseteq Q$, we have that $w_i \in \mathsf{MaxW}_Q(r)$ and thus Item 2 of closure is satisfied. Otherwise, for each $i \in [2]$ there exists a write event $w_i \in \mathsf{GoodW}(r) \cap \mathsf{MaxW}_{Q_i}(r) \cap X_{3-\mathcal{I}_\mathcal{P}}(r)$ such that $w_i <_{Q_i} r$. Since $\mathsf{Mwidth}(P|X_{3-\mathcal{I}_\mathcal{P}(r)}) = 1$, we have that $w_{3-i} <_Q w_i$ for some $i \in [2]$. Since $Q_i \sqsubseteq Q$, we have $w_i \in \mathsf{VisibleW}_Q(r)$ and it remains to argue that $w_i \in \mathsf{MaxW}_Q(r)$. Indeed, if that is not the case then there exists a write event $w' \in \mathsf{MaxW}_Q(r)$ such that $w_i <_Q w$. But then $w_i <_{Q_j} w$ for each $j \in [2]$ and since $w \not\in \mathsf{MaxW}_{Q_j}$, we have $r <_{Q_j} w$ for each $j \in [2]$. Hence $r <_Q w$, a contradiction.

3. Consider any write event $w' \in \mathsf{BadW}(r) \cap \mathsf{MinW}_Q(r)$ such that $w' <_Q r$, and we have $w' <_{Q_i} r$ for each $i \in [2]$.

   First assume that $\mathcal{I}_\mathcal{P}(w) = \mathcal{I}_\mathcal{P}(r)$. It follows that for each $i \in [2]$ there exists a write event $w_i \in \mathsf{GoodW}(r) \cap \mathsf{MaxW}_{Q_i}(r) \cap X_{3-\mathcal{I}_\mathcal{P}}(r)$ such that $w' <_Q w_i$. Since $\mathsf{Mwidth}(P|X_{3-\mathcal{I}_\mathcal{P}}(r)) = 1$, we have $w_{3-i} <_P w_i$ for some $i \in [2]$, and thus $w' <_{Q_j} w_i$ for each $j \in [2]$. Hence $w' <_Q w_i$. Since for each $j \in [2]$ we have $Q_j \sqsubseteq Q$, it is $w_i \in \mathsf{VisibleW}_Q(r)$, as desired.

   Finally, assume that $\mathcal{I}_\mathcal{P}(w) = 3 - \mathcal{I}_\mathcal{P}(r)$. If for some $i \in [2]$ there exists a write event $w_i \in \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_{Q_i} \cap X_{\mathcal{I}_\mathcal{P}}(w')$ such that $w' <_{Q_i} w$, since $Q_i \sqsubseteq Q$ and $\mathsf{Mwidth}(P|X_{\mathcal{I}_\mathcal{P}}(w')) = 1$ we have $w_i \in \mathsf{VisibleW}_Q(r)$ and $w' <_Q w$ as desired. Otherwise, due to Item 2 of closure it follows that for the unique write event $w \in X_{\mathcal{I}_\mathcal{P}}(r) \cap \mathsf{VisibleW}_Q(r)$ we have $w \in \mathsf{GoodW}(r)$.

It follows that Q is closed, a contradiction. The desired result follows. □

Feasible annotated partial orders. In light of Lemma 3.2, we define the *closure* of $\mathcal{P}$ as the unique annotated partial order $\mathcal{Q}$ that is a closure of $\mathcal{P}$, if such $\mathcal{Q}$ exists, and $\bot$ otherwise. We call $\mathcal{P}$ *feasible* if its closure is not $\bot$. We have the following lemma.

Lemma 3.3. *$\mathcal{P}$ is realizable if and only if it is feasible.*

Proof. Let $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$. We prove each direction separately.

($\Rightarrow$). If $\mathcal{P}$ is feasible, let $\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ be the closure of $\mathcal{P}$. Since $\mathcal{Q}$ is closed, by Lemma 3.1 we have that $\mathcal{Q}$ is linearizable to a trace σ. Since $Q \sqsubseteq P$, we have that σ is also a linearization of $\mathcal{P}$.

($\Leftarrow$). If $\mathcal{P}$ is realizable, there exists a trace σ such that $\sigma \sqsubseteq P$ and for every read event $r \in \mathcal{R}(\sigma)$ we have $\mathsf{RF}_\sigma(r) \in \mathsf{GoodW}(r)$. We can view σ as a partial (total) order, and observe that the annotated partial order $(X_1, X_2, \sigma, \mathsf{val}, S, \mathsf{GoodW})$ is closed. Hence P is feasible.

The desired result follows. □

Intuitively, Lemma 3.3 states that the closure rules give the weakest strengthening of $\mathcal{P}$ that is met by any witness of $\mathcal{P}$. If that strengthening can be made (i.e., $\mathcal{P}$ is feasible), then $\mathcal{P}$ has a witness. Hence, to decide whether $\mathcal{P}$ is realizable, it suffices to decide whether it is feasible, by computing its closure. In the next section we show that this computation can be done efficiently.

3.2.3 Computing the Closure

We now present an algorithm Closure that computes the closure of annotated partial orders. This will provide us with a way of solving the problem of verifying annotated partial orders.

Closure algorithm. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ and let $X = X_1 \cup X_2$. The algorithm Closure($\mathcal{P}$) either computes the closure of $\mathcal{P}$, or concludes that $\mathcal{P}$ is not feasible, and returns $\bot$. Intuitively, the algorithm maintains a partial order Q, initially identical to P. The algorithm iterates over every read event r and tests whether r violates Item 1, Item 2 or Item 3 of the definition of closed annotated partial orders. When it discovers that r violates one such closure rule, Closure calls one of the *closure methods* Rule1(r), Rule2(r), Rule3(r), for violation of Item 1, Item 2 and Item 3 of the definition, respectively. In turn, each of these methods inserts a new ordering $e_1 \rightarrow e_2$ in Q, with the guarantee that if $\mathcal{P}$ has a closure $\mathcal{K} = (X_1, X_2, K, \mathsf{val}, S, \mathsf{GoodW})$, then $e_1 <_K e_2$. Hence, $e_1 \rightarrow e_2$ is a *necessary ordering* in the closure of $\mathcal{P}$. Finally, when the algorithm discovers that all closure rules are satisfied by every read event in Q, it returns the annotated partial order $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$, which, due to Lemma 3.2, is guaranteed to be the closure of $\mathcal{P}$. We refer to Algorithm 3.1 for a formal description.

Algorithm 3.1: Closure($\mathcal{P}$)

```
Input: An annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$.
Output: The closure of $\mathcal{P}$ if it exists, else $\bot$.
1  $Q \leftarrow P$                          // We will strengthen $Q$ during the closure computation
2  Flag ← True
3  while Flag do
4      Flag ← False
5      foreach $r \in \mathcal{R}(X_1 \cup X_2)$ do                   // Iterate over the reads
6          if $r$ violates Item 1 of closure then
7              Call Rule1($r$)                    // Strengthen $Q$ to remove violation
8              Flag ← True               // Repeat as new violations might have appeared
9          if $r$ violates Item 2 of closure then
10             Call Rule2($r$)                    // Strengthen $Q$ to remove violation
11             Flag ← True               // Repeat as new violations might have appeared
12         if $r$ violates Item 3 of closure then
13             Call Rule3($r$)                    // Strengthen $Q$ to remove violation
14             Flag ← True               // Repeat as new violations might have appeared
15 return $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$                     // The closure of $\mathcal{P}$
```

Algorithm 3.2: Rule1(r)

```
1 $Y \leftarrow \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_Q(r)$
2 if $Y = \emptyset$ then return $\bot$
3 $w \leftarrow \min_Q(Y)$                // Since Rule 1 is violated, $\min_Q(Y)$ is unique
4 Insert $w \to r$ in $Q$
```

Algorithm 3.3: Rule2(r)

```
1 $w \leftarrow$ the unique event in $\mathsf{MaxW}_Q(r) \cap X_{3-\mathcal{I}_{\mathcal{P}}(r)}$        // $w$ exists since Item 1 holds
2 Insert $r \to w$ in $Q$
```

Algorithm 3.4: Rule3(r)

```
1 $\overline{w} \leftarrow$ the unique event in $\mathsf{MinW}_Q(r) \cap \mathsf{BadW}(r)$   // exists since Items 1 and 2 hold
2 $w \leftarrow$ the unique event in $\mathsf{MaxW}_Q(r) \cap X_{3-\mathcal{I}_{\mathcal{P}}(\overline{w})}$  // exists since Items 1 and 2 hold
3 Insert $\overline{w} \to w$ in $Q$
```

We now provide some intuition behind each of the closure methods. Given two events $e_1, e_2 \in X$, we say that e_2 is *local* to e_1 if $\mathcal{I}_{\mathcal{P}}(e_1) = \mathcal{I}_{\mathcal{P}}(e_2)$, i.e., e_1 and e_2 belong to the same set X_i. If e_2 is not local to e_1, then it is *remote* to e_1. We illustrate the three closure rules in Fig. 3.3, where we follow the convention that barred and unbarred write events ($\overline{w}$ and w) are bad writes and good writes for r, respectively. In each case in Fig. 3.3, the dashed edge shows the new order introduced by the algorithm in Q.

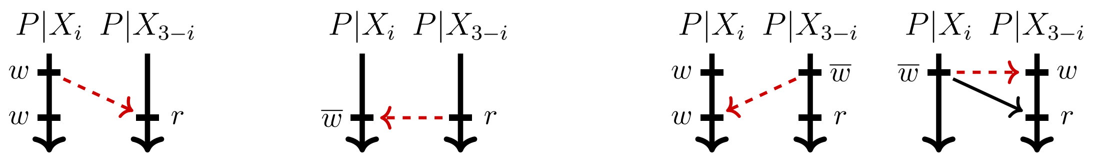
1. Rule1(r). This rule is called when Item 1 of closure is violated, i.e., there exists no write event $w \in \mathsf{GoodW}(r) \cap \mathsf{MinW}_Q(r)$ such that $w <_Q r$. Observe that in this case there is no write event that is (i) local to r, (ii) good for r and (iii) visible to r. To make r respect this rule, the algorithm finds the first write event w that is (i) good for r and (ii) visible to r, and orders $w \to r$ in Q. See Fig. 3.3a provides an illustration.

2. Rule2(r). This rule is violated when $\mathsf{MaxW}_Q(r) \cap \mathsf{GoodW}(r) = \emptyset$, i.e., every maximal write event is bad for r. To make r respect this rule, the algorithm finds the unique maximal write event w that is remote to r and orders $r \to w$ in Q. Rule2(r) is called only if r does not violate Item 1 of closure, which guarantees that w exists. Fig. 3.3b provides an illustration.

3. Rule3(r). This rule is violated when there exists a write event $\overline{w} \in \mathsf{BadW}(r) \cap \mathsf{MinW}_Q(r)$ such that (i) $\overline{w} <_Q r$, and (ii) there exists no write event $w' \in \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_Q(r)$ such that $\overline{w} <_Q w'$. To make r respect this rule, the algorithm determines a maximal write event w that is (i) remote to $\overline{w}$ and (ii) a good write for r, and orders $\overline{w} \to w$ in Q. Rule3(r) is called only if r does not violate either Item 1 or Item 2 of closure, which guarantees that w exists. Fig. 3.3c provides an illustration, depending on whether $\overline{w}$ is local or remote to r.

We have the following lemma regarding the correctness and complexity of Closure.

Lemma 3.4. Closure *correctly computes the closure of* $\mathcal{P}$ *and requires* $O(\mathsf{poly}(n))$ *time.*



(a) Rule1(r) **(b)** Rule2(r) **(c)** Rule3(r)

Figure 3.3: The three closure operations Rule1(r) (a), Rule2(r) (b) and Rule3(r) (c).

Proof. We prove the correctness of Closure and then argue about its complexity.

Correctness. We prove the following two assertions.

1. If Closure$(\mathcal{P})$ returns $\mathcal{Q} \neq \perp$ then $\mathcal{Q}$ is the closure of $\mathcal{P}$.
2. If Closure$(\mathcal{P})$ returns $\perp$ then $\mathcal{P}$ is not feasible.

Invariant. We first show that the following invariant holds at all times: if $\mathcal{P}$ has a closure $\mathcal{K} = (X_1, X_2, K, \mathsf{val}, S)$ then $K \sqsubseteq Q$. The claim holds trivially in the beginning of Closure since $Q = P$. Now assume that the algorithm inserts an ordering $e_1 \rightarrow e_2$ in Q, let Q' be the resulting partial order, and we will argue that $K \sqsubseteq Q'$. By the induction hypothesis, we have that $K \sqsubseteq Q$. We split cases based on which closure rule inserted the ordering $e_1 \rightarrow e_2$.

1. Rule1(r). In this case $e_2 = r$ and $e_1 = w$ as instantiated in Line 3 of Algorithm 3.2. By Item 1 of closure, there exists a write event $w' \in \mathsf{GoodW}(r) \cap \mathsf{MinW}_K(r)$ such that $w' <_K r$. By the induction hypothesis, we have that $K \sqsubseteq Q$, thus $w' \in \mathsf{VisibleW}_Q(r)$. Observe that since $\mathsf{Mwidth}(P|X_1) = \mathsf{Mwidth}(P|X_2) = 1$ and the rule is violated, we have that the set $Y = \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_Q(r)$ is totally ordered in Q, thus $w \leq_Q w'$, and thus $w <_K r$, as desired.

2. Rule2(r). In this case $e_1 = r$ and $e_2 = w$ as instantiated in Line 3 of Algorithm 3.3. By Item 2 of closure, there exists a write event $w' \in \mathsf{MaxW}_K \cap \mathsf{GoodW}(r)$. By the induction hypothesis, we have that $K \sqsubseteq Q$, thus $w' \in \mathsf{VisibleW}_Q(r)$. Observe that $\mathcal{I}_{\mathcal{P}}(w') = X_{3-\mathcal{I}_{\mathcal{P}}(r)}$, otherwise since $\mathsf{Mwidth}(P|X_{\mathcal{I}_{\mathcal{P}}(r)}) = 1$ we would have $w' \in \mathsf{MaxW}_Q(r)$ and thus Item 2 of closure would not be violated. Since $\mathsf{Mwidth}(P|X_{3-\mathcal{I}_{\mathcal{P}}}(r)) = 1$, it follows that $w' <_Q w$ and thus $r <_K w$, as desired.

3. Rule3(r). In this case $e_1 = \overline{w}$ and $e_2 = w$ as instantiated in Line 1 and Line 2 of Algorithm 3.4, respectively. Observe that at this point Item 1 of closure is not violated for r, and thus $|\mathsf{MinW}_Q(r) \cap \mathsf{BadW}_Q(r)| = 1$, and $\overline{w}$ is the unique event in $\mathsf{MinW}_Q(r) \cap \mathsf{BadW}(r)$. By Item 3 of closure, either $\overline{w} \not\in \mathsf{MinW}_K(r)$, or there exists a write event $w' \in \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_K(r)$ such that $\overline{w} <_K w'$. Since $\mathsf{Mwidth}(K|X_1) = \mathsf{Mwidth}(K|X_2) = 1$, it is easy to verify that in both cases there exists a write event $w'' \in \mathsf{MaxW}_K(r) \cap X_{3-\mathcal{I}_{\mathcal{P}}(\overline{w})}$ such that $\overline{w} <_K w''$ and $w'' \leq_K w$, thus $\overline{w} <_K w$, as desired.

Main correctness proof. We are now ready to prove the correctness. We examine each item separately.

1. If Closure($\mathcal{P}$) returns $\mathcal{Q} = (X_1, X_2, Q, \text{val}, S, \text{GoodW})$ then we have that $\mathcal{Q}$ is closed and $Q \sqsubseteq P$. It follows that the closure of $\mathcal{P}$ exists, and the above invariant establishes that $\mathcal{Q}$ is the closure of $\mathcal{P}$.

2. If Closure($\mathcal{P}$) returns $\perp$, then at some point the algorithm discovers a read event r such that $\text{GoodW}(r) \cap \text{VisibleW}_Q(r) = \emptyset$. Assume towards contradiction that $\mathcal{P}$ is feasible and $\mathcal{K} = (X_1, X_2, K, \text{val}, S, \text{GoodW})$ is the closure of $\mathcal{P}$. By our invariant above, it follows that $K \sqsubseteq Q$. But then $\text{GoodW}(r) \cap \text{VisibleW}_K(r) = \emptyset$, which contradicts Item 1 of closure, a contradiction.

The correctness follows.

Complexity. Let $n = |X_1 \cup X_2|$. It is straightforward to see that testing whether $\mathcal{Q}$ violates any of the closure rules in Line 6, Line 9 and Line 12 requires polynomial time in n. Every time one of these rules is violated, Closure strengthens $\mathcal{Q}$ by inserting some new orderings in $\mathcal{Q}$. Since Closure can insert at most n^2 such new orderings, it follows that the running time of Closure is $O(\text{poly}(n))$.

The desired result follows.

□

3.2.4 Verification Algorithm

Finally, we address the question of verification of annotated partial orders. Lemma 3.3 implies that in order to decide whether an annotated partial order is realizable, it suffices to compute its closure, and Lemma 3.4 states that the closure can be computed efficiently. Together, these two lemmas yield a simple algorithm for solving the realizability problem.

Verification algorithm. We describe a simple algorithm Witness that decides whether an annotated partial order $\mathcal{P}$ is realizable. The algorithms runs in two steps.

1. Use Lemma 3.4 to compute the closure of $\mathcal{P}$. If the closure is $\perp$, report that $\mathcal{P}$ is not realizable. Otherwise, the closure is an annotated partial order $\mathcal{Q}$.

2. Use Lemma 3.1 to obtain a witness trace σ of $\mathcal{Q}$. Report that $\mathcal{P}$ is realizable, and σ is the witness trace.

We conclude the results of this section with the following theorem.

Theorem 3.2. *Let $\mathcal{P}$ be an annotated partial order of n events. Deciding whether $\mathcal{P}$ is realizable requires $O(\text{poly}(n))$ time. If $\mathcal{P}$ is realizable, a witness trace can be produced in $O(\text{poly}(n))$ time.*

Proof. By Lemma 3.3, $\mathcal{P}$ is realizable if and only if it is feasible. By Lemma 3.4 the algorithm Closure($\mathcal{P}$) runs in $O(\text{poly}(n))$ time and returns the annotated partial order $\mathcal{Q}$ that is the closure of $\mathcal{P}$ if and only if $\mathcal{P}$ is feasible. If $\mathcal{P}$ is realizable, Lemma 3.1 provides a simple construction of a witness trace in $O(\text{poly}(n))$ time. □

Example (Verification of annotated partial orders)**.** We illustrate Witness on a simple example in Fig. 3.4 with an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$, which we assume to be consistent. We have a concurrent program $\mathscr{P}$ of two threads. To represent $\mathcal{P}$, we make the following conventions. We have three global variables x, y, z, and a unique read event per variable. Event subscripts denote the variable accessed by the corresponding event. For each variable, we have a unique read event, and barred and unbarred events denote the good and bad write events, respectively, for that read event. Since we have specified the good-writes for each read event, the value function val is not important for this example. Note also that $S(r_x) = 2$ (resp., $S(r_z) = 1$) since the good writes of r_x (resp., r_z) are remote (resp., local) to the read event. The partial order P of $\mathcal{P}$ consists of the thread orders of each thread, shown in solid lines in Fig. 3.4a. The dashed edges of Fig. 3.4a show the strengthening of P performed by the algorithm Closure (Algorithm 3.1). The numbers above the dashed edges denote both the order in which these orderings are added and the closure rule that is responsible for the corresponding ordering. In particular, algorithm Closure performs the following steps.

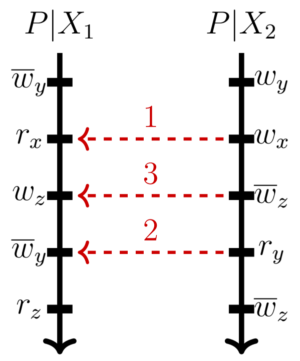
1. Initially there are no dashed edges, and r_x violates Item 1 of closure, as there is no good write event for r_x that is ordered before r_x. Rule1 inserts an ordering $w_x \to r_x$ (dashed edge 1).
2. After the previous step, r_y violates Item 2 of closure, as at this point, r_y has only one maximal write event $\overline{w}_y$, which is bad for r_y. Rule2 inserts an ordering $r_y \to \overline{w}_y$ (dashed edge 2).
3. After the previous step, r_z violates Item 3 of closure, as at this point, r_z has a bad minimal write event $\overline{w}_z$ that is ordered before r_z but not before any good write event. Rule3 inserts an ordering $\overline{w}_z \to w_z$ (dashed edge 3).

At this point no closure rule is violated, and Closure returns $\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$, the closure of $\mathcal{P}$, where P has been strengthened to Q with the dashed edges. Observe that Q has Mazurkiewicz width 2 (and not 1), as there still exist pairs of conflicting events that are unordered, both on variable y and variable z. For example, there exist two write events on variable y that are unordered, and hence there exist some linearizations that are "bad" in the sense that the read event r_y does not observe the good write event w_y. Nevertheless, Lemma 3.1 guarantees that the corresponding annotated partial order is linearizable to a valid trace, which is shown in Fig. 3.4b We make two final remarks for this example.

1. Not every linearization of Q produces a valid witness trace for the verification of $\mathcal{Q}$, as some linearizations violate the additional constraints that every read event must observe a write event that is good for the read event. Hence, the challenge is to find a correct witness (which Lemma 3.1 always achieves for a closed annotated partial order).
2. $\mathcal{Q}$ has more than one witness of realizability. Fig. 3.4b shows one such witness σ, as constructed by Lemma 3.1. It is easy to verify that σ is a valid witness. Due to Remark 3.2, the consistency of $\mathcal{P}$ guarantees that σ is a valid trace of the program $\mathscr{P}$.

3.3 Stateless Model Checking

We now present our SMC algorithm $\mathrm{VC\text{-}DPOR}$ for exploring the partitioning $\mathcal{T}^{max}/VC$. Intuitively, the algorithm manipulates annotated partial orders $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$



(a) The closure of $\mathcal{P}$ (dashed edges).

	τ_1	τ_2
1	$\overline{w}_y$	
2		w_y
3		w_x
4	r_x	
5		$\overline{w}_z$
6	w_z	
7		r_y
8	$\overline{w}_y$	
9	r_z	
10		$\overline{w}_z$

(b) A witness trace of $\mathcal{P}$.

Figure 3.4: An annotated partial order $\mathcal{P}$ and its witness trace.

where $X_1 \subseteq \mathcal{E}_{\mathsf{thr}_1}$ and $X_2 \subseteq \mathcal{E}_{\neq \mathsf{thr}_1}$, i.e., X_1 (resp., X_2) contains events of the root thread (resp., leaf threads). We first introduce some useful concepts and then proceed with the main algorithm.

Trace extensions and inevitable sets. Given a trace σ, an *extension* of σ is a trace σ' such that σ is a prefix of σ'. We say that σ' is a *maximal extension* of σ if σ' is an extension of σ and σ' is maximal. A set of events X is *inevitable* for σ if for every maximal extension σ' of σ we have $X \in \mathcal{E}(\sigma')$. A *write extension* of σ, denoted by $\mathsf{WExtend}(\sigma)$, is any arbitrary largest extension σ' of σ such that $\mathcal{E}(\sigma') \setminus \mathcal{E}(\sigma) \subseteq \mathcal{W}$. In words, we obtain each σ' by extending σ arbitrarily until (but not included) the next read event of each thread. Note that for every such write extension σ' of σ, for every thread thr, the local trace $\sigma'|\mathcal{E}(\mathsf{thr})$ is unique, and the set $\mathcal{E}(\sigma')$ is inevitable for σ. Let $\mathcal{P}$ be a closed annotated partial order over a set X. A set of events Y is *inevitable* for $\mathcal{P}$ if for every linearization σ of $\mathcal{P}$ and every maximal extension σ' of σ, we have that $Y \subseteq \mathcal{E}(\sigma')$.

Leaf refinement and minimal annotated partial orders. Consider two partial orders P, Q over a set X. We say that Q *leaf-refines* P, denoted by $Q \preccurlyeq P$ if for every pair of events $e_1, e_2 \in X \cap \mathcal{E}_{\neq \mathsf{thr}_1}$, if $e_1 \bowtie e_2$ and $e_1 <_P e_2$ then $e_1 <_Q e_2$. In words, Q leaf-refines P if Q agrees with P on the order of every pair of conflicting events that belong to leaf threads. Consider an annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$. We call $\mathcal{P}$ *minimal* if for every closed annotated partial order $\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$, if $Q \preccurlyeq P$ then $Q \sqsubseteq P$. Intuitively, the minimality of $\mathcal{P}$ guarantees that P is the weakest partial order among all partial orders Q that

1. agree with P on the order of conflicting pairs of events that belong to leaf threads, and
2. make the resulting annotated partial order $(X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW})$ closed.

Hence P does not contain any unnecessary orderings, given these two constraints. Observe that if $\mathcal{P}$ is minimal and $\mathcal{K}$ is the closure of $\mathcal{P}$ then $\mathcal{K}$ is also minimal. Afterwards, our algorithm VC-DPOR will use minimal annotated partial orders to represent different classes of the VC partitioning.

Algorithm Extend$(\mathcal{P}, X', \mathsf{val}', S', \mathsf{GoodW}')$. Let $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ be a minimal, closed annotated partial order, and $X = X_1 \cup X_2$. Consider

1. a set X' with (i) $X' \setminus X \subseteq \mathcal{W}$ or $|X' \setminus X| = 1$ and (ii) X' is inevitable for $\mathcal{P}$,
2. a value function val$'$ over X' such that val $\subseteq$ val$'$,
3. a side function S' over X' such that $S \subseteq S'$, and
4. a good-writes set GoodW$'$ over X' such that GoodW $\subseteq$ GoodW$'$.

We rely on an algorithm called Extend that constructs an *extension* of $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ to X', val$'$, S' and GoodW$'$ as a set of minimal closed annotated partial orders $\{\mathcal{K}_i = (X_1', X_2', K_i, \mathsf{val}', S', \mathsf{GoodW}')\}_i$, where $X_1' \cup X_2' = X'$. Intuitively, if σ is a linearization of $\mathcal{P}$, then for every extension σ' of σ such that $\mathcal{E}(\sigma') = X'$, $\mathsf{val}_{\sigma'} = \mathsf{val}'$ and $S_{\sigma'} = S'$, there exists some $\mathcal{K}_i$ that linearizes to σ'. In VC-DPOR, we will use Extend to extend annotated partial orders with new events.

We describe Extend for the special case where $|X' \setminus X| = 1$. When $|X' \setminus X| = q > 1$, Extend calls itself recursively for every annotated partial order of its output set on a sequence of sets $Y_1, \dots, Y_q$ where $Y_q = X'$, $Y_0 = X$ and $|Y_{i+1} \setminus Y_i| = 1$. Let $X' \setminus X = \{e\}$.

1. If thr(e) = thr$_1$ (i.e., e belongs to the root thread), the algorithm simply constructs a partial order K over the set X' such that $K|X = P$ and $e' <_K e$ for every event $e \in X'$ such that $e' <_{\mathsf{PO}} e$. Afterwards, the algorithm constructs the annotated partial order $\mathcal{K} = (X_1', X_2', K, \mathsf{val}', S', \mathsf{GoodW}')$ and returns the singleton set $\mathcal{A}_w = \{\mathsf{Closure}(\mathcal{K})\}$.
2. If thr$(e) \neq$ thr$_1$ (i.e., e belongs to the leaf threads), the algorithm first constructs a partial order K as in the previous item. Afterwards, it creates a new partial order K_i for every possible ordering of e with all events $e' \in X_2$ such that $e \bowtie e'$. Finally, the algorithm constructs the annotated partial orders $A = \mathcal{K}_i = (X_1', X_2', K_i, \mathsf{val}', S', \mathsf{GoodW}')$, and returns the set $\mathcal{A} = \{\mathsf{Closure}(\mathcal{K}_i) : \mathcal{K}_i \in A$ and $\mathsf{Closure}(\mathcal{K}_i) \neq \bot\}$.

Causal maps, guarding reads and candidate writes. A *causal map* is a map $\mathcal{C} : \mathcal{R} \to \mathscr{P} \to \mathcal{R} \cup \{\bot, \perp\!\!\!\perp\}$ such that for each read event $r \in \mathsf{dom}(\mathcal{C})$ and thread thr $\in \mathscr{P}$ we have that $\mathcal{C}(r)(\mathsf{thr}) \in \mathcal{R}_{\mathsf{thr}} \cup \{\bot, \perp\!\!\!\perp\}$. In words, $\mathcal{C}$ maps read events to functions that map every thread thr $\in \mathscr{P}$ to a read event of thr, or to some initial values $\{\bot, \perp\!\!\!\perp\}$. Given a trace σ and an event $e \in \mathcal{E}(\sigma)$, we define the *guarding read* Guard$_\sigma(e)$ of e in σ as the last read event of thr(e) that happens before e in σ, and Guard$_\sigma(e) = \bot$ if no such read event exists. Formally,

$$\mathsf{Guard}_\sigma(e) = \max_\sigma(\{r \in \mathcal{R}(\sigma|\mathsf{thr}(e)) : r <_{\mathsf{PO}} e\})$$

where we take the maximum of the empty set to be $\bot$. Given a trace σ, a causal map $\mathcal{C}$ and a read event $r \in \mathsf{enabled}(\sigma)$, we define the *candidate write set* $\mathrm{F}_\sigma^{\mathcal{C}}(r)$ of r in σ given $\mathcal{C}$ as follows:

$$\begin{aligned}\mathrm{F}_\sigma^{\mathcal{C}}(r) = \{ & w \in \mathcal{W}(\sigma) : \; r \bowtie w \quad \text{and} \\ & \text{either} \quad \mathsf{Guard}_\sigma(w) = \bot \quad \text{and} \quad \mathcal{C}(r)(\mathsf{thr}(w)) = \perp\!\!\!\perp \\ & \quad \text{or} \quad \mathsf{Guard}_\sigma(w) \neq \bot \quad \text{and also} \quad \mathcal{C}(r)(\mathsf{thr}(w)) \in \{\perp\!\!\!\perp, \bot\} \quad \text{or} \quad \mathcal{C}(r)(\mathsf{thr}(w)) <_{\mathsf{PO}} \mathsf{Guard}_\sigma(w)\end{aligned}$$

We refer to Fig. 3.5 for an illustration of the above notation, where we have a trace (Fig. 3.5a) and candidate write sets of read events given their causally-happens-before maps (Fig. 3.5b). Intuitively, $\mathcal{C}(r)(\mathsf{thr})$ encodes the prefix of the local trace of thread thr that contains write events which have already been considered by the algorithm as good writes for r. Instead of the whole prefix, we store the last read of that prefix. The two special values $\perp\!\!\!\perp$ and $\perp$ encode the empty prefix, and the prefix before the first read. The guarding read of a write w is the last local read event the same thread that appears before w in the execution so far. Hence, if the guarding read of w appears before $\mathcal{C}(r)(\text{thr})$, we know that w has been considered as a good write for r. The candidate write set for r contains writes that are considered as good writes for r in the current recursive step.

	τ_1	τ_2	τ_3
11	w_x		
12		w_y	
13		w_x	
14		r_y	
15		w_x	
16			r_y
17			w_x
18			r_x
19			w_x

(a) A trace σ. Threads thr_1 and thr_3 have enabled events r_x^1 and r_x^3 (not shown), which access the variable x.

$$\text{enabled}(\sigma) \cap \mathcal{E}_{\text{thr}_1} = r_x^1$$
$$\text{enabled}(\sigma) \cap \mathcal{E}_{\text{thr}_3} = r_x^3$$

$$\mathcal{C}(r_x^1) = \{(\text{thr}_1, \perp), (\text{thr}_2, e_4), (\text{thr}_3, e_6)\}$$
$$\mathcal{C}(r_x^3) = \{(\text{thr}_1, \perp\!\!\!\perp), (\text{thr}_2, \perp\!\!\!\perp), (\text{thr}_3, \perp\!\!\!\perp)\}$$

$$\mathrm{F}_{\sigma}^{\mathcal{C}}(r_x^1) = \{e_9\}$$
$$\mathrm{F}_{\sigma}^{\mathcal{C}}(r_x^3) = \{e_1, e_3, e_5, e_7, e_9\}$$

(b) The candidate write sets of the read events r_x^1 and r_x^3 given the causally-happens-before map $\mathcal{C}$. We denote by e_i the i-th event of σ.

Figure 3.5: Example of candidate write sets.

Algorithm 3.5: VC-DPOR($\mathcal{P} = (X_1, X_2, P, \text{val}, S, \text{GoodW}), \mathcal{C}$)

```
Input: A minimal closed annotated partial order $\mathcal{P}$, a causal map $\mathcal{C}$.
1 $\sigma' \leftarrow$ Witness($\mathcal{P}$)                        // $\mathcal{P}$ is closed hence realizable
2 $\sigma \leftarrow$ WExtend($\sigma'$)       // Extend $\sigma'$ until before the next read of each thread
3 foreach $\mathcal{Q} \in$ Extend($\mathcal{P}, \mathcal{E}(\sigma), \text{val}_\sigma, S, \text{GoodW}$) do     // Extensions of $\mathcal{P}$ to $\mathcal{E}(\sigma)$
4     $\mathcal{C}_\mathcal{Q} \leftarrow \mathcal{C}$                              // Create a copy of the CHB $\mathcal{C}$
5     ExtendRoot($\mathcal{Q}, \sigma, \mathcal{C}_\mathcal{Q}$)                   // Process the root thread
6     foreach thr $\in \mathscr{P} \setminus \{\text{thr}_1\}$ do                 // Process the leaf threads
7         ExtendLeaf($\mathcal{Q}, \sigma, \mathcal{C}_\mathcal{Q}$, thr)
```

Algorithm VC-DPOR. We are now ready to describe our main algorithm VC-DPOR for the enumerative exploration of the partitioning $\mathcal{T}^{all}/VC$. The algorithm takes as input a minimal closed annotated partial order $\mathcal{P}$ and a causal map $\mathcal{C}$. First, in Line 1 VC-DPOR calls Witness to obtain a witness σ' of $\mathcal{P}$, and in Line 2 it constructs the write-extension σ of σ' which reveals new write events in σ. Afterwards, in Line 3 the algorithm extends $\mathcal{P}$ to the set $\mathcal{E}(\sigma)$ by calling Extend. Recall that Extend returns a set of minimal closed annotated partial orders. For every annotated partial order $\mathcal{Q}$ returned by Extend, the algorithm first calls ExtendRoot in Line 5 to process the read event of the root thread thr_1 that is enabled in σ, and then the algorithm calls ExtendLeaf in Line 7 for every leaf thread $\text{thr} \neq \text{thr}_1$ to process the read event of thr that is enabled in σ. For the initial call, we construct an empty annotated partial order $\mathcal{P}$ and an initial causal map $\mathcal{C}$ that for every read event $r \in \mathcal{R}$ and thread $\text{thr} \in \mathscr{P}$ maps $\mathcal{C}(r)(\text{thr}) = \{\perp\!\!\!\perp\}$.

Algorithm ExtendRoot. The algorithm takes as input a minimal closed annotated partial order $\mathcal{Q}$, a trace σ and a causal map $\mathcal{C}_\mathcal{Q}$, and attempts all possible extensions of $\mathcal{Q}$ with the

Algorithm 3.6: ExtendRoot($\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW}), \sigma, \mathcal{C}_{\mathcal{Q}}$)

```
Input: A minimal closed annotated partial order $\mathcal{Q}$, a trace $\sigma$, a causal map $\mathcal{C}_{\mathcal{Q}}$.
1  $r \leftarrow \mathsf{enabled}(\sigma, \mathsf{thr}_1)$                  // The next enabled event in $\mathsf{thr}_1$ is a read
2  $Y_1 \leftarrow \mathrm{F}^{\mathcal{C}_{\mathcal{Q}}}_{\sigma}(r) \cap \mathcal{W}_{\mathsf{thr}_1}$                  // The set of local candidate writes of $r$
3  $Y_2 \leftarrow \mathrm{F}^{\mathcal{C}_{\mathcal{Q}}}_{\sigma}(r) \cap \mathcal{W}_{\neq \mathsf{thr}_1}$                  // The set of remote candidate writes of $r$
4  foreach $i \in [2]$ do                  // $i=1$ ($i=2$) reads from local (remote) writes
5      $S_r \leftarrow S \cup \{(r, i)\}$                  // The new side function
6      $\mathcal{D}_r \leftarrow \{\mathsf{val}_{\sigma}(w) :\ w \in Y_i\}$                  // The set of values of candidate writes of $r$
7      foreach $v \in \mathcal{D}_r$ do                  // Every value $v$ that $r$ may read
8          $\mathsf{val}_r \leftarrow \mathsf{val}_{\sigma} \cup \{(r, v)\}$                  // The new value function
9          $\mathsf{GoodW}_r \leftarrow \mathsf{GoodW} \cup \{(r, \{w \in Y_i :\ \mathsf{val}_{\sigma}(w) = v\})\}$                  // The new good-writes
10         $\mathcal{K} \leftarrow \mathsf{Extend}(\mathcal{Q}, X_1 \cup X_2 \cup \{r\}, \mathsf{val}_r, S_r, \mathsf{GoodW}_r)$                  // Returns one element
11         if $\mathcal{K} \neq \bot$ then                  // Extension is successful
12             Call $\mathrm{VC\text{-}DPOR}(\mathcal{K}, \mathcal{C}_{\mathcal{Q}})$                  // Recurse
13 $\mathcal{C}_{\mathcal{Q}}(r) \leftarrow \{(\mathsf{thr}, \max_{\sigma}(\{\mathcal{R}(\sigma)|\mathsf{thr}\})) :\ \mathsf{thr} \in \mathscr{P}\}$ // The last read of each thread in $\sigma$
```

read event r of thr_1 that is enabled in σ to all possible values that are written in σ. The algorithm first in Line 2 and Line 3 constructs two sets Y_1 and Y_2 which hold the local and remote, respectively, write events of σ that are candidate writes for r according to the causal map $\mathcal{C}_{\mathcal{Q}}$. Then, it iterates over the local ($i = 1$) and remote ($i = 2$) write choices for r in Y_i. Finally, the algorithm (i) collects all possible values that r may read from the set Y_i (Line 6), (ii) constructs the appropriate new side function, value function and good-writes function (Lines 5, 8 and 9), and (iii) calls Extend on these new parameters in Line 10, in order to establish the respective extension for r. For every such case, Extend returns a new minimal, closed annotated partial order $\mathcal{K}$ which is passed recursively to $\mathrm{VC\text{-}DPOR}$ in Line 12.

Algorithm 3.7: ExtendLeaf($\mathcal{Q} = (X_1, X_2, Q, \mathsf{val}, S, \mathsf{GoodW}), \sigma, \mathcal{C}_{\mathcal{Q}}, \mathsf{thr}$)

```
Input: Minimal closed annotated partial order $\mathcal{Q}$, trace $\sigma$, causal map $\mathcal{C}_{\mathcal{Q}}$, thread thr.
1  $r \leftarrow \mathsf{enabled}(\sigma, \mathsf{thr})$                  // The next enabled event in thr is a read
2  $\mathcal{D}_r \leftarrow \{\mathsf{val}_{\sigma}(w) :\ w \in \mathrm{F}^{\mathcal{C}_{\mathcal{Q}}}_{\sigma}(r)\}$                  // The set of values of candidate writes of $r$
3  foreach $v \in \mathcal{D}_r$ do                  // Every value $v$ that $r$ may read
4      $\mathsf{val}_r \leftarrow \mathsf{val}_{\sigma} \cup \{(r, v)\}$                  // The new value function
5      $\mathsf{GoodW}_r \leftarrow \mathsf{GoodW} \cup \{(r, \{w \in \mathrm{F}^{\mathcal{C}_{\mathcal{Q}}}_{\sigma}(r) :\ \mathsf{val}_{\sigma}(w) = v\})\}$                  // The new good-writes
6      foreach $\mathcal{K} \in \mathsf{Extend}(\mathcal{Q}, X_1 \cup X_2 \cup \{r\}, \mathsf{val}_r, S, \mathsf{GoodW}_r)$ do // Returns many elements
7          Call $\mathrm{VC\text{-}DPOR}(\mathcal{K}, \mathcal{C}_{\mathcal{Q}})$                  // Recurse
8  $\mathcal{C}_{\mathcal{Q}}(r) \leftarrow \{(\mathsf{thr}, \max_{\sigma}(\{\mathcal{R}(\sigma)|\mathsf{thr}\})) :\ \mathsf{thr} \in \mathscr{P}\}$ // The last read of each thread in $\sigma$
```

Algorithm ExtendLeaf. The algorithm ExtendLeaf takes as input a minimal closed partial order $\mathcal{Q}$, a trace σ, a causal map $\mathcal{C}_{\mathcal{Q}}$, and a leaf thread $\mathsf{thr} \in \mathscr{P} \setminus \{\mathsf{thr}_1\}$. Similarly to ExtendRoot, ExtendLeaf attempts all possible extensions of $\mathcal{Q}$ with the read event r of thr that is enabled in σ to all possible values that are written in σ. The main difference compared to ExtendRoot is that since r belongs to a leaf thread, Extend returns a set of minimal, closed annotated partial orders (as opposed to just one) which result from all possible orderings of r with the write events of X_2 that are conflicting with r. Then ExtendLeaf makes a recursive call to $\mathrm{VC\text{-}DPOR}$ for each such annotated partial order.

Example ($\mathrm{VC\text{-}DPOR}$). Fig. 3.6 illustrates the main aspects of $\mathrm{VC\text{-}DPOR}$ (Algorithms 3.5, 3.6, and 3.7) on a small example. We start with an empty annotated partial order $\mathcal{P}$ and a

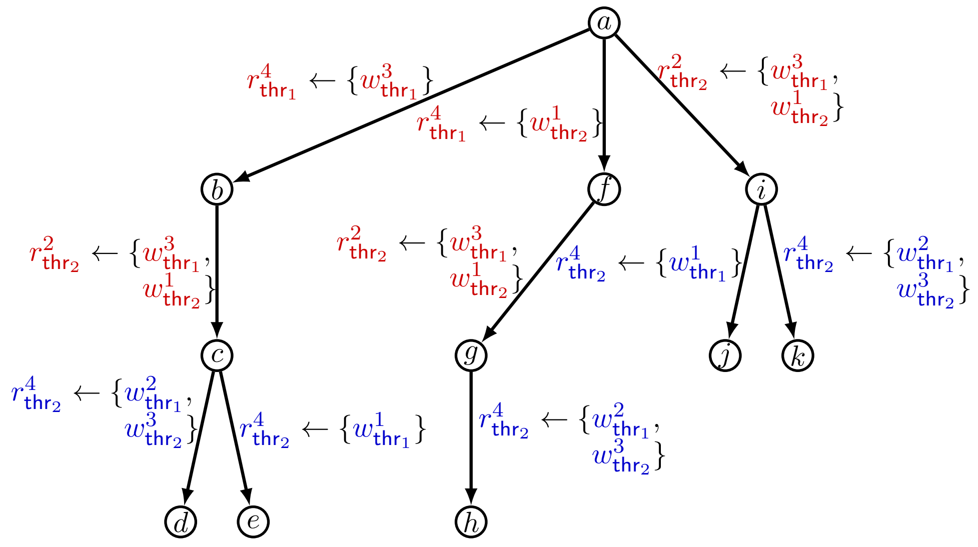
Thread thr_1 :

1. $w(y, 1)$
2. $w(y, 2)$
3. $w(x, 1)$
4. $r(x)$

Thread thr_2 :

1. $w(x, 1)$
2. $r(x)$
3. $w(y, 2)$
4. $r(y)$

(a) Program $\mathscr{P}$.



a
$r^4_{\mathsf{thr}_1} \leftarrow \{w^3_{\mathsf{thr}_1}\}$
$r^4_{\mathsf{thr}_1} \leftarrow \{w^1_{\mathsf{thr}_2}\}$
$r^2_{\mathsf{thr}_2} \leftarrow \{w^3_{\mathsf{thr}_1}, w^1_{\mathsf{thr}_2}\}$
b
f
i
$r^2_{\mathsf{thr}_2} \leftarrow \{w^3_{\mathsf{thr}_1}, w^1_{\mathsf{thr}_2}\}$
$r^2_{\mathsf{thr}_2} \leftarrow \{w^3_{\mathsf{thr}_1}, w^1_{\mathsf{thr}_2}\}$
$r^4_{\mathsf{thr}_2} \leftarrow \{w^1_{\mathsf{thr}_1}\}$
$r^4_{\mathsf{thr}_2} \leftarrow \{w^2_{\mathsf{thr}_1}, w^3_{\mathsf{thr}_2}\}$
c
g
j
k
$r^4_{\mathsf{thr}_2} \leftarrow \{w^2_{\mathsf{thr}_1}, w^3_{\mathsf{thr}_2}\}$
$r^4_{\mathsf{thr}_2} \leftarrow \{w^1_{\mathsf{thr}_1}\}$
$r^4_{\mathsf{thr}_2} \leftarrow \{w^2_{\mathsf{thr}_1}, w^3_{\mathsf{thr}_2}\}$
d
e
h

(b) The corresponding VC-DPOR exploration tree.

Figure 3.6: VC-DPOR exploration example.

causal map $\mathcal{C}$ that is empty (i.e., $\mathcal{C}(r)(\mathsf{thr}) = \{\perp\!\!\!\perp\}$ for every read event $r \in \mathcal{R}$ and thread $\mathsf{thr} \in \mathscr{P}$). The initial trace obtained in Line 1 of Algorithm 3.5 is $\sigma' = \varepsilon$. Its write-extension σ in Line 2 contains the three writes of thr_1 and the first write of thr_2. Next, Line 3 returns an annotated partial order $\mathcal{Q}_a$ that corresponds to the program order $\mathsf{PO}|\mathcal{E}(\sigma)$. In σ, the root thread thr_1 has an enabled event (which is always a read), so ExtendRoot (Algorithm 3.6) is called on $\mathcal{Q}_a$ and the (empty) causal map $\mathcal{C}_{\mathcal{Q}_a}$. (†)

The enabled read in Line 1 is $r^4_{\mathsf{thr}_1}$, its local candidate write (computed in Line 2) is $w^3_{\mathsf{thr}_1}$ and its remote candidate write (computed in Line 3) is $w^1_{p_2}$. This holds because $\mathcal{C}_{\mathcal{Q}_a}(r)(\mathsf{thr}_1) = \{(\mathsf{thr}_1, \perp\!\!\!\perp), (\mathsf{thr}_2, \perp\!\!\!\perp)\}$, which allows any write event to be observed. For the local (Line 4, $i = 1$) candidate $w^3_{\mathsf{thr}_1}$, first the side function is updated with $\{(r^4_{\mathsf{thr}_1}, 1)\}$ in Line 5. Then in Line 6, the only considered value is 1. Thus, in Line 8 the value function is updated with $\{(r^4_{p_1}, 1)\}$, and in Line 9 the good-writes function is updated with $\{(r^4_{\mathsf{thr}_1}, \{w^3_{\mathsf{thr}_1}\})\}$. Then, such an update is successfully realized in Line 10 by Extend, where the partial order is extended with $r^4_{\mathsf{thr}_1}$ and afterwards it is closed using algorithm Closure (Algorithm 3.1). Thus VC-DPOR (Algorithm 3.5) is recursively called on the corresponding annotated partial order $\mathcal{K}_a$ (and the empty causal map $\mathcal{C}_{\mathcal{Q}_a}$), and we proceed to the child b of a.

In node b, no new event is added during the write-extension (Line 2), as $r^4_{p_1}$ is the last event of thr_1, and in Line 3 we obtain $\mathcal{Q}_b$. The only thread with an enabled read event is thr_2, so ExtendLeaf (Algorithm 3.7) is called on $\mathcal{Q}_b$ and thr_2 (and empty causal map $\mathcal{C}_{\mathcal{Q}_b}$). The enabled read $r^2_{\mathsf{thr}_2}$ has candidate writes $w^3_{p_1}$ and $w^1_{p_2}$, both of which write the same value (c.f. Line 2), and hence the algorithm will allow $r^2_{\mathsf{thr}_2}$ to observe either. This is an example of the value-centric gains we obtain in this work. In Line 4 the value function is updated with $\{(r^2_{\mathsf{thr}_2}, 1)\}$, and in Line 5 the good-writes function is updated with $\{(r^2_{\mathsf{thr}_2}, \{w^3_{\mathsf{thr}_1}, w^1_{\mathsf{thr}_2}\})\}$. The realization of this update happens in Line 6 by Extend, where the partial order is extended with $r^2_{\mathsf{thr}_2}$ and then closed using algorithm Closure (Algorithm 3.1). One annotated partial order $\mathcal{K}_b$ is returned and it is the argument of the further VC-DPOR call (with an empty causal map $\mathcal{C}_{\mathcal{Q}_b}$), we proceed to the child c of b. In node c, the write-extension adds the event

$w^3_{\mathsf{thr}_2}$, which, in similar steps as before, will lead to nodes d and e.

Next, the recursion backtracks to the call of ExtendRoot in the node a (†). The second iteration $(i = 2)$ of the loop in Line 4 proceeds, where the remote candidate write $w^1_{\mathsf{thr}_2}$ is considered for $r^4_{\mathsf{thr}_1}$. In a similar fashion, the descendants f, g, and h are created and h concludes with a maximal trace.

Finally, the recursion backtracks to the node a again, where ExtendRoot (†) concludes with updating the causal map as follows: $\mathcal{C}_{\mathcal{Q}_a}(r^4_{\mathsf{thr}_1}) = \{(\mathsf{thr}_1, \bot),\ (\mathsf{thr}_2, \bot)\}$. The control-flow comes back to the initial $\mathrm{VC\text{-}DPOR}$ call (from Line 5), where the annotated partial order $\mathcal{Q}_a$ with the (now updated) causal map $\mathcal{C}_{\mathcal{Q}_a}$ is considered. The thread thr_2 has an enabled read $(r^2_{\mathsf{thr}_2})$ in σ, hence ExtendLeaf is called on $\mathcal{Q}_a$, $\mathcal{C}_{\mathcal{Q}_a}$, and thr_2. Eventually, the descendants i, j, and k are created and the exploration concludes. Note that in each of i, j, k, the thread thr_1 has an enabled read $r^4_{\mathsf{thr}_1}$. However, note that $\mathsf{Guard}_\sigma(w^3_{\mathsf{thr}_1}) = \mathsf{Guard}_\sigma(w^1_{\mathsf{thr}_2}) = \bot$ and in all those nodes we have $\mathcal{C}(r^4_{\mathsf{thr}_1})(\mathsf{thr}_1) = \{(\mathsf{thr}_1, \bot), (\mathsf{thr}_2, \bot)\}$, and thus $w^3_{p\mathsf{thr}1}$ and $w^1_{\mathsf{thr}_2}$ are never considered as candidate writes for $r^4_{\mathsf{thr}_1}$. This illustrates how $\mathrm{VC\text{-}DPOR}$ never explores the same class of VC twice.

3.3.1 Properties of $\mathrm{VC\text{-}DPOR}$

The following theorem states the main result of this chapter.

Theorem 3.3. *Consider a concurrent program $\mathscr{P}$ over a constant number of threads, and let $\mathcal{T}^{max}$ be the maximal trace space of $\mathscr{P}$. $\mathrm{VC\text{-}DPOR}$ solves the local-state reachability problem on $\mathscr{P}$ and requires $O\left(|\mathcal{T}^{max}/VC| \cdot \mathsf{poly}(n)\right)$ time, where n is the length of the longest trace in $\mathcal{T}^{max}$.*

We prove the above theorem by establishing a sequence lemmas. The proof concepts rely on the tree T induced by the recursive calls of $\mathrm{VC\text{-}DPOR}$. We start with introducing the tree T and proceed with the correctness and complexity statements of $\mathrm{VC\text{-}DPOR}$.

The induced tree T. An execution of $\mathrm{VC\text{-}DPOR}$ induces a tree T, where each node u is labeled with an annotated partial order $\mathcal{P}^u$ constructed at some recursive step by the algorithm. We have two types of nodes.

1. A type 1 node u is labeled with an annotated partial order $\mathcal{P}^u$ such that $\mathcal{P}^u$ was passed as an argument to a recursive call of $\mathrm{VC\text{-}DPOR}$. These nodes correspond to all annotated partial orders returned by the algorithm Extend when invoked from within ExtendRoot or ExtendLeaf.
2. A type 2 node u is labeled with with an annotated partial order $\mathcal{P}^u$ such that $\mathcal{P}^u$ was not passed as an argument to $\mathrm{VC\text{-}DPOR}$. These nodes correspond to all annotated partial orders returned by the algorithm Extend when invoked from within $\mathrm{VC\text{-}DPOR}$.

We will use the induced tree T to reason about the correctness and complexity of $\mathrm{VC\text{-}DPOR}$. Note that for every node u of the tree T, the annotated partial order $\mathcal{P}^u$ is closed and minimal.

Correctness of Extend. We first prove correctness of the algorithm Extend that constructs extensions of annotated partial orders in $\mathrm{VC\text{-}DPOR}$.

Lemma 3.5. *Let $\mathcal{A} = \mathsf{Extend}(\mathcal{P}, X', \mathsf{val}', S', \mathsf{GoodW}')$. Then* Extend *runs in $O(m \cdot \mathsf{poly}(n))$ time, where $n = |X'|$ and $m = |\mathcal{A}| + 1$, and the following assertions hold.*

1. *Every annotated partial order $\mathcal{K}_i$ is closed and minimal.*
2. *For every pair K_i, K_j, we have that $K_i \not\preccurlyeq K_j$ and $K_j \not\preccurlyeq K_i$.*
3. *For every trace σ such that (i) $\mathcal{E}(\sigma) = X'$, (ii) for each read event $r \in \mathcal{R}(\sigma)$ we have $\mathsf{RF}_\sigma(r) \in \mathsf{GoodW}'$ and (iii) $(\sigma|X) \preccurlyeq P$, there exists an annotated partial order $\mathcal{K}_i$ such that σ is a linearization of $\mathcal{K}_i$.*

Proof. We argue separately about correctness and complexity.

Correctness. We first argue about the correctness of the algorithm, i.e., the assertions in Item 1-Item 3 above.

1. This assertion is an immediate consequence of the facts that (i) $\mathcal{P}$ is closed and minimal, (ii) Extend constructs each annotated partial order simply by ordering conflicting events that belong to the leaf threads, and (iii) the closure of a minimal annotated partial order is also minimal.
2. This assertion holds trivially by construction.
3. Since $\mathcal{P}$ is minimal, Extend creates an annotated partial order $\mathcal{Q} = (X'_1, X'_2, Q, \mathsf{val}', S', \mathsf{GoodW}')$ such that $\sigma \preccurlyeq Q$ and $\mathcal{Q}$ is also minimal. Observe that $\mathcal{Q}$ is feasible, since $\sigma \sqsubseteq Q$ and $(X'_1, X'_2, \sigma, \mathsf{val}', S'\mathsf{GoodW}')$ is closed. Thus the algorithm will construct $\mathcal{K}_i = \mathsf{Closure}(\mathcal{Q})$ and include $\mathcal{K}_i$ in $\mathcal{A}$.

Complexity. Since the number of threads is constant, for every recursive call of Extend, Item 2 of the algorithm creates $O(\mathsf{poly}(n))$ partial orders K_i, and since computing the closure of K_i requires $O(\mathsf{poly}(n))$ time, we have that Extend spends $O(\mathsf{poly}(n))$ in each recursive call. It follows that constructing the whole set $\mathcal{A}$ takes $O(m \cdot \mathsf{poly}(n))$ time, since m is the size of the output (i.e., the number of leaves in the recursion) and every recursive step takes $O(\mathsf{poly}(n))$ time.

□

Correctness of VC-DPOR. We now turn our attention to the correctness of VC-DPOR. We will argue that for every target trace σ^*, the algorithm discovers the value function val_{σ^*}. In particular, the induced tree T has a node u such that $\mathcal{P}^u$ is of the form $\mathcal{P}^u = (X_1, X_2, P, \mathsf{val}_{\sigma^*}, S, \mathsf{GoodW})$, i.e., the value function of $\mathcal{P}^u$ is the value function of the target trace σ^*. In our discussion below, we fix such a target σ^* and introduce some notation around it.

Compatible and witness nodes. An annotated partial order $\mathcal{P} = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ is called *compatible* with σ^* if the following conditions hold. Let $X = X_1 \cup X_2$.

1. $X \subseteq \mathcal{E}(\sigma^*)$, $\mathsf{val} \subseteq \mathsf{val}_{\sigma^*}$, $S \subseteq S_{\sigma^*}$ and $(\sigma^*|X) \preccurlyeq P$.
2. For every read event $r \in \mathcal{R}(X)$ we have that $\mathsf{RF}_{\sigma^*}(r) \in \mathsf{GoodW}(r)$.

A node u of the induced tree T is called *compatible* with σ^* if $\mathcal{P}^u$ is compatible with σ^*. We call u a *witness* if $\mathsf{val}^u = \mathsf{val}_\sigma$, where val^u is the value function of the annotated partial order $\mathcal{P}^u$. Observe that if u is compatible with σ^* then every ancestor of u is also compatible with σ^*.

Left and leftmost movers. Consider a node u of the induced tree T such that u is compatible with σ^*. A child z of u in T is called a *left mover* if

1. z is compatible with σ^* and
2. z is the first child of u with this property, in the order the execution of $\mathrm{VC\text{-}DPOR}$.

We call u a *leftmost mover* if u and every ancestor of u (except for the root of T) is a left mover. The correctness of $\mathrm{VC\text{-}DPOR}$ is based on the following lemma.

Lemma 3.6. *If u is a leftmost mover then either u is a witness or u has a child that is a leftmost mover.*

Proof. Assume that u is not a witness and we argue that u has a child z such that z is compatible with σ^*. Since u is a leftmost mover, it will follow that u has a child that is a leftmost mover. We split cases based on whether u is a type 1 or type 2 node.

The node u is a type 1 node. Recall that $\mathcal{P}^u = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$ is a minimal, closed annotated partial order. Consider the trace σ constructed by $\mathrm{VC\text{-}DPOR}$ in Line 2, and observe that $\mathcal{E}(\sigma) \subseteq \mathcal{E}(\sigma^*)$, $\mathsf{val}_\sigma \subseteq \mathsf{val}_{\sigma^*}$ and $S_\sigma \subseteq S_{\sigma^*}$. Consider the trace $\overline{\sigma} = \sigma^*|\mathcal{E}(\sigma)$, and observe that (i) for every read event $r \in \mathcal{R}(\overline{\sigma})$ we have $\mathsf{RF}_{\overline{\sigma}}(r) \in \mathsf{GoodW}(r)$, and (ii) $\overline{\sigma} \preccurlyeq P$. By Lemma 3.5, Extend in Line 3 returns an annotated partial order $\mathcal{K}_i$ such that $\overline{\sigma}$ is a linearization of $\mathcal{K}_i$. We associate z with $\mathcal{K}_i$.

The node u is a type 2 node. Consider any linearization σ of $\mathcal{P}^u = (X_1, X_2, P, \mathsf{val}, S, \mathsf{GoodW})$, and since u is compatible with σ^*, for every read event r that is enabled in σ we have that $r \in \mathcal{E}(\sigma^*)$. In addition, there exists a read event r that is enabled in σ and $\mathsf{RF}_{\sigma^*}(r) \in \mathcal{E}(\sigma)$. Let $w = \mathsf{RF}_{\sigma^*}(r)$, and we argue that $w \in \mathrm{F}_\sigma^{\mathcal{C}^u}(r)$. We distinguish between the following cases.

1. $\mathcal{C}^u = \perp\!\!\!\perp$. Then by definition, $w \in \mathrm{F}_\sigma^{\mathcal{C}^u}(r)$.
2. $\mathcal{C}^u = \perp$ or $\mathcal{C}^u \in \mathcal{R}$. Then, there exists a type 2 ancestor q of u and a trace σ_q that is a linearization of $\mathcal{P}^q = (X_1', X_2', P', \mathsf{val}', S', \mathsf{GoodW}')$, r is enabled in σ^q and $w \in \mathrm{F}_{\sigma^q}^{\mathcal{C}^q}(r)$. It is straightforward to see that at that point the algorithm extended $\mathcal{P}^q$ with r and a good-writes function GoodW^q such that $w \in \mathsf{GoodW}^q(r)$. A similar analysis as in the previous item shows that Extend returned an annotated partial order $\mathcal{P}'$ that is compatible. In addition, $\mathcal{P}'$ is associated with a node of T that is a child of q and that was visited before the ancestor of u which is also a child of q . This contradicts the fact that u is a leftmost mover. It follows that $w \in \mathrm{F}_\sigma^{\mathcal{C}^u}(r)$. The rest follows by Lemma 3.5, similar to the previous case.

The desired result follows. □

Lemma 3.7. *For every pair of traces σ_1', σ_2' constructed by* $\mathrm{VC\text{-}DPOR}$ *in Line 1 in two recursive calls, we have that $\sigma_1' \not\sim_{VC} \sigma_2'$.*

Proof. Consider the nodes u_1, u_2 of the induced tree T that correspond to the recursive calls in which $\mathrm{VC\text{-}DPOR}$ constructed the traces σ'_1 and σ'_2, respectively. If u_i is ancestor of u_{3-i}, for some $i \in [2]$, then clearly $\mathcal{E}(\sigma'_i) \neq \mathcal{E}(\sigma'_{3-i})$. Otherwise, let u be the lowest common ancestor of u_1 and u_2 in T. For each $i \in [2]$, let z_i be the child of u that is also an ancestor of u_i, and let $\mathcal{P}^{z_i} = (X^i_1, X^i_2, P^i, \mathsf{val}^i, S^i, \mathsf{GoodW}^i)$ be the annotated partial order that labels node z_i. We distinguish between the following cases.

1. If z is a type 2 node, then $\mathcal{P}^{z_i}$ only differ on P^i. By Lemma 3.5, there exists a pair of events $e_1, e_2 \in X^1_2$ such that (i) $e_1 \bowtie e_2$ and (ii) $e_1 \hookrightarrow_{P^1} e_2$ and $e_2 \hookrightarrow_{P^2} e_1$. It follows that $e_1 \hookrightarrow_{\sigma'_1} e_2$ and $e_2 \hookrightarrow_{\sigma'_2} e_1$, and since $e_1, e_2 \in \mathcal{E}_{\neq \mathsf{thr}_1}$, we have that $\sigma'_1 \not\sim_{VC} \sigma'_2$.
2. If z is a type 1 node, let σ' be the trace constructed in Line 1 by the recursive call to $\mathrm{VC\text{-}DPOR}$ for node z. We distinguish between the following cases.
    a) If $\mathcal{P}^{z_1}$ and $\mathcal{P}^{z_2}$ occur from the same invocation to Extend, then the proof is similar to the previous item.
    b) If $\mathcal{P}^{z_1}$ and $\mathcal{P}^{z_2}$ occur from different invocations to Extend, we examine whether both $\mathcal{P}^{z_1}$ and $\mathcal{P}^{z_2}$ were constructed by extending to the same read event r or not. In the former case, we examine the values $\mathsf{val}^1(r)$ and $\mathsf{val}^2(r)$ that r was forced to read. If $\mathsf{val}^1(r) \neq \mathsf{val}^2(r)$ then $\mathsf{val}_{\sigma'_1}(r) \neq \mathsf{val}_{\sigma'_2}$, whereas if $\mathsf{val}^1(r) = \mathsf{val}^2(r)$ then $\mathsf{thr}(r) = \mathsf{thr}_1$ and $S^1(r) \neq S^2(r)$ and thus $S_{\sigma'_1}(r) \neq S_{\sigma'_2}(r)$. We are left with the case where $\mathcal{P}^{z_1}$ and $\mathcal{P}^{z_2}$ were constructed by extending to two different read events r_1 and r_2, respectively. Assume wlog that $\mathcal{P}^{z_2}$ was constructed after $\mathcal{P}^{z_1}$. If $\mathsf{val}_{\sigma'_2}(r_1) \neq \mathsf{val}_{\sigma'_1}(r_1)$ we are done. Otherwise, let $r = \mathsf{Guard}_{\sigma'_2}(\mathsf{RF}_{\sigma}(r_1))$ and note that $r \mapsto_{\sigma'_2} r_1$. Due to the causally-happens-before map $\mathcal{C}$ in that recursive call of $\mathrm{VC\text{-}DPOR}$, we have that $r \notin \mathcal{E}(\sigma')$ and thus $r \not\mapsto_{\sigma'_1} r_1$.

In all cases, we have $\sigma'_1 \not\sim_{VC} \sigma'_2$, as desired. □

Lemma 3.8. $\mathrm{VC\text{-}DPOR}$ *runs in time* $O\left(|\mathcal{T}^{max}/VC| \cdot \mathsf{poly}(n)\right)$, *where* n *is the length of the longest trace in* $\mathcal{T}^{max}$.

Proof. Consider two maximal traces $\sigma_1, \sigma_2 \in \mathcal{T}^{max}$ such that $\sigma_1 \sim_{VC} \sigma_2$. Let σ'_1, σ'_2 be prefixes of σ_1, σ_2, respectively, such that $\mathcal{E}(\sigma'_1) = \mathcal{E}(\sigma'_2)$, and observe that $\sigma'_1 \sim_{VC} \sigma'_2$. Since we have constantly many threads, it follows that given a maximal trace σ, there exist $O(\mathsf{poly}(n))$ different sets $X \subseteq \mathcal{E}(\sigma)$ for which there exists a trace σ' such that (i) $\mathcal{E}(\sigma') = X$ and (ii) σ is a maximal extension of σ'. It follows that $|\mathcal{T}^{all}/VC| = O\left(|\mathcal{T}^{max}/VC| \cdot \mathsf{poly}(n)\right)$, and thus it suffices to argue that $\mathrm{VC\text{-}DPOR}$ runs in time $O\left(|\mathcal{T}^{all}/VC| \cdot \mathsf{poly}(n)\right)$. By Lemma 3.7, for every pair of traces σ'_1 and σ'_2 constructed by $\mathrm{VC\text{-}DPOR}$ in Line 1, we have that $\sigma'_1 \not\sim_{VC} \sigma'_2$, and hence each such trace falls into a different class of $\mathcal{T}^{all}/VC$. Thus the size of the induced tree T is bounded by $|\mathcal{T}^{all}/VC|$. For every internal node u of T, the children of u in T are produced by $O(\mathsf{poly}(n))$ calls to Extend, which, by Lemma 3.5, requires $O(\mathsf{poly}(n))$ time per child of u. Hence the total time spent by $\mathrm{VC\text{-}DPOR}$ is

$$O(|T| \cdot \mathsf{poly}(n)) = O\left(|\mathcal{T}^{all}/VC| \cdot \mathsf{poly}(n)\right) = O\left(|\mathcal{T}^{max}/VC| \cdot \mathsf{poly}(n)\right).$$

The desired result follows.

□

Finally, the results of Lemma 3.6 and Lemma 3.8 together provide proof of Theorem 3.3, as desired. We conclude with two remarks on space usage and the way lock events can be handled.

Remark 3.3 (Space complexity)**.** To make our presentation simpler so far, $\mathrm{VC\text{-}DPOR}$ and ExtendLeaf iterate over the set of annotated partial orders returned by Extend, which can be exponentially large. An efficient variant of $\mathrm{VC\text{-}DPOR}$ shall explore these sets recursively, instead of computing all elements of each set imperatively. This results in polynomial space complexity for $\mathrm{VC\text{-}DPOR}$.

Remark 3.4 (Handling locks)**.** For simplicity of presentation, so far we have neglected locks. However, lock events can be incorporated naturally, as follows.

1. Each lock-release event is a write event, writing an arbitrary value.
2. Each lock-acquire event is a read event. Given two lock-acquire events r_1, r_2 the algorithm maintains that $\mathsf{GoodW}(r_1) \cap \mathsf{GoodW}(r_2) = \emptyset$

3.4 Experiments

We have established that VC is a coarse partitioning that can be explored efficiently by $\mathrm{VC\text{-}DPOR}$. In this section we present an experimental evaluation of $\mathrm{VC\text{-}DPOR}$ on various classes of concurrent benchmarks, to assess

1. the reduction of the trace-space partitioning achieved by VC, and
2. the efficiency with which this partitioning is explored by $\mathrm{VC\text{-}DPOR}$.

Implementation and experiments. To address the above questions, we have made a prototype implementation of $\mathrm{VC\text{-}DPOR}$ in the stateless model checker Nidhugg [AAA$^+$15], which works on LLVM IR. We have tested $\mathrm{VC\text{-}DPOR}$ on benchmarks coming in four classes:

1. The TACAS Software Verification Competition (SV-COMP).
2. Mutual-exclusion algorithms from the literature.
3. Multi-threaded dynamic-programming algorithms that use memoization.
4. Individual benchmarks that exercise various concurrency patterns.

Each benchmark comes with a scaling parameter, which is either the number of threads, or an unroll bound on all loops of the benchmark (often the unroll bound also controls the number of threads that are spawned.) We have compared our algorithm with three other state-of-the-art SMC algorithms that are implemented in Nidhugg, namely Nidhugg/source [AAJS14], Optimal [AAJS14] and Optimal* ("optimal with observers") [AJLS18], as well as our own implementation of $\mathrm{DC\text{-}DPOR}$ [CCP$^+$17].

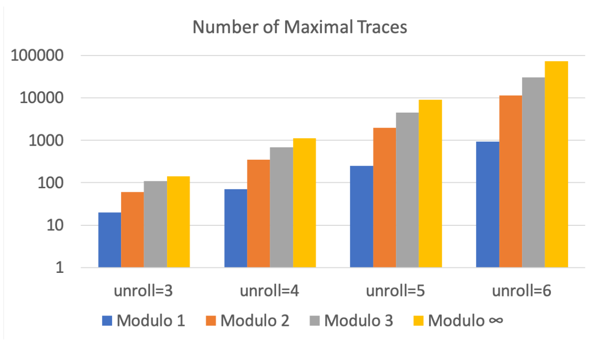
Technical details. For our experiments we have used a Linux machine with Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz (12 CPUs) and 128GB of RAM. We have run Nidhugg with

Clang and LLVM version 3.8. In all cases, we report the number of maximal traces and the total running time of each algorithm, subject to a timeout of 4 hours, indicated by "-".

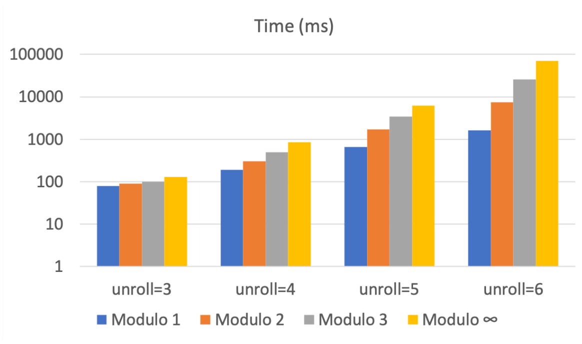
Implementation details. Here we clarify some details regarding our implementation.

1. The root thread is chosen as the first thread that is spawned from the main thread. We make this choice instead of the main thread as in many benchmarks, the main thread mainly spawns worker threads and performs only a few concurrent operations.

2. In our presentation of Extend$(\mathcal{P}, X', \mathsf{val}', S', \mathsf{GoodW}')$, given $X' \setminus X = \{e\}$ such that e belongs to a leaf thread, we consider all possible orderings of e with conflicting events from all leaf threads. In our implementation, we relax this in two ways. Given a write event e_w, we say it is *never-good* if it does not belong to $\mathsf{GoodW}'(r)$ for any read event r. Further, given e_w and an annotated partial order $\mathcal{K}$, we say that e_w is *unobservable* in $\mathcal{K}$, if for every linearization of $\mathcal{K}$ no read event can observe e_w. Given two unordered conflicting write events from leaf threads, we do not order them if (i) both are never-good, or (ii) at least one is unobservable.



(a) Number of traces.



(b) Running time.

Figure 3.7: VC-DPOR on variants of the `fib_bench` benchmark.

Value-centric gains. As a preliminary experimental step, we explore the gains of our value-centric technique on small variants of the simple benchmark `fib_bench` from SV-COMP. This benchmark consists of a main thread and two worker threads, and two global variables x and y. The first worker thread enters a loop in which it performs the update $x \leftarrow x + y$. Similarly, the second worker thread enters a loop in which it performs the update $y \leftarrow y + x$. To explore the sensitivity of our value-centric approach to values, we have created three variants `fib_bench_1`, `fib_bench_2`, `fib_bench_3` of the main benchmark. In variant `fib_bench_i` each worker thread performs the addition modulo i. Hence, the first and the second worker performs the update $x \leftarrow (x + y) \mod i$ and $y \leftarrow (y + x) \mod i$, respectively. For smaller values of i, we expect more write events to write the same value, and thus VC-DPOR to benefit both in terms of the traces explored and the running time. Although simple, this experiment serves the purpose of quantifying the value-centric gains of VC-DPOR in a controlled benchmark. Fig. 3.7 depicts the obtained results for the three variants of `fib_bench`, where Modulo $= \infty$ represents the original benchmark (i.e., without the modulo operation). We see that indeed, as i gets smaller, VC-DPOR benefits significantly in both number of traces and running time. Moreover, this benefit gets amplified with higher unroll bounds.

Benchmark	Maximal Traces					Time				
	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR
parker(6)	**38670**	1100917	1100917	1023567	985807	**1m29s**	23m5s	24m29s	24m54s	46m41s
parker(7)	**52465**	1735432	1735432	1613807	1554237	**2m23s**	41m28s	44m41s	45m13s	1h27m
parker(8)	**68360**	2576147	2576147	2395947	2307467	**3m35s**	1h9m	1h15m	1h17m	2h29m
27_Boop(6)	**248212**	35079696	35079696	4750426	1468774	**3m26s**	2h54m	2h49m	26m22s	12m33s
27_Boop(7)	**420033**	-	-	10134616	2874202	**6m33s**	-	-	1h0m	27m21s
27_Boop(8)	**677870**	-	-	20003512	5268064	**11m54s**	-	-	2h7m	56m13s
30_Fun_Point(6)	**5040**	665280	665280	665280	665280	**5.52s**	4m2s	4m14s	4m36s	1m34s
30_Fun_Point(7)	**40320**	17297280	17297280	17297280	17297280	**57.50s**	2h7m	2h15m	2h29m	51m46s
30_Fun_Point(8)	**362880**	-	-	-	-	**10m51s**	-	-	-	-
45_monabsex(5)	**600**	14400	14400	9745	6197	**0.44s**	2.28s	2.36s	1.86s	1.50s
45_monabsex(6)	**13152**	518400	518400	291546	180126	**14.93s**	1m41s	1m41s	1m5s	1m0s
45_monabsex(7)	**423360**	25401600	25401600	11710405	7073803	**13m30s**	1h43m	1h40m	51m57s	56m16s
46_monabsex(5)	**1064**	14400	14400	5566	2653	**0.32s**	1.98s	2.02s	0.87s	0.51s
46_monabsex(6)	**21371**	518400	518400	157717	62864	**6.26s**	1m29s	1m23s	28.04s	10.33s
46_monabsex(7)	**621948**	25401600	25401600	6053748	2057588	**4m9s**	1h38m	1h23m	21m3s	7m24s
fk2012_true(3)	**12400**	42144	42144	42144	33886	**5.55s**	9.34s	10.59s	11.08s	13.13s
fk2012_true(4)	**252586**	1217826	1217826	1217826	888404	**2m3s**	5m6s	5m35s	6m11s	6m30s
fk2012_true(5)	**3757292**	24580886	24580886	24580886	16494444	**37m3s**	2h0m	2h12m	2h26m	2h28m
fkp2013_true(5)	**17751**	86400	86400	48591	25626	**3.75s**	16.40s	15.20s	9.70s	4.90s
fkp2013_true(6)	**513977**	3628800	3628800	1672915	786499	**2m18s**	14m27s	12m55s	6m34s	3m18s
fkp2013_true(7)	**20043857**	-	-	-	32244120	**2h16m**	-	-	-	3h11m
nondet-array(4)	**404**	2616	2616	688	592	**0.13s**	0.88s	0.80s	0.27s	0.20s
nondet-array(5)	**10804**	128760	128760	18665	15449	**3.11s**	46.23s	46.99s	8.66s	4.26s
nondet-array(6)	**430004**	9854640	9854640	711276	571476	**2m36s**	1h15m	1h14m	7m45s	3m30s
pthread-de(7)	**327782**	4027216	4027216	4027216	829168	**1m10s**	12m9s	13m32s	17m36s	2m12s
pthread-de(8)	**2457752**	43976774	43976774	43976774	6984234	**10m29s**	2h29m	2h46m	3h24m	22m1s
pthread-de(9)	**18568126**	-	-	-	59287740	**1h33m**	-	-	-	3h37m
reorder_5(5)	**1016**	1755360	1755360	68206	4978	**0.21s**	9m0s	9m22s	26.45s	0.34s
reorder_5(8)	**247684**	-	-	-	437725	1m47s	-	-	-	**1m29s**
reorder_5(9)	**1644716**	-	-	-	1792290	22m53s	-	-	-	**12m38s**
scull_true(3)	**3426**	617706	617706	436413	172931	**19.77s**	9m46s	10m22s	9m7s	4m46s
scull_true(4)	**8990**	2732933	2732933	1840022	656100	**1m7s**	51m37s	54m33s	46m12s	25m56s
scull_true(5)	**19881**	9488043	9488043	6070688	1988798	**3m8s**	3h29m	3h42m	2h54m	1h47m
sigma_false(7)	**12509**	135135	135135	30952	30952	**10.52s**	55.87s	1m0s	18.65s	17.87s
sigma_false(8)	**133736**	2027025	2027025	325488	325488	**2m4s**	16m21s	18m45s	4m12s	3m44s
sigma_false(9)	**1625040**	-	-	3845724	3845724	**31m53s**	-	-	1h6m	53m28s
check_bad_arr(5)	**4046**	12838	12838	10989	6689	2.74s	6.98s	6.83s	6.49s	**2.72s**
check_bad_arr(6)	**87473**	357368	357368	307097	187377	1m47s	5m21s	4m36s	4m24s	**1m33s**
check_bad_arr(7)	**1856332**	8245810	8245810	6943293	4069592	2h11m	3h9m	2h19m	2h12m	**1h7m**
32_pthread5(1)	**20**	24	24	24	**20**	**0.05s**	**0.04s**	**0.04s**	0.06s	0.06s
32_pthread5(2)	**1470**	1890	1890	1806	**1470**	0.67s	**0.38s**	0.45s	0.54s	0.67s
32_pthread5(3)	**226800**	302400	302400	280800	**226800**	2m30s	**1m14s**	1m17s	1m17s	2m21s
fkp2014_true(2)	**16**	**16**	**16**	**16**	**16**	**0.05s**	**0.05s**	**0.04s**	**0.04s**	**0.05s**
fkp2014_true(3)	**1098**	**1098**	**1098**	**1098**	**1098**	0.86s	**0.19s**	**0.20s**	0.21s	0.72s
fkp2014_true(4)	**207024**	**207024**	**207024**	**207024**	**207024**	3m40s	**39.84s**	41.70s	44.67s	3m15s
singleton(8)	**2**	40320	40320	8	8	0.06s	14.92s	15.24s	**0.04s**	0.09s
singleton(9)	**2**	362880	362880	9	9	0.09s	2m31s	2m32s	**0.05s**	0.15s
singleton(10)	**2**	3628800	3628800	10	10	0.16s	27m33s	28m9s	**0.05s**	0.19s
stack_true(9)	**48620**	**48620**	**48620**	**48620**	**48620**	2m24s	**37.55s**	38.47s	40.06s	2m23s
stack_true(10)	**184756**	**184756**	**184756**	**184756**	**184756**	11m58s	**2m31s**	2m40s	2m50s	11m1s
stack_true(11)	**705432**	**705432**	**705432**	**705432**	**705432**	58m34s	**10m32s**	11m8s	11m48s	54m42s
48_ticket_lock(2)	**6**	**6**	**6**	**6**	**6**	0.05s	**0.03s**	**0.04s**	**0.04s**	0.05s
48_ticket_lock(3)	**204**	**204**	**204**	**204**	**204**	0.25s	**0.08s**	0.10s	**0.09s**	0.34s
48_ticket_lock(4)	**41400**	**41400**	**41400**	**41400**	**41400**	55.67s	**13.88s**	15.27s	16.56s	52.57s

Table 3.1: Experimental comparison on SV-COMP benchmarks.

Benchmark	Maximal Traces					Time				
	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR
rod_cut_td3(7)	**4324**	102128	102128	51974	23143	**33.23s**	4m14s	7m43s	3m47s	1m28s
rod_cut_td3(8)	**14744**	508646	508646	257707	114624	**3m4s**	27m32s	57m42s	28m2s	12m9s
rod_cut_td3(9)	**50320**	2574752	-	1300067	577682	**17m24s**	3h0m	-	3h27m	1h39m
rod_cut_td4(3)	**1478**	91592	91592	17451	4810	**0.97s**	1m29s	1m49s	21.79s	1.46s
rod_cut_td4(4)	**21358**	2459640	2459640	359609	85203	**28.55s**	1h6m	1h33m	14m2s	57.94s
rod_cut_td4(5)	**433371**	-	-	-	2551714	**20m57s**	-	-	-	1h22m
rod_cut_bu3(6)	**19933**	183516	183516	147746	71670	**56.15s**	2m23s	3m59s	3m26s	2m3s
rod_cut_bu3(7)	**99622**	1101084	1101084	886466	429494	**8m6s**	17m52s	33m33s	29m19s	21m40s
rod_cut_bu3(8)	**498061**	6606492	-	-	2574902	**1h6m**	2h12m	-	-	3h30m
rod_cut_bu4(2)	**1901**	33912	33912	14667	5377	**0.70s**	11.76s	13.36s	6.75s	1.15s
rod_cut_bu4(3)	**74541**	2246424	2246424	913299	292633	**46.95s**	18m50s	24m12s	11m37s	1m52s
rod_cut_bu4(4)	**3007476**	-	-	-	-	**1h17m**	-	-	-	-
lis_bu3(8)	**118812**	1744064	1744064	475986	358347	**4m24s**	33m22s	1h0m	18m27s	7m24s
lis_bu3(9)	**368400**	7001792	-	1439130	1092553	**15m49s**	2h38m	-	1h10m	27m6s
lis_bu3(10)	**3133740**	-	-	-	-	**3h59m**	-	-	-	-
lis_bu4(2)	**1137**	18522	18522	7936	2828	**0.45s**	8.45s	9.41s	4.42s	0.52s
lis_bu4(3)	**29931**	1024002	1024002	364560	101766	**12.70s**	10m36s	12m49s	5m0s	19.41s
lis_bu4(4)	**1222278**	-	-	-	5679067	**16m34s**	-	-	-	37m20s
coin_all_td3(9)	**4015**	566214	566214	23308	8071	22.23s	34m25s	1h20m	2m36s	**21.13s**
coin_all_td3(10)	**9052**	2444048	-	59168	19829	**1m2s**	2h56m	-	8m20s	1m3s
coin_all_td3(19)	**637859**	-	-	-	1528102	**2h43m**	-	-	-	3h5m
coin_all_td4(2)	**5938**	6406248	-	74153	20668	**4.86s**	3h46m	-	3m27s	6.47s
coin_all_td4(3)	**68966**	-	-	1549115	319142	**1m36s**	-	-	2h15m	2m34s
coin_all_td4(5)	**379086**	-	-	-	2857926	**16m12s**	-	-	-	36m32s
coin_min_td3(8)	**46535**	1902262	1902262	981936	382275	**3m0s**	1h13m	2h12m	1h12m	14m0s
coin_min_td3(9)	**154663**	-	-	-	1634899	**11m36s**	-	-	-	1h8m
coin_min_td3(11)	**1312252**	-	-	-	-	**2h4m**	-	-	-	-
coin_min_td4(4)	**9912**	1470312	1470312	208367	46634	**30.52s**	36m17s	51m36s	7m6s	47.93s
coin_min_td4(5)	**102154**	-	-	3534815	718883	**6m7s**	-	-	2h59m	14m59s
coin_min_td4(6)	**1490420**	-	-	-	-	**1h52m**	-	-	-	-
bin_nocon_td3(7)	**13202**	1664672	1664672	471151	121350	**29.57s**	48m34s	1h26m	26m11s	2m4s
bin_nocon_td3(8)	**44802**	-	-	2825725	603668	**1m54s**	-	-	3h17m	12m32s
bin_nocon_td3(11)	**922114**	-	-	-	-	**1h0m**	-	-	-	-
bin_nocon_bu3(6)	**52500**	773122	773122	115625	75000	1m15s	12m37s	19m44s	3m5s	**1m8s**
bin_nocon_bu3(7)	**262500**	5411854	5411854	578125	375000	7m27s	1h45m	2h52m	19m54s	**6m50s**
bin_nocon_bu3(8)	**1312500**	-	-	2890625	1875000	45m2s	-	-	2h1m	**41m9s**

Table 3.2: Experimental comparison on dynamic-programming benchmarks.

Benchmarks from SV-COMP. In Table 3.1 we present experiments on benchmarks from SV-COMP (along the industrial benchmark `parker`). We have replaced all assertions with simple read events. This way we ensure a fair comparison among all algorithms in exploring the trace-space of each benchmark, as an assertion violation would halt the search. We have verified that all assertion violations present in these benchmarks are detected by all algorithms before this modification. The scaling parameter in each case controls the size of the input benchmark in terms of loop unrolls.

Dynamic-programming benchmarks. In Table 3.2 we present experiments on various multi-threaded dynamic-programming algorithms. For efficiency, these algorithms use memoization to avoid recomputing instances that correspond to the same sub-problem. The benchmarks consist of three or four threads. In each case, all-but-one threads are performing the dynamic programming computation, and one thread reads a flag signaling that the computation is finished, as well as the result of the computation. Each benchmark name contains either the substring "td" or the substring"bu", denoting that the dynamic programming table is computed top-down or bottom-up, respectively. The scaling parameter of each benchmark controls the different sizes of the input problem. The dynamic programming problems we use as benchmarks are the following.

- `rod_cut` computes, given one rod of a given length and prices for rods of shorter lengths, the maximum profit achievable by cutting the given rod.

- `lis` computes, given an array of non-repeating integers, the length of the longest increasing subsequence (not necessarily contiguous) in the array.
- `coin_all` computes, given an unlimited supply of coins of given denominations, the total number of distinct ways to get a desired change.
- `coin_min` computes, given an unlimited supply of coins of given denominations, the minimum number of coins required to get a desired change.
- `bin_nocon` computes the number of binary strings of a given length that do not contain the substring '11'.

Mutual-exclusion benchmarks. In Table 3.3 we present experiments on various mutual-exclusion algorithms from the literature. In particular, we use the two-thread solutions of Dijkstra [Dij83], Kessels [Kes82], Tsay [Tsa98], Peterson [Pet81], Peterson-Fischer [PF77], Szymanski [Szy88], Dekker [Knu66], as well as various solutions of Correia-Ramalhete [CR16]. In addition, we use the two-thread and three-thread versions of Burns's algorithm [BL80]. These protocols exercise a wide range of communication patterns, based, e.g., on the number of shared variables and the number of sequentially consistent stores/loads required to enter/leave the critical section. In all these benchmarks, each thread executes the corresponding protocol to enter a (empty) critical section a number of times, the latter controlled by the scaling parameter.

Individual benchmarks. In Table 3.4 we present experiments on individual benchmarks: `eratosthenes` consists of two threads computing the sieve of Eratosthenes in parallel; `redundant_co` consists of three threads, two of which repeatedly write to a variable and one reads from it; `float_read` consists of several threads, each writing once to a variable, and one reading from it (adapted from [AJLS18]); `opt_lock` consists of three threads in an optimistic-lock scheme. The scaling parameter controls the size in terms of loop unrolls.

Summary. For the sake of completeness, we refer to Table 3.5 for some statistics on our benchmark set. Entries marked with "U" denote that the corresponding parameter is controlled by the unroll bound of the respective benchmark. In a variety of cases, the VC partitioning is significantly coarser than each of the partitionings constructed by the other algorithms. This coarseness makes $\mathrm{VC\text{-}DPOR}$ more efficient in its exploration than the alternatives. We note that in some cases, VC offers little-to-no reduction, and then $\mathrm{VC\text{-}DPOR}$ becomes slower than the alternatives, due to the overhead incurred in constructing VC. For example, for the benchmark `reorder_5` of Table 3.1, the partitioning reduction achieved by $\mathrm{VC\text{-}DPOR}$ is large enough compared to Nidhugg/source, Optimal and Optimal* that makes $\mathrm{VC\text{-}DPOR}$ significantly faster than each of these techniques. However, although the partitioning of $\mathrm{VC\text{-}DPOR}$ is smaller than $\mathrm{DC\text{-}DPOR}$, the corresponding reduction is not large enough to make $\mathrm{VC\text{-}DPOR}$ faster than $\mathrm{DC\text{-}DPOR}$ in this benchmark (in general, $\mathrm{VC\text{-}DPOR}$ has a larger polynomial overhead than $\mathrm{DC\text{-}DPOR}$.) Similarly, for the benchmark `X2Tv9` of Table 3.3, the reduction of the VC partitioning is quite small, and although Nidhugg/source is the slowest algorithm in theory, its more lightweight nature makes it faster in practice for this benchmark. Finally, we also identify benchmarks such as `stack_true` and `48_ticket_lock` where there is no trace reduction at all, and are better handled by existing methods. We note that our approach is fairly different from the literature, and our implementation of $\mathrm{VC\text{-}DPOR}$ still largely unoptimized. We identify potential for improving the performance of $\mathrm{VC\text{-}DPOR}$ by improving the closure computation, as well as reducing (or eliminating) the number of non-maximal traces explored by the algorithm.

Benchmark	Maximal Traces					Time				
	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR
tsay(2)	**2488**	7469	7469	7469	7469	**0.81s**	2.46s	2.76s	2.99s	1.82s
tsay(3)	**241822**	1414576	1414576	1414576	1414576	**1m38s**	10m2s	10m54s	12m1s	7m42s
tsay(4)	**24609389**	-	-	-	-	**3h51m**	-	-	-	-
peter_fisch(2)	**1371**	4386	4386	4386	4386	**0.69s**	1.56s	1.61s	1.73s	1.16s
peter_fisch(3)	**70448**	430004	430004	430004	430004	**34.03s**	2m54s	3m10s	3m31s	2m20s
peter_fisch(4)	**3747718**	-	-	-	-	**41m31s**	-	-	-	-
peterson(5)	**86929**	268706	268706	268706	256457	**32.42s**	49.22s	54.60s	1m4s	1m32s
peterson(6)	**880069**	3462008	3462008	3462008	3303617	**7m10s**	11m50s	13m18s	15m51s	25m29s
peterson(7)	**9013381**	45046254	45046254	-	-	**1h30m**	2h56m	3h21m	-	-
lamport(2)	**958**	3940	3940	2454	1456	**0.39s**	0.75s	0.77s	0.59s	0.45s
lamport(3)	**57436**	741370	741370	328764	130024	**28.14s**	2m24s	2m43s	1m29s	52.24s
lamport(4)	**3723024**	-	-	-	13088038	**49m40s**	-	-	-	2h26m
dekker(5)	**89647**	435245	435245	435245	435245	**29.78s**	1m14s	1m23s	1m37s	2m14s
dekker(6)	**932559**	6745775	6745775	6745775	6745775	**6m44s**	21m36s	24m12s	28m22s	42m46s
dekker(7)	**9837974**	-	-	-	-	**1h28m**	-	-	-	-
X2Tv6(3)	**7859**	20371	20371	20371	20371	**3.89s**	5.35s	5.68s	6.58s	7.69s
X2Tv6(4)	**152999**	596354	596354	596354	596354	**1m38s**	3m6s	3m23s	3m47s	5m17s
X2Tv6(5)	**3058189**	17836411	17836411	17836411	17836411	**46m41s**	1h51m	2h3m	2h21m	3h36m
kessels(3)	**8900**	13856	13856	13856	13856	**2.80s**	5.07s	5.45s	5.98s	3.70s
kessels(4)	**194858**	323400	323400	323400	323400	**1m13s**	2m19s	2m30s	2m48s	1m41s
kessels(5)	**4379904**	7763704	7763704	7763704	7763704	**35m59s**	1h8m	1h13m	1h22m	53m50s
X2Tv7(9)	**452142**	2004774	2004774	2004774	2004774	**7m34s**	24m59s	27m10s	29m54s	13m36s
X2Tv7(10)	**1721564**	7708671	7708671	7708671	7708671	**35m19s**	1h47m	1h58m	2h10m	1h1m
X2Tv7(11)	**6584004**	-	-	-	-	**2h37m**	-	-	-	-
X2Tv2(2)	**894**	1293	1293	1293	1293	**0.32s**	0.46s	0.46s	0.51s	0.50s
X2Tv2(3)	**42141**	69316	69316	69316	69316	**17.73s**	29.21s	31.04s	34.65s	22.01s
X2Tv2(4)	**1827915**	3552837	3552837	3552837	3552837	**17m21s**	31m13s	33m46s	37m35s	25m52s
burns(4)	**381**	140380	140380	140380	140380	**0.31s**	1m24s	1m28s	1m37s	1m8s
burns(5)	**1415**	2916980	2916980	2916980	2916980	**0.98s**	35m29s	38m9s	41m55s	29m25s
burns(11)	**4114995**	-	-	-	-	**1h48m**	-	-	-	-
burns3(1)	**67**	849	849	849	849	**0.09s**	0.45s	0.40s	0.44s	0.49s
burns3(2)	**11297**	1490331	1490331	1490331	1490331	**16.27s**	16m49s	17m32s	20m4s	26m4s
burns3(3)	**1638338**	-	-	-	-	**1h0m**	-	-	-	-
X2Tv10(2)	**4130**	5079	5079	5079	5079	1.81s	1.94s	1.95s	2.18s	**1.71s**
X2Tv10(3)	**213381**	308433	308433	308433	308433	**1m47s**	2m15s	2m26s	2m39s	1m56s
X2Tv10(4)	**10274441**	17910500	17910500	17910500	17910500	**1h58m**	2h48m	3h2m	3h29m	2h35m
X2Tv5(4)	**38743**	46161	46161	46161	46161	**14.34s**	21.05s	22.57s	24.92s	15.35s
X2Tv5(5)	**595527**	730647	730647	730647	730647	**4m37s**	6m28s	6m57s	7m50s	5m2s
X2Tv5(6)	**9312813**	11755440	11755440	11755440	11755440	**1h26m**	2h2m	2h17m	2h33m	1h37m
X2Tv1(6)	**224803**	253042	253042	253042	253042	1m45s	2m19s	2m27s	2m46s	**1m42s**
X2Tv1(7)	**1880095**	2115302	2115302	2115302	2115302	18m4s	21m56s	23m59s	26m35s	**17m31s**
X2Tv1(8)	**15873308**	17857733	17857733	-	17857733	2h59m	3h29m	3h49m	-	**2h51m**
X2Tv8(3)	**6168**	9894	9894	8700	8434	2.79s	**2.56s**	2.63s	2.64s	3.15s
X2Tv8(4)	**122932**	228417	228417	194206	186040	1m8s	**1m7s**	1m13s	1m10s	1m30s
X2Tv8(5)	**2503292**	5391534	5391534	4428748	4192466	31m12s	**31m4s**	34m43s	32m37s	44m43s
X2Tv9(3)	**7234**	7304	7304	7304	7304	2.53s	**2.11s**	2.23s	2.49s	2.41s
X2Tv9(4)	**150535**	153725	153725	153725	153725	1m3s	**52.80s**	56.85s	1m3s	56.86s
X2Tv9(5)	**3261067**	3324991	3324991	3324991	3324991	29m53s	**22m17s**	24m10s	27m11s	27m10s
szymanski(3)	**27892**	27951	27951	27951	27951	12.06s	**5.06s**	5.66s	6.69s	9.81s
szymanski(4)	**395743**	396583	396583	396583	396583	4m0s	**1m26s**	1m39s	1m49s	3m14s
szymanski(5)	**5734528**	5746703	5746703	5746703	5746703	1h17m	**25m17s**	28m59s	32m36s	1h1m

Table 3.3: Experimental comparison on mutual-exclusion benchmarks.

Benchmark	Maximal Traces					Time				
	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR	VC-DPOR	Nidhugg/source	Optimal	Optimal*	DC-DPOR
eratosthenes(5)	**3500**	1527736	1527736	27858	19991	**16.92s**	18m37s	20m39s	41.14s	1m29s
eratosthenes(7)	**29320**	-	-	253792	189653	**3m37s**	-	-	9m29s	19m41s
eratosthenes(8)	**110380**	-	-	938756	710551	**11m29s**	-	-	42m27s	1h4m
redundant_co(2)	**11**	1969110	1969110	5401	729	**0.06s**	7m16s	7m32s	1.51s	0.07s
redundant_co(8)	**35**	-	-	1118305	35937	**0.09s**	-	-	13m24s	0.97s
redundant_co(9)	**39**	-	-	1778221	50653	**0.07s**	-	-	23m49s	1.35s
float_read(9)	**9**	3628800	3628800	2305	10	**0.05s**	26m30s	26m38s	1.27s	**0.04s**
float_read(15)	**15**	-	-	245761	16	**0.65s**	-	-	3m52s	0.74s
float_read(16)	**16**	-	-	524289	17	**1.42s**	-	-	9m25s	1.44s
opt_lock(2)	**2497**	69252	69252	11982	6475	**1.50s**	15.10s	15.53s	3.25s	2.50s
opt_lock(3)	**80805**	15036174	15036174	416850	212877	**52.13s**	1h5m	1h9m	2m9s	1m29s
opt_lock(4)	**2543298**	-	-	14038926	6743831	**37m41s**	-	-	1h27m	1h2m

Table 3.4: Experimental comparison on individual benchmarks.

Benchmark	LOC	Var	Locks	Threads	**Benchmark**	LOC	Var	Locks	Threads	**Benchmark**	LOC	Var	Locks	Threads
parker	134	4	0	2	48_ticket_lock	52	3	1	U	dekker	91	4	0	2
27_Boop	74	4	0	4	rod_cut_td3	50	51	0	3	X2Tv6	75	4	0	2
30_Fun_Point	67	1	1	U	rod_cut_td4	62	51	0	4	kessels	44	3	0	2
45_monabsex	24	1	0	U	rod_cut_bu3	36	51	0	3	X2Tv7	83	3	0	2
46_monabsex	22	2	0	U	rod_cut_bu4	37	51	0	4	X2Tv2	65	3	0	2
fk2012_true	100	1	2	3	lis_bu3	47	51	0	3	burns	70	3	0	2
fkp2013_true	26	1	0	U	lis_bu4	48	51	0	4	burns3	70	4	0	3
nondet-array	29	1	0	U	coin_all_td3	51	151	0	3	X2Tv10	91	3	0	2
pthread-de	67	1	1	U	coin_all_td4	53	151	0	4	X2Tv5	55	4	0	2
reorder_5	1227	4	0	U	coin_min_td3	46	51	0	3	X2Tv1	56	3	0	2
scull_true	389	7	1	3	coin_min_td4	52	51	0	4	X2Tv8	64	4	0	2
sigma_false	36	1	0	U	bin_nocon_td3	43	101	0	3	X2Tv9	61	3	0	2
check_bad_arr	33	1	0	U	bin_nocon_bu3	53	101	0	3	szymanski	93	3	0	2
32_pthread5	87	4	1	U	tsay	54	3	0	2	eratosthenes	25	U	0	2
fkp2014_true	36	2	1	U	peter_fisch	59	3	0	2	redundant_co	23	1	0	2
singleton	43	1	0	U	peterson	68	4	0	2	float_read	25	1	0	U
stack_true	104	U	1	2	lamport	83	5	0	2	opt_lock	31	2	0	3

Table 3.5: Benchmark statistics.

CHAPTER 4

The Reads-Value-From Equivalence for the SC Memory Model

In this chapter we present RVF-SMC, a new stateless model checking (SMC) algorithm that uses a novel *reads-value-from (RVF)* partitioning. Intuitively, two interleavings are deemed equivalent if they agree on the value obtained in each read event, and read events induce consistent causal orderings between them. The RVF partitioning is provably coarser than recent approaches based on Mazurkiewicz and reads-from (RF) partitionings. Our experimental evaluation reveals that RVF is quite often a very effective equivalence, as the underlying partitioning is exponentially coarser than other approaches. Moreover, RVF-SMC generates representatives very efficiently, as the reduction in the partitioning is often met with significant speed-ups in the model checking task.

We illustrate the benefits of value-based partitionings with a motivating example. Consider a simple concurrent program shown in Fig. 4.1. The program has 98 different orderings of the conflicting memory accesses, and each ordering corresponds to a separate class of the Mazurkiewicz partitioning. Utilizing the reads-from abstraction reduces the number of partitioning classes to 9. However, when taking into consideration the values that the events can read and write, the number of cases to consider can be reduced even further. In this specific example, there is only a single behavior the program may exhibit, in which both read events read the only observable value.

Thread$_1$	Thread$_2$	Thread$_3$
1. $w(x,1)$	1. $w(x,1)$	1. $w(x,1)$
2. $w(y,1)$	2. $w(y,1)$	2. $w(y,1)$
	3. $r(x)$	3. $r(y)$

Equivalence classes:	
Mazurkiewicz [AAJS14]	98
reads-from [AAJ$^+$19]	9
value-centric [CPT19]	7
reads-value-from	1

Figure 4.1: Concurrent program and its underlying partitioning classes.

The above benefits have led to recent attempts in performing SMC using a value-based equivalence [Hua15, CPT19]. However, as the realizability problem is NP-hard in general [GK97], both approaches suffer significant drawbacks. In particular, the work of [CPT19] combines the value-centric approach with the Mazurkiewicz partitioning, which creates a refinement with exponentially many more classes than potentially necessary. The example program in Fig. 4.1 illustrates this, where while both read events can only observe one possible value, the work of [CPT19] further enumerates all Mazurkiewicz orderings of all-but-one threads, resulting in 7 partitioning classes. Separately, the work of [Hua15] relies on satisfiability modulo theories (SMT) solvers, thus spending exponential time to solve the realizability problem. Hence, each approach suffers an exponential blow-up a-priori, which motivates the following question: is there an efficient *parameterized* algorithm for the consistency problem? That is, we are interested in an algorithm that is exponential-time in the worst case (as the problem is NP-hard in general), but it is efficient when certain natural parameters of the input are small, and thus it only becomes slow in extreme cases.

Another disadvantage of these previous value-based works is that each of the exploration algorithms can end up to the same class of the partitioning many times, further hindering performance. To see an example, consider the program in Fig. 4.1 again. The work of [CPT19] assigns values to reads one by one, and in this example, it needs to consider as separate cases both permutations of the two reads as the orders for assigning the values. This is to ensure completeness in cases where there are write events causally dependent on some read events (e.g., a write event appearing only if its thread-predecessor reads a certain value). However, no causally dependent write events are present in this program, and our work uses a principled approach to detect this and avoid the redundant exploration. While an example to demonstrate [Hua15] revisiting partitioning classes is a bit more involved one, this property follows from the lack of information sharing between spawned subroutines, enabling the approach to be massively parallelized, which has been discussed already in prior works [CCP+17, AAJN18, CPT19].

In this work we tackle the two challenges illustrated in the motivating example in a principled, algorithmic way. In particular, our contributions are as follows.

1. We study the problem of verifying the sequentially consistent executions. The problem is known to be NP-hard [GK97] in general, already for 3 threads. We show that the problem can be solved in $O(k^{d+1} \cdot n^{k+1})$ time for an input of n events, k threads and d variables. Thus, although the problem is NP-hard in general, it can be solved in polynomial time when the number of threads and number of variables is bounded. Moreover, our bound reduces to $O(n^{k+1})$ in the class of programs where every variable is written by only one thread (while read by many threads). Hence, in this case the bound is polynomial for a fixed number of threads and without any dependence on the number of variables.

2. We define a new equivalence between concurrent traces, called the *reads-value-from (RVF)* equivalence. Intuitively, two traces are RVF-equivalent if they agree on the value obtained in each read event, and read events induce consistent causal orderings between them. We show that RVF induces a coarser partitioning than the partitionings explored by recent well-studied SMC algorithms [AAJS14, CCP+17, CPT19, AAJ+19], and thus reduces the search space of the model checker.

3. We develop a novel SMC algorithm called $\mathrm{RVF\text{-}SMC}$, and show that it is sound and complete for local safety properties such as assertion violations. Moreover, $\mathrm{RVF\text{-}SMC}$ has complexity $k^d \cdot n^{O(k)} \cdot \beta$, where β is the size of the underlying RVF partitioning. Under the hood, $\mathrm{RVF\text{-}SMC}$ uses our consistency-checking algorithm of Item 1 to visit each RVF class during the exploration. Moreover, $\mathrm{RVF\text{-}SMC}$ uses a novel heuristic

to significantly reduce the number of revisits in any given RVF class, compared to the value-based explorations of [Hua15, CPT19].

4. We implement RVF-SMC in the stateless model checker Nidhugg [AAA+15]. Our experimental evaluation reveals that RVF is quite often a very effective equivalence, as the underlying partitioning is exponentially coarser than other approaches. Moreover, RVF-SMC generates representatives very efficiently, as the reduction in the partitioning is often met with significant speed-ups in the model checking task.

4.1 Reads-Value-From Equivalence

In this section we present our new equivalence on traces, called the *reads-value-from* equivalence (RVF equivalence, or $\sim_{RVF}$, for short). Then we illustrate that $\sim_{RVF}$ has some desirable properties for stateless model checking.

Reads-Value-From equivalence. Given two traces σ_1 and σ_2, we say that they are *reads-value-from-equivalent*, written $\sigma_1 \sim_{RVF} \sigma_2$, if the following hold.

1. $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$, i.e., they consist of the same set of events.
2. $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$, i.e., each event reads resp. writes the same value in both.
3. $\mapsto_{\sigma_1}|\mathcal{R} = \mapsto_{\sigma_2}|\mathcal{R}$, i.e., their causal orderings agree on the read events.

Fig. 4.2 presents an intuitive example of RVF-(in)equivalent traces. It presents three traces σ_1, σ_2, σ_3, and events of each trace are vertically ordered as they appear in the trace. Traces σ_1 and σ_2 are RVF-equivalent ($\sigma_1 \sim_{RVF} \sigma_2$), as they have the same events, same value function, and the two read events are causally unordered in both. Trace σ_3 is not RVF-equivalent with either of σ_1 and σ_2. Compared to σ_1 resp. σ_2, the value function of σ_3 differs ($r(y)$ reads a different value), and the causal orderings of the reads differ ($r(x) \mapsto_{\sigma_3} r(y)$).

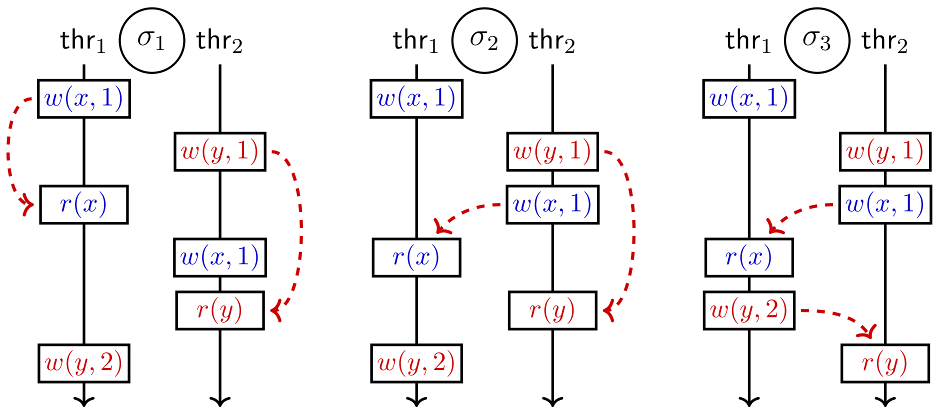


Figure 4.2: RVF-(in)equivalent traces.

Soundness. The RVF equivalence induces a partitioning on the maximal traces of $\mathscr{P}$. Any algorithm that explores each class of this partitioning provably discovers every reachable local state of every thread, and thus RVF is a sound equivalence for local safety properties, such as assertion violations, in the same spirit as in other recent works [AAJ+19, CCP+17, CPT19, Hua15]. This follows from the fact that for any two traces σ_1 and σ_2 with $\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$ and $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$, the local states of each thread are equal after executing σ_1 and σ_2.

Coarseness. Here we describe the coarseness properties of the RVF equivalence, as compared to other equivalences used by state-of-the-art approaches in stateless model checking. Fig. 4.3 summarizes the comparison, where an edge from X to Y signifies that Y is always at least as coarse, and sometimes coarser, than X.

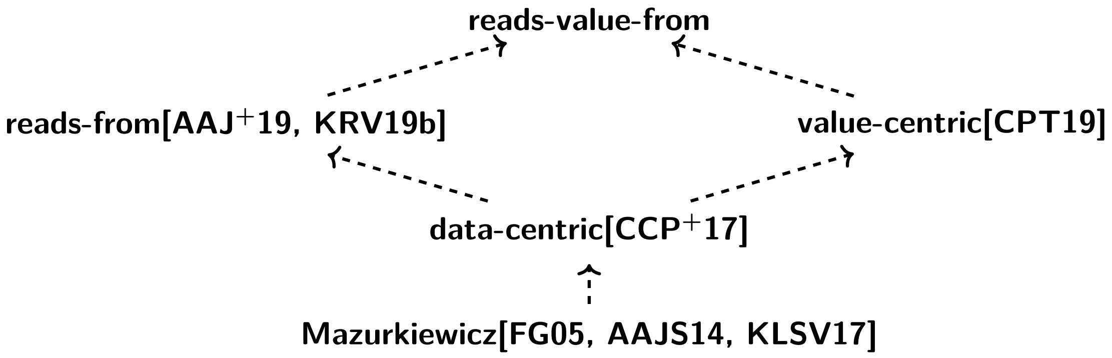

reads-value-from
reads-from[AAJ+19, KRV19b]
value-centric[CPT19]
data-centric[CCP+17]
Mazurkiewicz[FG05, AAJS14, KLSV17]

Figure 4.3: SMC trace equivalences.

The SMC algorithms of [AAJ+19] and [KRV19b] operate on a *reads-from equivalence*, which deems two traces σ_1 and σ_2 equivalent if

1. they consist of the same events ($\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$), and
2. their reads-from functions coincide ($\mathsf{RF}_{\sigma_1} = \mathsf{RF}_{\sigma_2}$).

The above two conditions imply that the induced causally-happens-before partial orders are equal, i.e., $\mapsto_{\sigma_1} = \mapsto_{\sigma_2}$, and thus trivially also $\mapsto_{\sigma_1}|\mathcal{R} = \mapsto_{\sigma_2}|\mathcal{R}$. Further, by a simple inductive argument the value functions of the two traces are also equal, i.e., $\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$. Hence any two reads-from-equivalent traces are also RVF-equivalent, which makes the RVF equivalence always at least as coarse as the reads-from equivalence.

The work of [CPT19] utilizes a *value-centric equivalence*, which deems two traces equivalent if they satisfy all the conditions of our RVF equivalence, and also some further conditions (note that these conditions are necessary for correctness of the SMC algorithm in [CPT19]). Thus the RVF equivalence is trivially always at least as coarse. The value-centric equivalence preselects a single thread thr, and then requires two extra conditions for the traces to be equivalent, namely:

1. For each read of thr, either the read reads-from a write of thr in both traces, or it does not read-from a write of thr in either of the two traces.
2. For each conflicting pair of events not belonging to thr, the ordering of the pair is equal in the two traces.

Both the reads-from equivalence and the value-centric equivalence are in turn as coarse as the *data-centric equivalence* of [CCP+17]. Given two traces, the data-centric equivalence has the equivalence conditions of the reads-from equivalence, and additionally, it preselects a single thread thr (just like the value-centric equivalence) and requires the second extra condition of the value-centric equivalence, i.e., equality of orderings for each conflicting pair of events outside of thr.

Finally, the data-centric equivalence is as coarse as the classical *Mazurkiewicz equivalence* [Maz87], the baseline equivalence for stateless model checking [FG05, AAJS14, KLSV17]. Mazurkiewicz equivalence deems two traces equivalent if they consist of the same set of events and they agree on their ordering of conflicting events.

While RVF is always at least as coarse, it can be (even exponentially) coarser, than each of the other above-mentioned equivalences. Consider the simple programs of Fig. 4.4. In each program, all traces of the program are pairwise RVF-equivalent (and hence there is only one RVF-equivalence class), while the reads-from, value-centric, data-centric, and Mazurkiewicz equivalence all induce exponentially many inequivalent traces.

Thread$_1$	Thread$_2$	Thread$_3$
$w(x,1)$	$w(x,1)$	$w(x,1)$
$w(x,1)$	$r(x)$	$w(x,1)$
...	...	...
$w(x,1)$	$w(x,1)$	$w(x,1)$
$w(x,1)$	$r(x)$	$w(x,1)$

(a) Many operations on one variable.

Thread$_1$	Thread$_2$	Thread$_3$
$w(x_1,1)$	$w(x_1,1)$	$w(x_1,1)$
$w(x_2,1)$	$r(x_1)$	$w(x_2,1)$
...	...	...
$w(x_{n-1},1)$	$w(x_n,1)$	$w(x_{n-1},1)$
$w(x_n,1)$	$r(x_n)$	$w(x_n,1)$

(b) Few operations on many variables.

Thread$_1$		Thread$_n$
$w(x,1)$	.....	$w(x,1)$
$r(x)$		$r(x)$

(c) Many threads and one variable.

Figure 4.4: Programs with one RVF-equivalence class.

We summarize the above observations in the following proposition.

Proposition 4.1. *RVF is at least as coarse as each of the Mazurkiewicz equivalence ([AAJS14]), the data-centric equivalence ([CCP+17]), the reads-from equivalence ([AAJ+19]), and the value-centric equivalence ([CPT19]). Moreover, RVF can be exponentially coarser than each of these equivalences.*

In this chapter we develop our SMC algorithm RVF-SMC around the RVF equivalence, with the guarantee that the algorithm explores at most one maximal trace per class of the RVF partitioning, and thus can perform significantly fewer steps than algorithms based on the above equivalences. To utilize RVF, the algorithm in each step solves an instance of the verification of sequential consistency problem, which we tackle in the next section. Afterwards, we present RVF-SMC.

4.2 Verifying Sequential Consistency

In this section we present our contributions towards the problem of verifying sequential consistency (VSC). We present an algorithm VerifySC for VSC, and we show how it can be efficiently used in stateless model checking.

The VSC problem. Consider an input pair (X, GoodW) where

1. $X \subseteq \mathcal{E}$ is a proper set of events, and
2. $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$ is a good-writes function such that $w \in \mathsf{GoodW}(r)$ only if $r \bowtie w$.

A *witness* of (X, GoodW) is a linearization τ of X (i.e., $\mathcal{E}(\tau) = X$) respecting the program order (i.e., $\tau \sqsubseteq \mathsf{PO}|X$), such that each read $r \in \mathcal{R}(\tau)$ reads-from one of its good-writes in τ, formally $\mathsf{RF}_\tau(r) \in \mathsf{GoodW}(r)$ (we then say that τ *satisfies* the good-writes function GoodW). The task is to decide whether (X, GoodW) has a witness, and to construct one in case it exists.

VSC in Stateless Model Checking. The VSC problem naturally ties in with our SMC approach enumerating the equivalence classes of the RVF trace partitioning. In our approach, we shall generate instances (X, GoodW) such that (i) each witness σ of (X, GoodW) is a valid program trace, and (ii) all witnesses σ_1, σ_2 of (X, GoodW) are pairwise RVF-equivalent ($\sigma_1 \sim_{RVF} \sigma_2$).

Hardness of VSC. Given an input (X, GoodW) to the VSC problem, let $n = |X|$, let k be the number of threads appearing in X, and let d be the number of variables accessed in X. The classic work of [GK97] establishes two important lower bounds on the complexity of VSC:

1. VSC is NP-hard even when restricted only to inputs with $k = 3$.
2. VSC is NP-hard even when restricted only to inputs with $d = 2$.

The first bound eliminates the possibility of any algorithm with time complexity $O(n^{f(k)})$, where f is an arbitrary computable function. Similarly, the second bound eliminates algorithms with complexity $O(n^{f(d)})$ for any computable f.

In this work we show that the problem is parameterizable in $k + d$, and thus admits efficient (polynomial-time) solutions when both parameters are bounded.

4.2.1 Algorithm for VSC

In this section we present our algorithm $\mathrm{VerifySC}$ for the problem VSC. First we define some relevant notation. In our definitions we consider a fixed input pair (X, GoodW) to the VSC problem, and a fixed sequence τ with $\mathcal{E}(\tau) \subseteq X$.

Active writes. A write $w \in \mathcal{W}(\tau)$ is *active* in τ if it is the last write of its variable in τ. Formally, for each $w' \in \mathcal{W}(\tau)$ with $\mathsf{var}(w') = \mathsf{var}(w)$ we have $w' \leq_\tau w$. We can then say that w is the active write of the variable $\mathsf{var}(w)$ in τ.

Held variables. A variable $x \in \mathcal{G}$ is *held* in τ if there exists a read $r \in \mathcal{R}(X) \setminus \mathcal{E}(\tau)$ with $\mathsf{var}(r) = x$ such that for each its good-write $w \in \mathsf{GoodW}(r)$ we have $w \in \tau$. In such a case we say that r *holds* x in τ. Note that several distinct reads may hold a single variable in τ.

Executable events. An event $e \in \mathcal{E}(X) \setminus \mathcal{E}(\tau)$ is *executable* in τ if $\mathcal{E}(\tau) \cup \{e\}$ is a lower set of (X, PO) and the following hold.

1. If e is a read, it has an active good-write $w \in \text{GoodW}(e)$ in τ.
2. If e is a write, its variable var(e) is not held in τ.

Memory maps. A *memory map* of τ is a function from global variables to thread indices $\mathrm{MMap}_\tau \colon \mathcal{G} \to [k]$ where for each variable $x \in \mathcal{G}$, the map $\mathrm{MMap}_\tau(x)$ captures the thread of the active write of x in τ.

Witness states. The sequence τ is a *witness prefix* if the following hold.

1. τ is a witness of $(\mathcal{E}(\tau), \text{GoodW}|\mathcal{R}(\tau))$.
2. For each $r \in X \setminus \mathcal{R}(\tau)$ that holds its variable var(r) in τ, one of its good-writes $w \in \text{GoodW}(r)$ is active in τ.

Intuitively, τ is a witness prefix if it satisfies all VSC requirements modulo its events, and if each read not in τ has at least one good-write still available to read-from in potential extensions of τ. For a witness prefix τ we call its corresponding event set and memory map a *witness state*.

Fig. 4.5 provides an example illustrating the above concepts, where for brevity of presentation, the variables are subscripted and the values are not displayed. The example in Fig. 4.5 presents an event set X, and a good-writes function GoodW denoted by the green dotted edges. The solid nodes are ordered vertically as they appear in a sequence τ. The grey dashed nodes are in $X \setminus \mathcal{E}(\tau)$. Events r_x and w'_x are executable in τ. Event $\overline{r_y}$ is not, its good-write is not active in τ. Event $\overline{w_y}$ is also not executable, as its variable y is held by r_y. The memory map of τ is $\mathrm{MMap}_\tau(x) = 1$ and $\mathrm{MMap}_\tau(y) = 3$. τ is a witness prefix, and $\mathcal{E}(\tau)$ with MMap_τ together form its witness state.

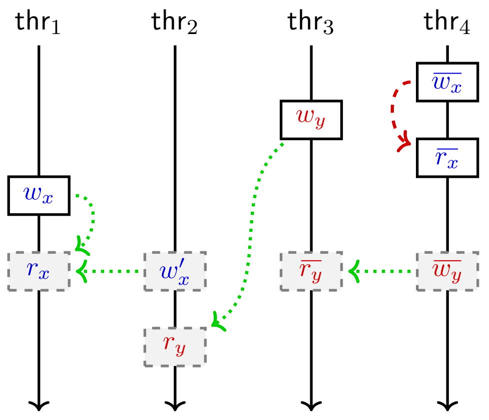


Figure 4.5: Illustration of the concepts used by $\mathrm{VerifySC}$ (Algorithm 4.1).

Algorithm. We are now ready to describe our algorithm $\mathrm{VerifySC}$, in Algorithm 4.1 we present the pseudocode. We attempt to construct a witness of (X, GoodW) by enumerating the witness states reachable by the following process. We start (Line 1) with an empty sequence ϵ as the first witness prefix (and state). We maintain a worklist $\mathcal{S}$ of so-far unprocessed witness prefixes, and a set Done of reached witness states. Then we iteratively obtain new witness prefixes (and states) by considering an already obtained prefix (Line 3) and extending it with each possible executable event (Line 6). Crucially, when we arrive at a sequence τ_e, we include it only if no sequence τ' with equal corresponding witness state has been reached yet (Line 7). We stop when we successfully create a witness (Line 4) or when we process all reachable witness states (Line 9).

Algorithm 4.1: $\mathrm{VerifySC}(X, \mathsf{GoodW})$

```
Input: Proper event set $X$ and good-writes function $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$
Output: A witness $\tau$ of $(X, \mathsf{GoodW})$ if $(X, \mathsf{GoodW})$ has a witness, else $\tau = \bot$
1 $\mathcal{S} \leftarrow \{\epsilon\}$; Done $\leftarrow \{\epsilon\}$
2 while $\mathcal{S} \neq \emptyset$ do
3     Extract a sequence $\tau$ from $\mathcal{S}$
4     if $\mathcal{E}(\tau) = X$ then return $\tau$          // All events executed, witness found
5     foreach event $e$ executable in $\tau$ do
6         Let $\tau_e \leftarrow \tau \circ e$                  // Execute $e$
7         if $\nexists \tau' \in$ Done s.t. $\mathcal{E}(\tau_e) = \mathcal{E}(\tau')$ and $\mathrm{MMap}_{\tau_e} = \mathrm{MMap}_{\tau'}$ then
8             Insert $\tau_e$ in $\mathcal{S}$ and in Done       // New witness state reached
9 return $\bot$                                              // No witness exists
```

Correctness and Complexity. We now highlight the correctness and complexity properties of VerifySC. The soundness follows straightforwardly by the fact that each sequence in $\mathcal{S}$ is a witness prefix. This follows from a simple inductive argument that extending a witness prefix with an executable event yields another witness prefix. The completeness follows from the fact that given two witness prefixes τ_1 and τ_2 with equal induced witness state, these prefixes are "equi-extendable" to a witness. Indeed, if a suffix τ^* exists such that $\tau_1 \circ \tau^*$ is a witness of (X, GoodW), then $\tau_2 \circ \tau^*$ is also a witness of (X, GoodW). The time complexity of VerifySC is bounded by $O(n^{k+1} \cdot k^{d+1})$, for n events, k threads and d variables. The bound follows from the fact that there are at most $n^k \cdot k^d$ pairwise distinct witness states. We thus have the following theorem.

Theorem 4.1. VSC *for n events, k threads and d variables is solvable in $O(n^{k+1} \cdot k^{d+1})$ time. Moreover, if each variable is written by only one thread,* VSC *is solvable in $O(n^{k+1} \cdot k)$ time.*

Proof. We argue separately about soundness, completeness, and complexity of VerifySC (Algorithm 4.1).

Soundness. We prove by induction that each sequence in the worklist $\mathcal{S}$ is a witness prefix. The base case with an empty sequence trivially holds. For an inductive case, observe that extending a witness prefix τ with an event e executable in τ yields a witness prefix. Indeed, if e is a read, it has an active good-write in τ, thus its good-writes condition is satisfied in $\tau \circ e$. If e is a write, new reads r may start holding the variable var(e) in $\tau \circ e$, but for all these reads e is its good-write, and it shall be active in $\tau \circ e$. Hence the soundness follows.

Completeness. First notice that for each witness τ of $\mathrm{VSC}(X, \mathsf{GoodW})$, each prefix of τ is a witness prefix. What remains to prove is that given two witness prefixes τ_1 and τ_2 with equal induced witness state, if a suffix exists to extend τ_1 to a witness of $\mathrm{VSC}(X, \mathsf{GoodW})$, then such suffix also exists for τ_2. Note that since τ_1 and τ_2 have an equal witness state, their length equals too (since $\mathcal{E}(\tau_1) = \mathcal{E}(\tau_2)$). We thus prove the argument by induction with respect to $|X \setminus \mathcal{E}(\tau_1)|$, i.e., the number of events remaining to add to τ_1 resp. τ_2. The base case with

$|X \setminus \mathcal{E}(\tau_1)| = 0$ is trivially satisfied. For the inductive case, let there be an arbitrary suffix τ^* such that $\tau_1 \circ \tau^*$ is a witness of $\mathrm{VSC}(X, \mathsf{GoodW})$. Let e be the first event of τ^*, we have that $\tau_1 \circ e$ is a witness prefix. Note that $\tau_2 \circ e$ is also a witness prefix. Indeed, if e is a read, the equality of the memory maps MMap_{τ_1} and MMap_{τ_2} implies that since e reads a good-write in $\tau_1 \circ e$, it also reads the same good-write in $\tau_2 \circ e$. If e is a write, since $\mathcal{E}(\tau_1) = \mathcal{E}(\tau_2)$, each read either holds its variable in both τ_1 and τ_2 or it does not hold its variable in either of τ_1 and τ_2. Finally observe that $\mathrm{MMap}_{\tau_1 \circ e} = \mathrm{MMap}_{\tau_2 \circ e}$. We have $\mathrm{MMap}_{\tau_1} = \mathrm{MMap}_{\tau_2}$, if e is a read both memory maps do not change, and if e is a write the only change of the memory maps as compared to MMap_{τ_1} and MMap_{τ_2} is that $\mathrm{MMap}_{\tau_1 \circ e}(\mathsf{var}(e)) = \mathrm{MMap}_{\tau_2 \circ e}(\mathsf{var}(e)) = \mathsf{thr}(e)$. Hence we have that $\tau_1 \circ e$ and $\tau_2 \circ e$ are both witness prefixes with the same induced witness state, and we can apply our induction hypothesis.

Complexity. There are at most $n^k \cdot k^d$ pairwise distinct witness states, since the number of different lower sets of (X, PO) is bounded by n^k, and the number of different memory maps is bounded by k^d. Hence we have a bound $n^k \cdot k^d$ on the number of iterations of the main while-loop in Line 2. Further, each iteration of the main while-loop spends $O(n \cdot k)$ time. Indeed, there are at most k iterations of the for-loop in Line 5, in each iteration it takes $O(n)$ time to check whether the event is executable, and the other items take constant time (manipulating Done in Line 7 and Line 8 takes amortized constant time with hash sets).

Let us now consider the special case when each variable is written by only one thread. In this case, the algorithm performs at most n^k iterations of the main while-loop in Line 2. This is due to the fact that each of the n^k different lower sets of (X, PO) has a unique corresponding memory map. Since each iteration of the main while-loop spends $O(n \cdot k)$ time, the overall time complexity of the algorithm in this case becomes $O(n^{k+1} \cdot k)$.

□

Implications. We now highlight some important implications of Theorem 4.1. Although VSC is NP-hard [GK97], the theorem shows that the problem is parameterizable in $k + d$, and thus in polynomial time when both parameters are bounded. Moreover, even when only k is bounded, the problem is fixed-parameter tractable in d, meaning that d only exponentiates a constant as opposed to n (e.g., we have a polynomial bound even when $d = \log n$). Finally, the algorithm is polynomial for a fixed number of threads regardless of d, when every memory location is written by only one thread (e.g., in producer-consumer settings, or in the concurrent-read-exclusive-write (CREW) concurrency model). These important facts brought forward by Theorem 4.1 indicate that VSC is likely to be efficiently solvable in many practical settings, which in turn makes RVF a good equivalence for SMC.

4.2.2 Practical heuristics for $\mathrm{VerifySC}$ in SMC

We now turn our attention to some practical heuristics that are expected to further improve the performance of $\mathrm{VerifySC}$ in the context of SMC.

1. Limiting the Search Space. We employ two straightforward improvements to $\mathrm{VerifySC}$ that significantly reduce the search space in practice. Consider the for-loop in Line 5 of Algorithm 4.1 enumerating the possible extensions of τ. This enumeration can be sidestepped by the following two greedy approaches.

1. If there is a read r executable in τ, then extend τ with r and do not enumerate other options.

2. Let $\overline{w}$ be an active write in τ such that $\overline{w}$ is not a good-write of any $r \in \mathcal{R}(X) \setminus \mathcal{E}(\tau)$. Let $w \in \mathcal{W}(X) \setminus \mathcal{E}(\tau)$ be a write of the same variable ($\mathsf{var}(w) = \mathsf{var}(\overline{w})$), note that w is executable in τ. If w is also not a good-write of any $r \in \mathcal{R}(X) \setminus \mathcal{E}(\tau)$, then extend τ with w and do not enumerate other options.

The enumeration of Line 5 then proceeds only if neither of the above two techniques can be applied for τ. This extension of $\mathrm{VerifySC}$ preserves completeness (not only when used during SMC, but in general), and it can be significantly faster in practice. For clarity of presentation we do not fully formalize this extended version, as its worst-case complexity remains the same.

2. Closure. We introduce *closure*, a low-cost filter for early detection of VSC instances (X, GoodW) with no witness. The notion of closure, its beneficial properties and construction algorithms are well-studied for the *reads-from consistency verification* problems [CCP+17, AAJ+19, Pav19], i.e., problems where a desired reads-from function is provided as input instead of a desired good-writes function GoodW. Further, the work of [CPT19] studies closure with respect to a good-writes function, but only for partial orders of Mazurkiewicz width 2 (i.e., for partial orders with no triplet of pairwise conflicting and pairwise unordered events). Here we define closure for all good-writes instances (X, GoodW), with the underlying partial order (in our case, the program order PO) of arbitrary Mazurkiewicz width.

Given a VSC instance (X, GoodW), its closure $P(X)$ is the weakest partial order that refines the program order ($P \sqsubseteq \mathsf{PO}|X$) and further satisfies the following conditions. Given a read $r \in \mathcal{R}(X)$, let $Cl(r) = \mathsf{GoodW}(r) \cap \mathsf{VisibleW}_P(r)$. The following must hold.

1. $Cl(r) \neq \emptyset$.
2. If $(Cl(r),\ P|Cl(r))$ has a least element w, then $w <_P r$.
3. If $(Cl(r),\ P|Cl(r))$ has a greatest element w, then for each $\overline{w} \in \mathcal{W}(X) \setminus \mathsf{GoodW}(r)$ with $r \bowtie \overline{w}$, if $\overline{w} <_P r$ then $\overline{w} <_P w$.
4. For each $\overline{w} \in \mathcal{W}(X) \setminus \mathsf{GoodW}(r)$ with $r \bowtie \overline{w}$, if each $w \in Cl(r)$ satisfies $w <_P \overline{w}$, then we have $r <_P \overline{w}$.

If (X, GoodW) has no closure (i.e., there is no P with the above conditions), then (X, GoodW) provably has no witness. If (X, GoodW) has closure P, then each witness τ of $\mathrm{VSC}(X, \mathsf{GoodW})$ provably refines P (i.e., $\tau \sqsubseteq P$).

Finally, we explain how closure can be used by $\mathrm{VerifySC}$. Given an input (X, GoodW), the closure procedure is carried out before $\mathrm{VerifySC}$ is called. Once the closure P of (X, GoodW) is constructed, since each solution of $\mathrm{VSC}(X, \mathsf{GoodW})$ has to refine P, we restrict $\mathrm{VerifySC}$ to only consider sequences refining P. This is ensured by an extra condition in Line 5 of Algorithm 4.1, where we proceed with an event e only if it is minimal in P restricted to events not yet in the sequence. This preserves completeness, while further reducing the search space to consider for $\mathrm{VerifySC}$.

3. $\mathrm{VerifySC}$ guided by auxiliary trace. In our SMC approach, each time we generate a VSC instance (X, GoodW), we further have available an auxiliary trace $\widetilde{\sigma}$. In $\widetilde{\sigma}$, either all-but-one, or all, good-writes conditions of GoodW are satisfied. If all good writes in GoodW are satisfied, we already have $\widetilde{\sigma}$ as a witness of (X, GoodW) and hence we do not need to run

$\mathrm{VerifySC}$ at all. On the other hand, if case all-but-one are satisfied, we use $\widetilde{\sigma}$ to guide the search of $\mathrm{VerifySC}$, as described below.

We guide the search by deciding the order in which we process the sequences of the worklist $\mathcal{S}$ in Algorithm 4.1. We use the auxiliary trace $\widetilde{\sigma}$ with $\mathcal{E}(\widetilde{\sigma}) = X$. We use $\mathcal{S}$ as a last-in-first-out stack, that way we search for a witness in a depth-first fashion. Then, in Line 5 of Algorithm 4.1 we enumerate the extension events in the reverse order of how they appear in $\widetilde{\sigma}$. We enumerate in reverse order, as each resulting extension is pushed into our worklist $\mathcal{S}$, which is a stack (last-in-first-out). As a result, in Line 3 of the subsequent iterations of the main while loop, we pop extensions from $\mathcal{S}$ in order induced by $\widetilde{\sigma}$.

4.3 Stateless Model Checking

We are now ready to present our SMC algorithm $\mathrm{RVF\text{-}SMC}$ that uses RVF to model check a concurrent program. $\mathrm{RVF\text{-}SMC}$ is a sound and complete algorithm for local safety properties, i.e., it is guaranteed to discover all local states that each thread visits.

$\mathrm{RVF\text{-}SMC}$ is a recursive algorithm. Each recursive call of $\mathrm{RVF\text{-}SMC}$ is argumented by a tuple $(X, \mathsf{GoodW}, \sigma, \mathcal{C})$ where:

1. X is a proper set of events.
2. $\mathsf{GoodW}\colon \mathcal{R}(X) \to 2^{\mathcal{W}(X)}$ is a desired good-writes function.
3. σ is a valid trace that is a witness of (X, GoodW).
4. $\mathcal{C}\colon \mathcal{R} \to \mathrm{Threads} \to \mathbb{N}$ is a partial function called *causal map* that tracks implicitly, for each read r, the writes that have already been considered as reads-from sources of r.

Further, we maintain a function ancestors: $\mathcal{R}(X) \to \{\mathsf{true}, \mathsf{false}\}$, where for each read $r \in \mathcal{R}(X)$, ancestors(r) stores a boolean *backtrack signal* for r. We now provide details on the notions of causal maps and backtrack signals.

Causal maps. The causal map $\mathcal{C}$ serves to ensure that no more than one maximal trace is explored per RVF partitioning class. Given a read $r \in \mathsf{enabled}(\sigma)$ enabled in a trace σ, we define $\mathsf{forbids}^{\mathcal{C}}_{\sigma}(r)$ as the set of writes in σ such that $\mathcal{C}$ forbids r to read-from them. Formally, $\mathsf{forbids}^{\mathcal{C}}_{\sigma}(r) = \emptyset$ if $r \notin \mathsf{dom}(\mathcal{C})$, otherwise $\mathsf{forbids}^{\mathcal{C}}_{\sigma}(r) = \{w \in \mathcal{W}(\sigma) \mid w \text{ is within first } \mathcal{C}(r)(\mathsf{thr}(w)) \text{ events of } \sigma_{\mathsf{thr}}\}$. We say that a trace σ *satisfies* $\mathcal{C}$ if for each $r \in \mathcal{R}(\sigma)$ we have $\mathsf{RF}_{\sigma}(r) \notin \mathsf{forbids}^{\mathcal{C}}_{\sigma}(r)$.

Backtrack signals. Each call of $\mathrm{RVF\text{-}SMC}$ (with its GoodW) operates with a trace $\widetilde{\sigma}$ satisfying GoodW that has only reads as enabled events. Consider one of those enabled reads $r \in \mathsf{enabled}(\widetilde{\sigma})$. Each maximal trace satisfying GoodW shall contain r, and further, one of the following two cases is true:

1. In all maximal traces σ' satisfying GoodW, we have that r reads-from some write of $\mathcal{W}(\widetilde{\sigma})$ in σ'.
2. There exists a maximal trace σ' satisfying GoodW, such that r reads-from a write not in $\mathcal{W}(\widetilde{\sigma})$ in σ'.

Whenever we can prove that the first above case is true for r, we can use this fact to prune away some recursive calls of $\mathrm{RVF\text{-}SMC}$ while maintaining completeness. Specifically, we leverage the following crucial lemma.

Lemma 4.1. *Consider a call* $\mathrm{RVF\text{-}SMC}(X, \mathsf{GoodW}, \sigma, \mathcal{C})$ *and a trace* $\widetilde{\sigma}$ *extending* σ *maximally such that no event of the extension is a read. Let* $r \in \mathsf{enabled}(\widetilde{\sigma})$ *such that* $r \not\in \mathsf{dom}(\mathcal{C})$*. If there exists a trace* σ' *that (i) satisfies* GoodW *and* $\mathcal{C}$*, and (ii) contains* r *with* $\mathsf{RF}_{\sigma'}(r) \not\in \mathcal{W}(\widetilde{\sigma})$*, then there exists a trace* $\overline{\sigma}$ *that (i) satisfies* GoodW *and* $\mathcal{C}$*, (ii) contains* r *with* $\mathsf{RF}_{\overline{\sigma}}(r) \in \mathcal{W}(\widetilde{\sigma})$*, and (iii) contains a write* $w \not\in \mathcal{W}(\widetilde{\sigma})$ *with* $r \bowtie w$ *and* $\mathsf{thr}(r) \neq \mathsf{thr}(w)$*.*

Proof. We prove the statement by a sequence of reasoning steps.

1. Let S be the set of writes $w^* \not\in \mathcal{W}(\widetilde{\sigma})$ such that there exists a trace σ^* that (i) satisfies GoodW and $\mathcal{C}$, and (ii) contains r with $\mathsf{RF}_{\sigma^*}(r) = w^*$. Observe that $\mathsf{RF}_{\sigma'}(r) \in S$, hence S is nonempty.

2. Let Y be the set of events containing r and the causal future of r in σ', formally $Y = \{e \in \mathcal{E}(\sigma') \mid r \mapsto_{\sigma} e\} \cup \{r\}$. Observe that $Y \cap \mathcal{E}(\widetilde{\sigma}) = \emptyset$. Consider a subsequence $\sigma'' = \sigma' | (\mathcal{E}(\sigma') \setminus Y)$ (i.e., subsequence of all events except Y). σ'' is a valid trace, and from $Y \cap \mathcal{E}(\widetilde{\sigma}) = \emptyset$ we get that σ'' satisfies GoodW and $\mathcal{C}$.

3. Let T_1 be the set of all partial traces that (i) contain all of $\mathcal{E}(\widetilde{\sigma})$, (ii) satisfy GoodW and $\mathcal{C}$, (iii) do not contain r, and (iv) contain some $w^* \in S$. T_1 is nonempty due to (2).

4. Let $T_2 \subseteq T_1$ be the traces σ^* of T_1 where for each $w^* \in S \cap \mathcal{E}(\sigma^*)$, we have that w^* is the last event of its thread in σ^*. The set T_2 is nonempty: since $w^* \not\in \mathcal{W}(\widetilde{\sigma})$, the events of its causal future in σ^* are also not in $\mathcal{E}(\widetilde{\sigma})$, and thus they are not good-writes to any read in GoodW.

5. Let $T_3 \subseteq T_2$ be the traces of T_2 with the least amount of read events in total. Trivially T_3 is nonempty. Further note that in each trace $\sigma^* \in T_3$, no read reads-from any write $w^* \in S \cap \mathcal{E}(\sigma^*)$. Indeed, such write can only be read-from by reads r^* out of $\mathcal{E}(\widetilde{\sigma})$ (traces of T_3 satisfy GoodW). Further, events of the causal future of such reads r^* are not good-writes to any read in GoodW (they are all out of $\mathcal{E}(\widetilde{\sigma})$). Thus the presence of r^* violates the property of having the least amount of read events in total.

6. Let σ_1 be an arbitrary partial trace from T_3. Let $S_1 = S \cap \mathcal{E}(\sigma_1)$, by (3) we have that S_1 is nonempty. Let $\sigma_2 = \sigma_1 | (\mathcal{E}(\sigma_1) \setminus S_1)$. Note that σ_2 is a valid trace, as for each $w^* \in S_1$, by (4) it is the last event of its thread, and by (5) it is not read-from by any read in σ_1.

7. Since $r \in \mathsf{enabled}(\widetilde{\sigma})$ and $\mathcal{E}(\widetilde{\sigma}) \subseteq \mathcal{E}(\sigma_2)$ and $r \not\in \mathcal{E}(\sigma_2)$, we have that $r \in \mathsf{enabled}(\sigma_2)$. Let $w^* \in S_1$ arbitrary, by the previous step we have $w^* \in \mathsf{enabled}(\sigma_2)$. Now consider $\overline{\sigma} = \sigma_2 \circ r \circ w^*$. Notice that (i) $\overline{\sigma}$ satisfies GoodW and $\mathcal{C}$, (ii) $\mathsf{RF}_{\overline{\sigma}}(r) \in \mathcal{W}(\widetilde{\sigma})$ (there is no write out of $\mathcal{W}(\widetilde{\sigma})$ present in σ_2), and (iii) for $w^* \in \mathcal{E}(\overline{\sigma})$, since $w^* \in S$ we have $r \bowtie w^*$ and $\mathsf{thr}(r) \neq \mathsf{thr}(w^*)$.

□

Algorithm 4.2: $\mathrm{RVF\text{-}SMC}(X, \mathsf{GoodW}, \sigma, \mathcal{C})$

```
Input: Proper set of events $X$, good-writes function GoodW, valid trace $\sigma$ that is a witness
       of $(X, \mathsf{GoodW})$, causal map $\mathcal{C}$.
1  $\widetilde{\sigma} \leftarrow \sigma \circ \widehat{\sigma}$ where $\widehat{\sigma}$ extends $\sigma$ maximally such that no event of $\widehat{\sigma}$ is a read
2  foreach $w \in \mathcal{E}(\widehat{\sigma})$ do                        // All extension events are writes
3      foreach $r \in$ dom(ancestors) do                      // All ancestor mutations are reads
4          if $r \bowtie w$ and thr$(r) \neq$ thr$(w)$ then        // Potential new source for $r$ to read-from
5              ancestors$(r) \leftarrow$ true                          // Set backtrack signal to true
6  mutate $\leftarrow \epsilon$                                  // Construct a sequence of enabled reads
7  foreach $r \in$ enabled$(\widetilde{\sigma})$ do                           // Enabled events in $\widetilde{\sigma}$ are reads
8      if $r \in$ dom$(\mathcal{C})$ then                                // Causal map $\mathcal{C}$ is defined for $r$
9          mutate $\leftarrow$ mutate $\circ\, r$                           // Insert $r$ to the end of mutate
10     else                                                // Causal map $\mathcal{C}$ is undefined for $r$
11         mutate $\leftarrow r \circ$ mutate                     // Insert $r$ to the beginning of mutate
12 backtrack $\leftarrow$ true
13 while backtrack = true and mutate $\neq \epsilon$ do
14     $r \leftarrow$ pop front of mutate                           // Process next read of mutate
15     if $r \notin$ dom$(\mathcal{C})$ then
16         backtrack $\leftarrow$ false
17     $\mathrm{F}_r \leftarrow \mathsf{VisibleW}_{\mathsf{PO}|\mathcal{E}(\widetilde{\sigma})}(r) \setminus \mathsf{forbids}^{\mathcal{C}}_{\widetilde{\sigma}}(r)$         // Visible writes not forbidden by $\mathcal{C}$
18     $\mathcal{D}_r \leftarrow \{\mathsf{val}_{\widetilde{\sigma}}(w) : w \in \mathrm{F}_r\}$                 // The set of values that $r$ may read
19     foreach $v \in \mathcal{D}_r$ do                                         // Process each value
20         $X' \leftarrow X \cup \mathcal{E}(\widetilde{\sigma}) \cup \{r\}$                                  // New event set
21         GoodW' $\leftarrow$ GoodW $\cup \{(r, \{ w \in \mathrm{F}_r \mid \mathsf{val}_{\widetilde{\sigma}}(w) = v \})\}$          // New good-writes
22         $\sigma' \leftarrow \mathrm{VerifySC}(X', \mathsf{GoodW}')$                         // VerifySC guided by $\widetilde{\sigma} \circ r$
23         if $\sigma' \neq \bot$ then                                   // $(X', \mathsf{GoodW}')$ has a witness
24             $\mathcal{C}' \leftarrow \mathcal{C}$
25             ancestors$(r) \leftarrow$ backtrack                                // Record ancestor
26             $\mathrm{RVF\text{-}SMC}(X', \mathsf{GoodW}', \sigma', \mathcal{C}')$
27             backtrack $\leftarrow$ ancestors$(r)$                     // Retrieve backtrack signal
28             delete $r$ from ancestors                              // Unrecord ancestor
29     foreach thr $\in$ Threads do                 // Update causal map $\mathcal{C}(r)$ for each thread
30         $\mathcal{C}(r)$(thr) $\leftarrow |\mathcal{E}(\widetilde{\sigma})_{\mathsf{thr}}|$                        // Number of events of thr in $\widetilde{\sigma}$
```

Given Lemma 4.1, we compute a boolean *backtrack signal* for a given RVF-SMC call and read $r \in$ enabled$(\widetilde{\sigma})$ to capture satisfaction of the consequent of Lemma 4.1. If the computed backtrack signal is false, we can safely stop the RVF-SMC exploration of this specific call and backtrack to its recursion parent.

Algorithm. We are now ready to describe our algorithm RVF-SMC in detail, Algorithm 4.2 captures the pseudocode of $\mathrm{RVF\text{-}SMC}(X, \mathsf{GoodW}, \sigma, \mathcal{C})$. First, in Line 1 we extend σ to $\widetilde{\sigma}$ maximally such that no event of the extension is a read. Then in Lines 2–5 we update the backtrack signals for ancestors of our current recursion call. After this, in Lines 6–11 we construct a sequence of reads enabled in $\widetilde{\sigma}$. Finally, we proceed with the main while-loop in Line 13. In each while-loop iteration we process an enabled read r (Line 14), and we perform no more while-loop iterations in case we receive a false backtrack signal for r. When processing r, first we collect its viable reads-from sources in Line 17, then we group the sources by value they write in Line 18, and then in iterations of the for-loop in Line 19 we consider each value-group. In Line 20 we form the event set, and in Line 21 we form the good-write function that designates the value-group as the good-writes of r. In Line 22 we use $\mathrm{VerifySC}$ to generate a witness, and in case it exists, we recursively call $\mathrm{RVF\text{-}SMC}$ in Line 26 with the newly obtained events, good-write constraint for r, and witness.

To preserve completeness of $\mathrm{RVF\text{-}SMC}$, the backtrack-signals technique can be utilized only for reads r with undefined causal map $r \notin \mathsf{dom}(\mathcal{C})$ (cf. Lemma 4.1). The order of the enabled reads imposed by Lines 6–11 ensures that subsequently, in iterations of the loop in Line 13 we first consider all the reads where we can utilize the backtrack signals. This is an insightful heuristic that often helps in practice, though it does not improve the worst-case complexity.

Thread thr_1

1. $w_1(x, 1)$
2. $r_1(x)$
3. $w_2(y, 1)$

Thread thr_2

1. $w_3(x, 1)$
2. $w_4(y, 1)$
3. $r_2(y)$
4. $w_5(x, 2)$

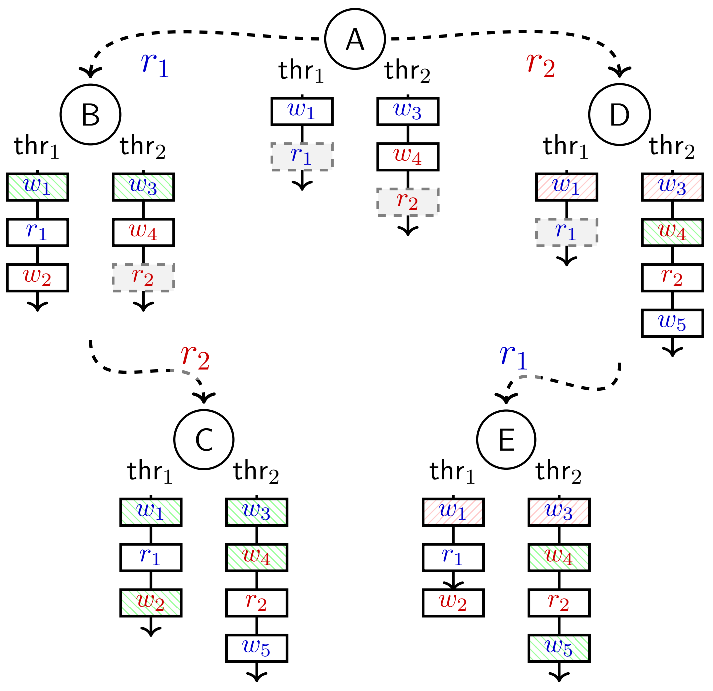


Figure 4.6: Example of $\mathrm{RVF\text{-}SMC}$ (Algorithm 4.2).

Example. Fig. 4.6 displays a simple concurrent program on the left, and its corresponding $\mathrm{RVF\text{-}SMC}$ (Algorithm 4.2) run on the right. Circles represent nodes of the recursion tree. Below each circle is its corresponding event set $\mathcal{E}(\widetilde{\sigma})$ and the enabled reads (dashed grey). Writes with green background are good-writes (GoodW) of its corresponding-variable read. Writes with red background are forbidden by $\mathcal{C}$ for its corresponding-variable read. Dashed arrows represent recursive calls.

We start with $\mathrm{RVF\text{-}SMC}(\emptyset, \emptyset, \epsilon, \emptyset)$ (A). By performing the extension (Line 1) we obtain the events and enabled reads as shown below (A). First we process read r_1 (Line 14). The read can read-from w_1 and w_3, both write the same value so they are grouped together as good-writes of r_1. A witness is found and a recursive call to (B) is performed. In (B), the only enabled event is r_2. It can read-from w_2 and w_4, both write the same value so they are grouped for r_2. A witness is found, a recursive call to (C) is performed, and (C) concludes with a maximal trace. Crucially, in (C) the event w_5 is discovered, and since it is a potential new reads-from source for r_1, a backtrack signal is sent to (A). Hence after $\mathrm{RVF\text{-}SMC}$ backtracks to (A), in (A) it needs to perform another iteration of Line 13 while-loop. In (A), first the causal map $\mathcal{C}$ is updated to forbid w_1 and w_3 for r_1. Then, read r_2 is processed from (A), creating (D). In (D), r_1 is the only enabled event, and w_5 is its only $\mathcal{C}$-allowed write. This results in (E) which reports a maximal trace. The algorithm backtracks and concludes, reporting two maximal traces in total.

Theorem 4.2. *Consider a concurrent program $\mathscr{P}$ of k threads and d variables, with n the length of the longest trace in $\mathscr{P}$. $\mathrm{RVF\text{-}SMC}$ is a sound and complete algorithm for local safety properties in $\mathscr{P}$. The time complexity of $\mathrm{RVF\text{-}SMC}$ is $k^d \cdot n^{O(k)} \cdot \beta$, where β is the size of the RVF trace partitioning of $\mathscr{P}$.*

Proof. We argue separately about soundness, completeness, and complexity.

Soundness. The soundness of $\mathrm{RVF\text{-}SMC}$ follows from the soundness of $\mathrm{VerifySC}$ used as a subroutine to generate traces that $\mathrm{RVF\text{-}SMC}$ considers.

Completeness. Let $nd = \mathrm{RVF\text{-}SMC}(X, \mathsf{GoodW}, \sigma, \mathcal{C})$ be an arbitrary recursion node of $\mathrm{RVF\text{-}SMC}$. Let σ' be an arbitrary valid full program trace satisfying GoodW and $\mathcal{C}$. The goal is to prove that the exploration rooted at a explores a good-writes function $\mathsf{GoodW}' \colon \mathcal{R}(\sigma') \to 2^{\mathcal{W}(\sigma')}$ such that for each $r \in \mathcal{R}(\sigma')$ we have $\mathsf{RF}_{\sigma'}(r) \in \mathsf{GoodW}'(r)$.

We prove the statement by induction in the length of maximal possible extension, i.e., the largest possible number of reads not defined in GoodW that a valid full program trace satisfying GoodW and $\mathcal{C}$ can have. As a reminder, given $nd = \mathrm{RVF\text{-}SMC}(X, \mathsf{GoodW}, \sigma, \mathcal{C})$ we first consider a trace $\widetilde{\sigma} = \sigma \circ \widehat{\sigma}$ where $\widehat{\sigma}$ is a maximal extension such that no event of $\widehat{\sigma}$ is a read.

Base case: 1. There is exactly one enabled read $r \in \mathsf{enabled}(\widetilde{\sigma})$. All other threads have no enabled event, i.e., they are fully extended in $\widetilde{\sigma}$. Because of this, our algorithm considers every possible source r can read-from in traces satisfying GoodW and $\mathcal{C}$. Completeness of $\mathrm{VerifySC}$ then implies completeness of this base case.

Inductive case. Let MAXEXT be the length of maximal possible extension of a recursion node $nd = \mathrm{RVF\text{-}SMC}(X, \mathsf{GoodW}, \sigma, \mathcal{C})$. By induction hypothesis, $\mathrm{RVF\text{-}SMC}$ is complete when rooted at any node with maximal possible extension length $<$ MAXEXT. The rest of the proof is to prove completeness when rooted at nd, and the desired result then follows.

Inductive case: $\mathrm{RVF\text{-}SMC}$ *without backtrack signals.* We first consider a simpler version of $\mathrm{RVF\text{-}SMC}$, where the boolean signal backtrack is always set to true (i.e., Algorithm 4.2 without Line 16). After we prove the inductive case of this version, we use it to prove the inductive case of the full version of $\mathrm{RVF\text{-}SMC}$.

Let $r_1, ..., r_k$ be the enabled events in $\widetilde{\sigma}$, nd proceeds with recursive calls in that order (i.e., first with r_1, then with r_2, ..., last with r_k). Let σ' be an arbitrary valid full program trace satisfying GoodW and $\mathcal{C}$. Trivially, σ' contains all of $r_1, ..., r_k$. Consider their reads-from sources $\mathsf{RF}_{\sigma'}(r_1), ..., \mathsf{RF}_{\sigma'}(r_k)$. Now consider two cases:

1. There exists $1 \leq i \leq k$ such that $\mathsf{RF}_{\sigma'}(r_i) \in \mathcal{E}(\widetilde{\sigma}) \cup \{\mathsf{init\_event}\}$.
2. There exists no such i.

Let us prove that (2) is impossible. By contradiction, consider it possible, let r_j be the first read out of $r_1, ..., r_k$ in the order as appearing in σ'. Consider the thread of $\mathsf{RF}_{\sigma'}(r_j)$. It has to be one of $\mathsf{thr}r_1, ..., \mathsf{thr}r_k$, as other threads have no enabled event in $\widetilde{\sigma}$, thus they are fully extended in $\widetilde{\sigma}$. It cannot be $\mathsf{thr}r_j$, because all thread-predecessors of r_j are in $\mathcal{E}(\widetilde{\sigma})$. Thus let it be a thread $1 \leq m \leq k, m \neq j$. Since $\mathsf{RF}_{\sigma'}(r_j) \notin \mathcal{E}(\widetilde{\sigma})$, $\mathsf{RF}_{\sigma'}(r_j)$ comes after r_m in σ'. This gives us $r_m <_{\sigma'} \mathsf{RF}_{\sigma'}(r_j) <_{\sigma'} r_j$, which is a contradiction with r_j being the first out of $r_1, ..., r_k$ in σ'. Hence we know that above case (1) is the only possibility.

Let $1 \leq j \leq k$ be the smallest with $\mathsf{RF}_{\sigma'}(r_j) \in \mathcal{E}(\widetilde{\sigma}) \cup \{\mathsf{init\_event}\}$. Since σ' satisfies GoodW and $\mathcal{C}$, we have $\mathsf{RF}_{\sigma'}(r_j) \notin \mathcal{C}(r_j)$. Consider nd performing a recursive call with r_j. Since

$\mathsf{RF}_{\sigma'}(r_j) \in \mathcal{E}(\widetilde{\sigma}) \cup \{\text{init\_event}\}$ and $\mathsf{RF}_{\sigma'}(r_j) \notin \mathcal{C}(r_j)$, nd considers for r_j (among others) a good-writes set GoodW_j that contains $\mathsf{RF}_{\sigma'}(r_j)$, and by completeness of VSC, we correctly classify $\mathsf{GoodW} \cup \{(r_j, \mathsf{GoodW}_j)\}$ as realizable. This creates a recursive call with the following $\overline{nd} = \text{RVF-SMC}(\overline{X}, \overline{\mathsf{GoodW}}, \overline{\sigma}, \overline{\mathcal{C}})$:

1. $\overline{X} = \mathcal{E}(\widetilde{\sigma}) \cup \{r_j\}$.
2. $\overline{\mathsf{GoodW}} = \mathsf{GoodW} \cup \{(r_j, \mathsf{GoodW}_j)\}$.
3. $\overline{\sigma}$ is a witness trace, i.e., valid program trace satisfying $\overline{\mathsf{GoodW}}$.
4. $\overline{\mathcal{C})} = \mathcal{C} \cup \{(r_i, \mathcal{C}_i) \mid 1 \leq i < j\}$ where $\mathcal{C}_i$ are the writes of var(r_i) in $\mathcal{E}(\widetilde{\sigma}) \cup \{\text{init\_event}\}$.

Clearly σ' satisfies $\overline{\mathsf{GoodW}}$. Note that σ' also satisfies $\overline{\mathcal{C})}$, as it satisfies $\mathcal{C}$, and for each $1 \leq i < j$, we have $\mathsf{RF}_{\sigma'}(r_i) \notin \mathcal{C}_i$, as $\mathsf{RF}_{\sigma'}(r_i) \notin \mathcal{E}(\widetilde{\sigma}) \cup \{\text{init\_event}\}$. Hence we can apply our inductive hypothesis for $\overline{nd}$, and we're done.

Inductive case: RVF-SMC. Let $r_1, ..., r_m$ be the enabled events (reads) in $\widetilde{\sigma}$ not defined in $\mathcal{C}$, and let $r_{m+1}, ..., r_k$ be the enabled events defined in $\mathcal{C}$. The node nd proceeds with the recursive calls as follows:

1. nd processes calls with r_1, stops if backtrack = false (Line 13), else:
2. nd processes calls with r_2, stops if backtrack = false, else .....
3. nd processes calls with r_m, stops if backtrack = false, else
4. nd processes calls with r_m, then r_{m+1}, ..., finally r_k.

Let σ' be an arbitrary valid full program trace satisfying GoodW and $\mathcal{C}$. Trivially, σ' contains all of $r_1, ..., r_k$. Consider their reads-from sources $\mathsf{RF}_{\sigma'}(r_1), ..., \mathsf{RF}_{\sigma'}(r_k)$. Let $1 \leq j \leq k$ be the smallest with $\mathsf{RF}_{\sigma'}(r_j) \in \mathcal{E}(\widetilde{\sigma}) \cup \{\text{init\_event}\}$. From the above paragraph, we have that such j exists. From the above paragraph we also have that nd processing calls with r_j explores a good-writes function $\mathsf{GoodW}' \colon \mathcal{R}(\sigma') \to 2^{\mathcal{W}(\sigma')}$ such that for each $r \in \mathcal{R}(\sigma')$ we have $\mathsf{RF}_{\sigma'}(r) \in \mathsf{GoodW}'(r)$. What remains to prove is that nd will reach the point of processing calls with r_j. That amounts to proving that for each $1 \leq x \leq \min(m, j-1)$, nd receives backtrack = true when processing calls with r_x. For each such x, $\mathsf{RF}_{\sigma'}(r_x) \notin \mathcal{E}(\widetilde{\sigma}) \cup \{\text{init\_event}\}$. We construct a trace that matches the antecedent of Lemma 4.1.

First denote $\sigma' = \sigma_1 \circ \mathsf{RF}_{\sigma'}(r_x) \circ \sigma_2$. Let σ_3 be the subsequence of σ_2, containing only events of $\mathcal{E}(\widetilde{\sigma}) \cap \mathcal{E}(\sigma_2)$. Now consider $\sigma_4 = \sigma_1 \circ \sigma_3 \circ \mathsf{RF}_{\sigma'}(r_x) \circ r_x$. Note that σ_4 is a valid trace because:

1. Each event of $\mathcal{E}(\sigma_2) \setminus \mathcal{E}(\widetilde{\sigma})$ appears in its thread only after all events of that thread in $\mathcal{E}(\widetilde{\sigma})$.
2. Each read of σ_3 has GoodW defined, and σ_3 does not contain $\mathsf{RF}_{\sigma'}(r_x)$ nor any events of $\mathcal{E}(\sigma_2) \setminus \mathcal{E}(\widetilde{\sigma})$.
3. $\mathsf{RF}_{\sigma'}(r_x)$ is enabled in $\sigma_1 \circ \sigma_3$, since it is already enabled in σ_1.
4. r_1 is not in σ_1 as those events appear before $\mathsf{RF}_{\sigma'}(r_x)$ in σ', also r_1 is not in σ_3 because $r_1 \notin \mathcal{E}(\widetilde{\sigma})$. r_1 is enabled in $\sigma_1 \circ \sigma_3 \circ \mathsf{RF}_{\sigma'}(r_x)$ as it contains all events of $\mathcal{E}(\widetilde{\sigma})$.

Hence σ_4 is a valid trace containing all events $\mathcal{E}(\widetilde{\sigma})$. Further σ_4 satisfies GoodW and $\mathcal{C}$, because σ' satisfies GoodW and $\mathcal{C}$ and σ_4 contains the same subsequence of the events $\mathcal{E}(\widetilde{\sigma})$. Finally, $\mathsf{RF}_{\sigma_4}(r_x) = \mathsf{RF}_{\sigma'}(r_x) \notin \mathcal{W}(\widetilde{\sigma})$. The inductive case, and hence the completeness result, follows.

Complexity. Each recursive call of RVF-SMC (Algorithm 4.2) trivially spends $n^{O(k)}$ time in total except the VerifySC subroutine of Line 22. For the VerifySC subroutine we utilize the complexity bound $O(n^{k+1} \cdot k^{d+1})$ from Theorem 4.1, thus the total time spent in each call of RVF-SMC is $n^{O(k)} \cdot O(k^d)$.

Next we argue that no two leaves of the recursion tree of $\mathrm{RVF\text{-}SMC}$ correspond to the same class of the RVF trace partitioning. For the sake of reaching contradiction, consider two such distinct leaves l_1 and l_2. Let a be their last (starting from the root recursion node) common ancestor. Let c_1 and c_2 be the child of a on the way to l_1 and l_2 respectively. We have $c_1 \neq c_2$ since a is the last common ancestor of l_1 and l_2. The recursion proceeds from a to c_1 (resp. c_2) by issuing a good-writes set to some read r_1 (resp. r_2). If $r_1 = r_2$, then the two good-writes set issued to $r_1 = r_2$ in a differ in the value that the writes of the two sets write (see Line 18 of Algorithm 4.2). Hence l_1 and l_2 cannot represent the same RVF partitioning class, as representative traces of the two classes shall differ in the value that $r_1 = r_2$ reads. Hence the only remaining possibility is $r_1 \neq r_2$. In iterations of Line 13 in a, wlog assume that r_1 is processed before r_2. For any pair of traces σ_1 and σ_2 that are class representatives of l_1 and l_2 respectively, we have that $\mathsf{RF}_{\sigma_1}(r_1) \neq \mathsf{RF}_{\sigma_2}(r_1)$. This follows from the update of the causal map $\mathcal{C}$ in Line 29 of the Line 13-iteration of a processing r_1. Further, we have that $\mathsf{RF}_{\sigma_2}(r_1)$ is a thread-successor of a read $\bar{r} \neq r_1$ that was among the enabled reads of mutate in a. From this we have $\bar{r} \mapsto_{\sigma_2} r_1$ and $\bar{r} \not\mapsto_{\sigma_1} r_1$. Thus the traces σ_1 and σ_2 differ in the causal orderings of the read events, contradicting that l_1 and l_2 correspond to the same class of the RVF trace partitioning.

Finally we argue that for each class of the RVF trace partitioning, represented by the (X, GoodW) of its $\mathrm{RVF\text{-}SMC}$ recursion leaf, at most n^k calls of $\mathrm{RVF\text{-}SMC}$ can be performed where its X' and GoodW' are subsets of X and GoodW, respectively. This follows from two observations. First, in each call of $\mathrm{RVF\text{-}SMC}$, the event set is extended maximally by enabled writes, and further by one read, while the good-writes function is extended by defining one further read. Second, the amount of lower sets of the partial order $(\mathcal{R}(X), \mathsf{PO})$ is bounded by n^k.

The desired complexity result follows. $\square$

Novelties of the exploration. Here we highlight some key aspects of $\mathrm{RVF\text{-}SMC}$. First, we note that $\mathrm{RVF\text{-}SMC}$ constructs the traces incrementally with each recursion step, as opposed to other approaches such as [AAJS14, AAJ+19] that always work with maximal traces. The reason of incremental traces is technical and has to do with the value-based treatment of the RVF partitioning. We note that the other two value-based approaches [Hua15, CPT19] also operate with incremental traces. However, $\mathrm{RVF\text{-}SMC}$ brings certain novelties compared to these two methods. First, the exploration algorithm of [Hua15] can visit the same class of the partitioning (and even the same trace) an exponential number of times by different recursion branches, leading to significant performance degradation. The exploration algorithm of [CPT19] alleviates this issue using the causal map data structure, similar to our algorithm. The causal map data structure can provably limit the number of revisits to polynomial (for a fixed number of threads), and although it offers an improvement over the exponential revisits, it can still affect performance. To further improve performance in this work, our algorithm combines causal maps with a new technique, which is the backtrack signals. Causal maps and backtrack signals together are very effective in avoiding having different branches of the recursion visit the same RVF class.

Beyond RVF partitioning. While $\mathrm{RVF\text{-}SMC}$ explores the RVF partitioning in the worst case, in practice it often operates on a partitioning coarser than the one induced by the RVF equivalence. Specifically, $\mathrm{RVF\text{-}SMC}$ may treat two traces σ_1 and σ_2 with same events ($\mathcal{E}(\sigma_1) = \mathcal{E}(\sigma_2)$) and value function ($\mathsf{val}_{\sigma_1} = \mathsf{val}_{\sigma_2}$) as equivalent even when they differ in some causal orderings ($\mapsto_{\sigma_1}|\mathcal{R} \neq \mapsto_{\sigma_2}|\mathcal{R}$). To see an example of this, consider the program and the $\mathrm{RVF\text{-}SMC}$ run in Fig. 4.6. The recursion node (C) spans all traces where (i) r_1 reads-from either w_1 or w_3, and (ii) r_2 reads-from either w_2 or w_4. Consider two such traces σ_1 and σ_2, with $\mathsf{RF}_{\sigma_1}(r_2) = w_2$ and $\mathsf{RF}_{\sigma_2}(r_2) = w_4$. We have $r_1 \mapsto_{\sigma_1} r_2$ and $r_1 \not\mapsto_{\sigma_2} r_2$, and yet σ_1 and σ_2 are (soundly) considered equivalent by $\mathrm{RVF\text{-}SMC}$. Hence the RVF partitioning is used to upper-bound the time complexity of $\mathrm{RVF\text{-}SMC}$. We remark that the algorithm is always sound, i.e., it is guaranteed to discover all thread states even when it does not explore the RVF partitioning in full.

4.4 Extensions of the Concurrent Model

For presentation clarity, in our exposition we considered a simple concurrent model with only read and write events. Here we describe how our approach handles the following extensions of the concurrent model:

1. Read-modify-write and compare-and-swap events.
2. Mutex events lock-acquire and lock-release.

Read-modify-write and compare-and-swap events. We model a read-modify-write atomic operation on a variable x as a pair of two events rmw_r and rmw_w, where rmw_r is a read event of x, rmw_w is a write event of x, and for each trace σ either the events are both not present in σ, or they are both present and appearing together in σ (rmw_r immediately followed by rmw_w in σ). We model a compare-and-swap atomic operation similarly, obtaining a pair of events cas_r and cas_w. In addition we consider a local event happening immediately after the read event cas_r, evaluating the "compare" condition of the compare-and-swap instruction. Thus, in traces σ that contain cas_r and the "compare" condition evaluates to true, we have that cas_r is immediately followed by cas_w in σ. In traces σ' that contain cas_r and the "compare" condition evaluates to false, we have that cas_w is not present in σ'.

We now discuss our extension of $\mathrm{VerifySC}$ to handle the $\mathrm{VSC}(X, \mathsf{GoodW})$ problem (Section 4.2) in presence of read-modify-write and compare-and-swap events. First, observe that as the event set X and the good-writes function GoodW are fixed, we possess the information on whether each compare-and-swap instruction satisfies its "compare" condition or not. Then, in case we have in our event set a read-modify-write event pair $e_1 = rmw_r$ and $e_2 = rmw_w$ (resp. a compare-and-swap event pair $e_1 = cas_r$ and $e_2 = cas_w$), we proceed as follows. When the first of the two events e_1 becomes executable in Line 5 of Algorithm 4.1 for τ, we proceed only in case e_2 is also executable in $\tau \circ e_1$, and in such a case in Line 6 we consider straight away a sequence $\tau \circ e_1 \circ e_2$. This ensures that in all sequences we consider, the event pair of the read-modify-write (resp. compare-and-swap) appears as one event immediately followed by the other event.

In the presence of read-modify-write and compare-and-swap events, the SMC approach $\mathrm{RVF\text{-}SMC}$ can be utilized as presented in Section 4.3, after an additional corner case is handled for backtrack signals. Specifically, when processing the extension events in Line 2 of Algorithm 4.2, we additionally process in the same fashion reads cas_r enabled in $\widetilde{\sigma}$ that are part of a compare-and-swap instruction. These reads cas_r are then treated as potential novel reads-from sources for ancestor mutations $cas_r^* \in \mathsf{dom}(\mathsf{ancestors})$ (Line 4) where cas_r^* is also a read-part of a compare-and-swap instruction.

Mutex events. Mutex events $acquire$ and $release$ are naturally handled by our approach as follows. We consider each lock-release event $release$ as a write event and each lock-acquire event $acquire$ as a read event, the corresponding unique mutex they access is considered a global variable of $\mathcal{G}$.

In SMC, we enumerate good-writes functions whose domain also includes the lock-acquire events. Further, a good-writes set of each lock-acquire admits only a single conflicting lock-release event, thus obtaining constraints of the form $\mathsf{GoodW}(acquire) = \{release\}$. During closure (Section 4.2.2), given $\mathsf{GoodW}(acquire) = \{release\}$, we consider the following condition: $\mathsf{thr}(acquire) \neq \mathsf{thr}(release)$ implies $release <_P acquire$. Thus P totally orders the critical sections of each mutex, and therefore $\mathrm{VerifySC}$ does not need to take additional care for mutexes. Indeed, respecting P trivially solves all GoodW constraints of lock-acquire events, and further preserves the property that no thread tries to acquire an already acquired (and so-far unreleased) mutex. No modifications to the $\mathrm{RVF\text{-}SMC}$ algorithm are needed to incorporate mutex events.

4.5 Experiments

In this section we describe the experimental evaluation of our SMC approach $\mathrm{RVF\text{-}SMC}$. We have implemented $\mathrm{RVF\text{-}SMC}$ as an extension in Nidhugg [AAA$^+$15], a state-of-the-art stateless model checker for multithreaded C/C++ programs that operates on LLVM Intermediate Representation. First we assess the advantages of utilizing the RVF equivalence in SMC as compared to other trace equivalences. Then we perform ablation studies to demonstrate the impact of the backtrack signals technique (cf. Section 4.3) and the $\mathrm{VerifySC}$ heuristics (cf. Section 4.2.2).

In our experiments we compare $\mathrm{RVF\text{-}SMC}$ with several state-of-the-art SMC tools utilizing different trace equivalences. First we consider $\mathrm{VC\text{-}DPOR}$ [CPT19], the SMC approach operating on the value-centric equivalence. Then we consider Nidhugg/rfsc [AAJ$^+$19], the SMC algorithm utilizing the reads-from equivalence. Further we consider $\mathrm{DC\text{-}DPOR}$ [CCP$^+$17] that operates on the data-centric equivalence, and finally we compare with Nidhugg/source [AAJS14] utilizing the Mazurkiewicz equivalence.[1] The works of [AAJ$^+$19] and [LS20] in turn compare the Nidhugg/rfsc algorithm with additional SMC tools, namely GenMC [KRV19b] (with reads-from equivalence), RCMC [KLSV17] (with Mazurkiewicz equivalence), and CDSChecker [ND16] (with Mazurkiewicz equivalence), and thus we omit those tools from our evaluation.

There are two main objectives to our evaluation. First, from Section 4.1 we know that the RVF equivalence can be up to exponentially coarser than the other equivalences, and we want to discover how often this happens in practice. Second, in cases where RVF does provide reduction in the trace-partitioning size, we aim to see whether this reduction is accompanied by the reduction in the runtime of $\mathrm{RVF\text{-}SMC}$ operating on RVF equivalence.

Setup. We consider 119 benchmarks in total in our evaluation. Each benchmark comes with a scaling parameter, called the *unroll* bound. The parameter controls the bound on the number of iterations in all loops of the benchmark. For each benchmark and unroll bound, we capture the number of explored maximal traces, and the total running time, subject to a timeout of one hour.

[1]The MCR algorithm [Hua15] is beyond the experimental scope of this work, as that tool handles Java programs and uses heavyweight SMT solvers that require fine-tuning.

Handling assertion violations. Some of the benchmarks in our experiments contain assertion violations, which are successfully detected by all algorithms we consider in our experimental evaluation. After performing this sanity check, we have disabled all assertions, in order to not have the measured parameters be affected by how fast a violation is discovered, as the latter is arbitrary. Our primary experimental goal is to characterize the size of the underlying partitionings, and the time it takes to explore these partitionings.

Identifying events. As mentioned in Section 2.2, an event is uniquely identified by its predecessors in PO, and by the values its PO-predecessors have read. In our implementation, we rely on the interpreter built inside Nidhugg to identify events. An event e is defined by a pair (a_e, b_e), where a_e is the thread identifier of e and b_e is the sequential number of the last LLVM instruction (of the corresponding thread) that is part of e (the e corresponds to zero or several LLVM instructions not accessing shared variables, and exactly one LLVM instruction accessing a shared variable). It can happen that there exist two traces σ_1 and σ_2, and two different events $e_1 \in \sigma_1$, $e_2 \in \sigma_2$, such that their identifiers are equal, i.e., $a_{e_1} = a_{e_2}$ and $b_{e_2} = b_{e_2}$. However, this means that the control-flow leading to each event is different. In this case, σ_1 and σ_2 differ in the value read by a common event that is ordered by the program order PO both before e_1 and before e_2, hence e_1 and e_2 are treated as inequivalent.

Technical details. For our experiments we have used a Linux machine with Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz (12 CPUs) and 128GB of RAM. We have run Nidhugg with Clang and LLVM version 8.

Scatter plots setup. Each scatter plot compares our algorithm $\mathrm{RVF\text{-}SMC}$ with some other algorithm X. In a fixed plot, each benchmark provides a single data point, obtained as follows. For the benchmark, we consider the highest unroll bound where neither of the algorithms $\mathrm{RVF\text{-}SMC}$ and X timed out.[2] Then we plot the times resp. traces obtained on that benchmark and unroll bound by the two algorithms $\mathrm{RVF\text{-}SMC}$ and X.

Experimental tables setup. Here we provide several details regarding our experimental tables. The unroll bound is shown in the column **U**. Symbol "-" indicates one-hour timeout. Bold-font entries indicate the smallest numbers for respective benchmark and unroll. Symbol † indicates that a particular benchmark operation is not handled by the tool.

Results. We provide a number of scatter plots summarizing the comparison of $\mathrm{RVF\text{-}SMC}$ with other state-of-the-art tools. In Fig. 4.7, Fig. 4.8, Fig. 4.9 and Fig. 4.10 we provide comparison both in runtimes and explored traces, for $\mathrm{VC\text{-}DPOR}$, Nidhugg/rfsc, $\mathrm{DC\text{-}DPOR}$, and Nidhugg/source, respectively. In each scatter plot, both its axes are log-scaled, the opaque red line represents equality, and the two semi-transparent lines represent an order-of-magnitude difference. The points are colored green when $\mathrm{RVF\text{-}SMC}$ achieves trace reduction in the underlying benchmark, and blue otherwise.

Discussion: Significant trace reduction. In Table 4.1 we provide the results for several benchmarks where RVF achieves significant reduction in the trace-partitioning size. This is typically accompanied by significant runtime reduction, allowing is to scale the benchmarks to unroll bounds that other tools cannot handle. Examples of this are 27_Boop4 and scull_loop, two toy Linux kernel drivers.

[2]In case one of the algorithms timed out on all attempted unroll bounds, we do not consider this benchmark when reporting on explored traces, and when reporting on execution times we consider the results on the lowest unroll bound, reporting the time-out accordingly.

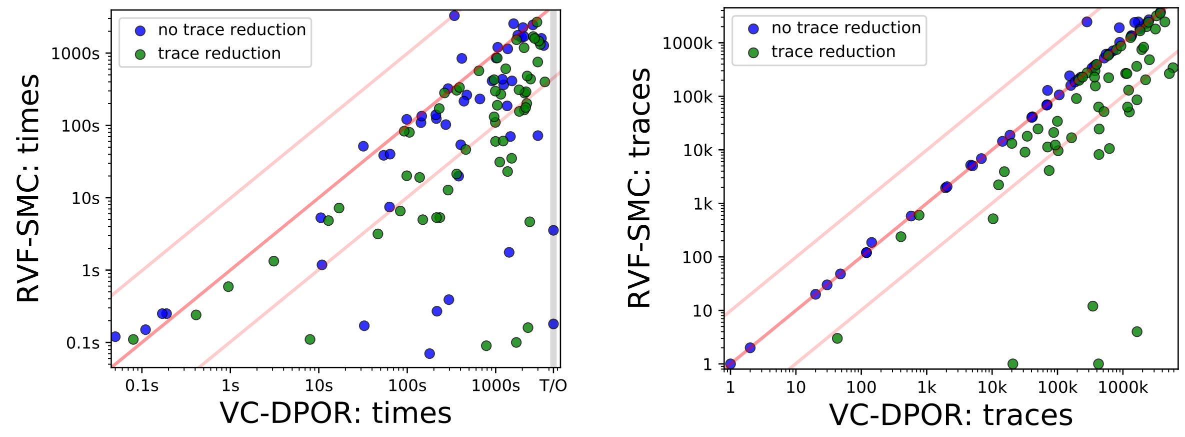


Figure 4.7: Runtime and traces comparison of RVF-SMC with VC-DPOR.

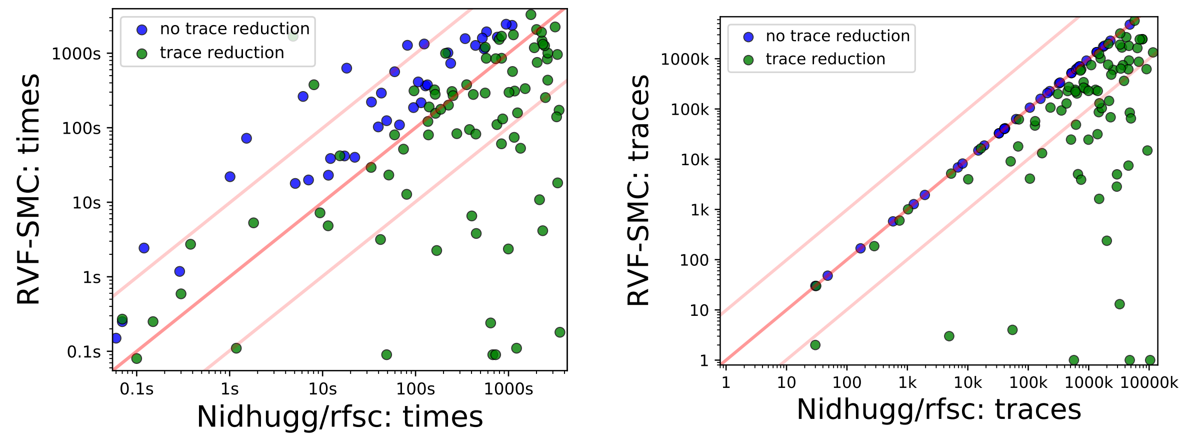


Figure 4.8: Runtime and traces comparison of RVF-SMC with Nidhugg/rfsc.

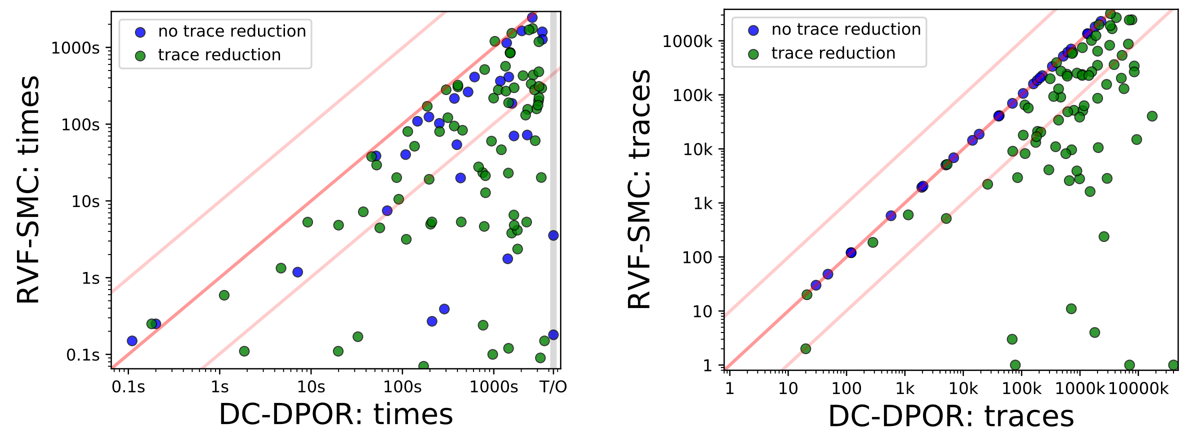


Figure 4.9: Runtime and traces comparison of RVF-SMC with DC-DPOR.

In several benchmarks the number of explored traces remains the same for RVF-SMC even when scaling up the unroll bound, see 45_monabsex1, reorder_5 and singleton in Table 4.1. The singleton example is further interesting, in that while VC-DPOR and DC-DPOR also explore few traces, they still suffer in runtime due to additional redundant exploration, as described in the introduction of Chapter 4, and in Section 4.3.

Discussion: Little-to-no trace reduction. Table 4.2 presents several benchmarks where

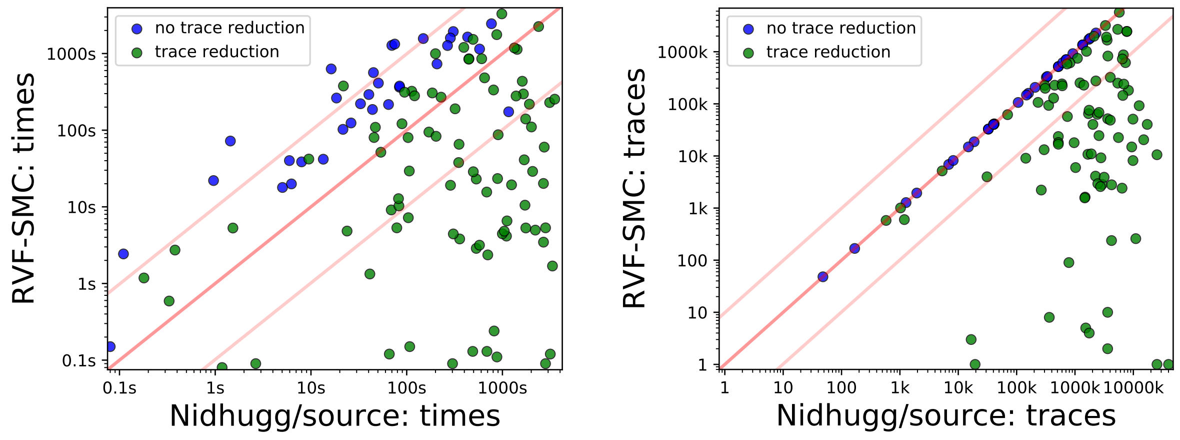


Figure 4.10: Runtime and traces comparison of RVF-SMC with Nidhugg/source.

Benchmark		U	RVF-SMC	VC-DPOR	Nidh/rfsc	DC-DPOR	Nidh/source
27_Boop4 threads: 4	Traces	10	**1337215**	1574287	11610040	-	-
		12	**2893039**	-	-	-	-
	Times	10	**837s**	1946s	2616s	-	-
		12	**2017s**	-	-	-	-
45_monabsex1 threads: U	Traces	7	**1**	423360	262144	7073803	25401600
		8	**1**	-	4782969	-	-
	Times	7	**0.09s**	784s	33s	3239s	2819s
		8	**0.09s**	-	677s	-	-
reorder_5 threads: U+1	Traces	9	**4**	1644716	1540	1792290	-
		30	**4**	-	54901	-	-
	Times	9	**0.10s**	1711s	0.44s	974s	-
		30	**0.09s**	-	49s	-	-
scull_loop threads: 3	Traces	2	**3908**	15394	749811	884443	3157281
		3	**115032**	-	-	-	-
	Times	2	**6.55s**	83s	403s	1659s	1116s
		3	**266s**	-	-	-	-
singleton threads: U+1	Traces	20	**2**	**2**	20	20	-
		30	**2**	-	30	-	-
	Times	20	**0.07s**	179s	0.08s	171s	-
		30	**0.08s**	-	0.10s	-	-

Table 4.1: Benchmarks with trace reduction achieved by RVF-SMC.

the RVF partitioning achieves little-to-no reduction. In these cases the well-engineered Nidhugg/rfsc and Nidhugg/source dominate the runtime.

RVF-SMC ablation studies. Here we demonstrate the effect that follows from our RVF-SMC algorithm utilizing the approach of backtrack signals (see Section 4.3) and the heuristics of VerifySC (see Section 4.2.2). These techniques have no effect on the number of the explored traces, thus we focus on the runtime. The left plot of Fig. 4.11 compares RVF-SMC as is with a RVF-SMC version that does not utilize the backtrack signals (achieved by simply keeping the backtrack flag in Algorithm 4.2 always true). The right plot of Fig. 4.11 compares RVF-SMC as is with a RVF-SMC version that employs VerifySC without the closure and auxiliary-trace heuristics. We can see that the techniques almost always result in improved runtime. The improvement is mostly within an order of magnitude, and in a few cases there is several-orders-of-magnitude improvement.

Finally, in Fig. 4.12 we illustrate how much time during RVF-SMC is typically spent on VerifySC (i.e., on solving VSC instances generated during RVF-SMC).

Benchmark		U	RVF-SMC	VC-DPOR	Nidh/rfsc	DC-DPOR	Nidh/source
13_unverif threads: U	Traces	5	**14400**	**14400**	**14400**	**14400**	**14400**
		6	**518400**	-	**518400**	-	**518400**
	Times	5	7.45s	63s	3.33s	68s	**2.72s**
		6	376s	-	134s	-	**84s**
approxds_append threads: U	Traces	6	**50897**	1256381	198936	1114746	9847080
		7	**923526**	-	4645207	-	-
	Times	6	**60s**	995s	67s	944s	2733s
		7	2078s	-	**2003s**	-	-
chase-lev-dq threads: 3	Traces	4	**87807**	†	175331	†	175331
		5	**227654**	†	448905	†	448905
	Times	4	289s	†	**71s**	†	**71s**
		5	995s	†	210s	†	**200s**
linuxrwlocks threads: U+1	Traces	1	**56**	†	59	†	59
		2	**62018**	†	70026	†	70026
	Times	1	0.12s	†	**0.09s**	†	0.13s
		2	42s	†	15s	†	**9.50s**
pgsql threads: 2	Traces	3	**3906**	**3906**	**3906**	**3906**	**3906**
		4	**335923**	**335923**	**335923**	**335923**	**335923**
	Times	3	3.30s	5.98s	1.01s	4.00s	**0.51s**
		4	412s	911s	107s	616s	**51s**

Table 4.2: Benchmarks with little-to-no trace reduction by RVF-SMC.

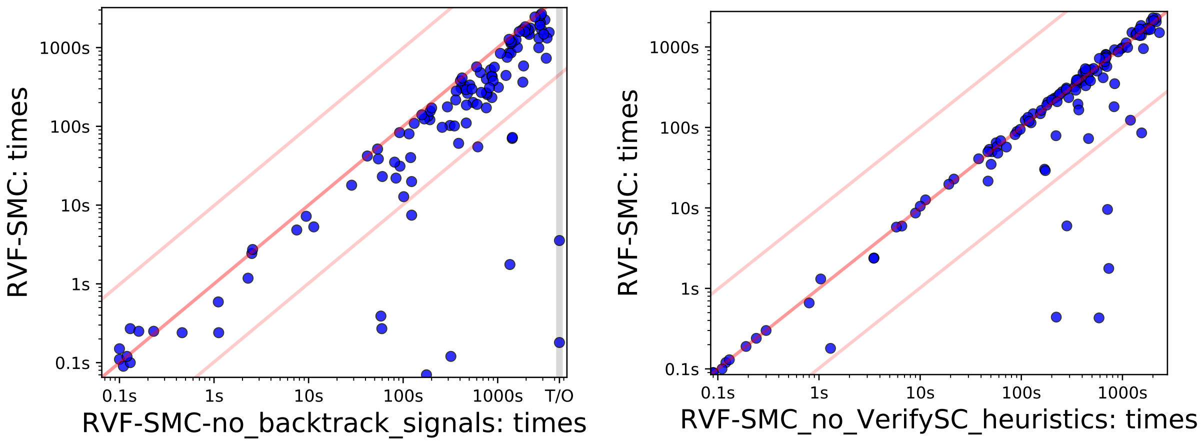


Figure 4.11: RVF-SMC ablation studies (backtrack signals and Section 4.2.2 heuristics).

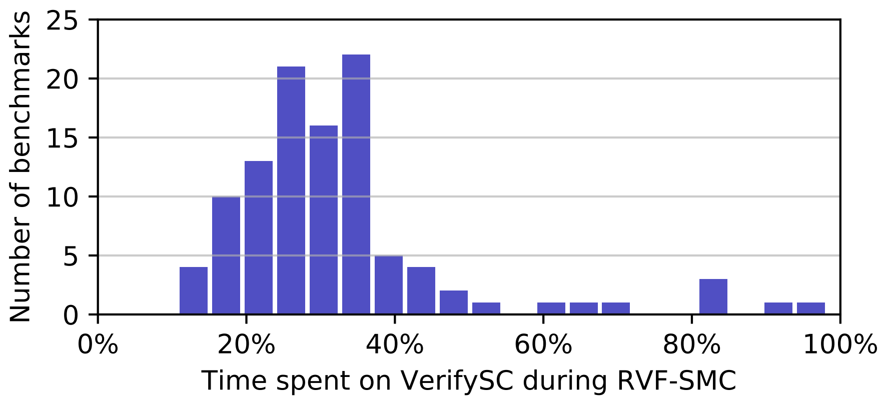


Figure 4.12: Percentage of time spent solving VSC instances during RVF-SMC.

CHAPTER 5

The Reads-From Equivalence for the TSO and PSO Memory Models

In this chapter we solve the algorithmic problem of consistency verification for the total store order (TSO) and partial store order (PSO) memory models given a *reads-from (RF)* map, denoted $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$, respectively. For an execution of n events over k threads and d variables, we establish novel bounds that scale as n^{k+1} for TSO and as $n^{k+1} \cdot \min(n^{k^2}, 2^{k \cdot d})$ for PSO. Consistency verification under a reads-from map allows to compute the *reads-from (RF) equivalence* between concurrent traces, with direct applications to areas such as stateless model checking (SMC). Hence, based on our solutions to $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$, we develop an SMC algorithm under TSO and PSO that uses the RF equivalence. The algorithm is *exploration-optimal*, in the sense that it is guaranteed to explore each class of the RF partitioning exactly once, and spends polynomial time per class when k is bounded. Finally, we implement all our algorithms in the SMC tool Nidhugg, and perform a large number of experiments over benchmarks from existing literature. Our experimental results show that our algorithms for $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$ provide significant scalability improvements over standard alternatives. Moreover, when used for SMC, the RF partitioning is often much coarser than the standard Shasha–Snir partitioning for TSO/PSO, which yields a significant speedup in the model checking task.

Recall that the TSO and PSO memory models introduce buffering mechanisms that can defer when write operations become visible to the shared memory. To illustrate the intricacies under the TSO and PSO memory models, consider the examples in Fig. 5.1. On the left, under sequential consistency (SC), in every execution at least one of $r(y)$ and $r'(x)$ will observe the corresponding $w'(y)$ and $w(x)$. Under TSO, however, the write events may become visible on the shared memory only after the read events have executed, and hence both write events go unobserved. Executions under PSO are even more involved, see the example on the right of

Thread$_1$	Thread$_2$		Thread$_1$	Thread$_2$
$1.\ w(x)$	$1.\ w'(y)$		$1.\ w(x)$	$1.\ r(y)$
$2.\ r(y)$	$2.\ r'(x)$		$2.\ w'(y)$	$2.\ r'(x)$

Figure 5.1: A TSO example (left) and a PSO example (right).

Fig. 5.1. Under either SC or TSO, if $r(y)$ observes $w'(y)$, then $r'(x)$ must observe $w(x)$, as $w(x)$ becomes visible on the shared memory before $w'(y)$. Under PSO, however, there is a single local buffer for each variable. Hence the order in which $w(x)$ and $w'(y)$ become visible in the shared memory can be reversed, allowing $r(y)$ to observe $w'(y)$ while $r'(x)$ does not observe $w(x)$.

The great challenge in verification under relaxed memory is to systematically, yet efficiently, explore all such extra behaviors of the system as illustrated in Fig. 5.1, i.e., account for the additional non-determinism that comes from the buffers. In this chapter we tackle this challenge for two verification tasks under TSO and PSO, namely, (A) for verifying the consistency of executions, and (B) for stateless model checking.

5.1 Summary of Results

Here we present formally the main results of this chapter. In later sections we present the details, algorithms, examples and proofs.

A. Verifying execution consistency for TSO and PSO. Our first set of results and the main contribution of this chapter is on the problems $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$ for verifying TSO- and PSO-consistent executions, respectively. The corresponding problem $\mathrm{VSC\text{-}rf}$ for SC was recently shown to be in polynomial time for a constant number of threads [AAJ+19, BE19].

Consider an input to the corresponding problem that consists of k threads and n operations, where each thread executes write and read operations, as well as *fence* operations that flush each thread-local buffer to the main memory. The solution for SC is obtained by essentially enumerating all the n^k possible lower sets of the program order upon the input set of events, where k is the number of threads, and hence yields a polynomial when $k = O(1)$. For TSO, the number of possible lower sets is $n^{2\cdot k}$, since there are k threads and k buffers (one for each thread). For PSO, the number of possible lower sets is $n^{k\cdot(d+1)}$, where d is the number of variables, since there are k threads and $k \cdot d$ buffers (d buffers for each thread). Hence, following an approach similar to [AAJ+19, BE19] would yield a running time of a polynomial with degree $2 \cdot k$ for TSO, and with degree $k \cdot (d + 1)$ for PSO (thus the solution for PSO is not polynomial-time even when the number of threads is bounded). In this chapter we show that both problems can be solved significantly faster.

Our results are as follows.

1. We present an algorithm that solves $\mathrm{VTSO\text{-}rf}$ in $O(k \cdot n^{k+1})$ time. Hence, although for TSO there are k additional buffers, our result shows that the complexity is only minorly impacted by an additional factor n, as opposed to n^k.

2. We present an algorithm that solves $\mathrm{VPSO\text{-}rf}$ in $O(k \cdot n^{k+1} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d}))$ time, where d is the number of variables. Note that even though there are $k \cdot d$ buffers, one of our two bounds is independent of d and thus yields polynomial time when the number of threads is bounded. Moreover, our bound collapses to $O(k \cdot n^{k+1})$ when there are no fences, and hence this case is no more difficult that $\mathrm{VTSO\text{-}rf}$.

Theorem 5.1. $\mathrm{VTSO\text{-}rf}$ *for n events and k threads is solvable in $O(k \cdot n^{k+1})$ time.*

Theorem 5.2. $\mathrm{VPSO\text{-}rf}$ *for n events, k threads and d variables is solvable in $O(k \cdot n^{k+1} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d}))$. Moreover, if there are no fences, the problem is solvable in $O(k \cdot n^{k+1})$ time.*

Novelty. For TSO, Theorem 5.1 yields an improvement of order n^{k-1} compared to the naive $n^{2\cdot k}$ bound. For PSO, perhaps surprisingly, the first upper-bound of Theorem 5.2 does not depend on the number of variables. Moreover, when there are no fences, the cost for PSO is the same as for TSO (with or without fences).

B. Stateless Model Checking for TSO and PSO. Our second result concerns stateless model checking (SMC) under TSO and PSO using the RF equivalence. We introduce an SMC algorithm $\mathrm{RF\text{-}SMC}$ that explores the RF partitioning in the TSO and PSO settings. The algorithm is based on the RF algorithm for SC [AAJ+19] and uses our solutions to $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$ for visiting each class of the respective partitioning. Moreover, $\mathrm{RF\text{-}SMC}$ is *exploration-optimal*, in the sense that it explores only maximal traces and further it is guaranteed to explore each class of the RF partitioning exactly once. The properties of $\mathrm{RF\text{-}SMC}$ are summarized in the following theorem.

Theorem 5.3. *Consider a concurrent program $\mathscr{P}$ with k threads and d variables, under a memory model $\mathcal{M} \in \{TSO, PSO\}$ with trace space $\mathcal{T}_{\mathcal{M}}^{max}$ and n being the number of events of the longest trace in $\mathcal{T}_{\mathcal{M}}^{max}$. $\mathrm{RF\text{-}SMC}$ is a sound, complete and exploration-optimal algorithm for local state reachability in $\mathscr{P}$, i.e., it explores only maximal traces and visits each class of the RF partitioning exactly once. The time complexity is $O\left(\alpha \cdot |\mathcal{T}_{\mathcal{M}}^{max}/\sim_{\mathsf{RF}}|\right)$, where*

1. *$\alpha = n^{O(k)}$ under $\mathcal{M}$ =TSO, and*
2. *$\alpha = n^{O(k^2)}$ under $\mathcal{M}$ =PSO.*

Note that the time complexity per class is polynomial in n when k is bounded. An algorithm with RF exploration-optimality in SC is presented by [AAJ+19]. Our $\mathrm{RF\text{-}SMC}$ algorithm generalizes the above approach to achieve RF exploration-optimality in the relaxed memory models TSO and PSO. Further, the time complexity of $\mathrm{RF\text{-}SMC}$ per class of RF partitioning is equal between PSO and TSO for programs with no fence instructions.

$\mathrm{RF\text{-}SMC}$ uses the verification algorithms developed in Theorem 5.1 and Theorem 5.2 as black-boxes to decide whether any specific class of the RF partitioning is TSO- or PSO-consistent, respectively. We remark that these theorems can potentially be used as black-boxes to other SMC algorithms that explore the RF partitioning (e.g., [CCP+17, KRV19b, KV20]).

C. Implementation and experiments. We have implemented $\mathrm{RF\text{-}SMC}$ in the stateless model checker Nidhugg [AAA+15], and performed an evaluation on an extensive set of benchmarks from the recent literature. Our results show that our algorithms for $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$ provide significant scalability improvements over standard alternatives, often by orders of magnitude. Moreover, when used for SMC, the RF partitioning is often much coarser than the standard Shasha–Snir partitioning for TSO/PSO, which yields a significant speedup in the model checking task.

5.2 Verifying TSO and PSO Executions with a Reads-From Function

In this section we tackle the verification problems $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$. In each case, the input is a pair (X, RF), where X is a proper set of events of $\mathscr{P}$, and $\mathsf{RF}\colon \mathcal{R}(X) \to \mathcal{W}(X)$ is a reads-from function. The task is to decide whether there exists a trace σ that is a linearization of (X, PO) with $\mathsf{RF}_\sigma = \mathsf{RF}$, where RF_σ is wrt TSO/PSO memory semantics. In case such σ exists, we say that (X, RF) is *realizable* and σ is its *witness* trace. We first define some relevant notation, and then establish upper bounds for $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$, i.e., Theorem 5.1 and Theorem 5.2.

Held variables. Given a trace σ and a memory-write $wM \in \mathcal{W}^M(\sigma)$ present in the trace, we say that wM *holds* variable $x = \mathsf{var}(wM)$ in σ if the following hold.

1. wM is the last memory-write event of σ on variable x.
2. There exists a read event $r \in X \setminus \mathcal{E}(\sigma)$ such that $\mathsf{RF}(r) = (\_, wM)$.

We similarly say that the thread $\mathsf{thr}(wM)$ *holds* x in σ. Finally, a variable x is *held* in σ if it is held by some thread in σ. Intuitively, wM holds x until all reads that need to read-from wM get executed.

Witness prefixes. Throughout this section, we use the notion of witness prefixes. Formally, a *witness prefix* is a trace σ that can be extended to a trace σ^* that realizes (X, RF), under the respective memory model. Our algorithms for $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$ operate by constructing traces σ such that if (X, RF) is realizable, then σ is a witness prefix that can be extended with the remaining events and finally realize (X, RF).

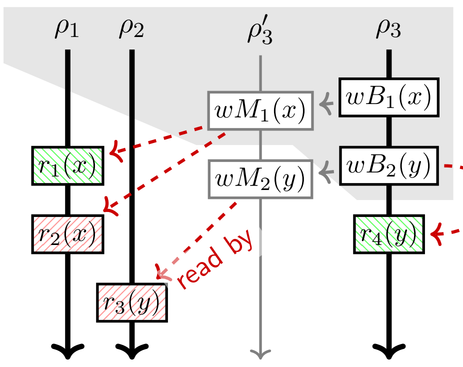
Throughout, we assume wlog that whenever $\mathsf{RF}(r) = (wB, wM)$ with $\mathsf{thr}(r) = \mathsf{thr}(wB)$, then wB is the last buffer-write on $\mathsf{var}(wB)$ before r in their respective thread. Clearly, if this condition does not hold, then the corresponding pair (X, RF) is not realizable in TSO nor PSO.

5.2.1 Verifying TSO Executions

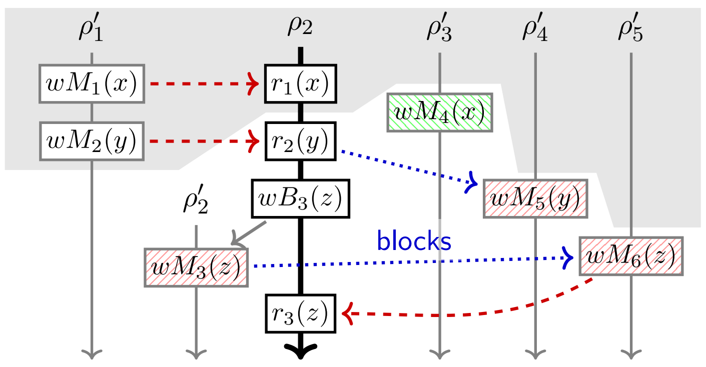
In this section we establish Theorem 5.1, i.e., we present an algorithm $\mathrm{VerifyTSO}$ that solves $\mathrm{VTSO\text{-}rf}$ in $O(k \cdot n^{k+1})$ time. The algorithm relies crucially on the notion of TSO-executable events, defined below. Throughout this section we consider fixed an instance (X, RF) of $\mathrm{VTSO\text{-}rf}$, and all traces σ considered in this section are such that $\mathcal{E}(\sigma) \subseteq X$.

TSO-executable events. Consider a trace σ. An event $e \in X \setminus \mathcal{E}(\sigma)$ is *TSO-executable* (or executable for short) in σ if $\mathcal{E}(\sigma) \cup \{e\}$ is a lower set of (X, PO) and the following conditions hold.

1. *If e is a read event r*, let $\mathsf{RF}(r) = (wB, wM)$. If $\mathsf{thr}(r) \neq \mathsf{thr}(wM)$, then $wM \in \sigma$.
2. *If e is a memory-write event wM* then the following hold.
   a) Variable $\mathsf{var}(wM)$ is not held in σ.
   b) Let $r \in \mathcal{R}(X)$ be an arbitrary read with $\mathsf{RF}(r) = (wB, wM)$ and $\mathsf{thr}(r) \neq \mathsf{thr}(wM)$. For each two-phase write (wB', wM') with $\mathsf{var}(r) = \mathsf{var}(wB')$ and $wB' <_{\mathsf{PO}} r$, we have $wM' \in \sigma$.



(a) The reads r_1 and r_4 are TSO-executable. The read r_2 is not TSO-executable, because $\mathcal{E}(\sigma) \cup \{r_2\}$ is not a lower set; neither is the read r_3, because $\mathsf{RF}(r_3) = (\_, wM_2)$ has not been executed yet.



(b) The memory-write wM_4 is TSO-executable. The other memory-writes are not; $\mathcal{E}(\sigma) \cup \{wM_3\}$ is not a lower set, for wM_5 resp. wM_6, the blue dotted arrows show the events that they have to wait for, because of Item 2a resp. Item 2b (some buffer-writes are not displayed here for brevity).

Figure 5.2: Example on TSO-executability.

Intuitively, the conditions of executable events ensure that executing an event does not immediately create an invalid witness prefix. The lower-set condition ensures that the program order PO is respected. This is a sufficient condition for a buffer-write or a fence (in particular, for a fence this implies that the respective buffer is currently empty). The extra condition for a read ensures that its reads-from constraint is satisfied. The extra conditions for a memory-write prevent it from causing some reads-from constraint to become unsatisfiable.

Fig. 5.2 illustrates the notion of TSO-executability on several examples. In the examples, the already executed events (i.e., $\mathcal{E}(\sigma)$) are in the gray zone, and the remaining events are outside the gray zone. The buffer threads are gray and thin, the main threads are black and thick. Observe that if σ is a valid trace, extending σ with an executable event (i.e., $\sigma \circ e$) also yields a valid trace that is well-formed, as, by definition, $\mathcal{E}(\sigma) \cup \{e\}$ is a lower set of (X, PO).

Algorithm $\mathrm{\mathbf{VerifyTSO}}$. We are now ready to describe our algorithm $\mathrm{VerifyTSO}$ for the problem $\mathrm{VTSO\text{-}rf}$. At a high level, the algorithm enumerates all lower sets of $(\mathcal{W}^M(X), \mathsf{PO})$ by constructing a trace σ with $\mathcal{W}^M(\sigma) = Y$ for every lower set Y of $(\mathcal{W}^M(X), \mathsf{PO})$. The crux of the algorithm is to maintain the following. Each constructed trace σ is *maximal* in the set of thread events, among all witness prefixes with the same set of memory-writes. That is, for every witness prefix σ' with $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$, we have that $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$. Thus, the algorithm will only explore n^k traces, as opposed to $n^{2 \cdot k}$ from a naive enumeration of all lower sets of (X, PO).

The formal description of $\mathrm{VerifyTSO}$ is in Algorithm 5.1. The algorithm maintains a worklist $\mathcal{S}$ of prefixes and a set Done of already-explored lower sets of $(\mathcal{W}^M(X), \mathsf{PO})$. In each iteration, the Line 4 loop makes the prefix maximal in the thread events, then Line 6 checks if we are done, otherwise the loop in Line 7 enumerates the executable memory-writes to extend the prefix with.

We now provide the insights behind the correctness of $\mathrm{VerifyTSO}$. The correctness proof has two components: (i) soundness and (ii) completeness, which we present below.

Soundness. The soundness follows directly from the definition of TSO-executable events. In

Algorithm 5.1: VerifyTSO(X, RF)

```
Input: An event set $X$ and a reads-from function RF: $\mathcal{R}(X) \to \mathcal{W}(X)$
Output: A witness $\sigma$ that realizes $(X, \mathsf{RF})$ if $(X, \mathsf{RF})$ is realizable under TSO, else $\perp$
1  $\mathcal{S} \leftarrow \{\epsilon\}$; Done $\leftarrow \{\emptyset\}$
2  while $\mathcal{S} \neq \emptyset$ do
3      Extract a trace $\sigma$ from $\mathcal{S}$
4      while $\exists$ thread event $e$ TSO-executable in $\sigma$ do
5          $\sigma \leftarrow \sigma \circ e$                              // Execute the thread event $e$
6      if $\mathcal{E}(\sigma) = X$ then return $\sigma$                       // Witness found
7      foreach memory-write $wM$ that is TSO-executable in $\sigma$ do
8          $\sigma_{wM} \leftarrow \sigma \circ wM$                             // Execute $wM$
9          if $\nexists \sigma' \in$ Done s.t. $\mathcal{W}^M(\sigma_{wM}) = \mathcal{W}^M(\sigma')$ then
10             Insert $\sigma_{wM}$ in $\mathcal{S}$ and in Done          // Continue from $\sigma_{wM}$
11 return $\perp$
```

particular, when the algorithm extends a trace σ with a read r, where $\mathsf{RF}(r) = (wB, wM)$, the following hold.

1. If $\mathsf{thr}(r) \neq \mathsf{thr}(wB)$, then $wM \in \sigma$, since r became executable. Moreover, when wM appeared in σ, the variable $x = \mathsf{var}(wM)$ became held by wM, and remained held at least until the current step where r is executed. Hence, no other memory-write wM' with $\mathsf{var}(wM') = x$ could have become executable in the meantime, to violate the observation of r. Moreover, r cannot read-from a local buffer write wB' with $\mathsf{var}(wB') = x$, as by definition, when wM became executable, all buffer-writes on x that are local to r and precede r must have been flushed to the main memory (i.e., wM' must have also appeared in the trace).
2. If $\mathsf{thr}(r) = \mathsf{thr}(wB)$, then either wM has not appeared already in σ, in which case r reads-from wB from its local buffer, or wM has appeared in the trace and held its variable until r is executed, as in the previous item.

Completeness. Let σ' be an arbitrary witness prefix, VerifyTSO constructs a trace σ such that $\mathcal{W}^M(\sigma) = \mathcal{W}^M(\sigma')$ and $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$. This is because VerifyTSO constructs for every lower set Y of $(\mathcal{W}^M(X), \mathsf{PO})$ a single representative trace σ with $\mathcal{W}^M(\sigma) = Y$. The key is to make σ *maximal* on the thread events, i.e., $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$ for any witness prefix σ' with $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$, and thus any memory-write wM that is executable in σ' is also executable in σ.

We now present the above insight in detail. Indeed, if wM is not executable in σ, one of the following holds. Let $\mathsf{var}(wM) = x$.

1. x is already held in σ. But since $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$ and any read of σ' also appears in σ, the variable x is also held in σ', thus wM is not executable in σ' either.
2. There is a later read $r \notin \sigma$ that must read-from wM, but r is preceded by a local write (wB', wM') (i.e., $wB' <_{\mathsf{PO}} r$) also on x, for which $wM' \notin \sigma$. Since $\mathcal{L}(\sigma) \supseteq \mathcal{L}(\sigma')$, we have $r \notin \sigma'$, and as $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\sigma)$, also $wM' \notin \sigma'$. Thus wM is also not executable in σ'.

The final insight is on how the algorithm maintains the maximality invariant as it extends σ with new events. This holds because read events become executable as soon as their corresponding remote observation wM appears in the trace, and hence all such reads are executable for a given lower set of $(\mathcal{W}^M(X), \mathsf{PO})$. All other thread events are executable without any further conditions. Fig. 5.3 illustrates the intuition behind the maximality invariant. In Fig. 5.3, the gray zone shows the events of some witness prefix σ'; the lighter gray shows the events of the corresponding trace σ, constructed by the algorithm, which is maximal on thread events. Yellow writes (wM_2 and wM_4) are those that are TSO-executable in σ but not in σ'. Green writes (wM_3) and red writes (wM_5) are TSO-executable and non TSO-executable, respectively.

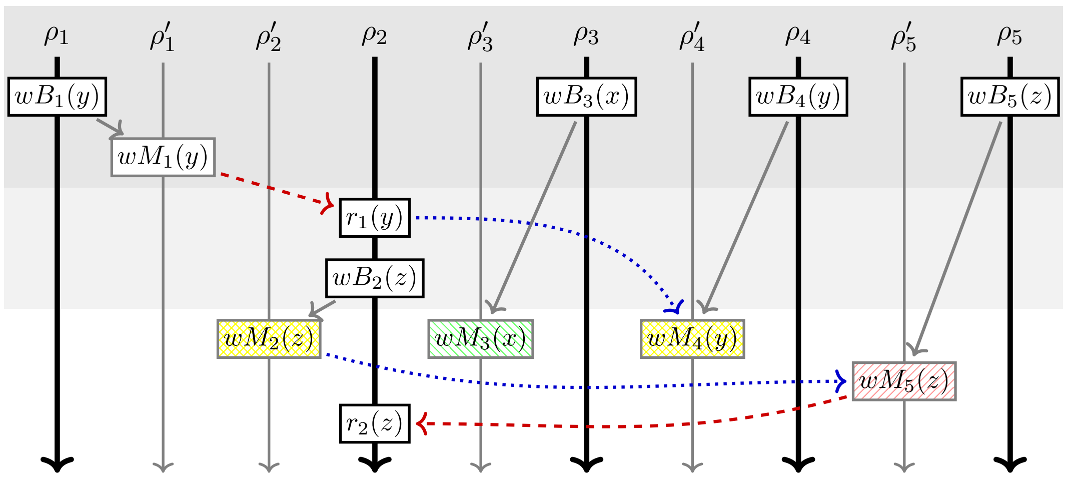


Figure 5.3: VerifyTSO maximality invariant.

The following lemma states the formal correctness, which together with the complexity argument gives us Theorem 5.1.

Lemma 5.1. (X, RF) *is realizable under TSO iff* VerifyTSO *returns a trace* $\sigma \neq \epsilon$.

Proof. We argue separately about soundness and completeness.

Soundness. We prove by induction that every trace σ extracted from $\mathcal{S}$ in Line 3 is a trace that realizes $(X|\mathcal{E}(\sigma), \mathsf{RF}|\mathcal{E}(\sigma))$ under TSO. The claim clearly holds for $\sigma = \epsilon$. Now consider a trace σ such that $\sigma \neq \emptyset$, hence σ was inserted in $\mathcal{S}$ in Line 10 while executing a previous iteration of the while-loop in Line 2. Let σ' be the trace that was extracted from $\mathcal{S}$ in that iteration. Observe that σ' is extended with TSO-executable events in Line 5 and Line 8, hence it is well-formed. It remains to argue that for every new read r executed in Line 5, we have $\mathsf{RF}_{\sigma'}(r) = \mathsf{RF}(r)$. Assume towards contradiction otherwise, and let r be the first read for which this equality fails. For the remaining of the proof, we let σ' be the trace in the iteration of Line 5 that executed r, i.e., σ' ends in r. Let $\mathsf{RF}(r) = (wB, wM)$ and $\mathsf{RF}_{\sigma'}(r) = (wB', wM')$. We distinguish the following cases.

1. If r reads-from wB' in σ', then $\mathsf{thr}(r) \neq \mathsf{thr}(wB)$, while also $wM' \notin \mathcal{E}(\sigma)$. Since r became TSO-executable, we have $wM \in \mathcal{E}(\sigma')$, hence wM has already become TSO-executable. This violates Item 2b of the definition of TSO-executable memory-writes for wM, a contradiction.

2. If r reads-from wM' in σ', then $wM \in \mathcal{E}(\sigma')$ and wM' was executed after wM was executed in σ'. This violates Item 2a of the definition of TSO-executable memory-writes for wM', a contradiction.

It follows that $\mathsf{RF}_{\sigma'}(r) = \mathsf{RF}(r)$ for all reads $r \in \mathcal{R}(\sigma')$, and hence σ' realizes $(X|\mathcal{E}(\sigma'), \mathsf{RF}|\mathcal{E}(\sigma'))$ under TSO. The above soundness argument carries over to executions containing RMW and CAS instructions, since (i) such instructions are modeled by events of already considered types (c.f. Section 5.2.4), while respecting the TSO-executability requirements of these events (as were defined in Section 5.2.1), and (ii) in Line 4 resp. Line 7 we only consider TSO-executable atomic blocks (described in detail in Section 5.2.4).

Completeness. Consider any trace σ^* that realizes (X, RF). We show by induction that for every prefix $\overline{\sigma}$ of σ^*, the algorithm examines a trace σ in Line 3 such that (i) $\mathcal{W}^M(\overline{\sigma}) = \mathcal{W}^M(\sigma)$, and (ii) $\mathcal{L}(\overline{\sigma}) \subseteq \mathcal{L}(\sigma)$. The proof is by induction on the number of memory-writes of $\overline{\sigma}$. Let $\overline{\sigma} = \overline{\sigma}' \circ \kappa \circ wM$, where κ is a sequence of thread events. Assume by the induction hypothesis that the algorithm extracts a trace σ' in Line 3 such that (i) $\mathcal{W}^M(\overline{\sigma}') = \mathcal{W}^M(\sigma')$, and (ii) $\mathcal{L}(\overline{\sigma'}) \subseteq \mathcal{L}(\sigma')$. (note that the statement clearly holds for the base case where $\overline{\sigma}' = \epsilon$). By a straightforward induction, all the events of κ not already present in σ' become eventually TSO-executable in σ', and thus appended in σ', as the algorithm executes the while-loop in Line 4. Hence, at the end of this while-loop, we have (i) $\mathcal{W}^M(\overline{\sigma}') = \mathcal{W}^M(\sigma')$, and (ii) $\mathcal{L}(\overline{\sigma}') \cup \mathcal{E}(\kappa) \subseteq \mathcal{L}(\sigma')$.

It remains to argue that wM is TSO-executable in σ' at this point (i.e., in Line 7). Assume towards contradiction otherwise, hence one of the following hold.

1. There is a read $r \in \mathcal{R}(X)$ with $\mathsf{RF}(r) = (wB', wM')$ and such that (i) $r \bowtie wM$, (ii) $wM \neq wM'$, (iii) $wM' \in \sigma'$, and (iv) $r \notin \sigma'$. By the induction hypothesis, we have $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\overline{\sigma}')$ and thus $wM' \in \overline{\sigma}'$. Moreover, we have $\mathcal{E}(\overline{\sigma}' \circ \kappa) \subseteq \mathcal{E}(\sigma')$, and thus $r \notin \overline{\sigma}' \circ \kappa$. This violates the fact that $\overline{\sigma}$ is a witness prefix for (X, RF).

2. There is a read $r \in \mathcal{R}(X)$ with $\mathsf{RF}(r) = (wB, wM)$ and such that there exists a two-phase write (wB', wM') with (i) $r \bowtie wB'$, (ii) $wB' <_{\mathsf{PO}} r$, (iii) $wM' \notin \sigma'$. By the induction hypothesis, we have $\mathcal{W}^M(\sigma') = \mathcal{W}^M(\overline{\sigma}')$ and thus $wM' \notin \overline{\sigma}'$. Moreover, we have $\mathcal{E}(\overline{\sigma}' \circ \kappa) \subseteq \mathcal{E}(\sigma')$, and thus $r \notin \overline{\sigma}' \circ \kappa$. This violates the fact that $\overline{\sigma}$ is a witness prefix for (X, RF).

Hence wM is TSO-executable in σ' in Line 7, and thus the algorithm will construct the trace $\sigma'_{wM} = \sigma' \circ wM$ in Line 8. If $\mathcal{W}^M(\sigma'_{wM}) \notin$ Done, the test in Line 9 succeeds, and the statement holds for σ being σ'_{wM} extracted from $\mathcal{S}$ in a later iteration. Otherwise, the algorithm previously constructed a trace σ'' with $\mathcal{W}^M(\sigma'') = \mathcal{W}^M(\sigma'_{wM})$, and the statement holds for σ being σ'' extracted from $\mathcal{S}$ in a later iteration.

When arguing about completeness in the presence of RMW and CAS instructions, additional care needs to be taken, as follows. The above induction argument applies, but it needs to additionally consider a case with $\overline{\sigma} = \overline{\sigma}' \circ \kappa \circ r \circ wB \circ wM$ and $\mathrm{fnc} \in \mathcal{E}(\overline{\sigma}' \circ \kappa)$, where fnc together with $r \circ wB \circ wM$ represent an atomic RMW resp. CAS instruction with the write-part designated to be immediately propagated to the shared memory. Let us consider this case in what follows.

As above, we start with the induction hypothesis that in Line 3 we have σ' with (i) $\mathcal{W}^M(\overline{\sigma}') = \mathcal{W}^M(\sigma')$, and (ii) $\mathcal{L}(\overline{\sigma'}) \subseteq \mathcal{L}(\sigma')$. Further, by an argument similar to the above, we reach

Line 7 where σ' now satisfies (i) $\mathcal{W}^M(\overline{\sigma}') = \mathcal{W}^M(\sigma')$, and (ii) $\mathcal{L}(\overline{\sigma}') \cup \mathcal{E}(\kappa) \subseteq \mathcal{L}(\sigma')$. At this point, we have $\mathrm{fnc} \in \mathcal{E}(\overline{\sigma}' \circ \kappa)$ and $\mathrm{fnc} \in \mathcal{L}(\sigma')$. Further, since in our approach we emplace r, wB and wM in an atomic block, and we never allow execution of a singular event that is part of some atomic block (described in detail in Section 5.2.4), we have that $\mathcal{E}(\sigma') \cap \{r, wB, wM\} = \emptyset$. As a result, since there are no events between fnc and r in the thread of the atomic instruction, we have that the buffer of the thread of the atomic instruction is empty in both $\overline{\sigma}' \circ \kappa$ and σ'. What remains to argue is that the atomic block $r \circ wB \circ wM$ is TSO-executable in σ'. For this, we refer to the TSO-executable conditions of atomic blocks defined in Section 5.2.4. In turn, utilizing the TSO-executable conditions of (i) reads, (ii) buffer-writes, and (iii) memory-writes, defined in Section 5.2.1, we show that (i) r is TSO-executable in σ', (ii) wB is TSO-executable in $\sigma' \circ r$, and (iii) wM is TSO-executable in $\sigma' \circ r \circ wB$. This together with $\mathcal{E}(\sigma') \cap \{r, wB, wM\} = \emptyset$ gives us that the atomic block $r \circ wB \circ wM$ is TSO-executable in σ', and thus in Line 8 the algorithm will construct the trace $\sigma'' = \sigma' \circ r \circ wB \circ wM$.

The desired completeness result follows. □

Now we can conclude the section with the proof of Theorem 5.1.

Theorem 5.1. VTSO-rf *for n events and k threads is solvable in $O(k \cdot n^{k+1})$ time.*

Proof. Lemma 5.1 establishes the correctness, so here we focus on the complexity, and the following argument applies also to executions containing RMW and CAS instructions.

Since there are k threads, there exist at most n^k distinct traces σ_1, σ_2 with $\mathcal{W}^M(\sigma_1) \neq \mathcal{W}^M(\sigma_2)$. Hence, the main loop in Line 2 is executed at most n^k times. For each of the $\leq n^k$ traces σ inserted in $\mathcal{S}$ in Line 10, there exist at most $k-1$ traces that are not inserted in $\mathcal{S}$ because $\mathcal{W}^M(\sigma) = \mathcal{W}^M(\sigma')$ (hence the test in Line 9 fails). Hence, the algorithm handles $O(k \cdot n^k)$ traces in total, while each trace is constructed in $O(n)$ time. Thus, the complexity of VerifyTSO is $O(k \cdot n^{k+1})$. The desired result follows. □

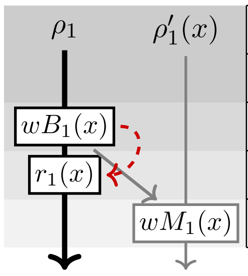
5.2.2 Verifying PSO Executions

In this section we show Theorem 5.2, i.e., we present an algorithm VerifyPSO that solves VPSO-rf in $O(k \cdot n^{k+1} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d}))$ time, while the bound becomes $O(k \cdot n^{k+1})$ when there are no fences. Similarly to the case of TSO, the algorithm relies on the notion of PSO-executable events, defined below. We first introduce some relevant notation that makes our presentation simpler.

Spurious and pending writes. Consider a trace σ with $\mathcal{E}(\sigma) \subseteq X$. A memory-write $wM \in \mathcal{W}^M(X)$ is called *spurious* in σ if the following conditions hold.

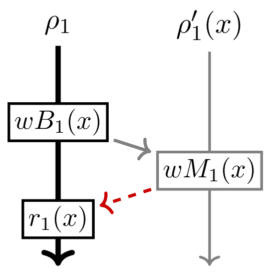
1. There is no read $r \in \mathcal{R}(X) \setminus \sigma$ with RF$(r) = (_, wM)$ (informally, no remaining read wants to read-from wM).
2. If $wM \in \sigma$, then for every read $r \in \sigma$ with RF$_\sigma(r) = (\_, wM)$ we have $r <_\sigma wM$ (informally, reads in σ that read-from this write read it from the local buffer).

Note that if wM is a spurious memory-write in σ then wM is spurious in all extensions of σ. We denote by $\mathcal{SW}^M(\sigma)$ the set of memory-writes of σ that are spurious in σ. A memory-write wM is *pending* in σ if $wB \in \sigma$ and $wM \notin \sigma$, where wB is the corresponding buffer-write of wM. We denote by $\mathcal{PW}^M(\sigma, \mathsf{thr})$ the set of all pending memory-writes wM in σ with $\mathsf{thr}(wM) = \mathsf{thr}$. See Fig. 5.4 for an intuitive illustration of spurious and pending memory-writes.



$\mathcal{SW}^M(\sigma)$	$\mathcal{PW}^M(\sigma, \mathsf{thr}_1)$
$\emptyset$	$\emptyset$
$\emptyset$	$wM_1(x)$
$wM_1(x)$	$wM_1(x)$
$wM_1(x)$	$\emptyset$

(a) Linearization where wM_1 is spurious. The table shows the spurious and pending writes after each step.



(b) Linearization where wM_1 is *not* spurious; here $\mathsf{RF}_\sigma(r_1) = (\_, wM_1)$ and $wM_1 <_\sigma r_1$.

Figure 5.4: Illustration of spurious and pending writes.

PSO-executable events. Similarly to the case of $\mathrm{VTSO\text{-}rf}$, we define the notion of PSO-executable events (executable for short). An event $e \in X \setminus \mathcal{E}(\sigma)$ is *PSO-executable* in σ if the following conditions hold.

1. *If e is a buffer-write or a memory-write*, then the same conditions apply as for TSO-executable.

2. *If e is a fence* fnc, then every pending memory-write from $\mathsf{thr}(\mathrm{fnc})$ is PSO-executable in σ, and these memory-writes together with fnc and $\mathcal{E}(\sigma)$ form a lower set of (X, PO).

3. *If e is a read r*, let $\mathsf{RF}(r) = (wB, wM)$. We have $wB \in \sigma$, and the following conditions.

   a) if $\mathsf{thr}(r) = \mathsf{thr}(wB)$, then $\mathcal{E}(\sigma) \cup \{r\}$ is a lower set of (X, PO).

   b) if $\mathsf{thr}(r) \neq \mathsf{thr}(wB)$, then $\mathcal{E}(\sigma) \cup \{wM, r\}$ is a lower set of (X, PO) and further either $wM \in \sigma$ or wM is PSO-executable in σ.

Fig. 5.5 illustrates several examples of PSO-(un)executable events, where the green events are PSO-executable and the red events are not. The memory-write $wM_2(x)$ is executable, and thus so are $r_1(x)$ and fnc_1. The memory-write $wM_3(y)$ is not executable, as the variable y is held by $wM_1(y)$ until $r_2(y)$ is executed. Consequently, fnc_2 and $r_3(y)$ are not executable.

Similarly to the case of TSO, the PSO-executable conditions ensure that we do not execute events creating an invalid witness prefix. The executability conditions for PSO are different (e.g., there are extra conditions for a fence), since our approach for $\mathrm{VPSO\text{-}rf}$ fundamentally differs from the approach for $\mathrm{VTSO\text{-}rf}$.

Fence maps. We define a *fence map* as a function $\mathrm{FMap}_\sigma \colon \mathrm{Threads} \times \mathrm{Threads} \to [n]$ as follows. First, $\mathrm{FMap}_\sigma(\mathsf{thr}, \mathsf{thr}) = 0$ for all $\mathsf{thr} \in \mathrm{Threads}$. In addition, if thr does not have a fence unexecuted in σ (i.e., a fence $\mathrm{fnc} \in (X_{\mathsf{thr}} \setminus \mathcal{E}(\sigma))$), then $\mathrm{FMap}_\sigma(\mathsf{thr}, \mathsf{thr}') = 0$ for all $\mathsf{thr}' \in \mathrm{Threads}$. Otherwise, consider the set of all reads $A_{\mathsf{thr},\mathsf{thr}'}$ such that every $r \in A_{\mathsf{thr},\mathsf{thr}'}$ with $\mathsf{RF}(r) = (wB, wM)$ satisfies the following conditions.

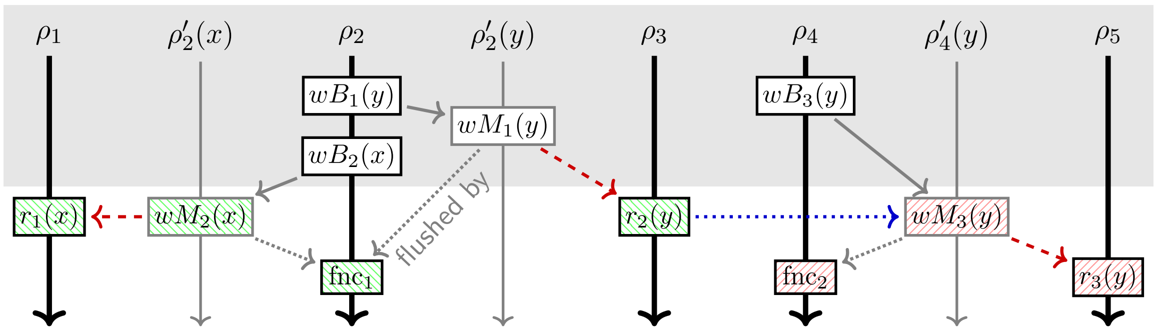


Figure 5.5: Example on PSO-executability.

1. $\mathsf{thr}(r) = \mathsf{thr}'$ and $r \notin \sigma$.
2. $\mathsf{thr}(wB) \notin \{\mathsf{thr}, \mathsf{thr}'\}$, and $\mathsf{var}(r)$ is held by wM in σ, and there is a pending memory write wM' in σ with $\mathsf{thr}(wM') = \mathsf{thr}$ and $\mathsf{var}(wM') = \mathsf{var}(r)$.

If $A_{\mathsf{thr},\mathsf{thr}'} = \emptyset$ then we let $\mathrm{FMap}_\sigma(\mathsf{thr}, \mathsf{thr}') = 0$, otherwise $\mathrm{FMap}_\sigma(\mathsf{thr}, \mathsf{thr}')$ is the largest index of a read in $A_{\mathsf{thr},\mathsf{thr}'}$. Given two traces σ_1, σ_2, $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\sigma_2}$ denotes that $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}, \mathsf{thr}') \leq \mathrm{FMap}_{\sigma_2}(\mathsf{thr}, \mathsf{thr}')$ for all $\mathsf{thr}, \mathsf{thr}' \in [k]$.

The intuition behind fence maps is as follows. Given a trace σ, the index $\mathrm{FMap}_\sigma(\mathsf{thr}, \mathsf{thr}')$ points to the *latest* (wrt PO) read r of thr' that must be executed in any extension of σ before thr can execute its next fence. This occurs because the following hold in σ.

1. The variable $\mathsf{var}(r)$ is held by the memory-write $wM \in \sigma$ with $\mathsf{RF}(r) = (\_, wM)$.
2. Thread thr has executed some buffer-write $wB' \in \sigma$ with $\mathsf{var}(wB') = \mathsf{var}(r) = \mathsf{var}(wM)$, but the corresponding memory-write wM' has not yet been executed in σ. Hence, thr cannot flush its buffers in any extension of σ that does not contain r (as wM' will not become executable until r gets executed).

The following lemmas state two key monotonicity properties of fence maps.

Lemma 5.2. *Consider two witness prefixes σ_1, σ_2 such that $\sigma_2 = \sigma_1 \circ wM$ for some memory-write wM executable in σ_1. We have $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\sigma_2}$. Moreover, if wM is a spurious memory-write in σ_1, then $\mathrm{FMap}_{\sigma_1} = \mathrm{FMap}_{\sigma_2}$.*

Proof. Since wM is executable in σ_1, the variable $\mathsf{var}(\sigma_1)$ is not held in σ_1. It follows directly from the definition of fence maps that the read sets $A_{\mathsf{thr},\mathsf{thr}'}$ can only increase in FMap_{σ_2} compared to FMap_{σ_1}. Hence, $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2) \leq \mathrm{FMap}_{\sigma_2}(\mathsf{thr}_1, \mathsf{thr}_2)$ for all $\mathsf{thr}_1, \mathsf{thr}_2$. Moreover, if wM is spurious then the sets $A_{\mathsf{thr},\mathsf{thr}'}$ are identical, thus $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2) = \mathrm{FMap}_{\sigma_2}(\mathsf{thr}_1, \mathsf{thr}_2)$ for all $\mathsf{thr}_1, \mathsf{thr}_2$.

The desired result follows. □

Lemma 5.3. *Consider two witness prefixes σ_1, σ_2 such that (i) $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$, (ii) $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\sigma_2}$, and (iii) $\mathcal{W}^M(\sigma_1) \setminus \mathcal{SW}^M(\sigma_1) \subseteq \mathcal{W}^M(\sigma_2)$. Let $e \in \mathcal{L}(X)$ be a thread event that is executable in σ_i for each $i \in [2]$, and let $\sigma_i' = \sigma_i \circ e$, for each $i \in [2]$. Then $\mathrm{FMap}_{\sigma_1'} \leq \mathrm{FMap}_{\sigma_2'}$.*

Proof. We distinguish cases based on the type of e.

1. *If e is a fence* fnc, the fence maps do not change, hence the claim holds directly from the fact that $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\sigma_2}$.

2. *If e is a read r*, observe that $\mathrm{FMap}_{\sigma'_i} \leq \mathrm{FMap}_{\sigma_i}$ for each $i \in [2]$. Hence we must have $\mathrm{FMap}_{\sigma'_2}(\mathsf{thr}_1, \mathsf{thr}_2) < \mathrm{FMap}_{\sigma_2}(\mathsf{thr}_1, \mathsf{thr}_2)$, for some thread $\mathsf{thr}_1 \in \mathrm{Threads}$ and $\mathsf{thr}_2 = \mathsf{thr}(r)$. Note that in fact $\mathrm{FMap}_{\sigma'_2}(\mathsf{thr}_1, \mathsf{thr}_2) = 0$, which occurs because $\mathrm{FMap}_{\sigma_2}(\mathsf{thr}_1, \mathsf{thr}_2)$ is the index of r in thr_2. Since $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\sigma_2}$, we have either $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2) = 0$ or $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2) = \mathrm{FMap}_{\sigma_2}(\mathsf{thr}_1, \mathsf{thr}_2)$. In either case, we have $\mathrm{FMap}_{\sigma'_1} \leq \mathrm{FMap}_{\sigma_2} = 0$, a contradiction.

3. *If e is a buffer-write wB*, observe that $\mathrm{FMap}_{\sigma_i} \leq \mathrm{FMap}_{\sigma'_i}$ for each $i \in [2]$. Hence we must have $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2) < \mathrm{FMap}_{\sigma'_1}(\mathsf{thr}_1, \mathsf{thr}_2)$, where $\mathsf{thr}_1 = \mathsf{thr}(wB)$ and thr_2 is some other thread. It follows that $v = \mathsf{var}(wB)$ is held in σ_1 by an active memory-write wM' (thus wM' is not spurious in σ_1), and $\mathrm{FMap}_{\sigma'_1}(\mathsf{thr}_1, \mathsf{thr}_2)$ is the index of thr_2 that contains a read r with $\mathsf{RF}(r) = (\_, wM')$. Since $\mathcal{W}^M(\sigma_1) \setminus \mathcal{SW}^M(\sigma_1) \subseteq \mathcal{W}^M(\sigma_2)$, we have $\mathcal{W}^M(') \in \sigma_2$ Since $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$, we have that wM' is an active memory-write in σ_2. Hence $\mathrm{FMap}_{\sigma'_2}(\mathsf{thr}_1, \mathsf{thr}_2) \geq \mathrm{FMap}_{\sigma'_1}(\mathsf{thr}_1, \mathsf{thr}_2)$, a contradiction.

The desired result follows. □

Note that there exist in total at most $n^{k \cdot k}$ different fence maps. Further, the following lemma gives a bound on the number of different fence maps among witness prefixes that contain the same thread events.

Lemma 5.4. *Let d be the number of variables. There exist at most $2^{k \cdot d}$ distinct witness prefixes σ_1, σ_2 such that $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$ and $\mathrm{FMap}_{\sigma_1} \neq \mathrm{FMap}_{\sigma_2}$.*

Proof. Given a trace σ, we define the *non-empty-buffer map* $\mathrm{NEBMap}_\sigma \colon \mathrm{Threads} \times \mathcal{G} \to \{\mathsf{True}, \mathsf{False}\}$, such that $\mathrm{NEBMap}_\sigma(\mathsf{thr}, v) = \mathsf{True}$ iff (i) thr does not hold variable v, and (ii) the buffer of thread thr on variable v is non-empty. Clearly there exist at most $2^{k \cdot d}$ different non-empty-buffer maps. We argue that for every two traces σ_1, σ_2, if $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$ and $\mathrm{NEBMap}_{\sigma_1} = \mathrm{NEBMap}_{\sigma_2}$ then $\mathrm{FMap}_{\sigma_1} = \mathrm{FMap}_{\sigma_2}$, from which the $2^{k \cdot d}$ bound of the lemma follows.

Assume towards contradiction that $\mathrm{FMap}_{\sigma_1} \neq \mathrm{FMap}_{\sigma_2}$. Hence, wlog, there exist two threads $\mathsf{thr}_1, \mathsf{thr}_2$ such that $\mathrm{FMap}_{\sigma_2}(\mathsf{thr}_1, \mathsf{thr}_2) > \mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2)$. Let $\mathrm{FMap}_{\sigma_2}(\mathsf{thr}_1, \mathsf{thr}_2) = m$, and consider the read r of thr_2 at index m. Let $v = \mathsf{var}(r)$ and $\mathsf{RF}(r) = (wB, wM)$ and $\mathsf{thr}_3 = \mathsf{thr}(wB)$. By the definition of fence maps, we have that thr_3 holds variable v in σ_2. By the definition of non-empty-buffer maps, we have that $\mathrm{NEBMap}_{\sigma_2}(\mathsf{thr}_3, v) = \mathsf{False}$, and since $\mathrm{NEBMap}_{\sigma_1} = \mathrm{NEBMap}_{\sigma_2}$, we also have $\mathrm{NEBMap}_{\sigma_1}(\mathsf{thr}_3, v) = \mathsf{False}$. Since $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$, we have that $wB \in \sigma_1$. Moreover, we have $wM \notin \sigma_1$, as otherwise, since $\mathrm{NEBMap}_{\sigma_1}(\mathsf{thr}_1) = \mathrm{NEBMap}_{\sigma_1}(\mathsf{thr}_2)$, we would have $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2) \geq m$. Hence, the buffer of thread thr_3 on variable v is non-empty in σ_1. Since $\mathrm{NEBMap}_{\sigma_1}(\mathsf{thr}_3, v) = \mathsf{False}$, we have that thr_3 holds v in σ_1. Thus, there is a read $r' \notin \sigma_1$ with $\mathsf{RF}(r') = (wB', wM')$, where $wM' <_{\mathsf{PO}} wM$. Since $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$, we have that $r' \notin \sigma_2$, which violates the observation of r' in any extension of σ_2.

The desired result follows. □

Algorithm VerifyPSO. We are now ready to describe our algorithm $\mathrm{VerifyPSO}$ for the problem $\mathrm{VPSO\text{-}rf}$. In high level, the algorithm enumerates all lower sets of $(\mathcal{L}(X), \mathsf{PO})$, i.e., the lower sets of the thread events. The crux of the algorithm is to guarantee that for every witness-prefix σ', the algorithm constructs a trace σ such that (i) $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, (ii) $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and (iii) $\mathrm{FMap}_\sigma \leq \mathrm{FMap}_{\sigma'}$. To achieve this, for a given lower set Y of $(\mathcal{L}(X), \mathsf{PO})$, the algorithm examines at most as many traces σ with $\mathcal{L}(\sigma) = Y$ as the number of different fence maps of witness prefixes with the same set of thread events. Hence, the algorithm examines significantly fewer traces than the $n^{k\cdot(d+1)}$ lower sets of (X, PO).

Algorithm 5.2 presents a formal description of $\mathrm{VerifyPSO}$. The algorithm maintains a worklist $\mathcal{S}$ of prefixes, and a set Done of explored pairs "(thread events, fence map)". Consider an iteration of the main loop in Line 2. First in the loop of Line 4 all spurious executable memory-writes are executed. Then Line 6 checks whether the witness is complete. In case it is not complete, the loop in Line 7 enumerates the possibilities to extend with a thread event. Crucially, the condition in Line 16 ensures that there are no duplicates with the same pair "(thread events, fence map)".

Algorithm 5.2: $\mathrm{VerifyPSO}(X, \mathsf{RF})$

```
Input: An event set $X$ and a reads-from function RF: $\mathcal{R}(X) \to \mathcal{W}(X)$
Output: A witness $\sigma$ that realizes $(X, \mathsf{RF})$ if $(X, \mathsf{RF})$ is realizable under PSO, else $\sigma = \bot$
1  $\mathcal{S} \leftarrow \{\epsilon\}$; Done $\leftarrow \{\emptyset\}$
2  while $\mathcal{S} \neq \emptyset$ do
3      Extract a trace $\sigma$ from $\mathcal{S}$
4      while $\exists$ spurious $wM$ PSO-executable in $\sigma$ do
5          $\sigma \leftarrow \sigma \circ wM$                         // Flush spurious memory-write $wM$
6      if $\mathcal{E}(\sigma) = X$ then return $\sigma$                // Witness found
7      foreach thread event $e$ PSO-executable in $\sigma$ do
8          Let $\sigma_e \leftarrow \sigma$
9          if $e$ is a read event with $\mathsf{RF}(r) = (wB, wM)$ then
10             if $\mathsf{thr}(r) \neq \mathsf{thr}(wB)$ and $wM \notin \sigma_e$ then
11                 $\sigma_e \leftarrow \sigma_e \circ wM$           // Execute the reads-from of $e$
12         else if $e$ is a fence event then
13             Let $\mu \leftarrow$ any linearization of $(\mathcal{PW}^M(\sigma, \mathsf{thr}(e)), \mathsf{PO})$
14             $\sigma_e \leftarrow \sigma_e \circ \mu$                 // Execute pending memory writes
15         $\sigma_e \leftarrow \sigma_e \circ e$                       // Finally, execute $e$
16         if $\nexists \sigma' \in$ Done s.t. $\mathcal{L}(\sigma_e) = \mathcal{L}(\sigma')$ and $\mathrm{FMap}_{\sigma_e} = \mathrm{FMap}_{\sigma'}$ then
17             Insert $\sigma_e$ in $\mathcal{S}$ and in Done        // Continue from $\sigma_e$
18 return $\bot$
```

Soundness. The soundness of $\mathrm{VerifyPSO}$ follows directly from the definition of PSO-executable events, and is similar to the case of $\mathrm{VerifyTSO}$.

Completeness. For each witness prefix σ', algorithm $\mathrm{VerifyPSO}$ generates a trace σ with (i) $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, (ii) $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and (iii) $\mathrm{FMap}_\sigma \leq \mathrm{FMap}_{\sigma'}$. This fact directly implies completeness, and it is achieved by the following key invariant. Consider that the algorithm has constructed a trace σ, and is attempting to extend σ with a thread event e. Further, let σ' be an arbitrary witness prefix with (i) $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, (ii) $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and (iii) $\mathrm{FMap}_\sigma \leq \mathrm{FMap}_{\sigma'}$. If σ' can be extended so that the next thread event is

e, then e is also executable in σ, and (by Lemma 5.2 and Lemma 5.3) the extension of σ with e maintains the invariant.

We now prove the argument in detail for the above σ, σ' and thread event e. Assume that $\sigma' \circ \kappa \circ e$ is a witness prefix as well, for a sequence of memory-writes κ. Consider the following cases.

1. If e is a read event, let $\boldsymbol{w} = (wB, wM) = \mathsf{RF}(e)$. If it is a local write (i.e., $\mathsf{thr}(\boldsymbol{w}) = \mathsf{thr}(e)$), necessarily $wB \in \sigma' \circ \kappa$, and since the traces agree on thread events, we have $wB \in \sigma$; thus e is executable in σ. Otherwise, $\boldsymbol{w}$ is a remote write (i.e., $\mathsf{thr}(\boldsymbol{w}) \neq \mathsf{thr}(e)$). Assume towards contradiction that e is not executable in σ; this can happen in two cases.

   In the first case, the variable $x = \mathsf{var}(e)$ is held by another (non-spurious) memory-write wM' in σ. Since $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\sigma')$, and $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$, the variable x is also held by wM' in $\sigma' \circ \kappa$. But then, both wM and wM' hold x in $\sigma' \circ \kappa$, a contradiction.

   In the second case, there is a write (wB', wM') with $\mathsf{var}(wM') = \mathsf{var}(e)$ and $wB' <_{\mathsf{PO}} e$ and $wM' \notin \sigma$. If $wM' \notin \sigma' \circ \kappa$, then e would read-from wB' from the buffer in $\sigma' \circ \kappa \circ e$, contradicting $\mathsf{RF}(e) = (\_, wM)$. Thus $wM' \in \sigma' \circ \kappa$, and further $wM \in \sigma' \circ \kappa$ with $wM' <_{\sigma' \circ \kappa} wM$. Since $\sigma' \circ \kappa \circ e$ is a witness prefix and $wB' <_{\mathsf{PO}} e$, we have $wB' \in \sigma'$. From this and $\mathcal{L}(\sigma) = \mathcal{L}(\sigma')$ we have that $wB' \in \sigma$ and wM' is pending in σ. This together gives us that wM' is spurious in σ. Consider the earliest memory-write pending in σ on the same buffer (i.e., $\mathsf{thr}(wM')$ and $\mathsf{var}(wM')$), denote it wM''. We have that $wM'' \leq_{\mathsf{PO}} wM'$ and wM'' is spurious in σ. Further, wM'' is executable in σ. But then it would have been added to σ in the while loop of Line 4, a contradiction.

2. Assume that e is a fence event, and let $wM_1, \dots, wM_j$ be the pending memory-writes of $\mathsf{thr}(e)$ in σ. Suppose towards contradiction that e is not executable. Then one of the wM_i is not executable, let $x = \mathsf{var}(wM_i)$. Similarly to the above, there can be two cases where this might happen.

   The first case is when wM_i must be read-from by some read event $r \notin \sigma$, but r is preceded by a local write (wB, wM) (i.e., $wB <_{\mathsf{PO}} r$) on the same variable x while $wM \notin \sigma$. A similar analysis to the previous case shows that the earliest pending write on $\mathsf{thr}(wM)$ for variable x is spurious, and thus already added to σ due to the while loop in Line 4, a contradiction.

   The second case is when the variable x is held in σ. Since $\mathrm{FMap}_\sigma \leq \mathrm{FMap}_{\sigma'}$, the variable x is also held in σ', and thus wM_i is not executable in σ' either. But then $\sigma' \circ \kappa \circ e$ cannot be a witness prefix, a contradiction.

In Fig. 5.6 we provide an intuitive illustration of the completeness idea. Consider the witness prefix σ' (lighter gray) and the corresponding trace σ constructed by the algorithm (darker gray). The fence fnc_1 is PSO-executable in σ but not in σ', since in the latter, $\mathsf{thr}(\mathrm{fnc}_1)$ has non-empty buffers, but the variables x and y are held by wM_1 and wM_2, respectively. This is equivalent to waiting until after r_1 and r_2 have been executed. Since executing r_2 implies having executed r_1, the fence map $\mathrm{FMap}_{\sigma'}(\mathsf{thr}_3, \mathsf{thr}_2)$ compresses this information by only pointing to r_2.

The following lemma states the correctness of $\mathrm{VerifyPSO}$, which together with the complexity argument establishes Theorem 5.2.

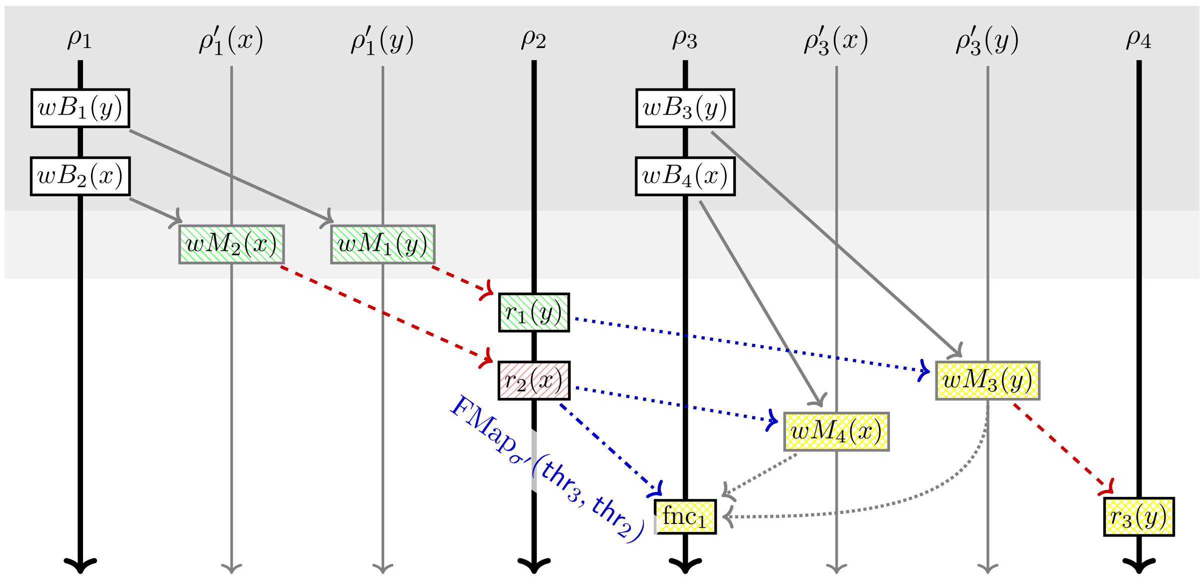


Figure 5.6: VerifyPSO completeness idea.

Lemma 5.5. (X, RF) *is realizable under PSO iff* VerifyPSO *returns a trace* $\sigma \neq \epsilon$*.*

Proof. We argue separately about soundness and completeness.

Soundness. We prove by induction that every trace σ extracted from $\mathcal{S}$ in Line 3 is a trace that realizes $(X|\mathcal{E}(\sigma), \mathsf{RF}|\mathcal{E}(\sigma))$ under PSO. The claim clearly holds for $\sigma = \epsilon$. Now consider a trace σ such that $\sigma \neq \emptyset$, hence σ was inserted in $\mathcal{S}$ in Line 17 while executing a previous iteration of the while-loop in Line 2. Let σ' be the trace that was extracted from $\mathcal{S}$ in that iteration, and consider the trace σ_e constructed in Line 15. Since σ_e is obtained by extending σ' with PSO-executable events, it follows that σ_e is well-formed. It remains to argue that $\mathsf{RF}_\sigma \subseteq \mathsf{RF}$. If e is not a read, then the claim holds by the induction hypothesis as $\mathcal{R}(\sigma_e) = \mathcal{R}(\sigma')$. Now assume that e is a read with $\mathsf{RF}_{\sigma_e}(e) = (wB', wM')$. Let $\mathsf{RF}(e) = (wB, wM)$, and assume towards contradiction that $wB \neq wB$. We distinguish the following cases.

1. If e reads-from wB' in σ_e, we have that $\mathsf{thr}(r) \neq \mathsf{thr}(wB)$. But then $wM \in \sigma_e$, hence wM has become PSO-executable, and thus $wM' \in \sigma_e$. Since e is the last event of σ_e this violates the fact that e reads-from wB' in σ_e.

2. If e reads-from wM' in σ_e, then $wM \in \sigma'$, and wM' was executed after wM in σ'. By the definition of PSO-executable events, wM' could not have been PSO-executable at that point, a contradiction.

It follows that $\mathsf{RF}_{\sigma_e}(e) = \mathsf{RF}(e)$, and this with the induction hypothesis gives us that $\mathsf{RF}_{\sigma_e}(r) = \mathsf{RF}(r)$ for all reads $r \in \mathcal{R}(\sigma_e)$. As a result, σ_e realizes $(X|\mathcal{E}(\sigma_e), \mathsf{RF}|\mathcal{E}(\sigma_e))$ under PSO. The soundness argument carries over directly to executions containing RMW and CAS instructions. Indeed, since in Line 7 we only consider atomic blocks that are PSO-executable (described in Section 5.2.4), consequently, the PSO-executable conditions of fences, reads, buffer-writes and memory-writes (as defined in Section 5.2.2) used to model RMW and CAS are preserved, which by the above argument implies soundness.

Completeness. Consider any trace σ^* that realizes (X, RF). We show by induction that for every prefix $\overline{\sigma}$ of σ^*, the algorithm examines a trace σ in Line 3 such that (i) $\mathcal{L}(\sigma) = \mathcal{L}(\overline{\sigma})$, (ii) $\mathcal{W}^M(\sigma) \setminus \mathcal{SW}^M(\sigma) \subseteq \mathcal{W}^M(\overline{\sigma})$, and (iii) $\mathrm{FMap}_\sigma \leq \mathrm{FMap}_{\overline{\sigma}}$.

The proof is by induction on the number of thread events of $\overline{\sigma}$. The statement clearly holds when $\overline{\sigma} = \epsilon$ due to the initialization of $\mathcal{S}$. For the inductive step, let $\overline{\sigma} = \overline{\sigma}' \circ \kappa \circ e$, where κ is a sequence of memory-writes and e is a thread event. By the induction hypothesis, the algorithm extracts a trace σ' in Line 3 such that (i) $\mathcal{L}(\sigma') = \mathcal{L}(\overline{\sigma}')$, (ii) $\mathcal{W}^M(\sigma') \setminus \mathcal{SW}^M(\sigma') \subseteq \mathcal{W}^M(\overline{\sigma}')$, (iii) $\mathrm{FMap}_{\sigma'} \leq \mathrm{FMap}_{\overline{\sigma}'}$. Let $\overline{\sigma}_1 = \overline{\sigma}' \circ \kappa$, and σ_1 be the trace σ' after the algorithm has extended σ' with all events in the while-loop of Line 4. By Lemma 5.2, we have $\mathrm{FMap}_{\overline{\sigma}'} \leq \mathrm{FMap}_{\overline{\sigma}_1}$. Since all events appended to σ' are spurious memory-writes in σ', by Lemma 5.2, we have $\mathrm{FMap}_{\sigma_1} = \mathrm{FMap}_{\sigma'}$ and thus $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\overline{\sigma}_1}$. Moreover, since the while-loop only appends spurious memory-writes to σ', we have $\mathcal{W}^M(\sigma_1) \setminus \mathcal{SW}^M(\sigma_1) \subseteq \mathcal{W}^M(\overline{\sigma}_1)$. Finally, we trivially have $\mathcal{L}(\sigma_1) = \mathcal{L}(\overline{\sigma}_1)$.

We now argue that e is PSO-executable in σ_1 in Line 7, and the statement holds for the new trace σ_e constructed in Line 15. We distinguish cases based on the type of e.

1. *If e is a buffer-write*, then $\mathcal{E}(\sigma_1) \cup \{e\}$ is a lower set of (X, RF), hence e is PSO-executable in σ_1. Thus, we have $\mathcal{L}(\overline{\sigma}) = \mathcal{L}(\sigma_e)$. Moreover, note that $\sigma_e = \sigma_1 \circ e$ and $\overline{\sigma} = \overline{\sigma}_1 \circ e$. By Lemma 5.3 on σ_1 and $\overline{\sigma}_1$, we have $\mathrm{FMap}_{\sigma_e} \leq \mathrm{FMap}_{\overline{\sigma}}$. Finally, we have $\mathcal{W}^M(\sigma_e) = \mathcal{W}^M(\sigma_1)$ and thus $\mathcal{W}^M(\sigma_e) \setminus \mathcal{SW}^M(\sigma_e) \subseteq \mathcal{W}^M(\overline{\sigma})$.

2. *If e is a read*, let $\mathsf{RF}(e) = (wB, wM)$ and $v = \mathsf{var}(e)$. We have $wB \in \overline{\sigma}_1$ and thus $wB \in \sigma_1$. If $\mathsf{thr}(wB) = \mathsf{thr}(e)$, then e is PSO-executable in σ_1. Now consider that $\mathsf{thr}(wB) \neq \mathsf{thr}(e)$, and assume towards contradiction that e is not PSO-executable in σ_1. There are two cases where this can happen.

   The first case is when the variable v is held by another memory write in σ_1. Since $\mathcal{L}(\sigma_1) = \mathcal{L}(\overline{\sigma}_1)$ and $\mathcal{W}^M(\sigma_1) \setminus \mathcal{SW}^M(\sigma_1) \subseteq \mathcal{W}^M(\overline{\sigma}_1)$, the variable v is also held by another memory write in $\overline{\sigma}_1$, and thus wM is neither in $\overline{\sigma}_1$ nor PSO-executable in $\overline{\sigma}_1$. Thus e is not PSO-executable in $\overline{\sigma}_1$ either, a contradiction.

   The second case is when there exists a read $r \notin \sigma_1$ such that $\mathsf{RF}(r) = (\_, wM)$, and there exists a local write event $w' = (wB', wM')$ with $\mathsf{thr}(wB') = \mathsf{thr}(r)$ but $wM' \notin \sigma_1$. Since $\overline{\sigma}_1$ is a witness prefix, we have $wM' \in \overline{\sigma}_1$, hence $wB' \in \overline{\sigma}_1$, and since $\mathcal{L}(\sigma_1) = \mathcal{L}(\overline{\sigma}_1)$, we also have $wB' \in \sigma_1$. Thus wM' is a pending memory write for the thread $\mathsf{thr}' = \mathsf{thr}(wM')$. Let wM'' be the earliest (wrt PO) pending memory-write of thr' for the variable v. Thus $wM'' <_{\mathsf{PO}} wM'$, and hence $wM'' \in \overline{\sigma}_1$. Note that wM'' is not read-from by any read not in σ_1, and hence wM'' is spurious in σ_1. But then, the while loop in Line 4 must have added wM'' in σ_1, a contradiction.

   It follows that e is PSO-executable in σ_1, and thus $\mathcal{L}(\overline{\sigma}) = \mathcal{L}(\sigma_e)$. Let $\sigma_2 = \sigma_1$ if $wM \in \sigma_1$, else $\sigma_2 = \sigma_1 \circ wM$. Observe that if wM is PSO-executable in σ_1, all pending memory-writes wM' on variable v of threads other than $\mathsf{thr}(wB)$ are spurious in σ', and thus all such buffers are empty in σ_1. It follows that $\mathrm{FMap}_{\sigma_2} \leq \mathrm{FMap}_{\sigma_1}$ and thus $\mathrm{FMap}_{\sigma_2} \leq \mathrm{FMap}_{\overline{\sigma}_1}$. Moreover, trivially $\mathcal{W}^M(\sigma_2) \setminus \mathcal{SW}^M(\sigma_2) \subseteq \overline{\sigma_1}$. Finally, executing e in σ_2 and $\overline{\sigma}_1$, we obtain respectively σ_e and $\overline{\sigma}$, and by Lemma 5.3, we have $\mathrm{FMap}_{\sigma_e} \leq \mathrm{FMap}_{\overline{\sigma}}$. Moreover, clearly $\mathcal{W}^M(\sigma_e) = \mathcal{W}^M(\sigma_2)$ and thus $\mathcal{W}^M(\sigma_e) \setminus \mathcal{SW}^M(\sigma_e) \subseteq \overline{\sigma}$.

3. *If e is a fence*, let $\mu = wM_1, \dots, wM_j$ be the sequence of pending memory-writes constructed in Line 13. By a similar analysis to the case where e is a a read event,

we have that μ contains at most one memory-write per variable, as all preceding ones (wrt PO) must be spurious. Assume towards contradiction that some pending memory write wM_i is not PSO-executable in σ, and let $v = \mathsf{var}(wM_i)$. There are two cases to consider.

a) wM_i is not PSO-executable because v is held in σ_1. Let wM be the memory-write that holds v in σ_1, and r be the corresponding read with $\mathsf{RF}(r) = (\_, wM)$ and $r \notin \sigma_1$. Since $\mathcal{L}(\sigma_1) = \mathcal{L}(\overline{\sigma}_1)$, we have $r \notin \overline{\sigma}_1$. Let $\mathsf{thr}_1 = \mathsf{thr}(e)$, $\mathsf{thr}_2 = \mathsf{thr}(r)$, and m be the index of r in thr_2. We have $\mathrm{FMap}_{\sigma_1}(\mathsf{thr}_1, \mathsf{thr}_2) \geq m$, and since $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\overline{\sigma}'}$, we also have $\mathrm{FMap}_{\overline{\sigma}'}(\mathsf{thr}_1, \mathsf{thr}_2) \geq m$. But then there is a pending memory-write $wM' \in \overline{\sigma}'$ with $\mathsf{thr}(wM') = \mathsf{thr}_1$ and $wM' \notin \overline{\sigma}_1$. Hence e is not PSO-executable in $\overline{\sigma}_1$, a contradiction.

b) wM_i is not PSO-executable because there exists a read $r \notin \sigma_1$ such that $\mathsf{RF}(r) = (\_, wM_i)$, and there exists a local write event $w = (wB, wM)$ with $\mathsf{thr}(wB) = \mathsf{thr}(r)$ but $wM \notin \sigma_1$. The analysis is similar to the case of e being a read, which leads to a contradiction.

Thus, we have that the fence e is PSO-executable in σ_1. It is straightforward to see that $\mathcal{W}^M(\sigma_e) \setminus \mathcal{SW}^M(\sigma_e) \subseteq \overline{\sigma}$, and thus it remains to argue that $\mathrm{FMap}_{\sigma_e} \leq \mathrm{FMap}_{\overline{\sigma}}$. Let $\sigma_1^j = \sigma_1 \circ wM_1, \dots, wM_{j-1}$. It suffices to argue that $\mathrm{FMap}_{\sigma_1^{j+1}} \leq \mathrm{FMap}_{\overline{\sigma}_1}$, as $\sigma_e = \sigma_1^{j+1} \circ e$ and $\overline{\sigma} = \overline{\sigma}_1 \circ e$, and the claim holds by Lemma 5.3 on σ_1^{j+1} and $\overline{\sigma}_1$. The proof is by induction on σ_1^i. The claim clearly holds for $i = 1$, as then $\sigma_1^1 = \sigma_1$ and we have $\mathrm{FMap}_{\sigma_1} \leq \mathrm{FMap}_{\overline{\sigma}_1}$. Now consider that for some $i > 1$, there exist two threads $\mathsf{thr}_1, \mathsf{thr}_2, \in \mathrm{Threads}$ such that $\mathrm{FMap}_{\sigma_1^i} > \mathrm{FMap}_{\sigma_1^{i-1}}(\mathsf{thr}_1, \mathsf{thr}_2)$. Hence, variable $v = \mathsf{var}(wM^i)$ is held in σ_1^i and wM^i is the respective active-memory-write, and thread thr_2 has a read r in index $m = \mathrm{FMap}_{\sigma_1^i}(\mathsf{thr}_1, \mathsf{thr}_2)$ with $\mathsf{RF}(r) = (\_, wM^i)$. In addition, there exists a buffer-write $wB \in \sigma_1$ such that $\mathsf{thr}(wB) = \mathsf{thr}_1$ and $\mathsf{var}(wB) = v$. Since $\mathcal{L}(\sigma_1^i) = \mathcal{L}(\overline{\sigma})$, we have that $wB \in \overline{\sigma}$ and $r \notin \overline{\sigma}$. Moreover, since $wM^i <_{\mathsf{PO}} e$, we have $wM^i \in \overline{\sigma}$. Hence wM^i is an active-memory-write in $\overline{\sigma}$ as well, and thus $\mathrm{FMap}_{\overline{\sigma}}(\mathsf{thr}_1, \mathsf{thr}_2) \geq \mathrm{FMap}_{\sigma_1^i}(\mathsf{thr}_1, \mathsf{thr}_2)$. At the end of the induction, we have $\mathrm{FMap}_{\sigma_1^{j+1}} \leq \mathrm{FMap}_{\overline{\sigma}}$, as desired.

This concludes the completeness argument for executions without RMW and CAS instructions.

When executions contain RMW and CAS instructions, additional argument has to be made for completeness, as follows. We proceed with the same induction argument as above, but additionally consider the inductive case where $\overline{\sigma} = \overline{\sigma}' \circ \kappa \circ e$ such that e is an atomic block corresponding to a RMW or a CAS instruction. In this case, e is a sequence of (i) a read r, (ii) a buffer-write wB, and optionally (in case the write-part of e is designated to proceed directly into the shared memory) (iii) a memory-write wM. Finally, e is preceded in its thread by a fence fnc.

First, since $\overline{\sigma}$ is a witness prefix we have $\mathrm{fnc} \in \mathcal{L}(\overline{\sigma}')$, and from the induction hypothesis regarding σ' such that $\mathcal{L}(\sigma') = \mathcal{L}(\overline{\sigma}')$ we also have $\mathrm{fnc} \in \mathcal{L}(\sigma')$. Thus all buffers of the thread of e are empty in both $\overline{\sigma}'$ and σ'. Then the argument is followed identically to above until Line 7, where we have to show that the atomic block e is PSO-executable in σ_1 in Line 7, and that consequently the induction statement holds for the new trace σ_e constructed in Line 15.

The crucial observation is that no event from the second event onward in the atomic block e is a read or a fence. This is important as reads and fences may need additional events executed right before them (see Lines 9–14), which would invalidate the atomicity of the atomic block. Given this observation, we simply utilize the PSO-executable requirements to prove the following. First, using the argument of Item 2 above we show that r is PSO-executable in σ_1, let σ_r denote the trace resulting after executing r. Second, using Item 1 above we show that wB is PSO-executable in σ_r, and further that (i) $\mathcal{L}(\sigma_r \circ wB) = \mathcal{L}(\overline{\sigma}' \circ \kappa \circ r \circ wB)$, (ii) $\mathcal{W}^M(\sigma_r \circ wB) \setminus \mathcal{SW}^M(\sigma_r \circ wB) \subseteq \mathcal{W}^M(\overline{\sigma}' \circ \kappa \circ r \circ wB)$, and (iii) $\mathrm{FMap}_{\sigma_r \circ wB} \leq \mathrm{FMap}_{\overline{\sigma}' \circ \kappa \circ r \circ wB}$. Finally, in the case where wM is part of the atomic block e, we have that wM is PSO-executable in $\sigma_r \circ wB$, resulting in the trace σ_e. Further, since the induction statement held already for $\sigma_r \circ wB$ with respect to $\overline{\sigma}' \circ \kappa \circ r \circ wB$ (see (i),(ii),(iii) above), we have that the induction statement holds also for σ_e with respect to $\overline{\sigma}$, which concludes the argument.

The desired result follows. □

We can now proceed with the proof of Theorem 5.2.

Theorem 5.2. $\mathrm{VPSO\text{-}rf}$ *for n events, k threads and d variables is solvable in $O(k \cdot n^{k+1} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d}))$. Moreover, if there are no fences, the problem is solvable in $O(k \cdot n^{k+1})$ time.*

Proof. Lemma 5.5 establishes the correctness, so here we focus on the complexity, and the following argument applies also for executions with RMW and CAS instructions. Since there are k threads, there exist at most n^k distinct traces σ_1, σ_2 with $\mathcal{L}(\sigma_1) \neq \mathcal{L}(\sigma_2)$. Because of the test in Line 16, for any two traces σ_1, σ_2 inserted in the worklist with $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$, we have $\mathrm{FMap}_{\sigma_1} \neq \mathrm{FMap}_{\sigma_2}$. If there are no fences, there is only one possible fence map, hence there are n^k traces inserted in $\mathcal{S}$. If there are fences, the number of different fence maps with $\mathrm{FMap}_{\sigma_1} \neq \mathrm{FMap}_{\sigma_2}$ when $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$ is bounded by $2^{k\cdot d}$ (by Lemma 5.4) and also by $n^{k\cdot(k-1)}$ (since there are at most that many difference fence maps). Hence the number of traces inserted in the worklist is bounded by $n^k \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d})$. Since there are k threads, for every trace σ_1 inserted in the worklist, the algorithm examines at most $k-1$ other traces σ_2 that are not inserted in the worklist because $\mathcal{L}(\sigma_1) = \mathcal{L}(\sigma_2)$ and $\mathrm{FMap}_{\sigma_1} = \mathrm{FMap}_{\sigma_2}$. Hence the algorithm examines at most $k \cdot n^k \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d})$ traces in total, while each such trace is handled in $O(n)$ time. Hence the total running time is $O(k \cdot n^{k+1} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d}))$.

Finally, note that if there are no fences present, we can completely drop the fence maps from the algorithm, which results in complexity $O(k \cdot n^{k+1})$.

The desired result follows. □

We now provide some insights on the relationship between $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$.

Relation between TSO and PSO verification. In high level, TSO might be perceived as a special case of PSO, where every thread is equipped with one buffer (TSO) as opposed to one buffer per global variable (PSO). However, the communication patterns between TSO and PSO are drastically different. As a result, our algorithm $\mathrm{VerifyPSO}$ is not applicable to TSO, and we do not see an extension of $\mathrm{VerifyTSO}$ for handling PSO efficiently. In particular, the minimal strategy of $\mathrm{VerifyPSO}$ on memory-writes is based on the following observation: for a read r observing a remote memory-write wM, it always suffices to execute wM exactly before executing r (unless wM has already been executed). This holds because the corresponding buffer contains memory-writes *only* on the same variable, and thus all such memory-writes that precede wM cannot be read-from by any subsequent read. This property does not hold for TSO: as there is a single buffer, wM might be executed as a result of flushing the buffer of thread thr(wM) to make another memory-write wM' visible, on a *different* variable than var(wM), and thus wM' might be observable by a subsequent read. Hence the minimal strategy of $\mathrm{VerifyPSO}$ on memory-writes does not apply to TSO. On the other hand, the maximal strategy of $\mathrm{VerifyTSO}$ is not effective for PSO, as it requires enumerating all lower sets of $(\mathcal{W}^M(X), \mathsf{RF})$, which are $n^{k\cdot d}$ many in PSO (where d is the number of variables), and thus this leads to worse bounds than the ones we achieve in Theorem 5.2.

Verifying PSO Executions with Store-store Fences. Here we describe our extension to handle $\mathrm{VPSO\text{-}rf}$ in the presence of store-store-fences.

A *store-store fence* event $\mathrm{storefnc}$ happening on a thread thr introduces further orderings into the program order PO, namely $wM <_{\mathsf{PO}} wM'$ for each $(wB, wM), (wB', wM') \in \mathcal{W}^M_{\mathsf{thr}}$ with $wB <_{\mathsf{PO}} \mathrm{storefnc} <_{\mathsf{PO}} wB'$.

Store-store fences are considered only for the PSO memory model, as they would have no effect in TSO, since in TSO all memory-writes within the same thread are already ordered. In fact, the TSO model can be seen as PSO with a store-store fence inserted after every buffer-write event.

We extend our notion of PSO-executability to accommodate store-store-fences. Given (X, PO) and σ with $\mathcal{E}(\sigma) \subseteq X$:

1. A store-store fence $\mathrm{storefnc} \in X \setminus \mathcal{E}(\sigma)$ is PSO-executable if $\mathcal{E}(\sigma) \cup \{\mathrm{storefnc}\}$ is a lower set of (X, PO).
2. An additional condition for a memory-write $wM \in X \setminus \mathcal{E}(\sigma)$ to be PSO-executable, is that every memory-write $wM' \in X \setminus \mathcal{E}(\sigma)$ with $wM' <_{\mathsf{PO}} wM$ is PSO-executable.

We consider a notion very similar to the fence maps introduced in Section 5.2.2, to efficiently represent the PSO-executability requirements introduced by store-store fences, namely *store-store fence maps* $\mathrm{SFMap}_\sigma\colon \mathrm{Threads} \times \mathrm{Threads} \to [n]$. While FMap_σ(thr) efficiently captures the requirements for executing a fence event of thr, SFMap_σ(thr) captures efficiently, in the same manner as FMap_σ(thr) does, the following. Consider the latest $\mathrm{storefnc} \in \mathcal{E}(\sigma)$ of thread thr, and consider that no memory-write of thr has been executed in σ after $\mathrm{storefnc}$ yet. Then, SFMap_σ(thr) captures the requirements for executing a memory-write of thr.

We utilize the store-store fence maps to refine our identification of duplicate witness-prefixes. This then gives us a time-complexity bound of $O(k \cdot n^{k+1} \cdot \min(n^{2\cdot k\cdot(k-1)}, 2^{k\cdot d}))$.

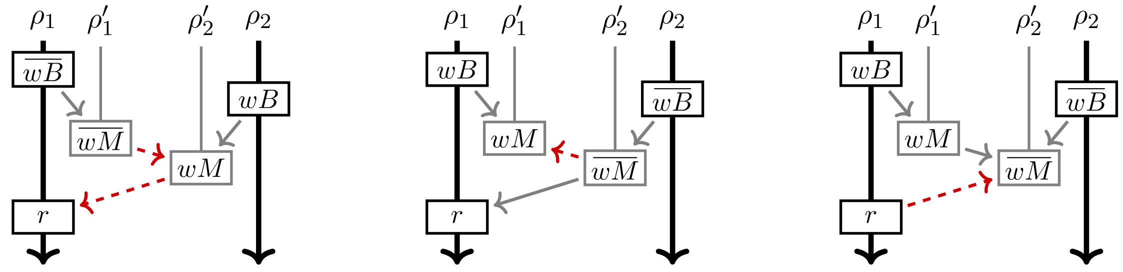
5.2.3 Closure for $\mathrm{VerifyTSO}$ and $\mathrm{VerifyPSO}$

In this section we introduce *closure*, a practical heuristic to efficiently detect whether a given instance (X, RF) of the verification problem $\mathrm{VTSO\text{-}rf}$ resp. $\mathrm{VPSO\text{-}rf}$ is unrealizable. Closure is sound, meaning that a realizable instance (X, RF) is never declared unrealizable by closure. Further, closure is not complete, which means there exist unrealizable instances (X, RF) not detected as such by closure. Finally, closure can be computed in time polynomial with respect to the number of events (i.e., size of X), irrespective of the underlying number of threads and variables.

Given an instance (X, RF), any solution of $\mathrm{VTSO\text{-}rf}/\mathrm{VPSO\text{-}rf}(X, \mathsf{RF})$ respects $\mathsf{PO}|X$, i.e., the program order upon X. Closure constructs the weakest partial order $P(X)$ that refines the program order (i.e., $P \sqsubseteq \mathsf{PO}|X$) and further satisfies for each read $r \in \mathcal{R}(X)$ with $\mathsf{RF}(r) = (wB, wM)$:

1. If $\mathsf{thr}(r) \neq \mathsf{thr}(\mathsf{RF}(r))$, then (i) $wM <_P r$ and (ii) $\overline{wM} <_P wM$ for any $(\overline{wB}, \overline{wM}) \in \mathcal{W}(X_{\mathsf{thr}(r)})$ such that $\overline{wM} \bowtie r$ and $\overline{wB} <_{\mathsf{PO}} r$.
2. For any $\overline{wM} \in \mathcal{W}^M(X_{\neq \mathsf{thr}(r)})$ such that $\overline{wM} \bowtie r$ and $\overline{wM} \neq wM$, $\overline{wM} <_P r$ implies $\overline{wM} <_P wM$.
3. For any $\overline{wM} \in \mathcal{W}^M(X_{\neq \mathsf{thr}(r)})$ such that $\overline{wM} \bowtie r$ and $\overline{wM} \neq wM$, $wM <_P \overline{wM}$ implies $r <_P \overline{wM}$.

If no above P exists, the instance $\mathrm{VTSO\text{-}rf/VPSO\text{-}rf}(X, \mathsf{RF})$ provably has no solution. In case P exists, each solution σ of $\mathrm{VTSO\text{-}rf/VPSO\text{-}rf}(X, \mathsf{RF})$ provably respects P (formally, $\sigma \sqsubseteq P$).



(a) Rule Item 1. Both new orderings are necessary, as a reversal of either of them would "hide" wM from r, making it impossible for r to read-from (wB, wM).

(b) Rule Item 2. The new ordering is necessary; its reversal would make $\overline{wM}$ appear between (wB, wM) and r, making it impossible for r to read-from (wB, wM).

(c) Rule Item 3. The new ordering is necessary; its reversal would make $\overline{wM}$ appear between (wB, wM) and r, making it impossible for r to read-from (wB, wM).

Figure 5.7: Illustration of the three closure rules.

The intuition behind closure is as follows. The construction starts with the program order $\mathsf{PO}|X$, and then, utilizing the above rules Item 1, Item 2 and Item 3, it iteratively adds further event orderings such that every witness execution provably has to follow the orderings. Consequently, if the added orderings induce a cycle, this serves as a proof that there exists no witness of the input instance (X, RF). The rules Item 1, Item 2 and Item 3 can intuitively be though of as simple reasoning arguments why specific orderings have to be present in each witness of (X, RF), and Fig. 5.7 provides an illustration of the rules. In each example of Fig. 5.7, the read r has to read-from the write (wB, wM), i.e., $\mathsf{RF}(r) = (wB, wM)$. All depicted events are on the same variable (which is omitted for clarity). The gray solid edges illustrate orderings already present in the partial order, and the red dashed edges illustrate the resulting new orderings enforced by the specific rule.

We leverage the guarantees of closure by computing it before executing $\mathrm{VerifyTSO}$ resp. $\mathrm{VerifyPSO}$. If no closure P of (X, RF) exists, the algorithm $\mathrm{VerifyTSO}$ resp. $\mathrm{VerifyPSO}$ does not need to be executed at all, as we already know that (X, RF) is unrealizable. Otherwise we obtain the closure P, we execute $\mathrm{VerifyTSO/VerifyPSO}$ to search for a witness of (X, RF), and we restrict $\mathrm{VerifyTSO/VerifyPSO}$ to only consider prefixes σ' respecting P (formally, $\sigma' \sqsubseteq P|\mathcal{E}(\sigma')$), since we know that each solution of $\mathrm{VTSO\text{-}rf/VPSO\text{-}rf}(X, \mathsf{RF})$ has to respect P.

The notion of closure, its beneficial properties, as well as construction algorithms are well-studied for the SC memory model [CCP$^+$17, AAJ$^+$19, Pav19]. Our conditions above extend this notion to TSO and PSO. Moreover, the closure we introduce here is *complete* for concurrent programs with two threads, i.e., if P exists then there is a valid trace realizing (X, RF) under the respective memory model.

5.2.4 Verifying Executions with Atomic Primitives

For clarity of presentation of the core algorithmic concepts, we have thus far neglected more involved atomic operations, namely atomic *read-modify-write* (RMW) and atomic *compare-and-swap* (CAS). We show how our approach handles verification of TSO and PSO executions that also include RMW and CAS operations here in a separate section. Importantly, our treatment retains the complexity bounds established in Theorem 5.1 and Theorem 5.2.

Atomic instructions. We consider the concurrent program under the TSO resp. PSO memory model, which can further atomically execute the following types of instructions.

1. A *read-modify-write* instruction rmw executes atomically the following sequence. It (i) reads, with respect to the TSO resp. PSO semantics, the value v of a global variable $x \in \mathcal{G}$, then (ii) uses v to compute a new value v', and finally (iii) writes the new value v' to the global variable x. An example of a typical rmw computation is fetch-and-add (resp. fetch-and-sub), where $v' = v + c$ for some positive (resp. negative) constant c.
2. A *compare-and-swap* instruction cas executes atomically the following sequence. It (i) reads, with respect to the TSO resp. PSO semantics, the value v of a global variable $x \in \mathcal{G}$, (ii) compares it with a value c, and (iii) if $v = c$ then it writes a new value v' to the global variable x.

Each instruction of the above two types blocks (i.e., it cannot get executed) until the buffer of its thread is empty (resp. all buffers of its thread are empty in PSO). Finally, the instruction specifies the nature of its final write. This write is either enqueued into its respective buffer (to be dequeued into shared memory at a later point), or it gets immediately flushed into the shared memory.

Atomic instructions modeling. In our approach we handle atomic RMW and CAS instructions without introducing them as new event types. Instead, we model these instructions as sequences of already considered events, i.e., reads, buffer-writes, memory-writes, and fences. We annotate some events of an atomic instruction to constitute an *atomic block*, which intuitively indicates that the event sequence of the atomic block cannot be interleaved with other events, thus respecting the semantics of the instruction.

1. A *read-modify-write* instruction rmw on a variable x is modeled as a sequence of four events: (i) a fence event, (ii) a read of x, (iii) a buffer-write of x, and (iv) a memory-write of x. The read and buffer-write events (ii)+(iii) are annotated as constituting an atomic block; in case the write of rmw is specified to proceed immediately to the shared memory, the memory-write event (iv) is also part of the atomic block.
2. For a *compare-and-swap* instruction cas we consider separately the following two cases. A *successful* cas (i.e., the write proceeds) is modeled the same way as a read-modify-write. A *failed* cas (i.e., the write does not proceed) is modeled simply as a fence followed by a read, with no atomic block.

Executable atomic blocks. Here we describe the TSO- and PSO-executability conditions for an atomic block. No further additions for executability are required, since no new event types are introduced to handle RMW and CAS instructions.

Consider an instance (X, RF) of $\mathrm{VTSO\text{-}rf}$, and a trace σ with $\mathcal{E}(\sigma) \subseteq X$. An atomic block containing a sequence of events $e_1, ..., e_j$ is TSO-executable in σ if:

1. for each $1 \leq i \leq j$ we have that $e_i \in X \setminus \mathcal{E}(\sigma)$, and
2. for each $1 \leq i \leq j$ we have that e_i is TSO-executable in $\sigma \circ e_1 ... e_{i-1}$.

Intuitively, an atomic block is TSO-executable if it can be executed as a sequence at once (i.e., without other events interleaved), and the TSO-executable conditions of each event (i.e., a read or a buffer-write or a memory-write or a fence) within the block are respected.

The PSO-executable conditions are analogous. Given an instance (X, RF) of $\mathrm{VPSO\text{-}rf}$ and a trace σ with $\mathcal{E}(\sigma) \subseteq X$, an atomic block of events $e_1, ..., e_j$ is PSO-executable in σ if:

1. for each $1 \leq i \leq j$ we have that $e_i \in X \setminus \mathcal{E}(\sigma)$, and
2. for each $1 \leq i \leq j$ we have that e_i is PSO-executable in $\sigma \circ e_1 ... e_{i-1}$.

Execution verification. Given the above executable conditions, the execution verification algorithms $\mathrm{VerifyTSO}$ and $\mathrm{VerifyPSO}$ only require minor technical modifications to verify executions including RMW and CAS instructions.

The core idea of the $\mathrm{VerifyTSO}$ resp. $\mathrm{VerifyPSO}$ modifications is to not extend prefixes with single events that are part of some atomic block, and instead extend the atomic blocks fully. This way, a lower set of (X, PO) is considered only if for each atomic block, the block is either fully present or fully not present in the lower set.

In $\mathrm{VerifyTSO}$ (Algorithm 5.1), in Line 4 we further consider each TSO-executable atomic block $e_1, ..., e_j$ not containing any memory-write event, and then in Line 5 we extend the prefix with the entire atomic block, i.e., $\sigma \leftarrow \sigma \circ e_1, ..., e_j$. Further, in Line 7 we further consider each TSO-executable atomic block $e_1, ..., e_j$ containing a memory-write event, and in Line 8 we then extend the prefix with the whole atomic block, i.e., $\sigma \leftarrow \sigma \circ e_1, ..., e_j$.

In $\mathrm{VerifyPSO}$ (Algorithm 5.2), in the loop of Line 7 we further consider each PSO-executable atomic block. Consider a fixed iteration of this loop with an atomic block $e_1, ..., e_j$. The first event of the atomic block e_1 is a read, thus the condition in Line 9 is evaluated true with e_1 and the control flow moves to Line 10. Later, the condition in Line 12 is evaluated false (since e_1 is a read). Finally, in Line 15 the prefix is extended with the whole atomic block, i.e., $\sigma_e \leftarrow \sigma_e \circ e_1, ..., e_j$.

For $\mathrm{VerifyTSO}$ the argument of maintaining maximality in the set of thread events applies also in the presence of RMW and CAS, and thus the bound of Theorem 5.1 is retained. Similarly, for $\mathrm{VerifyPSO}$ the enumeration of fence maps and the maximality in the spurious writes is preserved also with RMW and CAS, and hence the bound of Theorem 5.2 holds.

Closure. When verifying executions with RMW and CAS instructions, while the closure retains its guarantees as is, it can more effectively detect unrealizable instances with additional rules. Specifically, the closure P of (X, RF) satisfies the rules 1–3 described in Section 5.2.3, and additionally, given an event e and an atomic block $e_1, ..., e_j$, P satisfies the following.

4. If $e_i <_P e$ for any $1 \leq i \leq j$, then $e_j <_P e$ (i.e., if some part of the block is before e then the entire block is before e).

5. If $e <_P e_i$ for any $1 \leq i \leq j$, then $e <_P e_1$ (i.e., if e is before some part of the block then e is before the entire block).

5.3 Reads-From SMC for TSO and PSO

In this section we present $\mathrm{RF\text{-}SMC}$, an exploration-optimal reads-from SMC algorithm for TSO and PSO. The algorithm $\mathrm{RF\text{-}SMC}$ is based on the reads-from algorithm for SC [AAJ$^+$19], and adapted in this work to handle the relaxed memory models TSO and PSO. The algorithm uses as subroutines $\mathrm{VerifyTSO}$ (resp. $\mathrm{VerifyPSO}$) to decide whether any given class of the RF partitioning is consistent under the TSO (resp. PSO) semantics.

$\mathrm{RF\text{-}SMC}$ is a recursive algorithm, each call of $\mathrm{RF\text{-}SMC}$ is argumented by a tuple $(\tau, \mathsf{RF}, \sigma, \mathsf{mrk})$ where the following points hold:

- τ is a sequence of thread events. Let X denote the set of events of τ together with their memory-write counterparts, formally $X = \mathcal{E}(\tau) \cup \{wM : \exists (wB, wM) \in \mathcal{W} \text{ such that } wB \in \mathcal{W}^B(\tau)\}$.
- $\mathsf{RF}\colon \mathcal{R}(X) \to \mathcal{W}(X)$ is a desired reads-from function.
- σ is a concrete valid trace that is a witness of (X, RF), i.e., $\mathcal{E}(\sigma) = X$ and $\mathsf{RF}_\sigma = \mathsf{RF}$.
- $\mathsf{mrk} \subseteq \mathcal{R}(\tau)$ is a set of reads that are *marked* to be committed to the source they read-from in σ.

Further, a globally accessible set of schedule sets called schedules is maintained throughout the recursion. The schedules set is initialized empty (schedules $= \emptyset$) and the initial call of the algorithm is argumented with empty sequences and sets — $\mathrm{RF\text{-}SMC}(\epsilon, \emptyset, \epsilon, \emptyset)$.

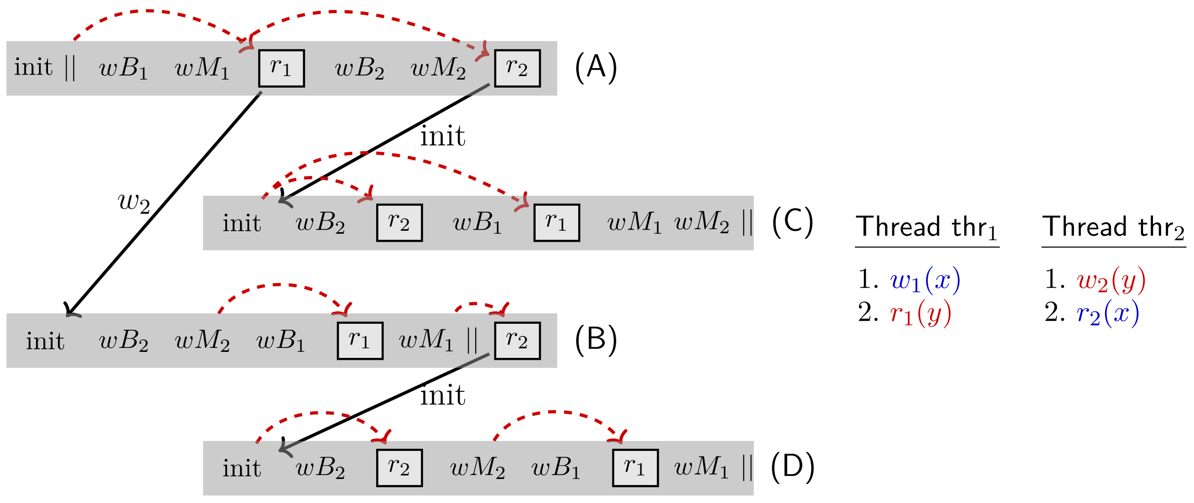
Algorithm 5.3 presents the pseudocode of $\mathrm{RF\text{-}SMC}$. In each call of $\mathrm{RF\text{-}SMC}$, a number of possible changes (or *mutations*) of the desired reads-from function RF is proposed in iterations of the loop in Line 5. Consider the read r of a fixed iteration of the Line 5 loop. First, in Lines 6–8 a partial order P is constructed to capture the causal past of write events. In Lines 9–11 the set of mutations for r is computed. Then in each iteration of the Line 12 loop a mutation is constructed (Lines 13–16). Here the partial order P is utilized in Line 13 to help determine the event set of the mutation. The constructed mutation, if deemed novel (checked in Line 17), is probed whether it is realizable (in Line 18). In case it is realizable, it gets added into schedules in Line 21. After all the mutations are proposed, then in Lines 22–25 a number of recursive calls of $\mathrm{RF\text{-}SMC}$ is performed, and the recursive $\mathrm{RF\text{-}SMC}$ calls are argumented by the specific schedules retrieved.

Example. Fig. 5.8 illustrates the run of $\mathrm{RF\text{-}SMC}$ on a simple concurrent program (the run is identical under both TSO and PSO). The gray boxes represent individual calls to $\mathrm{RF\text{-}SMC}$. The sequence of events inside a gray box is the trace $\widetilde{\sigma}$; the part left of the $||$-separator is σ (before extending), and to the right is $\widehat{\sigma}$ (the extension). The red dashed arrows represent the reads-from function $\mathsf{RF}_{\widetilde{\sigma}}$. Each black solid arrow represents a recursive call, where the arrow's outgoing tail and label describes the corresponding mutation. An initial trace (A) is obtained where $r_1(y)$ reads-from the initial event and $r_2(x)$ reads-from $w_1(x)$. Here two mutations

```
Algorithm 5.3: RF-SMC($\tau$, RF, $\sigma$, mrk)
   Input: Sequence $\tau$, desired reads-from RF, valid trace $\sigma$ with $\mathsf{RF}_\sigma = \mathsf{RF}$, marked reads mrk.
1  $\widetilde{\sigma} \leftarrow \sigma \circ \widehat{\sigma}$ where $\widehat{\sigma}$ is an arbitrary maximal extension of $\sigma$   // Maximally extend trace $\sigma$
2  $\widetilde{\tau} \leftarrow \tau \circ \widehat{\sigma}|\mathcal{L}(\widehat{\sigma})$   // Extend $\tau$ with the thread-events subsequence of extension $\widehat{\sigma}$
3  foreach $r \in \mathcal{R}(\widehat{\sigma})$ do   // Reads of the extension $\widehat{\sigma}$
4      schedules(pre$_{\widetilde{\tau}}(r)$) $\leftarrow \emptyset$   // Initialize new schedule set
5  foreach $r \in \mathcal{R}(\widetilde{\tau}) \setminus$ mrk do   // Unmarked reads
6      $P \leftarrow \mathsf{PO}|\mathcal{E}(\widetilde{\sigma})$   // Program order on all the events of $\widetilde{\sigma}$
7      foreach $r' \in \mathcal{R}(\widetilde{\tau}) \setminus \{r\}$ with thr$(r') \neq$ thr(RF$_{\widetilde{\sigma}}(r')$) do
8          insert $wM \rightarrow r'$ into $P$ where RF$_{\widetilde{\sigma}}(r') = wM$   // Add reads-from ordering into $P$
9      mutations $\leftarrow \{(wB, wM) \in \mathcal{W}(\widetilde{\sigma}) \mid r \bowtie wM\} \setminus \{\mathsf{RF}_{\widetilde{\sigma}}(r)\}$
10     if $r \notin \mathcal{R}(\widehat{\sigma})$ then   // If $r$ is not part of the extension then
11         mutations $\leftarrow$ mutations $\cap\, \mathcal{W}(\widehat{\sigma})$   // Only consider writes of the extension
12     foreach $(wB, wM) \in$ mutations do   // Considered mutations
13         causesafter $\leftarrow \{e \in \mathcal{E}(\widetilde{\tau}) \mid r <_{\widetilde{\tau}} e$ and $e \leq_P wB\}$   // Causal past of $wB$ after $r$
14         $\tau' \leftarrow$ pre$_{\widetilde{\tau}}(r) \circ \widetilde{\tau}|$causesafter   // $r$-prefix followed by causesafter
15         $X' \leftarrow \mathcal{E}(\tau') \cup \{wM' : (wB', wM') \in \mathcal{W}(\widetilde{\sigma})$ and $wB' \in \mathcal{W}^B(\tau')\}$   // New event set
16         RF$' \leftarrow \{(r', \mathsf{RF}_{\widetilde{\sigma}}(r')) : r' \in \mathcal{R}(\tau')$ and $r' \neq r\} \cup \{(r, (wB, wM))\}$   // New reads-from
17         if $(\tau', \mathsf{RF}', \_, \_) \notin$ schedules(pre$_{\widetilde{\tau}}(r)$) then   // If this is a new schedule
18             $\sigma' \leftarrow$ Witness$(X', \mathsf{RF}')$   // VerifyTSO or VerifyPSO
19             if $\sigma' \neq \perp$ then   // If the mutation is realizable
20                 mrk$' \leftarrow$ (mrk $\cap\, \mathcal{R}(\tau')) \cup \mathcal{R}$(causesafter)   // New marked reads
21                 add $(\tau', \mathsf{RF}', \sigma', \mathsf{mrk}')$ to schedules(pre$_{\widetilde{\tau}}(r)$)   // Add the new schedule
22 foreach $\widehat{r} \in \mathcal{R}(\widehat{\sigma})$ in the reverse order of $<_{\widehat{\sigma}}$ do   // Extension reads from the end
23     foreach $(\tau', \mathsf{RF}', \sigma', \mathsf{mrk}') \in$ schedules(pre$_{\widetilde{\tau}}(\widehat{r})$) do   // Schedules mutating $\widehat{r}$
24         RF-SMC$(\tau', \mathsf{RF}', \sigma', \mathsf{mrk}')$   // Recursive call on the schedule
25     delete schedules(pre$_{\widetilde{\tau}}(\widehat{r})$)   // The set is explored, hence it can be deleted
```

are probed and both are realizable. In the first mutation (B), $r_1(y)$ is mutated to read-from $w_2(y)$ and $r_2(x)$ is not retained (since it appears after $r_1(y)$ and it is not in the causal past of $w_2(y)$). In the second mutation (C), $r_2(x)$ is mutated to read-from the initial event and $r_1(y)$ is retained (since it appears before $r_2(x)$) with initial event as its reads-from. After both mutations are added to schedules, recursive calls are performed in the reverse order of reads appearing in the trace, thus starting with (C). Here no mutations are probed since there are no events in the extension, the algorithm backtracks to (A) and a recursive call to (B) is performed. Here one mutation (D) is added, where $r_2(x)$ is mutated to read-from the initial event and $r_1(y)$ is retained (it appears before $r_2(x)$) with $w_2(y)$ as its reads-from. The call to (D) is performed and here no mutations are probed (there are no events in the extension). The algorithm backtracks and concludes, exploring four RF partitioning classes in total.

Extension from SC to TSO and PSO. The fundamental challenge in extending the SC algorithm of [AAJ+19] to TSO and PSO is verifying execution consistency for TSO and PSO, which we address in Section 5.2 (Line 18 of Algorithm 5.3 calls our algorithms VerifyTSO and VerifyPSO). The main remaining challenge is then to ensure that the exploration optimality is preserved. To that end, we have to exclude certain events (in particular, memory-write events) from subsequences and event subsets that guide the exploration of Algorithm 5.3. Specifically, the sequences τ, τ', and $\widetilde{\tau}$ invariantly contain only the thread events, which is ensured in Line 2, Line 13 and Line 14, and then in Line 15 the absent memory-writes are reintroduced.



init || wB_1 wM_1 r_1 wB_2 wM_2 r_2 (A)
init
w_2
init wB_2 r_2 wB_1 r_1 wM_1 wM_2 || (C)
init wB_2 wM_2 wB_1 r_1 wM_1 || r_2 (B)
init
init wB_2 r_2 wM_2 wB_1 r_1 wM_1 || (D)
Thread thr$_1$
1. $w_1(x)$
2. $r_1(y)$
Thread thr$_2$
1. $w_2(y)$
2. $r_2(x)$

Figure 5.8: Visualization of a $\mathrm{RF\text{-}SMC}$ run.

No such distinction is required under SC.

$\mathrm{RF\text{-}SMC}$ is sound, complete and exploration-optimal, and we formally state this in Theorem 5.3.

Theorem 5.3. *Consider a concurrent program $\mathscr{P}$ with k threads and d variables, under a memory model $\mathcal{M} \in \{$TSO, PSO$\}$ with trace space $\mathcal{T}_{\mathcal{M}}^{max}$ and n being the number of events of the longest trace in $\mathcal{T}_{\mathcal{M}}^{max}$. $\mathrm{RF\text{-}SMC}$ is a sound, complete and exploration-optimal algorithm for local state reachability in $\mathscr{P}$, i.e., it explores only maximal traces and visits each class of the RF partitioning exactly once. The time complexity is $O\left(\alpha \cdot |\mathcal{T}_{\mathcal{M}}^{max}/\sim_{\mathsf{RF}}|\right)$, where*

1. *$\alpha = n^{O(k)}$ under $\mathcal{M}$ =TSO, and*
2. *$\alpha = n^{O(k^2)}$ under $\mathcal{M}$ =PSO.*

Proof. Let $\mathcal{M}$ be the memory model from $\{$TSO, PSO$\}$. We sketch the correctness (i.e., soundness and completeness), exploration-optimality, and time complexity of $\mathrm{RF\text{-}SMC}$.

Soudness. The soundness trivially follows from soundness of $\mathrm{VerifyTSO}$ used in TSO and of $\mathrm{VerifyPSO}$ used in PSO, which are used as subroutines for verifying execution consistency.

Completeness. The completeness of $\mathrm{RF\text{-}SMC}$ rests upon the completeness of its variant for SC introduced by [AAJ+19]. We now argue that the modifications to accomodate TSO and PSO have no effect on completeness. First, consider in each recursive call the sequences τ (argument of the call) and $\widetilde{\tau}$ (Line 2 of Algorithm 5.3). The sequence τ (resp. $\widetilde{\tau}$) in each call contains exactly the thread events of the trace σ (resp. $\widehat{\sigma}$) in that call. Thus τ (resp. $\widetilde{\tau}$) contains exactly the events of local traces of each thread in σ (resp. $\widehat{\sigma}$). This gives that the usage of $\widetilde{\tau}$ to manipulate schedules is equivalent to the SC case where there are only thread events. Second, the proper event set formed in Line 15 of Algorithm 5.3 is uniquely determined, and mirrors the set of events $\mathcal{E}(\tau')$ of the sequence τ' created in Line 14 of Algorithm 5.3. The set of events $\mathcal{E}(\tau')$ would be considered for the mutation in the SC case, given that we consider buffer-writes of $\mathcal{E}(\tau')$ as simply atomic write events that SC models. Finally, the witness subroutine is handled by $\mathrm{VerifyTSO}$ for TSO and $\mathrm{VerifyPSO}$ for PSO, whose completeness is established in Lemma 5.1 and Lemma 5.5. Thus the completeness of $\mathrm{RF\text{-}SMC}$ follows.

Exploration-optimality. The exploration-optimality argument mirrors the one made by [AAJ+19], and can be simply established by considering the sequence $\widetilde{\tau}$ (Line 2 of Algorithm 5.3) of each recursive call. The sequences $\widetilde{\tau}$ of all calls, coalesced together with equal events merged, form a rooted tree. Each node in the tree with multiple children is some read r. Let us label each child branch by the source r reads-from, in the trace of the same call that owns the sequence introducing the child branch. The source for r is different in each branch, and thus the same trace can never appear when following two different branches of r. The exploration-optimality follows.

Time complexity. From exploration-optimality we have that a run of $\mathrm{RF\text{-}SMC}$ performs exactly $|\mathcal{T}_{\mathcal{M}}^{max}/\sim_{\mathsf{RF}}|$ calls. It remains to argue that each class of $\mathcal{T}_{\mathcal{M}}^{max}/\sim_{\mathsf{RF}}$ spends time $O(\alpha)$ where

1. $\alpha = n^{k+O(1)}$ under $\mathcal{M} = \mathsf{TSO}$, and
2. $\alpha = n^{k+O(1)} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d})$ under $\mathcal{M} = \mathsf{PSO}$.

We split this argument to three parts.

1. Lines 1-4 spend $O(n)$ time per call.
2. One call of $\mathrm{VerifyTSO}$ resp. $\mathrm{VerifyPSO}$ spends $O(\alpha)$ time by Theorem 5.1 resp. Theorem 5.2. Thus Lines 5-21 spend $O(n^2 \cdot \alpha)$ time per call.
3. The total number of mutations added into schedules (on Line 21) equals $|\mathcal{T}_{\mathcal{M}}^{max}/\sim_{\mathsf{RF}}|-1$, i.e., it equals the total number of calls minus the initial call. However, we note that (i) each call adds only polynomialy many new schedules, and (ii) a call to a new schedule is considered work spent on the *class corresponding to* the new schedule. Thus Lines 22-25 spend $O(1)$ amortized time per recursive call, and $O(1)$ time is spent in this location per partitioning class.

The complexity result follows.

□

Remark 5.1 (Handling locks and atomic primitives)**.** For clarity of presentation, so far we have neglected locks in our model. However, lock events can be naturally handled by our approach as follows. We consider each lock-release event $release$ as an atomic write event (i.e., its effects are not deferred by a buffer but instead are instantly visible to each thread). Then, each lock-acquire event $acquire$ is considered as a read event that accesses the unique memory location.

In SMC, we enumerate the reads-from functions that also consider locks, thus having constraints of the form $\mathsf{RF}(acquire) = release$. This treatment totally orders the critical sections of each lock, which naturally solves all reads-from constraints of locks, and further ensures that no thread acquires an already acquired (and so-far unreleased) lock. Therefore $\mathrm{VerifyTSO}/\mathrm{VerifyPSO}$ need not take additional care for locks. The approach to handle locks by [AAJ+19] directly carries over to our exploration algorithm $\mathrm{RF\text{-}SMC}$.

The atomic operations read-modify-write (RMW) and compare-and-swap (CAS) are modeled as in Section 5.2.4, except for the fact that the atomic blocks are not necessary for SMC. Then $\mathrm{RF\text{-}SMC}$ can handle programs with such operations as described by [AAJ+19]. In particular, the modification of $\mathrm{RF\text{-}SMC}$ (Algorithm 5.3) to handle RMW and CAS operations is as follows.

Consider an iteration of the loop in Line 5 where r is the read-part of either a RMW or a successful CAS, denoted e, and let $(wB'', wM'') = \mathsf{RF}_{\widetilde{\sigma}}(r)$. Then, in Line 9 we additionally consider as an extra mutation each atomic instruction e' satisfying:

1. The read-part r' of e' reads-from the write-part (wB, wM) of e (i.e., $\mathsf{RF}_{\widetilde{\sigma}}(r') = (wB, wM)$), and
2. e' is either a RMW, or it will be a successful CAS when it reads-from (wB'', wM''). In this case, let (wB', wM') denote the write-part of e'.

When considering the above mutation in Line 12, we set $\mathsf{RF}'(r') = (wB'', wM'')$ and $\mathsf{RF}'(r) = (wB', wM')$ in Line 16, which intuitively aims to "reverse" e and e' in the trace.

5.4 Experiments

In this section we report on an experimental evaluation of the consistency verification algorithms $\mathrm{VerifyTSO}$ and $\mathrm{VerifyPSO}$, as well as the reads-from SMC algorithm $\mathrm{RF\text{-}SMC}$. We have implemented our algorithms as an extension in Nidhugg [AAA+15], a state-of-the-art stateless model checker for multithreaded C/C++ programs with pthreads library, operating on LLVM IR.

Benchmarks. For our experimental evaluation of both the consistency verification and SMC, we consider 109 benchmarks coming from four different categories, namely: (i) SV-COMP benchmarks, (ii) benchmarks from related papers and works [AAJ+19, AAA+15, HH16, CPT19], (iii) mutual-exclusion algorithms, and (iv) dynamic-programming benchmarks of [CPT19]. Although the consistency and SMC algorithms can be extended to support atomic compare-and-swap and read-modify-write primitives (cf. Remark 5.1), our current implementation does not support these primitives. Therefore, we used all benchmarks without such primitives that we could obtain (e.g., we include every benchmark of the relevant SC reads-from work [AAJ+19] except the one benchmark with compare-and-swap). Each benchmark comes with a scaling parameter, called the *unroll* bound, which controls the bound on the number of iterations in all loops of the benchmark (and in some cases it further controls the number of threads).

Technical details. For all our experiments we have used a Linux machine with Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz (12 CPUs) and 128GB of RAM. We have run the Nidhugg version of 26. November 2020, with Clang and LLVM version 8.

5.4.1 Experiments on Execution Verification for TSO and PSO

In this section we perform an experimental evaluation of our execution verification algorithms $\mathrm{VerifyTSO}$ and $\mathrm{VerifyPSO}$. For the purpose of comparison, we have also implemented within Nidhugg the naive lower-set enumeration algorithm of [AAJ+19, BE19], extended to TSO and PSO. Intuitively, this approach enumerates all lower sets of the program order restricted to the input event set, which yields a better complexity bound than enumerating write-coherence orders (even with just one location). The extensions to TSO and PSO are called NaiveVerifyTSO and NaiveVerifyPSO, respectively, and their worst-case complexity is $n^{2\cdot k}$ and $n^{k\cdot(d+1)}$, respectively (as discussed in Section 5.1). Further, for each of the above verification algorithms, we consider two variants, namely, with and without the closure heuristic of Section 5.2.3.

Setup. We evaluate the verification algorithms on execution consistency instances induced during SMC of the benchmarks. For TSO we have collected 9400 instances, 1600 of which are not realizable. For PSO we have collected 9250 instances, 1400 of which are not realizable. For each instance, we run the verification algorithms subject to a timeout of one minute, and we report the average time achieved over 5 runs.

We generate and collect the $\mathrm{VTSO\text{-}rf}$ and $\mathrm{VPSO\text{-}rf}$ instances that appear during SMC of our benchmarks using the reads-from SMC algorithm $\mathrm{RF\text{-}SMC}$. We supply the unroll bounds to the benchnmarks so that the created $\mathrm{VTSO\text{-}rf}/\mathrm{VPSO\text{-}rf}$ instances are solvable in a time reasonable for experiments (i.e., more than a tiny fraction of a second, and within a minute). We run each benchmark with several such unroll bounds. Further, as a filter of too small instances, we only consider realizable instances where at least one verification algorithm without closure took at least 0.05 seconds.

For each SMC run, to collect a diverse set of instances, we collect every fifth realizable instance we encounter, and every fifth unrealizable instance we encounter. In this way we collect 50 realizable instances and 20 unrealizable instances. For each collected instance, we run all verification algorithms and closure/no-closure variants 5 times, and average the results. We run all verification algorithms subject to a timeout of one minute.

Below we present the results using logarithmically scaled plots, where the opaque and semi-transparent red lines represent identity and an order-of-magnitude difference, respectively.

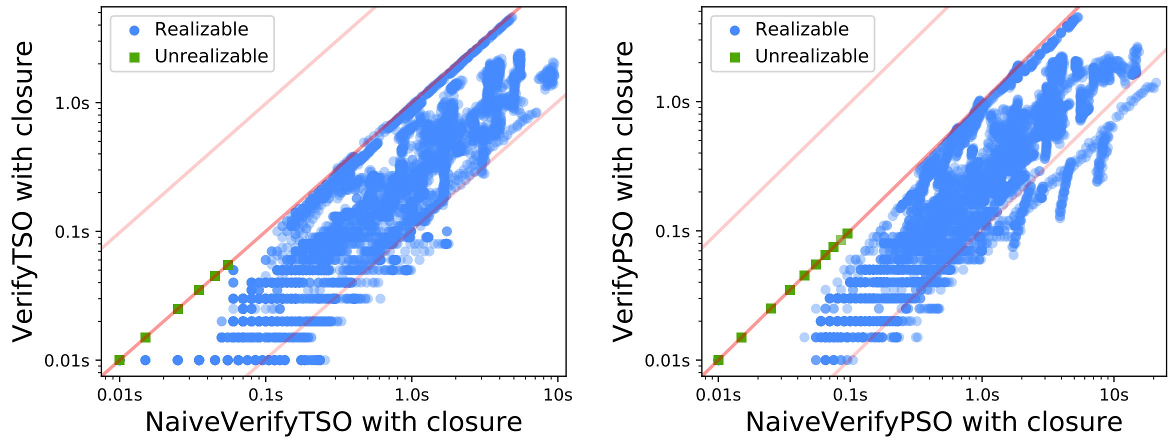


Figure 5.9: Verification in TSO (left) and PSO (right) with closure.

Results – algorithms with closure. Here we evaluate the verification algorithms that execute the closure as the preceding step. The plots in Fig. 5.9 present the results for TSO and PSO.

In TSO, our algorithm VerifyTSO is similar to or faster than NaiveVerifyTSO on the realizable instances (blue dots), and the improvement is mostly within an order of magnitude. All unrealizable instances (green squares) were detected as such by closure, and hence the closure-using VerifyTSO and NaiveVerifyTSO coincide on these instances.

We make similar observations in PSO, where $\mathrm{VerifyPSO}$ is similar or superior to $\mathrm{NaiveVerifyPSO}$ for the realizable instances, and the algorithms are indentical on the unrealizable instances, since these are all detected as unrealizable by closure.

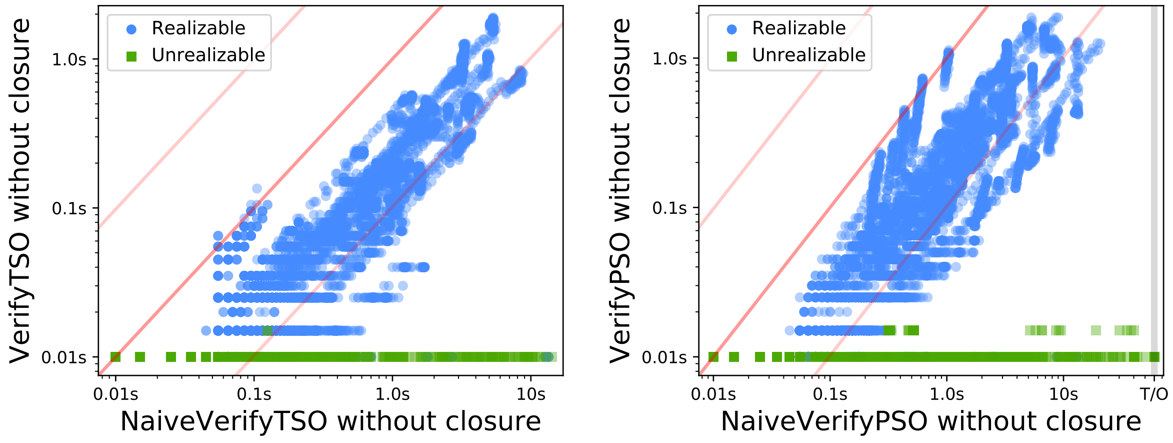


Figure 5.10: Verification in TSO (left) and PSO (right) without closure.

Results – algorithms without closure. Here we evaluate the verification algorithms without the closure. The plots in Fig. 5.10 present the results for TSO and PSO.

In TSO, the algorithm $\mathrm{VerifyTSO}$ outperforms $\mathrm{NaiveVerifyTSO}$ on most of the realizable instances (blue dots). Further, $\mathrm{VerifyTSO}$ significantly outperforms $\mathrm{NaiveVerifyTSO}$ on the unrealizable instances (green squares). This is because without closure, a verification algorithm can declare an instance unrealizable only after an exhaustive exploration of its respective lower-set space. $\mathrm{VerifyTSO}$ explores a significantly smaller space compared to $\mathrm{NaiveVerifyTSO}$, as outlined in Section 5.1.

Similar observations as above hold in PSO for the algorithms $\mathrm{VerifyPSO}$ and $\mathrm{NaiveVerifyPSO}$ without closure, both for the realizable and the unrealizable instances.

Results – effect of closure. Here we compare each verification algorithm against itself, where one version uses closure and the other one does not. Recall that closure constructs a partial order that each witness has to satisfy, and declares an instance unrealizable when it detects that the partial order cannot be constructed for this instance (we refer to Section 5.2.3 for details).

Fig. 5.11 presents the comparison of $\mathrm{VerifyTSO}$ (used for $\mathrm{VTSO\text{-}rf}$) and $\mathrm{VerifyPSO}$ (used for $\mathrm{VPSO\text{-}rf}$) with and without closure. For both memory models, we see that for instances that are realizable (blue dots), the version without closure is superior, sometimes even beyond an order of magnitude. This suggests that computing the closure-partial-order takes more time than is subsequently saved by utilizing it during the witness search. On the other hand, we observe that for the instances that are not realizable (green squares), the version with closure is orders-of-magnitude faster. This signifies that closure detects unrealizable instances much faster than complete exploration of a consistency verification algorithm.

Fig. 5.12 presents the comparison of $\mathrm{NaiveVerifyTSO}$ (used for $\mathrm{VTSO\text{-}rf}$) and $\mathrm{NaiveVerifyPSO}$ (used for $\mathrm{VPSO\text{-}rf}$) with and without closure. We observe trends similar to the paragraph above. Specifically, both $\mathrm{NaiveVerifyTSO}$ and $\mathrm{NaiveVerifyPSO}$ are mostly faster without closure on realizable instances, while they are significantly faster with closure on unrealizable instances.

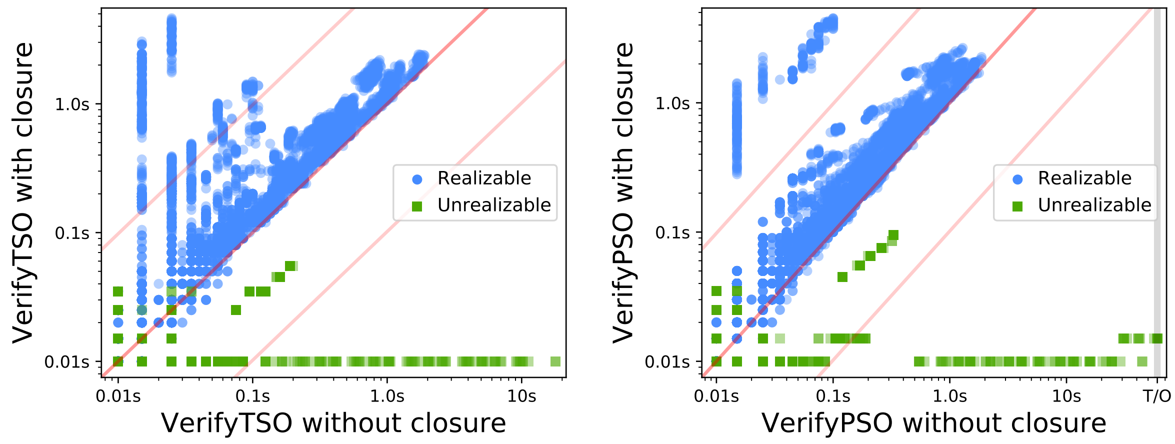


Figure 5.11: VerifyTSO (left) and VerifyPSO (right) with and without closure.

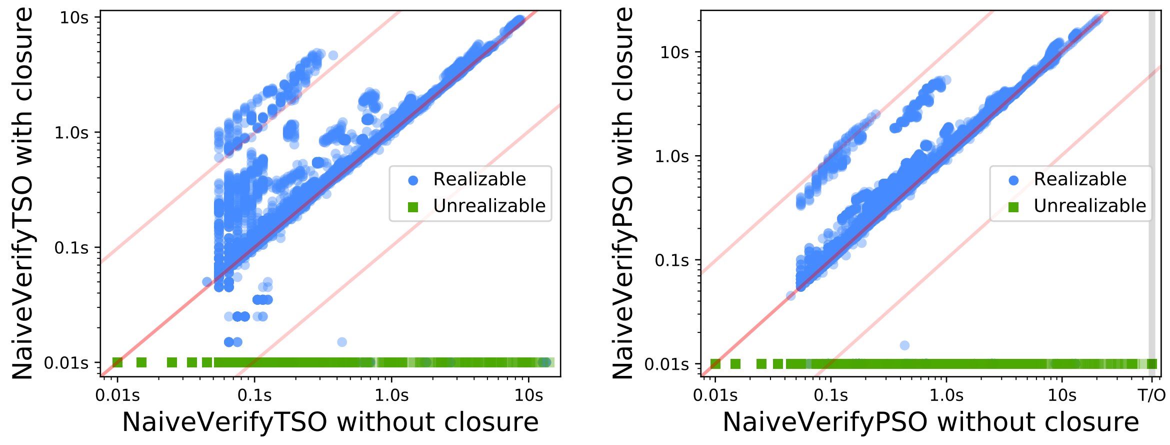


Figure 5.12: NaiveVerifyTSO (left) and NaiveVerifyPSO (right) with and without closure.

For each verification algorithm, its version without closure is faster on most instances that are realizable (i.e., a witness exists). This means that the overhead of computing the closure typically outweighs the consecutive benefit of the verification being guided by the partial order.

On the other hand, for each verification algorithm, its version with closure is significantly faster on the unrealizable instances (i.e., no witness exists). This is because a verification algorithm has to enumerate all its lower sets before declaring an instance unrealizable, and this is much slower than the polynomial closure computation.

Results – verification with atomic operations. Here we present additional experiments to evaluate TSO verification algorithms VerifyTSO and NaiveVerifyTSO on executions containing atomic operations read-modify-write (RMW) and compare-and-swap (CAS). To that end, we consider 1088 verification instances (779 realizable and 309 not realizable) that arise during stateless model checking of benchmarks containing RMW and CAS, namely:

- synthetic benchmarks casrot [AAJ$^+$19] and cinc [KRV19b],
- data structure benchmarks barrier, chase-lev, ms-queue and linuxrwlocks [ND13, KRV19b], and
- Linux kernel benchmarks mcs_spinlock and qspinlock [KRV19b].

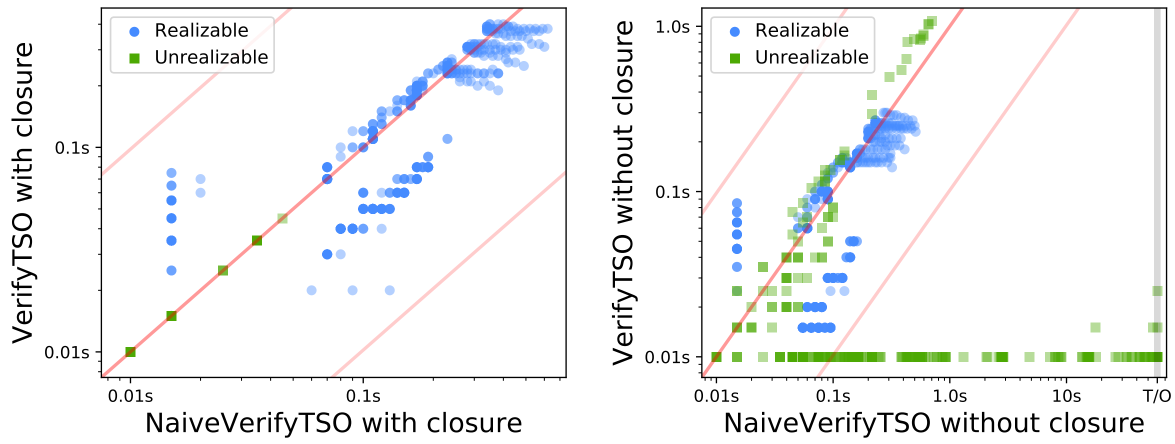


Figure 5.13: Verification in TSO with (left) and without (right) closure; RMW/CAS.

The results are presented in Fig. 5.13. The left plot depicts the results for VerifyTSO and NaiveVerifyTSO when closure is used as a preceding step. Here the results are all within an order-of-magnitude difference, and they are identical for unrealizable instances, since all of them were detected as unrealizable already by the closure. The right plot depicts the results for VerifyTSO and NaiveVerifyTSO without using the closure. Here the difference for realizable instances is also within an order of magnitude, but for some unrealizable instances the algorithm VerifyTSO is significantly faster. Generally, the observed improvement of our VerifyTSO as compared to NaiveVerifyTSO is somewhat smaller in Fig. 5.13, which could be due to the fact that executions with RMW and CAS instructions typically have fewer concurrent writes (indeed, in an execution where each write event is a part of a RMW/CAS instruction, each conflicting pair of writes is inherently ordered by the reads-from orderings together with PO).

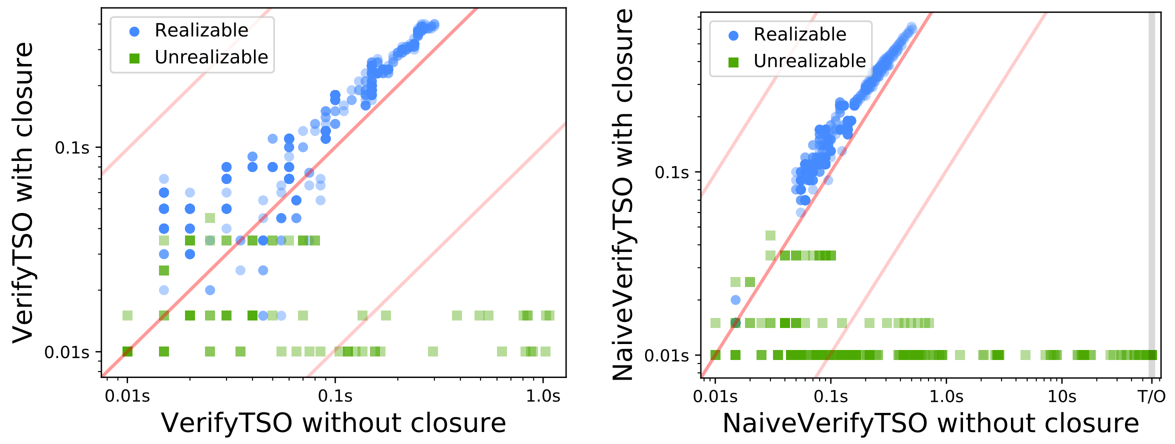


Figure 5.14: VerifyTSO (left) and NaiveVerifyTSO (right) closure effect; RMW/CAS.

Finally, Fig. 5.14 presents the effect of closure for VerifyTSO and NaiveVerifyTSO on verification instances that contain RMW and CAS instructions. Similarly to the verification without RMW and CAS instructions, both verification algorithms are somewhat slower when using closure on the realizable instances, and they are significantly faster when using closure on the unrealizable instances.

5.4.2 Experiments on SMC for TSO and PSO

In this section we focus on assessing the advantages of utilizing the reads-from equivalence for SMC in TSO and PSO.

Setup. We have used RF-SMC for stateless model checking of 109 benchmarks under each memory model $\mathcal{M} \in \{\text{SC, TSO, PSO}\}$, where SC is handled in our implementation as TSO with a fence after each thread event.

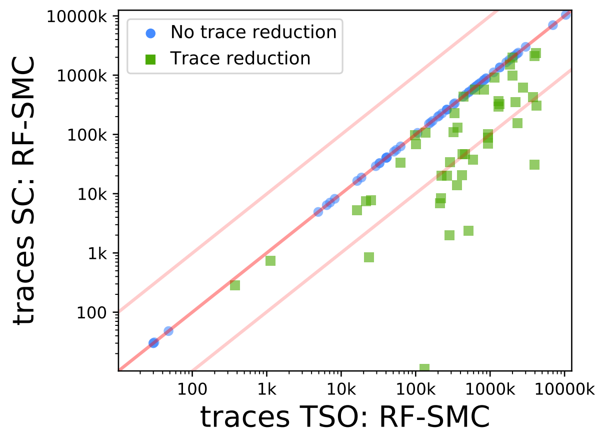
Handling assertion violations. We note that not all benchmarks behave as intended under all memory models, e.g., a benchmark might be correct under SC, but contain bugs under TSO. However, this is not an issue, as our goal is to characterize the size of the underlying partitionings, rather than detecting assertion violations. We have disabled all assertions, in order to not have the measured parameters be affected by how fast a violation is discovered, as the latter is arbitrary. As a sanity check, we have confirmed that for each memory model, all algorithms considered for that model discover the same bugs when assertions are enabled.

Identifying events. Our implementation extends the Nidhugg model checker and we rely on the interpreter built inside Nidhugg to identify events. An event e is defined by a triple (a_e, b_e, c_e), where a_e is the thread-id of e, b_e is the id of either the buffer of a_e or the main-thread of a_e that e is a part of, and c_e is the sequential number of the last LLVM instruction (of the corresponding thread/buffer) that is part of e. It can happen that there exist two traces σ_1 and σ_2, and two different events $e_1 \in \sigma_1$, $e_2 \in \sigma_2$, such that their identifiers are equal, i.e., $a_{e_1} = a_{e_2}$, $b_{e_2} = b_{e_2}$, and $c_{e_1} = c_{e_2}$. However, this means that the control-flow leading to each event is different. In this case, σ_1 and σ_2 differ in the reads-from of a common event that is ordered by the program order PO both before e_1 in σ_1 and before e_2 in σ_2, and hence e_1 and e_2 are treated as inequivalent.

Comparison. As a baseline for comparison, we have also executed Nidhugg/source [AAJS14], which is implemented in Nidhugg and explores the trace space using the partitioning based on the Shasha–Snir equivalence. In SC, we have further executed Nidhugg/rfsc [AAJ$^+$19], the Nidhugg implementation of the reads-from SMC algorithm for SC by [AAJ$^+$19]. Both Nidhugg/rfsc and Nidhugg/source are well-optimized, and recently started using advanced data-structures for SMC [LS20]. The works of [KRV19b, KV20] provide a general interface for reads-from SMC in relaxed memory models. However, they handle a given memory model assuming that an auxiliary consistency verification algorithm for that memory model is provided. No such consistency algorithm for TSO or PSO is presented by [KRV19b, KV20], and, to our knowledge, the tool implementations of [KRV19b, KV20] also lack a consistency algorithm for both TSO and PSO. Thus these tools are not included in the evaluation.[1]

Evaluation objective. Our objective for the SMC evaluation is three-fold. First, we want to quantify how each memory model $\mathcal{M} \in \{\text{SC, TSO, PSO}\}$ impacts the size of the RF partitioning. Second, we are interested to see whether, as compared to the baseline Shasha–Snir equivalence, the RF equivalence leads to coarser partitionings for TSO and PSO, as it does for SC [AAJ$^+$19]. Finally, we want to determine whether a coarser RF partitioning leads to faster exploration. Theorem 5.3 states that RF-SMC spends polynomial time per partitioning class, and we aim to see whether this is a small polynomial in practice.

[1]Another related work is MCR [HH16], however, the corresponding tool operates on Java programs and uses heavyweight SMT solvers that require fine tuning, and thus is beyond the experimental scope of this work.



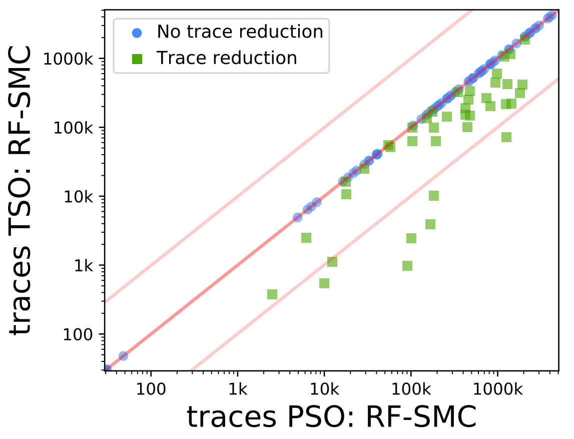
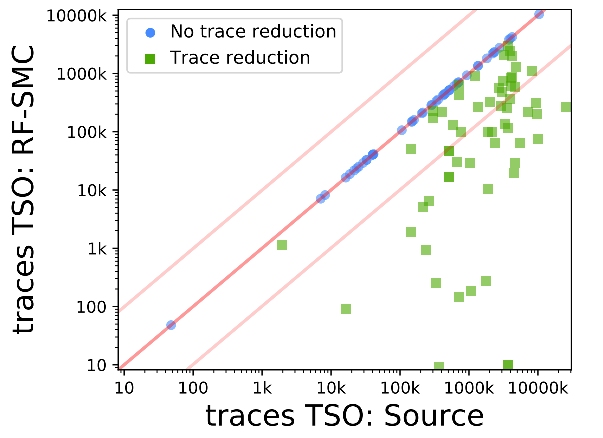


Figure 5.15: Traces as RF-SMC moves from SC to TSO (left) to PSO (right).

Results. We illustrate the obtained results with several scatter plots. Each plot compares two algorithms executing under specified memory models. Then for each benchmark, we consider the highest attempted unroll bound where both the compared algorithms finish before the one-hour timeout. Green squares indicate that a trace reduction was achieved on the underlying benchmark by the algorithm on the y-axis as compared to the algorithm on the x-axis. Benchmarks with no trace reduction are represented by the blue dots. All scatter plots are in log scale, the opaque and semi-transparent red lines represent identity and an order-of-magnitude difference, respectively.



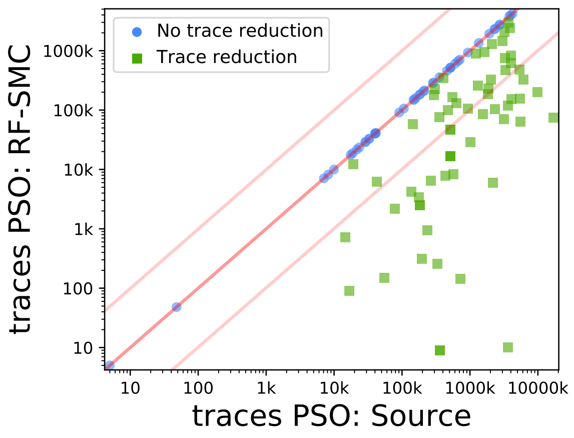
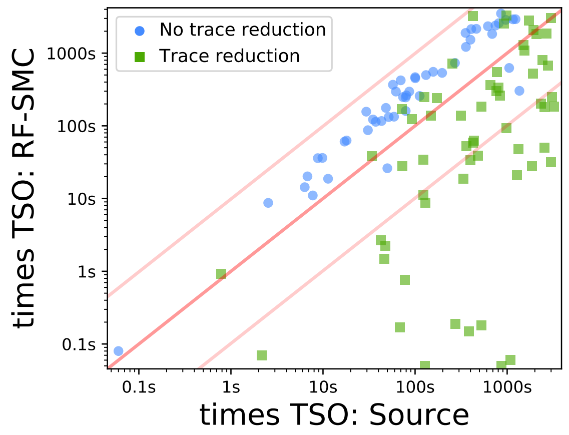


Figure 5.16: Traces for RF-SMC and Nidhugg/source on TSO (left) and PSO (right).

The plots in Fig. 5.15 illustrate how the size of the RF partitioning explored by RF-SMC changes as we move to more relaxed memory models (SC to TSO to PSO). The plots in Fig. 5.16 capture how the size of the RF partitioning explored by RF-SMC relates to the size of the Shasha–Snir partitioning explored by Nidhugg/source. Finally, the plots in Fig. 5.17 demonstrate the time comparison of RF-SMC and Nidhugg/source when there is some (green squares) or no (blue dots) RF-induced trace reduction.

Below we discuss the observations on the obtained results. Table 5.1 captures detailed results on several benchmarks that we refer to as examples in the discussion. In the table, **U** denotes the unroll bound, the timeout of one hour is indicated by "-", and bold-font entries indicate the smallest numbers for the respective memory model.



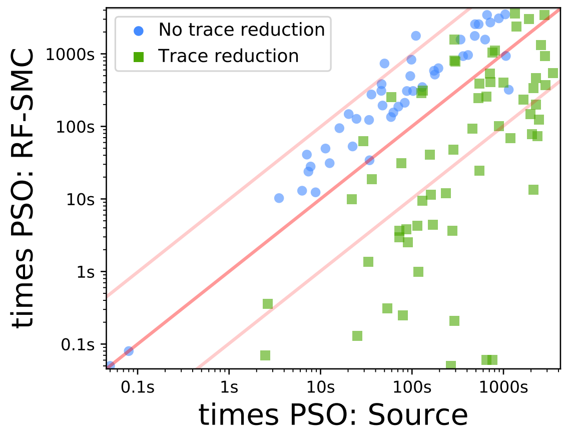


Figure 5.17: Times for RF-SMC and Nidhugg/source on TSO (left) and PSO (right).

Benchmark		U	Seq. Consistency		Total Store Order		Partial Store Order	
			RF-SMC	Source	RF-SMC	Source	RF-SMC	Source
27_Boop4 threads: 4	Traces	1	**2902**	21948	**3682**	36588	**8233**	572436
		4	**197260**	3873348	**313336**	9412428	**1807408**	-
	Times	1	**1.22s**	1.74s	**1.46s**	6.18s	**4.40s**	169s
		4	**124s**	550s	**182s**	2556s	**1593s**	-
eratosthenes threads: 2	Traces	17	**4667**	100664	**29217**	4719488	**253125**	-
		21	**19991**	1527736	**223929**	-	-	-
	Times	17	**6.70s**	46s	**32s**	2978s	**475s**	-
		21	**41s**	736s	**342s**	-	-	-
fillarray_false threads: 2	Traces	3	**14625**	47892	**14625**	59404	**14625**	63088
		4	**471821**	2278732	**471821**	3023380	**471821**	3329934
	Times	3	12s	**6.18s**	12s	**12s**	**18s**	39s
		4	553s	**331s**	**547s**	778s	**930s**	2844s

Table 5.1: SMC results on several benchmarks.

Discussion. We notice that the analysed programs can often exhibit additional behavior in relaxed memory settings. This causes an increase in the size of the partitionings explored by SMC algorithms (see 27_Boop4 in Table 5.1 as an example). Fig. 5.15 illustrates the overall phenomenon for RF-SMC, where the increase of the RF partitioning size (and hence the number of traces explored) is sometimes beyond an order of magnitude when moving from SC to TSO, or from TSO to PSO.

We observe that across all memory models, the reads-from equivalence can offer significant reduction in the trace partitioning as compared to Shasha–Snir equivalence. This leads to fewer traces that need to be explored, see the plots of Fig. 5.16. As we move towards more relaxed memory (SC to TSO to PSO), the reduction of RF partitioning often becomes more prominent (see 27_Boop4 in Table 5.1). Interestingly, in some cases the size of the Shasha–Snir partitioning explored by Nidhugg/source increases as we move to more relaxed settings, while the RF partitioning remains unchanged (cf. fillarray_false in Table 5.1). All these observations signify advantages of RF for analysis of the more complex program behavior that arises due to relaxed memory.

We now discuss how trace partitioning coarseness affects execution time, observing the plots of Fig. 5.17. We see that in cases where RF partitioning is coarser (green squares), our RF algorithm RF-SMC often becomes significantly faster than the Shasha–Snir-based Nidhugg/source, allowing us to analyse programs scaled several levels further (see `eratosthenes` in Table 5.1). In cases where the sizes of the RF partitioning and the Shasha–Snir partitioning coincide (blue dots), the well-engineered Nidhugg/source outperforms our RF-SMC implementation. The time differences in these cases are reasonably moderate, suggesting that the polynomial overhead incurred to operate on the RF partitioning is small in practice.

Further results. We now provide further results of the SMC experiments via several scatter plots to compactly illustrate the full experimental results. For each fixed plot comparing two algorithms, we plot the execution times and the numbers of explored maximal traces as follows. For each benchmark, we consider the highest attempted unroll bound where both compared algorithms finish before the one-hour timeout. Then we plot the time and the number of traces obtained by the two algorithms on the benchmark scaled with the above unroll bound.

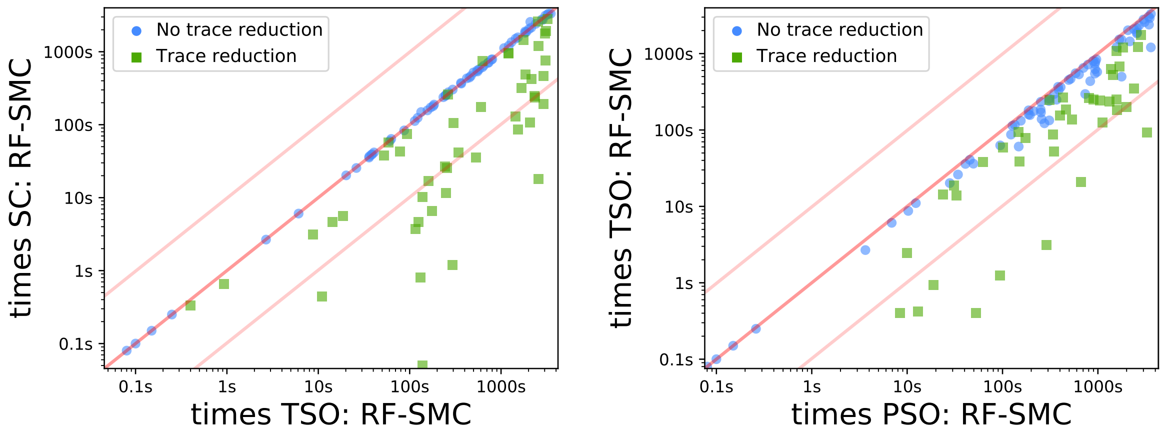


Figure 5.18: Times as RF-SMC moves from SC to TSO (left) to PSO (right).

Fig. 5.18 captures how analyzing a concurrent program by RF-SMC under more relaxed memory settings affects the execution time. Unsurprisingly, when a program exhibits additional behavior under a more relaxed model, more time is required to fully analyze it under the more relaxed model. Green squares represent such programs. On the other hand, for programs (represented by blue dots) where the number of traces stays the same in the more relaxed model, the time required for analysis is only minorly impacted.

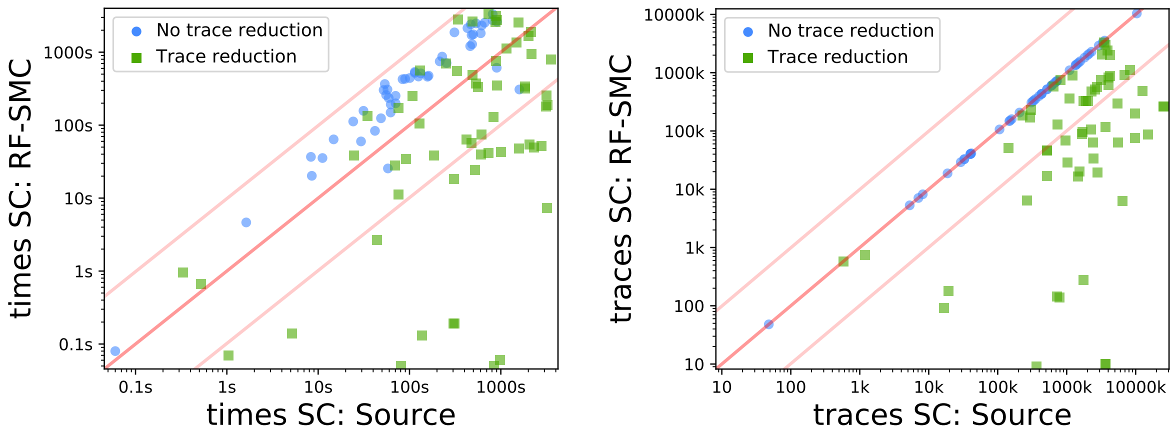


Figure 5.19: Times (left) and traces (right) for RF-SMC and Nidhugg/source on SC.

Fig. 5.19 compares in SC the algorithm Nidhugg/source with our algorithm RF-SMC that handles SC as TSO where a fence event is inserted after every buffer-write event. Similar

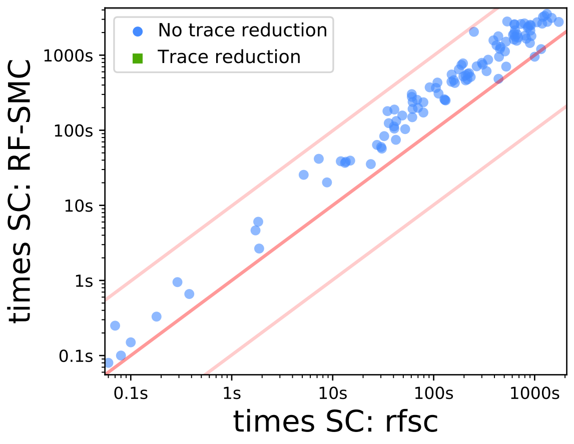


Figure 5.20: Times for RF-SMC and Nidhugg/rfsc on SC.

trends are observed as when Nidhugg/source and RF-SMC are compared in TSO and PSO. Specifically, there are cases where the RF partitioning offers reduction of the trace space size to be explored (green squares), and this often leads to significant speedup of the exploration. On the other hand, Nidhugg/source dominates in cases where no RF-based partitioning is induced (blue dots).

Further, Fig. 5.20 compares in SC the algorithm RF-SMC with Nidhugg/rfsc, the reads-from SMC algorithm for SC presented by [AAJ+19]. These two are essentially identical algorithms, thus unsurprisingly, the number of explored traces coincides in all cases. However, the well-engineered implementation of Nidhugg/rfsc is faster than our implementation of RF-SMC. This comparison provides a rough illustration of the effect of the optimizations and data-structures recently employed by Nidhugg/rfsc in the work of [LS20].

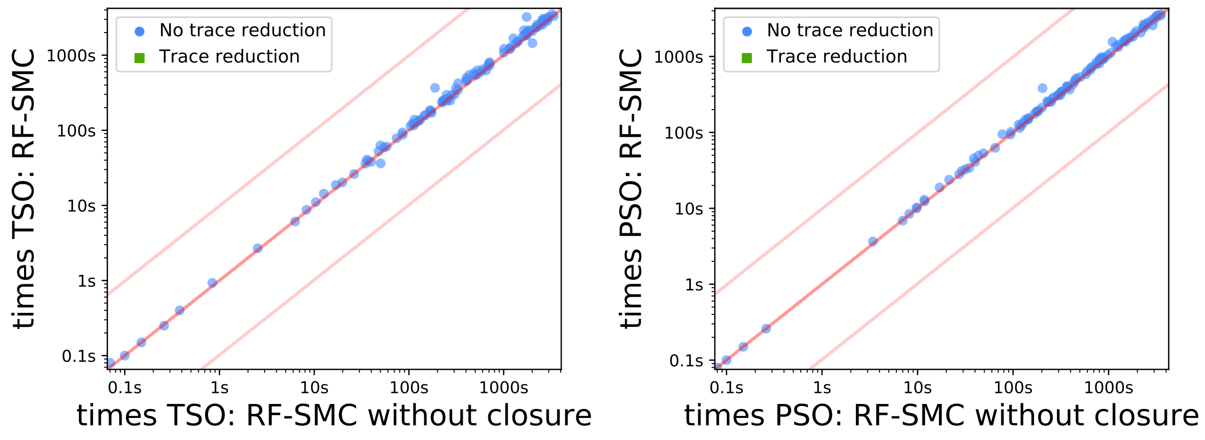


Figure 5.21: Times for RF-SMC with and without closure on TSO (left) and PSO (right).

In Section 5.4.1 we have seen that utilizing closure in consistency verification of realizable instances is mostly detrimental, whereas in consistency verification of unrealizable cases it is extremely helpful. This naturally begs a question whether it is overall beneficial to use closure in SMC. The plots in Fig. 5.21 present the time results for such an experiment (the number of explored traces coincide, as expected). The plots demonstrate that the time differences are negligible. The number of traces is, unsurprisingly, unaffected (it is also supposed to be unaffected, since closure is sound and VerifyTSO/VerifyPSO are sound and complete).

We have further considered an auxiliary-trace heuristic for guiding $\mathrm{VerifyTSO}$ resp. $\mathrm{VerifyPSO}$, similar to the heuristic reported by [AAJ$^+$19]. Similar to the paragraph above, this heuristic provided little-to-no time difference in the model checking task in our experiments.

CHAPTER 6

Symbolic Algorithms for Fairness Objectives

In this chapter we present faster symbolic algorithms for graphs and Markov decision processes (MDPs) with strong fairness (also known as *Streett*) objectives. While explicit algorithms for graphs and MDPs with Streett objectives have been widely studied, there has been no improvement of the basic symbolic algorithms. The worst-case numbers of symbolic steps required for the basic symbolic algorithms are as follows: quadratic for graphs and cubic for MDPs. In this work we present the first sub-quadratic symbolic algorithm for graphs with Streett objectives, and our algorithm is sub-quadratic even for MDPs. Based on our algorithmic insights we present an implementation of the new symbolic approach and we show that it improves the existing approach on several academic benchmark examples.

Previous Results. The most basic algorithm for the problem for graphs is based on repeated strongly connected component (SCC) computation, and informally can be described as follows: for a given SCC, (a) if for every request type that is present in the SCC the corresponding grant type is also present in the SCC, then the SCC is identified as "good", (b) else vertices of each request type that has no corresponding grant type in the SCC are removed, and the algorithm recursively proceeds on the remaining graph. Finally, reachability to good SCCs is computed. The current best-known symbolic algorithm for SCC computation requires $O(n)$ symbolic steps, for graphs with n vertices [GPP08], and moreover, the algorithm is optimal [CDHL18]. For MDPs, the SCC computation has to be replaced by maximal end-component (MEC) computation, and the current best-known symbolic algorithm for MEC computation requires $O(n^2)$ symbolic steps. While there have been several explicit algorithms for graphs with Streett objectives [HT96, CHL15], MEC computation [CH11, CH12, CH14], and MDPs with Streett objectives [CDHL16], as well as symbolic algorithms for MDPs with Büchi objectives [CHJS13], the current best-known bounds for symbolic algorithms with Streett objectives are obtained from the basic algorithms, which are $O(n \cdot \min(n, k))$ for graphs and $O(n^2 \cdot \min(n, k))$ for MDPs, where k is the number of types of request-grant pairs.

Our Contributions. In this work our main contributions are as follows:

1. We present a symbolic algorithm that requires $O(n \cdot \sqrt{m \log n})$ symbolic steps, both for graphs and MDPs, where m is the number of edges. In the case $k = O(n)$, the previous worst-case bounds are quadratic ($O(n^2)$) for graphs and cubic ($O(n^3)$) for MDPs. In

	Symbolic Operations		
Problem	Basic Algorithm	Improved Algorithm	Reference
Graphs with Streett	$O(n \cdot \min(n,k))$	$\mathbf{O(n\sqrt{m \log n})}$	Theorem 6.2
MDPs with Streett	$O(n^2 \cdot \min(n,k))$	$\mathbf{O(n\sqrt{m \log n})}$	Theorem 6.5
MEC decomposition	$O(n^2)$	$\mathbf{O(n\sqrt{m})}$	Theorem 6.4

Table 6.1: Symbolic algorithms for Streett objectives and MEC decomposition.

contrast, we present the first sub-quadratic symbolic algorithm both for graphs as well as MDPs. Moreover, in practice, since most graphs are sparse (with $m = O(n)$), the worst-case bounds of our symbolic algorithm in these cases are $O(n \cdot \sqrt{n \log n})$. Another interesting contribution of our work is that we also present an $O(n \cdot \sqrt{m})$ symbolic steps algorithm for MEC decomposition, which is relevant for our results as well as of independent interest, as MEC decomposition is used in many other algorithmic problems related to MDPs. Our results are summarized in Table 6.1.

2. While our main contribution is theoretical, based on the algorithmic insights we also present a new symbolic algorithm implementation for graphs and MDPs with Streett objectives. We show that the new algorithm improves (by around 30%) the basic algorithm on several academic benchmark examples from the VLTS benchmark suite [CWI].

Technical Contributions. The two key technical contributions of our work are as follows:

1. *Symbolic Lock Step Search:* We search for newly emerged SCCs by a local graph exploration around vertices that lost adjacent edges. In order to find small new SCCs first, all searches are conducted "in parallel", i.e., in lock-step, and the searches stop as soon as the first one finishes successfully. This approach has successfully been used to improve explicit algorithms [HT96, CJH03, CH12, CDHL16]. Our contribution is a non-trivial symbolic variant (Section 6.2) which lies at the core of the theoretical improvements.

2. *Symbolic Interleaved MEC Computation:* For MDPs the identification of vertices that have to be removed can be interleaved with the computation of MECs such that in each iteration the computation of SCCs instead of MECs is sufficient to make progress [CDHL16]. We present a symbolic variant of this interleaved computation. This interleaved MEC computation is the basis for applying the lock-step search to MDPs.

6.1 Definitions

6.1.1 Basic Problem Definitions

Markov decision processes (MDPs) and Graphs. An MDP $\mathscr{M} = ((V,E),(V_1,V_R),\delta)$ consists of a finite directed graph $G = (V,E)$ with a set of n vertices V and a set of m edges E, a partition of the vertices into *player 1 vertices* V_1 and *random vertices* V_R, and a probabilistic transition function δ. We call an edge (u,v) with $u \in V_1$ a *player 1 edge* and an edge (v,w) with $v \in V_R$ a *random edge*. For $v \in V$ we define *In*$(v) = \{w \in V \mid (w,v) \in E\}$

and *Out*$(v) = \{w \in V \mid (v, w) \in E\}$. The probabilistic transition function is a function from V_R to $\mathscr{D}(V)$, where $\mathscr{D}(V)$ is the set of probability distributions over V and a random edge $(v, w) \in E$ if and only if $\delta(v)[w] > 0$. Graphs are a special case of MDPs with $V_R = \emptyset$.

Plays and Strategies. A *play* or infinite path in $\mathscr{M}$ is an infinite sequence $\omega = \langle v_0, v_1, v_2, \dots \rangle$ such that $(v_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}$; we denote by Ω the set of all plays. A player 1 *strategy* $\lambda \colon V^* \cdot V_1 \to V$ is a function that assigns to every finite prefix $\omega \in V^* \cdot V_1$ of a play that ends in a player 1 vertex v a successor vertex $\lambda(\omega) \in V$ such that $(v, \lambda(\omega)) \in E$; we denote by Λ the set of all player 1 strategies. A strategy is *memoryless* if we have $\lambda(\omega) = \lambda(\omega')$ for any $\omega, \omega' \in V^* \cdot V_1$ that end in the same vertex $v \in V_1$.

Objectives. An *objective* ϕ is a subset of Ω said to be winning for player 1. We say that a play $\omega \in \Omega$ *satisfies the objective* ϕ if $\omega \in \phi$. For a vertex set $\mathscr{T} \subseteq V$ the *reachability objective* is the set of infinite paths that contain a vertex of $\mathscr{T}$, i.e., $\mathrm{Reach}(\mathscr{T}) = \{\langle v_0, v_1, v_2, \dots \rangle \in \Omega \mid \exists j \geq 0 : v_j \in \mathscr{T}\}$. Let $\mathrm{Inf}(\omega)$ for $\omega \in \Omega$ denote the set of vertices that occur infinitely often in ω. Given a set SP of k pairs (L_i, U_i) of vertex sets $L_i, U_i \subseteq V$ with $1 \leq i \leq k$, the *Streett objective* is the set of infinite paths for which it holds *for each* $1 \leq i \leq k$ that whenever a vertex of L_i occurs infinitely often, then a vertex of U_i occurs infinitely often, i.e., $\mathrm{Streett}(\mathrm{SP}) = \{\omega \in \Omega \mid L_i \cap \mathrm{Inf}(\omega) = \emptyset$ or $U_i \cap \mathrm{Inf}(\omega) \neq \emptyset$ for all $1 \leq i \leq k\}$.

Almost-Sure Winning Sets. For any measurable set of plays $A \subseteq \Omega$ we denote by $\Pr_v^\lambda(A)$ the probability that a play starting at $v \in V$ belongs to A when player 1 plays strategy λ. A strategy λ is *almost-sure (a.s.) winning* from a vertex $v \in V$ for an objective ϕ if $\Pr_v^\lambda(\phi) = 1$. The *almost-sure winning set* $\langle\!\langle 1 \rangle\!\rangle_{as}(\mathscr{M}, \phi)$ of player 1 is the set of vertices for which player 1 has an almost-sure winning strategy. In graphs the existence of an almost-sure winning strategy corresponds to the existence of a play in the objective, and the set of vertices for which player 1 has an (almost-sure) winning strategy is called the *winning set* $\langle\!\langle 1 \rangle\!\rangle(\mathscr{M}, \phi)$ of player 1.

Symbolic Encoding of MDPs. Symbolic algorithms operate on sets of vertices, which are usually described by Binary Decision Diagrams (BDDs) [Lee59, Ake78]. In particular Ordered Binary Decision Diagrams [Bry85] (OBDDs) provide a canonical symbolic representation of Boolean functions. For the computation of almost-sure winning sets of MDPs it is sufficient to encode MDPs with OBDDs and one additional bit that denotes whether a vertex is in V_1 or V_R.

Symbolic Steps. One symbolic step corresponds to one primitive operation as supported by standard symbolic packages like CUDD [Som15]. In this paper we only allow the same basic *set-based symbolic operations* as in [RBS00, GPP03, BGS06, CHJS13], namely set operations and the following one-step symbolic operations for a set of vertices Z: (a) the one-step predecessor operator $\mathsf{Pre}(Z) = \{v \in V \mid \mathit{Out}(v) \cap Z \neq \emptyset\}$; (b) the one-step successor operator $\mathsf{Post}(Z) = \{v \in V \mid \mathit{In}(v) \cap Z \neq \emptyset\}$; and (c) the one-step *controllable* predecessor operator $\mathsf{CPre}_R(Z) = \{v \in V_1 \mid \mathit{Out}(v) \subseteq Z\} \cup \{v \in V_R \mid \mathit{Out}(v) \cap Z \neq \emptyset\}$; i.e., the CPre_R operator computes all vertices such that the successor belongs to Z with positive probability. This operator can be defined using the Pre operator and basic set operations as follows: $\mathsf{CPre}_R(Z) = \mathsf{Pre}(Z) \setminus (V_1 \cap \mathsf{Pre}(V \setminus Z))$. We additionally allow cardinality computation and picking an arbitrary vertex from a set as in [CHJS13].

Symbolic Model. Informally, a symbolic algorithm does not operate on explicit representation of the transition function of a graph, but instead accesses it through Pre and Post operations. For explicit algorithms, a Pre/Post operation on a set of vertices (resp., a single vertex) requires $O(m)$ (resp., the order of indegree/outdegree of the vertex) time. In contrast, for symbolic algorithms Pre/Post operations are considered unit-cost. Thus an interesting algorithmic question is whether better algorithmic bounds can be obtained considering Pre/Post as unit-cost operations. Moreover, the basic set operations are computationally less expensive (as they encode the relationship between the state variables) compared to the Pre/Post symbolic operations (as they encode the transitions and thus the relationship between the present and the next-state variables). In all presented algorithms, the number of set operations is asymptotically at most the number of Pre/Post operations. Hence in the sequel we focus on the number of Pre/Post operations of algorithms.

Algorithmic Problem. Given an MDP $\mathscr{M}$ (resp. a graph G) and a set of Streett pairs SP, the problem we consider asks for a symbolic algorithm to compute the almost-sure winning set $\langle\langle 1 \rangle\rangle_{as}(\mathscr{M}, \mathrm{Streett(SP)})$ (resp. the winning set $\langle\langle 1 \rangle\rangle(G, \mathrm{Streett(SP)})$), which is also called the *qualitative analysis* of MDPs (resp. graphs).

6.1.2 Basic Concepts related to Algorithmic Solution

Reachability. For a graph $G = (V, E)$ and a set of vertices $S \subseteq V$ the set GraphReach(G, S) is the set of vertices of V that *can reach* a vertex of S within G, and it can be identified with at most $|\text{GraphReach}(G, S) \setminus S| + 1$ many Pre operations.

Strongly connected components (SCCs). For a set of vertices $S \subseteq V$ we denote by $G[S] = (S, E \cap (S \times S))$ the subgraph of the graph G induced by the vertices of S. An induced subgraph $G[S]$ is strongly connected if there exists a path in $G[S]$ between every pair of vertices of S. A *strongly connected component (SCC)* of G is a set of vertices $C \subseteq V$ such that the induced subgraph $G[C]$ is strongly connected and C is a maximal set in V with this property. Given an MDP $\mathscr{M} = ((V, E), (V_1, V_R), \delta)$, a set of vertices $C \subseteq V$ is an SCC of $\mathscr{M}$ if it is an SCC of the corresponding graph (V, E). We call an SCC *trivial* if it only contains a single vertex and no edges; and *non-trivial* otherwise. The SCCs of G partition its vertices and can be found in $O(n)$ symbolic steps [GPP08]. A bottom SCC C in a directed graph G is an SCC with no edges from vertices of C to vertices of $V \setminus C$, i.e., an SCC without *outgoing* edges. Analogously, a top SCC C is an SCC with no *incoming* edges from $V \setminus C$. For more intuition for bottom and top SCCs, consider the graph in which each SCC is contracted into a single vertex (ignoring edges within an SCC). In the resulting directed acyclic graph the sinks represent the bottom SCCs and the sources represent the top SCCs. Note that every graph has at least one bottom and at least one top SCC. If the graph is not strongly connected, then there exist at least one top and at least one bottom SCC that are disjoint and thus one of them contains at most half of the vertices of G.

Random Attractors. In an MDP $\mathscr{M}$ the *random attractor* $Attr_R(\mathscr{M}, W)$ of a set of vertices W is defined as $Attr_R(\mathscr{M}, W) = \bigcup_{j \geq 0} Z_j$ where $Z_0 = W$ and $Z_{j+1} = Z_j \cup \text{CPre}_R(Z_j)$ for all $j > 0$. The attractor can be computed with at most $|Attr_R(\mathscr{M}, W) \setminus W| + 1$ many CPre_R operations.

Maximal end-components (MECs). Let X be a vertex set without outgoing random edges, i.e., with $Out(v) \subseteq X$ for all $v \in X \cap V_R$. A sub-MDP of an MDP $\mathscr{M}$ induced by a vertex set $X \subseteq V$ without outgoing random edges is defined as $\mathscr{M}[X] = ((X, E \cap (X \times X), (V_1 \cap X, V_R \cap X), \delta)$. Note that the requirement that X has no outgoing random edges is necessary in order to use the same probabilistic transition function δ. An *end-component* of an MDP $\mathscr{M}$ is a set of vertices $X \subseteq V$ such that (a) X has no outgoing random edges, i.e., $\mathscr{M}[X]$ is a valid sub-MDP, (b) the induced sub-MDP $\mathscr{M}[X]$ is strongly connected, and

(c) $\mathcal{M}[X]$ contains at least one edge. Intuitively, an end-component is a set of vertices for which player 1 can ensure that the play stays within the set and almost-surely reaches all the vertices in the set (infinitely often). An end-component is a maximal end-component (MEC) if it is maximal under set inclusion. An end-component is *trivial* if it consists of a single vertex (with a self-loop), otherwise it is *non-trivial*. The *MEC decomposition* of an MDP consists of all MECs of the MDP.

Good End-Components. All algorithms for MDPs with Streett objectives are based on finding good end-components, defined below. Given the union of all good end-components, the almost-sure winning set for the Streett objective is obtained by computing the almost-sure winning set for the reachability objective with the union of all good end-components as the target set. The correctness of this approach is shown in [CDHL16, Loi16] (see also [BK08, Chap. 10.6.3]). For Streett objectives a good end-component is defined as follows. In the special case of graphs they are called good components.

Definition (Good end-component). *Given an MDP $\mathcal{M}$ and a set $\mathrm{SP} = \{(L_j, U_j) \mid 1 \le j \le k\}$ of target pairs, a* good end-component *is an end-component X of $\mathcal{M}$ such that for each $1 \le j \le k$ either $L_j \cap X = \emptyset$ or $U_j \cap X \neq \emptyset$. A maximal good end-component is a good end-component that is maximal with respect to set inclusion.*

Lemma 6.1 (Correctness of Computing Good End-Components [Loi16, Corollary 2.6.5, Proposition 2.6.9]). *For an MDP $\mathcal{M}$ and a set SP of target pairs, let $\mathcal{X}$ be the set of all maximal good end-components. Then $\langle\langle 1 \rangle\rangle_{as}(\mathcal{M}, \mathit{Reach}(\bigcup_{X \in \mathcal{X}} X))$ is equal to $\langle\langle 1 \rangle\rangle_{as}(\mathcal{M}, \mathit{Streett}(\mathrm{SP}))$.*

Iterative Vertex Removal. All the algorithms for Streett objectives maintain vertex sets that are candidates for good end-components. For such a vertex set S we (a) refine the maintained sets according to the SCC decomposition of $\mathcal{M}[S]$ and (b) for a set of vertices W for which we know that it cannot be contained in a good end-component, we remove its random attractor from S. The following lemma shows the correctness of these operations.

Lemma 6.2 (Correctness of Vertex Removal [Loi16, Lemma 2.6.10]). *Given an MDP $\mathcal{M} = ((V, E), (V_1, V_R), \delta)$, let X be an end-component with $X \subseteq S$ for some $S \subseteq V$. Then*

(*a*) *$X \subseteq C$ for one SCC C of $\mathcal{M}[S]$ and*

(*b*) *$X \subseteq S \setminus Attr_R(\mathcal{M}', W)$ for each $W \subseteq V \setminus X$ and each sub-MDP $\mathcal{M}'$ containing X.*

Let X be a good end-component. Then X is an end-component and for each index j, $X \cap U_j = \emptyset$ implies $X \cap L_j = \emptyset$. Hence we obtain the following corollary.

Corollary 6.1 ([Loi16, Corollary 4.2.2]). *Given an MDP $\mathcal{M}$, let X be a* good *end-component with $X \subseteq S$ for some $S \subseteq V$. For each i with $S \cap U_i = \emptyset$ it holds that $X \subseteq S \setminus Attr_R(\mathcal{M}[S], L_i \cap S)$.*

For an index j with $S \cap U_j = \emptyset$ we call the vertices of $S \cap L_j$ *bad vertices*. The set of all bad vertices $\mathrm{Bad}(S) = \bigcup_{1 \le i \le k}\{v \in L_i \cap S \mid U_i \cap S = \emptyset\}$ can be computed with $2k$ set operations.

6.2 Symbolic Divide-and-Conquer with Lock-Step Search

In this section we present a symbolic version of the lock-step search for strongly connected subgraphs [HT96]. This symbolic version is used in all subsequent results of this chapter, i.e., the sub-quadratic symbolic algorithms for graphs and MDPs with Streett objectives, and for MEC decomposition.

Divide-and-Conquer. The common property of the algorithmic problems we consider in this work is that the goal is to identify subgraphs of the input graph $G = (V, E)$ that are strongly connected and satisfy some additional properties. The difference between the problems lies in the required additional properties. We describe and analyze the Algorithm 6.1 that we use in all our improved algorithms to efficiently implement a divide-and-conquer approach based on the requirement of strong connectivity, that is, we divide a subgraph $G[S]$, induced by a set of vertices S, into two parts that are not strongly connected within $G[S]$, or we detect that $G[S]$ is indeed strongly connected.

Algorithm 6.1: Lock-Step-Search(G, S, H_S, T_S)

```
Input: Graph $G$, set of vertice $S$ and its subsets $H_S$ and $T_S$.
/* Pre and Post defined w.r.t. to $G$ */
1  foreach $v \in H_S \cup T_S$ do
2      $C_v \leftarrow \{v\}$
3  while true do
4      $H'_S \leftarrow H_S$, $T'_S \leftarrow T_S$
5      foreach $h \in H_S$ do                         // search for top SCC
6          $C'_h \leftarrow (C_h \cup \mathrm{Pre}(C_h)) \cap S$
7          if $|C'_h \cap H'_S| > 1$ then
8              $H'_S \leftarrow H'_S \setminus \{h\}$
9          else
10             if $C'_h = C_h$ then
11                 return ($C_h$, $H'_S$, $T_S$)
12             $C_h \leftarrow C'_h$
13     foreach $t \in T_S$ do                         // search for bottom SCC
14         $C'_t \leftarrow (C_t \cup \mathrm{Post}(C_t)) \cap S$
15         if $|C'_t \cap T'_S| > 1$ then
16             $T'_S \leftarrow T'_S \setminus \{t\}$
17         else
18             if $C'_t = C_t$ then
19                 return ($C_t$, $H'_S$, $T'_S$)
20             $C_t \leftarrow C'_t$
21     $H_S \leftarrow H'_S$, $T_S \leftarrow T'_S$
```

Start Vertices of Searches. The input to Algorithm 6.1 is a set of vertices $S \subseteq V$ and two subsets of S denoted by H_S and T_S. In the algorithms that call the procedure as a subroutine, vertices contained in H_S have lost incoming edges (i.e., they were a "head" of a lost edge) and vertices contained in T_S have lost outgoing edges (i.e., they were a "tail" of a lost edge) since the last time a superset of S was identified as being strongly connected. For each vertex h of H_S the procedure conducts a backward search (i.e., a sequence of Pre operations) within $G[S]$

to find the vertices of S that can reach h; and analogously a forward search (i.e., a sequence of Post operations) from each vertex t of T_S is conducted.

Intuition for the Choice of Start Vertices. If the subgraph $G[S]$ is not strongly connected, then it contains at least one top SCC and at least one bottom SCC that are disjoint. Further, if for a superset $S' \supset S$ the subgraph $G[S']$ was strongly connected, then each top SCC of $G[S]$ contains a vertex that had an additional incoming edge in $G[S']$ compared to $G[S]$, and analogously each bottom SCC of $G[S]$ contains a vertex that had an additional outgoing edge. Thus by keeping track of the vertices that lost incoming or outgoing edges, the following invariant will be maintained by all our improved algorithms.

Invariant 6.1 (Start Vertices Sufficient)**.** *We have $H_S, T_S \subseteq S$. Either (a) $H_S \cup T_S = \emptyset$ and $G[S]$ is strongly connected or (b) at least one vertex of each top SCC of $G[S]$ is contained in H_S and at least one vertex of each bottom SCC of $G[S]$ is contained in T_S.*

Lock-Step Search. The searches from the vertices of $H_S \cup T_S$ are performed in *lock-step*, that is, (a) one step is performed in each of the searches before the next step of any search is done and (b) all searches stop as soon as the first of the searches finishes. This is implemented in Algorithm 6.1 as follows. A step in the search from a vertex $t \in T_S$ (and analogously for $h \in H_S$) corresponds to the execution of the iteration of the for-each loop for $t \in T_S$. In an iteration of a for-each loop we might discover that we do not need to consider this search further (see the paragraph on ensuring strong connectivity below) and update the set T_S (via T'_S) for future iterations accordingly. Otherwise the set C_t is either strictly increasing in this step of the search or the search for t terminates and we return the set of vertices in $G[S]$ that are reachable from t. So the two for-each loops over the vertices of T_S and H_S that are executed in an iteration of the while-loop perform one step of each of the searches and the while-loop stops as soon as a search stops, i.e., a return statement is executed and hence this implements properties (a) and (b) of lock-step search. Note that the while-loop terminates, i.e., a return statement is executed eventually because for all $t \in T_S$ (and resp. for all $h \in H_S$) the sets C_t are monotonically increasing over the iterations of the while-loop, we have $C_t \subseteq S$, and if some set C_t does not increase in an iteration, then it is either removed from T_S and thus not considered further or a return statement is executed. Note that when a search from a vertex $t \in T_S$ stops, it has discovered a maximal set of vertices C that can be reached from t; and analogously for $h \in H_S$.

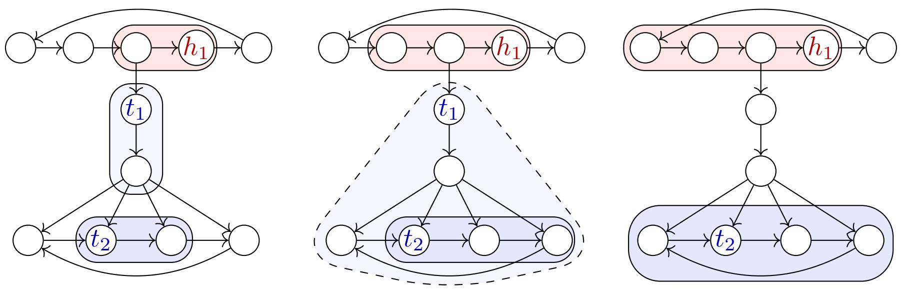


Figure 6.1: An example of symbolic lock-step search.

Figure 6.1 shows a small intuitive example of a call to Algorithm 6.1. The example shows the first three iterations of the main while-loop. Note that during the second iteration, the search started from t_1 is disregarded since it collides with t_2. In the subsequent fourth iteration, the search started from t_2 is returned by the algorithm.

Comparison to Explicit Algorithm. In the *explicit* version of the algorithm [HT96, CDHL16] the search from vertex $t \in T_S$ performs a depth-first search that terminates exactly when every *edge* reachable from t is explored. Since any search that starts outside of a bottom SCC but reaches the bottom SCC has to explore more edges than the search started inside of the bottom SCC, the first search from a vertex of T_S that terminates has exactly explored (one of) the smallest (in the number of edges) bottom SCC(s) of $G[S]$. Thus on explicit graphs the explicit lock-step search from the vertices of $H_S \cup T_S$ finds (one of) the smallest (in the number of edges) top or bottom SCC(s) of $G[S]$ in time proportional to the number of searches times the number of edges in the identified SCC. In *symbolically* represented graphs it can happen (1) that a search started outside of a bottom (resp. top) SCC terminates earlier than the search started within the bottom (resp. top) SCC and (2) that a search started in a larger (in the number of vertices) top or bottom SCC terminates before one in a smaller top or bottom SCC. We discuss next how we address these two challenges.

Ensuring Strong Connectivity. First, we would like the set returned by Algorithm 6.1 to indeed be a top or bottom SCC of $G[S]$. For this we use the following observation for bottom SCCs that can be applied to top SCCs analogously. If a search starting from a vertex of $t_1 \in T_S$ encounters another vertex $t_2 \in T_S$, $t_1 \neq t_2$, there are two possibilities: either (1) both vertices are in the same SCC or (2) t_1 can reach t_2 but not vice versa. In Case (1) the searches from both vertices can explore all vertices in the SCC and thus it is sufficient to only search from one of them. In Case (2) the SCC of t_1 has an outgoing edge and thus cannot be a bottom SCC. Hence in both cases we can remove the vertex t_1 from the set T_S while still maintaining Invariant 6.1. By Invariant 6.1 we further have that each search from a vertex of T_S that is not in a bottom SCC encounters another vertex of T_S in its search and therefore is removed from the set T_S during Algorithm 6.1 (if no top or bottom SCC is found earlier). This ensures that the returned set is either a top or a bottom SCC.[1]

Bound on Symbolic Steps. Second, observe that we can still bound the number of symbolic steps needed for the search that terminates first by the number of *vertices* in the smallest top or bottom SCC of $G[S]$, since this is an upper bound on the symbolic steps needed for the search started in this SCC. Thus provided Invariant 6.1, we can bound the number of symbolic steps in Algorithm 6.1 to identify a vertex set $C \subsetneq S$ such that C and $S \setminus C$ are not strongly connected in $G[S]$ by $O((|H_S| + |T_S|) \cdot \min(|C|, |S \setminus C|))$. In the algorithms that call Algorithm 6.1 we charge the number of symbolic steps in the procedure to the vertices in the smaller set of C and $S \setminus C$; this ensures that each vertex is charged at most $O(\log n)$ times over the whole algorithm. We obtain the following result.

Theorem 6.1 (Lock-Step Search). *Provided Invariant 6.1 holds, Algorithm 6.1(G, S, H_S, T_S) returns a top or bottom SCC C of $G[S]$. It uses $O((|H_S| + |T_S|) \cdot \min(|C|, |S \setminus C|))$ symbolic steps if $C \neq S$ and $O((|H_S| + |T_S|) \cdot |C|)$ otherwise.*

Proof. We argue separately about correctness and complexity.

Strong connectivity. We want to show that $C \leftarrow$ Algorithm 6.1(G, S, H_S, T_S) is a top or bottom SCC of $G[S]$ given Invariant 6.1 is satisfied. By the invariant at least one vertex of each top SCC of $G[S]$ is contained in H_S and at least one vertex of each bottom SCC of $G[S]$ is contained in T_S. Suppose C is the set obtained from a search conducted by Post operations that started from within a bottom SCC $\tilde{C}$ of $G[S]$. Since $\tilde{C}$ is a bottom SCC and we update

[1] To improve the practical performance, we return the updated sets H_S and T_S. By the above argument this preserves Invariant 6.1.

the search by executing Post operations (and moreover intersect with S at every update), we have $C \subseteq \tilde{C}$. Further, since $\tilde{C}$ is an SCC, the updates with Post eventually cover all vertices of $\tilde{C}$, which gives us $C = \tilde{C}$. A set C_t constructed with Post operations whose start vertex t is not contained in a bottom SCC of $G[S]$ can not yield the set C since eventually it contains a bottom SCC of $G[S]$, and by Invariant 6.1 this SCC contains a candidate in T_S; therefore $|C_t \cap T_S| > 1$ is satisfied at some point in the construction of C_t and then search is canceled by removing t from T_S; note that a search starting from a bottom SCC can be canceled only if another vertex of the bottom SCC remains in T_S. By the symmetric argument for searches conducted by Pre operations that started from a vertex of a top SCC we have that the returned set C is either a top or a bottom SCC of $G[S]$.

Bound on symbolic steps. Consider (one of) the smallest top or bottom SCCs $\tilde{C}$ of $G[S]$. Suppose w.l.o.g. that $\tilde{C}$ is a bottom SCC. By Invariant 6.1 there is a search, conducted by Post operations, that starts from a vertex $t \in T_S$ within $\tilde{C}$ and that is not canceled, and therefore this search terminates after at most $|\tilde{C}|$ many Post operations. Other searches may terminate earlier but this gives an upper bound of $O((|H_S| + |T_S|) \cdot |\tilde{C}|)$ on the number of symbolic steps until the lock-step search terminates. Finally, consider the returned set $C \leftarrow$ Algorithm 6.1(G, S, H_S, T_S). There are two possible cases: either (i) $S = C$, which implies $C = \tilde{C}$ so the number of symbolic steps can be bounded by $O((|H_S| + |T_S|) \cdot |C|)$, or (ii) $S \neq C$. In the second case, since $\tilde{C}$ is (some) smallest SCC, C is an SCC, and $S \setminus C$ contains at least one SCC, we have $|\tilde{C}| \leq |C|$ and $|\tilde{C}| \leq |S \setminus C|$, and hence we can bound the number of symbolic steps in this case by $O((|H_S| + |T_S|) \cdot \min(|C|, |S \setminus C|))$. □

6.3 Graphs with Streett Objectives

In this section we present the basic and improved symbolic algorithms for graphs with Streett objectives.

6.3.1 Basic Symbolic Algorithm for Graphs with Streett Objectives

Recall that for a given graph (with n vertices) and a Streett objective (with k target pairs) each non-trivial strongly connected subgraph without bad vertices is a good component. The basic symbolic algorithm for graphs with Streett objectives repeatedly removes bad vertices from each SCC and then recomputes the SCCs until all good components are found. The winning set then consists of the vertices that can reach a good component. We refer to this algorithm as $\mathrm{StreettGraphBasic}$. The pseudocode of the basic symbolic algorithm for graphs with Streett objectives is given in Algorithm 6.2.

The basic symbolic algorithm for Streett objectives on graphs $\mathrm{StreettGraphBasic}$ finds good components as follows. The algorithm maintains two sets of vertex sets: goodC contains identified good components and is initially empty; $\mathcal{X}$ contains candidates for good components and is initialized with the SCCs of the input graph G. The sets in $\mathcal{X}$ are strongly connected subgraphs of G throughout the algorithm. In each iteration of the while-loop one of the candidate sets S maintained in $\mathcal{X}$ is considered. If the set S does not contain bad vertices and contains at least one edge, then it is a good component and added to goodC. Otherwise, the set of bad vertices B in S is removed from S; the subgraph induced by $S' = S \setminus B$ might not be strongly connected but every good component contained in S' must still be strongly connected, therefore the maximal strongly connected subgraphs of $G[S']$ are added to $\mathcal{X}$ as new candidates for good components. By Lemma 6.2 and Corollary 6.1 this procedure maintains the property that every good component of G is completely contained in one of the vertex sets of goodC or $\mathcal{X}$. Further in each iteration either (a) vertices are removed or separated into different vertex sets or (b) a new good component is identified. Thus after at most $O(n)$ iterations the set $\mathcal{X}$ is empty and all good components of G are contained in goodC. Furthermore, whenever bad vertices are removed from a given candidate set, the number of target pairs this candidate set intersects is reduced by one. Thus each vertex is considered in at most $O(k)$ iterations of the main while-loop. Finally, the set of vertices that can reach a good component is determined (by $O(n)$ Pre operations) and output as the winning set. Since computing SCCs can be done in $O(n)$ symbolic steps, the total number of symbolic steps of the basic algorithm is bounded by $O(n \cdot \min(n, k))$.

Algorithm 6.2: StreettGraphBasic: Basic Algorithm for Graphs with Streett Obj.

```
Input  : graph $G = (V, E)$ and Streett pairs $\mathrm{SP} = \{(L_i, U_i) \mid 1 \le i \le k\}$
Output : $\langle\langle 1 \rangle\rangle (G, \mathrm{Streett}(\mathrm{SP}))$
1  $\mathcal{X} \leftarrow$ allSCCs($G$); goodC $\leftarrow \emptyset$
2  while $\mathcal{X} \neq \emptyset$ do
3      remove some $S \in \mathcal{X}$ from $\mathcal{X}$
4      $B \leftarrow \bigcup_{1 \le i \le k : U_i \cap S = \emptyset} (L_i \cap S)$
5      if $B \neq \emptyset$ then
6          $S \leftarrow S \setminus B$
7          $\mathcal{X} \leftarrow \mathcal{X} \cup$ allSCCs($G[S]$)
8      else
9          if Post($S$) $\cap S \neq \emptyset$ then        // $G[S]$ contains at least one edge
10             goodC $\leftarrow$ goodC $\cup \{S\}$
11 return GraphReach($G, \bigcup_{C \in \mathrm{goodC}} C$)
```

Proposition 6.1. *Algorithm 6.2 correctly computes the winning set in graphs with Streett objectives and requires $O(n \cdot \min(n, k))$ symbolic steps.*

6.3.2 Improved Symbolic Algorithm for Graphs with Streett Objectives

In our improved symbolic algorithm we replace the recomputation of all SCCs with the search for a new top or bottom SCC with Algorithm 6.1 from vertices that have lost adjacent edges whenever there are not too many such vertices. We present the improved symbolic algorithm for graphs with Streett objectives in more detail as it also conveys important intuition for the MDP case. The pseudocode is given in Algorithm 6.3.

Iterative Refinement of Candidate Sets. The improved algorithm maintains a set goodC of already identified good components that is initially empty and a set $\mathcal{X}$ of candidates for good components that is initialized with the SCCs of the input graph G. The difference to the basic algorithm lies in the properties of the vertex sets maintained in $\mathcal{X}$ and the way we identify sets that can be separated from each other without destroying a good component. In each iteration one vertex set S is removed from $\mathcal{X}$ and, after the removal of bad vertices from the set, either identified as a good component or split into several candidate sets. By Lemma 6.2 and Corollary 6.1 the following invariant is maintained throughout the algorithm for the sets in goodC and $\mathcal{X}$.

Invariant 6.2 (Maintained Sets)**.** *The sets in $\mathcal{X} \cup \text{goodC}$ are pairwise disjoint and for every good component C of G there exists a set $Y \supseteq C$ such that either $Y \in \mathcal{X}$ or $Y \in \text{goodC}$.*

Lost Adjacent Edges. In contrast to the basic algorithm, the subgraph induced by a set S contained in $\mathcal{X}$ is not necessarily strongly connected. Instead, we remember vertices of S that have lost adjacent edges since the last time a superset of S was determined to induce a strongly connected subgraph; vertices that lost incoming edges are contained in H_S and vertices that lost outgoing edges are contained in T_S. In this way we maintain Invariant 6.1 throughout the algorithm, which enables us to use Algorithm 6.1 with the running time guarantee provided by Theorem 6.1.

Identifying SCCs. Let S be the vertex set removed from $\mathcal{X}$ in a fixed iteration of Algorithm 6.3 after the removal of bad vertices in the inner while-loop. First note that if S is strongly connected and contains at least one edge, then it is a good component. If the set S was already identified as strongly connected in a previous iteration, i.e., H_S and T_S are empty, then S is identified as a good component in Line 14. If many vertices of S have lost adjacent edges since the last time a super-set of S was identified as a strongly connected subgraph, then the SCCs of $G[S]$ are determined as in the basic algorithm. To achieve the optimal asymptotic upper bound, we say that many vertices of S have lost adjacent edges when we have $|H_S| + |T_S| \geq \sqrt{m/\log n}$, while lower thresholds are used in our experimental results. Otherwise, if not too many vertices of S lost adjacent edges, then we start a symbolic *lock-step search* for top SCCs from the vertices of H_S and for bottom SCCs from the vertices of T_S using Procedure 6.1. The set returned by the procedure is either a top or a bottom SCC C of $G[S]$ (Theorem 6.1). Therefore we can from now on consider C and $S \setminus C$ separately, maintaining Invariants 6.1 and 6.2.

Algorithm 6.3 (StreettGraphImpr). A succinct description of the pseudocode is as follows: Lines 1–3 initialize the set of candidates for good components with the SCCs of the input graph. In each iteration of the main while-loop one candidate is considered and the following operations are performed: (a) Lines 6–11 iteratively remove all bad vertices; if afterwards the candidate is still strongly connected (and contains at least one edge), it is identified as a good component in the next step; otherwise it is partitioned into new candidates in one of the following ways: (b) if many vertices lost adjacent edges, Lines 15–23 partition the candidate into its SCCs (this corresponds to an iteration of the basic algorithm); (c) otherwise, Lines 24–33 use symbolic lock-step search to partition the candidate into one of its SCCs and the remaining vertices. The while-loop terminates when no candidates are left. Finally, vertices that can reach some good component are returned. We have the following result.

Theorem 6.2 (Improved Algorithm for Graphs)**.** *Algorithm 6.3 correctly computes the winning set in graphs with Streett objectives and requires $O(n \cdot \sqrt{m \log n})$ symbolic steps.*

To prove Theorem 6.2, we first establish the following lemma.

Lemma 6.3 (Invariants of Improved Algorithm for Graphs)**.** *Invariant 6.1 and Invariant 6.2 are preserved throughout Algorithm 6.3, i.e., they hold before the first iteration, after each iteration, and after termination of the main while-loop. Further, Invariant 6.1 is preserved during each iteration of the main while-loop.*

Proof.

Algorithm 6.3: StreettGraphImpr: Improved Alg. for Graphs with Streett Obj.

```
Input  : graph $G = (V, E)$ and Streett pairs $\mathrm{SP} = \{(L_i, U_i) \mid 1 \le i \le k\}$
Output : $\langle\langle 1 \rangle\rangle (G, \mathrm{Streett}(\mathrm{SP}))$
1  $\mathcal{X} \leftarrow$ allSCCs$(G)$; goodC $\leftarrow \emptyset$
2  foreach $C \in \mathcal{X}$ do
3      $H_C \leftarrow \emptyset$; $T_C \leftarrow \emptyset$
4  while $\mathcal{X} \neq \emptyset$ do
5      remove some $S \in \mathcal{X}$ from $\mathcal{X}$
6      $B \leftarrow \bigcup_{1 \le i \le k : U_i \cap S = \emptyset} (L_i \cap S)$
7      while $B \neq \emptyset$ do
8          $S \leftarrow S \setminus B$
9          $H_S \leftarrow (H_S \cup \mathsf{Post}(B)) \cap S$
10         $T_S \leftarrow (T_S \cup \mathsf{Pre}(B)) \cap S$
11         $B \leftarrow \bigcup_{1 \le i \le k : U_i \cap S = \emptyset} (L_i \cap S)$
12     if Post$(S) \cap S \neq \emptyset$ then                  // $G[S]$ contains at least one edge
13         if $|H_S| + |T_S| = 0$ then
14             goodC $\leftarrow$ goodC $\cup \{S\}$
15         else if $|H_S| + |T_S| \ge \sqrt{m / \log n}$ then
16             delete $H_S$ and $T_S$
17             $\mathcal{C} \leftarrow$ allSCCs$(G[S])$
18             if $|\mathcal{C}| = 1$ then
19                 goodC $\leftarrow$ goodC $\cup \{S\}$
20             else
21                 foreach $C \in \mathcal{C}$ do
22                     $H_C \leftarrow \emptyset$; $T_C \leftarrow \emptyset$
23                 $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{C}$
24         else
25             $(C, H_S, T_S) \leftarrow$ Lock-Step-Search$(G, S, H_S, T_S)$
26             if $C = S$ then
27                 goodC $\leftarrow$ goodC $\cup \{S\}$
28             else                                        // separate $C$ and $S \setminus C$
29                 $S \leftarrow S \setminus C$
30                 $H_C \leftarrow \emptyset$; $T_C \leftarrow \emptyset$
31                 $H_S \leftarrow (H_S \cup \mathsf{Post}(C)) \cap S$
32                 $T_S \leftarrow (T_S \cup \mathsf{Pre}(C)) \cap S$
33                 $\mathcal{X} \leftarrow \mathcal{X} \cup \{S\} \cup \{C\}$
34 return GraphReach$(G, \bigcup_{C \in \mathrm{goodC}} C)$
```

Invariant 6.1. Whenever a new candidate S is added as a result from allSCCs, it is strongly connected, and we set $H_S = T_S = \emptyset$; this in particular implies that the invariant is satisfied after the initialization of the algorithm.

By induction and Theorem 6.1, the invariant is satisfied whenever Procedure 6.1 returns a candidate C and we set $H_C = T_C = \emptyset$.

Now consider an update of a candidate S where some subset B is deleted from it and assume the invariant holds before the update. In these cases we update H_S and T_S by setting $H_S \leftarrow (H_S \cup \mathsf{Post}(B)) \cap S$ and $T_S \leftarrow (T_S \cup \mathsf{Pre}(B)) \cap S$. This adds the vertices that remain in S and have an edge from a vertex of B to H_S and those with an edge to B to T_S. Suppose a new top (resp. bottom) SCC $\tilde{S} \subseteq S$ emerges in S by the removal of B from S. Then some vertex of $\tilde{S}$ had an outgoing edge to B (resp. an incoming edge from B) and thus is contained in the updated set T_S (resp. H_S), maintaining the invariant. This happens whenever we remove Bad(S) from S, and whenever we subtract a result from Procedure 6.1 C from S.

Invariant 6.2 – Disjointness. The sets in $\mathcal{X} \cup$ goodC are pairwise disjoint at the initialization since goodC is initialized as $\emptyset$. Furthermore, whenever a set S is added to goodC in an iteration of the main while-loop, a superset $\tilde{S} \supseteq S$ is removed from $\mathcal{X}$ in the same iteration of the while-loop. Therefore by induction the disjointness of the sets in $\mathcal{X} \cup$ goodC is preserved.

Invariant 6.2 – Containment of good components. At initialization, $\mathcal{X}$ contains all SCCs of the input graph G. Each good component C of G is strongly connected, so there exists an SCC $Y \supseteq C$ such that $Y \in \mathcal{X}$ for each good component C.

Consider a set $S \in \mathcal{X}$ that is removed from $\mathcal{X}$ at the beginning of an iteration of the main while-loop. Consider further a good component C of G such that $C \subseteq S$. We require that a set $Y \supseteq C$ is added to either $\mathcal{X}$ or goodC in this iteration of the main while-loop.

First, whenever we remove Bad(S) from S, by Corollary 6.1 we maintain the fact that $C \subseteq S$. Second, $G[S]$ contains an edge since $C \subseteq S$. Finally, one of the three cases happens:

Case (1): If $|H_S| + |T_S| = 0$, then the set $S \supseteq C$ is added to goodC.

Case (2): If $|H_S| + |T_S| \geq \sqrt{m/\log n}$, then the algorithm computes the SCCs of $G[S]$. Since $C \subseteq S$ is strongly connected, it is completely contained in some SCC Y of $G[S]$, and Y is added either to $\mathcal{X}$ or to goodC.

Case (3): If $0 < |H_S| + |T_S| < \sqrt{m/\log n}$, then the algorithm either adds $S \supseteq C$ to goodC, or partitions S into $\tilde{S}$ and $S \setminus \tilde{S}$. Suppose the latter case happens, then by Theorem 6.1 we have that $\tilde{S}$ is an SCC of $G[S]$. Further, since $C \subseteq S$ is strongly connected, it is completely contained in some SCC of $G[S]$. Therefore either $C \subseteq \tilde{S}$ or $C \subseteq (S \setminus \tilde{S})$, and both $\tilde{S}$ and $S \setminus \tilde{S}$ are added to $\mathcal{X}$.

By the above case analysis we have that a set $Y \supseteq C$ is added to either $\mathcal{X}$ or goodC in the iteration of the main while-loop, and thus the invariant is preserved throughout the algorithm.

$\square$

We are now ready to prove the main result of this section, Theorem 6.2.

Proof of Theorem 6.2.

Correctness. Whenever a candidate set S is added to goodC, it contains an edge by the check at Line 12, and Bad$(S) = \emptyset$ by the check at Line 7. Furthermore, (a) at Line 14, S is strongly connected by Invariant 6.1, (b) at Line 19, S is strongly connected by the result of allSCCs, and (c) at Line 27, S is strongly connected by Theorem 6.1. Therefore we have that whenever a candidate set is added to goodC, it is indeed a good component (soundness).

Finally, by soundness, Invariant 6.2, the termination of the algorithm (shown below), and the fact that $\mathcal{X} = \emptyset$ at the termination of the algorithm, we have that goodC contains all good components of G (completeness).

Symbolic steps analysis. By [GPP08], the initialization with the SCCs of the input graph takes $O(n)$ symbolic steps. Furthermore, the reachability computation in the last step takes $O(n)$ Pre operations.

In each iteration of the outer while-loop, a set S is removed from $\mathcal{X}$ and either (a) a set $S' \subseteq S$ is added to goodC and no set is added to $\mathcal{X}$ or (b) at least two sets that are (proper subsets of) a partition of S are added to $\mathcal{X}$. Both can happen at most $O(n)$ times, thus there can be at most $O(n)$ iterations of the outer while-loop. The Pre and Post operations at Lines 12, 31, and 32 can be charged to the iterations of the outer while-loop.

An iteration of the inner while-loop (Lines 7-11) is executed only if some vertices B are removed from S; the vertices of B are then not considered further. Thus there can, in total, be at most $O(n)$ Pre and Post operations over all iterations of the inner while-loop.

Note that every vertex in each of H_S and T_S can be attributed to at least one unique implicit edge deletion since we only add vertices to H_S resp. T_S that are successors resp. predecessors of vertices that were separated from S (or deleted from the maintained graph). Whenever the case $|H_S| + |T_S| \geq \sqrt{m/\log n}$ occurs, for all subsets $C \subseteq S$ that are then added to $\mathcal{X}$, we initialize $H_C = T_C = \emptyset$. Therefore the case $|H_S| + |T_S| \geq \sqrt{m/\log n}$ can happen at most $O(\sqrt{m \log n})$ times throughout the algorithm since there are at most m edges that can be deleted, and hence in total takes $O(n \cdot \sqrt{m \log n})$ symbolic steps.

It remains to bound the number of symbolic steps in Procedure 6.1. Let C be the set returned by the procedure; we charge the symbolic steps in this call of the procedure to the vertices of the smaller set of C and $S \setminus C$. By Theorem 6.1 we have either (a) $C = S$, the number of symbolic steps in this call is bounded by $O(\sqrt{m/\log n} \cdot |C|)$, and the set S is added to goodC or (b) $\min(|C|, |S \setminus C|) \leq |S|/2$ and the number of symbolic steps in this call is bounded by $O(\sqrt{m/\log n} \cdot \min(|C|, |S \setminus C|))$. Case (a) can happen at most once for the vertices of C, and for case (b) note that the size of a set containing a specific vertex can be halved at most $O(\log n)$ times; thus we charge each vertex at most $O(\log n)$ times. Hence we can bound the total number of symbolic steps in all calls to the procedure by $O(n \cdot \sqrt{m \log n})$.

□

6.4 Symbolic MEC Decomposition

In this section we present a short description of the basic symbolic algorithm for MEC decomposition and then present the improved algorithm.

6.4.1 Basic Symbolic Algorithm for MEC decomposition

Recall that an end-component is a set of vertices that (a) has no random edges to vertices not in the set and its induced sub-MDP is (b) strongly connected and (c) contains at least one edge. The basic symbolic algorithm for MEC decomposition maintains a set of identified MECs and a set of candidates for MECs, initialized with the SCCs of the MDP. Whenever a candidate is considered, either (a) it is identified as a MEC or (b) it contains vertices with outgoing random edges, which are then removed together with their random attractor from the candidate, and the SCCs of the remaining sub-MDP are added to the set of candidates. We refer to the algorithm as MECBasic, and the pseudocode is in Algorithm 6.4.

Algorithm 6.4 computes all maximal end-components of a given MDP and is formulated as to highlight the similarities to the algorithms for graphs and MDPs with Streett objectives. The algorithm maintains two sets, the set goodC of identified maximal end-components that is initially empty and the set $\mathcal{X}$ of candidates for maximal end-components that is initialized with the SCCs of the MDP. In each iteration of the while-loop one set S is removed from $\mathcal{X}$ and either (1a) identified as a maximal end-component and added to goodC or (1b) removed because the induced sub-MDP does not contain an edge or (2) it contains vertices with outgoing random edges. In the latter case these vertices *rout* are identified and their random attractor is removed from S. After this step the sub-MDP induced by the remaining vertices of S might not be strongly connected any more. Therefore the SCCs of this sub-MDP are determined and added to $\mathcal{X}$ as new candidates for maximal end-components. Note that this maintains the invariants that (i) each set in $\mathcal{X}$ induces a strongly connected subgraph and (ii) each end-component is a subset of one set in either goodC or $\mathcal{X}$. By (i) a set in $\mathcal{X}$ is an end-component if it does not have outgoing random edges and the induced sub-MDP contains an edge, i.e., in particular this holds for the sets added to goodC (soundness). By (ii) and $\mathcal{X} = \emptyset$ at termination of the while-loop the algorithm identifies all maximal end-components of the MDP (completeness). Since both (1) and (2) can happen at most $O(n)$ times, there are $O(n)$ iterations of the while-loop. In each iteration the most expensive operations are the computation of a random attractor and of SCCs, which can both be done in $O(n)$ symbolic steps. Thus Algorithm 6.4 correctly computes all maximal end-components of an MDP and takes $O(n^2)$ symbolic steps.

Algorithm 6.4: MECBasic: Basic Algorithm for Maximal End-Components

```
Input  : an MDP $\mathscr{M} = (G = (V, E), (V_1, V_R))$
Output : the set of maximal end-components of $\mathscr{M}$
1  goodC ← $\emptyset$
2  $\mathcal{X}$ ← allSCCs($G$)
3  while $\mathcal{X} \neq \emptyset$ do
4      remove some $S \in \mathcal{X}$ from $\mathcal{X}$
5      rout ← $S \cap V_R \cap \text{Pre}(V \setminus S)$
6      if rout $\neq \emptyset$ then
7          $S \leftarrow S \setminus Attr_R(G, rout)$
8          $\mathcal{X} \leftarrow \mathcal{X} \cup$ allSCCs($G[S]$)
9      else
10         if Post($S$) $\cap S \neq \emptyset$ then          // $G[S]$ contains at least one edge
11             goodC ← goodC $\cup \{S\}$
12 return goodC
```

Proposition 6.2. *Algorithm 6.4 correctly computes the MEC decomposition of MDPs and requires $O(n^2)$ symbolic steps.*

6.4.2 Improved Symbolic Algorithm for MEC decomposition

The improved symbolic algorithm for MEC decomposition uses the ideas of symbolic lock-step search presented in Section 6.2. Informally, when considering a candidate that lost a few edges from the remaining graph, we use the symbolic lock-step search to identify some bottom SCC. We refer to the algorithm as MECImpr and present the pseudocode in Algorithm 6.5.

Algorithm 6.5: MECImpr: Improved Algorithm for Maximal End-Components

```
Input  : an MDP $\mathcal{M} = (G = (V, E), (V_1, V_R))$
Output : the set of maximal end-components of $\mathcal{M}$
1  $\mathcal{X} \leftarrow$ allSCCs($G$); goodC $\leftarrow \emptyset$
2  foreach $C \in \mathcal{X}$ do
3      $T_C \leftarrow \emptyset$
4  while $\mathcal{X} \neq \emptyset$ do
5      remove some $S \in \mathcal{X}$ from $\mathcal{X}$
6      $rout \leftarrow S \cap V_R \cap \text{Pre}(V \setminus S)$
7      $A \leftarrow Attr_R(G, rout)$
8      $S \leftarrow S \setminus A$
9      $T_S \leftarrow (T_S \cup \text{Pre}(A)) \cap S$
10     if Post($S$) $\cap S \neq \emptyset$ then                    // $G[S]$ contains at least one edge
11         if $|T_S| = 0$ then
12             goodC $\leftarrow$ goodC $\cup \{S\}$
13         else if $|T_S| \geq \sqrt{m}$ then
14             delete $T_S$
15             $\mathcal{C} \leftarrow$ allSCCs($G[S]$)
16             if $|\mathcal{C}| = 1$ then
17                 goodC $\leftarrow$ goodC $\cup \{S\}$
18             else
19                 foreach $C \in \mathcal{C}$ do
20                     $T_C \leftarrow \emptyset$
21                 $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{C}$
22         else
23             $C \leftarrow$ Lock-Step-Search($G$, $S$, $\emptyset$, $T_S$)
24             if Post($C$) $\cap C \neq \emptyset$ then                // $G[C]$ contains at least one edge
25                 goodC $\leftarrow$ goodC $\cup \{C\}$
26             $S \leftarrow S \setminus C$
27             $T_S \leftarrow (T_S \cup \text{Pre}(C)) \cap S$
28             $\mathcal{X} \leftarrow \mathcal{X} \cup \{S\}$
29 return goodC
```

Informal description. We show how to determine all maximal end-components (MECs) of an MDP in $O(n\sqrt{m})$ symbolic operations. The difference to the basic algorithm lies in the way strongly connected parts of the MDP are identified after the deletion of vertices that cannot be contained in a MEC. For this the symbolic lock-step search from Section 6.2 is used whenever not too many edges have been deleted since the last re-computation of SCCs.

Let $\mathcal{M}$ be the given MDP and $G = (V, E)$ its underlying graph. The algorithm maintains two sets of vertex sets: the set goodC of already identified MECs that is initialized with the empty set and the set $\mathcal{X}$ that is initialized with the SCCs of G and contains vertex sets that are candidates for MECs. The algorithm preserves the following invariant for the goodC and $\mathcal{X}$ over the iterations of the while-loop and returns the set goodC when the set $\mathcal{X}$ is empty after an iteration of the while-loop.

Invariant 6.3 (Maintained Sets). *The sets in $\mathcal{X} \cup$ goodC are pairwise disjoint and for every maximal end-component X of G there exists a set $Y \supseteq X$ such that either $Y \in \mathcal{X}$ or $Y \in$ goodC.*

For each vertex set S in $\mathcal{X}$ additionally a subset T_S of S is maintained that contains vertices that have lost outgoing edges since the last time a superset of S was identified as strongly connected. We use the following restrictions of Invariant 6.1 and Theorem 6.1 (presented in Section 6.2) to bottom SCCs only.

Invariant 6.4 (Start Vertices BSCC)**.** *Either* (*a*) T_S *is empty and* $G[S]$ *is strongly connected or* (*b*) *at least one vertex of each bottom SCC of* $G[S]$ *is contained in* T_S*.*

Theorem 6.3 (Lock-Step Search BSCC)**.** *Provided Invariant 6.4 holds, Algorithm 6.1*(G, S, $\emptyset$, T_S) *returns a bottom SCC* $C \subseteq S$ *of* $G[S]$ *in* $O(|T_S| \cdot |C|)$ *symbolic steps.*

Proof. The proof of Theorem 6.3 is a straightforward simplification of the proof of Theorem 6.1. □

Initially the sets T_S are empty. The algorithm maintains Invariant 6.4 for all $S \in \mathcal{X}$. This will ensure the correctness and the number of symbolic steps of Algorithm 6.1 (Section 6.2) as called by the algorithm.

In each iteration of the while-loop one vertex set S is removed from $\mathcal{X}$ and processed. First the random vertices of S with edges to vertices of $V \setminus S$ are identified and their random attractor is removed from S. After this step, there are no random vertices with edges from S to $V \setminus S$. The predecessors of the removed vertices that are contained in S are added to T_S and additionally T_S is updated to only include vertices that are still in S. This preserves Invariant 6.4 (see also [Loi16, Lemma 4.5.2]). The number of symbolic steps for the attractor computation can be charged to the removed vertices and is therefore bounded by $O(n)$ in total.

If afterwards $G[S]$ does not contain an edge anymore, then S is not considered further and the algorithm continues with the next iteration. Otherwise one of three cases happens.

Case (1): If T_S is empty, then by Invariant 6.4 $G[S]$ is strongly connected, contains at least one edge and does not contain a random vertex with edges to $V \setminus S$, i.e., S is an end-component, and by Invariant 6.3 it is a MEC. In this case the algorithm adds the set S to goodC, which preserves both invariants and can happen at most $O(n)$ times.

Case (2): If there are at least $\sqrt{m}$ vertices in T_S, then the set T_S is deleted and as in the basic algorithm all SCCs of $G[S]$ are computed and add to $\mathcal{X}$ as new candidates for MECs. For each of the SCCs C a set T_C is initialized with the empty set. As a vertex is added to a set T_S only if one of its incoming edges is removed by the algorithm, Case (2) can happen only $O(\sqrt{m})$ times over the whole algorithm. Thus the total number of symbolic steps for this case is $O(n\sqrt{m})$. Note that the Invariants 6.4 and 6.3 are preserved.

Case (3): If T_S contains less than $\sqrt{m}$ vertices, then Algorithm 6.1(G, S, $\emptyset$, T_S) is called. By Invariant 6.4 and Theorem 6.3 the procedure returns a bottom SCC C of $G[S]$ in $O(|T_S| \cdot |C|)$ many symbolic steps. Since there are no random edges between S and $V \setminus S$ in $\mathcal{M}$ and C has no outgoing edges in $G[S]$, we have that C is an end-component if it contains at least one edge. By Invariant 6.3 it is also a MEC and is correctly added to goodC. As the sets in goodC are not considered further by the algorithm, we can charge the symbolic steps of Algorithm 6.1 to the vertices of C. Thus this part takes at most $O(n\sqrt{m})$ symbolic steps over the whole algorithm. The vertices of $S \setminus C$ are added back to $\mathcal{X}$, which preserves Invariant 6.3. The predecessors of C in $S \setminus C$ are added to $T_{S \setminus C}$ and vertices of C are removed from $T_{S \setminus C}$, which preserves Invariant 6.4.

By the above case analysis we have that each vertex set that is added to goodC is indeed a MEC (soundness). By Invariant 6.3 and $\mathcal{X} = \emptyset$ at termination of the algorithm we further have completeness. In each iteration either S does not contain an edge and is not considered further, a set is added to goodC (and not contained in $\mathcal{X}$ after that) or case (2) happens. Thus there are at most $O(n + \sqrt{m})$ iterations of the algorithm. The symbolic operations we have not yet accounted for in the analysis of the number of symbolic steps are of $O(1)$ per iteration. Hence Algorithm 6.5 takes $O(n\sqrt{m})$ symbolic steps and correctly computes the MECs of the given MDP $\mathscr{M}$.

Lemma 6.4 (Invariants of Improved Algorithm for MEC)**.** *Invariant 6.4 and Invariant 6.3 are preserved throughout Algorithm 6.5, i.e., they hold before the first iteration, after each iteration, and after termination of the main while-loop. Further, Invariant 6.4 is preserved during each iteration of the main while-loop.*

Proof.

Invariant 6.4. The proof of maintaining Invariant 6.4 in Algorithm 6.5 is a straightforward simplification of the proof of maintaining Invariant 6.1 in Algorithm 6.3.

Invariant 6.3 – Disjointness. The sets in $\mathcal{X} \cup$ goodC are pairwise disjoint at the initialization since goodC is initialized as $\emptyset$. Furthermore, whenever a set S is added to goodC in an iteration of the main while-loop, a superset $\tilde{S} \supseteq S$ is removed from $\mathcal{X}$ in the same iteration of the while-loop. Therefore by induction the disjointness of the sets in $\mathcal{X} \cup$ goodC is preserved.

Invariant 6.3 – Containment of maximal end-components. At initialization, $\mathcal{X}$ contains all SCCs of G. Each maximal end-component X of $\mathscr{M} = (G = (V, E), (V_1, V_R), \delta)$ is strongly connected, so there exists an SCC $Y \supseteq X$ of G such that $Y \in \mathcal{X}$.

Consider a set $S \in \mathcal{X}$ that is removed from $\mathcal{X}$ at the beginning of an iteration of the main while-loop. Consider further a maximal end-component X of $\mathscr{M}$ such that $X \subseteq S$. We require that a set $Y \supseteq X$ is added to either $\mathcal{X}$ or goodC in this iteration of the main while-loop.

First, after we remove $\mathit{Attr}_R(G, S \cap V_R \cap \mathsf{Pre}(V \setminus S))$ from S, we maintain the fact that $X \subseteq S$ by Lemma 6.2. Second, $G[S]$ contains an edge since $X \subseteq S$. Finally, one of the three cases happens:

Case (1): If $|T_S| = 0$, then the set $S \supseteq X$ is added to goodC.

Case (2): If $|T_S| \geq \sqrt{m}$, then the algorithm computes the SCCs of $G[S]$. Since $X \subseteq S$ is strongly connected, it is completely contained in some SCC Y of $G[S]$, and Y is added to $\mathcal{X}$.

Case (3): If $0 < |T_S| < \sqrt{m}$, then the algorithm partitions S into C and $S \setminus C$. By Theorem 6.3 we have that C is a (bottom) SCC of $G[S]$. Since $X \subseteq S$ is strongly connected, it is completely contained in some SCC of $G[S]$. Therefore either $X \subseteq C$ or $X \subseteq (S \setminus C)$. The set $S \setminus C$ is added to $\mathcal{X}$. If $X \subseteq C$, then in particular $G[C]$ contains an edge, and C is added to goodC.

By the above case analysis we have that a set $Y \supseteq X$ is added to either $\mathcal{X}$ or goodC in the iteration of the main while-loop. □

We summarize with the main result of this section.

Theorem 6.4 (Improved Algorithm for MEC)**.** *Algorithm 6.5 correctly computes the MEC decomposition of MDPs and requires $O(n \cdot \sqrt{m})$ symbolic steps.*

Proof.

Correctness. A candidate set can be added to goodC in three cases. When S is added to goodC at Line 12 (resp. at Line 17), then it contains an edge by the check at Line 10, it is strongly connected by $|T_S| = 0$ and Invariant 6.4 (resp. by the result of allSCCs), and it has no random vertices with edges to $V \setminus S$ by the random attractor removal at Lines 6–9. When C is added at Line 25, then it contains an edge by the check at Line 24, it is strongly connected by Theorem 6.3, it contains no random vertices with edges to $V \setminus S$ by the random attractor removal at Lines 6–9, and it contains no random vertices with edges to $S \setminus C$ by the fact that C is a bottom SCC of $G[S]$ (see Theorem 6.3). Therefore we have that whenever a candidate set is added to goodC, it is an end-component, and by induction and Invariant 6.3 we have that it is a maximal end-component (soundness).

Finally, by soundness, Invariant 6.3, the termination of the algorithm (shown below), and the fact that $\mathcal{X} = \emptyset$ at the termination of the algorithm, we have that goodC contains all the maximal end-components of $\mathscr{M}$ (completeness).

Symbolic steps analysis. By [GPP08], the initialization with the SCCs of a given MDP takes $O(n)$ symbolic steps.

In each iteration of the outer while-loop, a set S is removed from $\mathcal{X}$ and (a) S is added to goodC, or (b) at least two sets that are (subsets of) a partition of S are added to $\mathcal{X}$, or (c) S is partitioned into two sets, one of them may be added to goodC and the other is added to $\mathcal{X}$. All three cases can happen at most $O(n)$ times, so there can be at most $O(n)$ iterations of the outer while-loop. The Pre and Post operations at Lines 6, 9, 10, 24, and 27 can be charged to the iterations of the outer while-loop.

Each CPre$_R$ operation executed as a part of the random attractor computation at Line 7 adds at least one vertex to A, and the vertices of A are then not considered any further in the algorithm. Therefore there can, in total, be at most $O(n)$ CPre$_R$ operations over all attractor computations at Line 7.

Note that every vertex in each of T_S can be attributed to at least one unique implicit edge deletion since we only add vertices to T_S that are predecessors of the vertices that were separated from S (or deleted from the maintained graph). Whenever the case $|T_S| \geq \sqrt{m}$ occurs, for all subsets $C \subseteq S$ that are then added to $\mathcal{X}$, we initialize $T_C = \emptyset$. Therefore, the case $|T_S| \geq \sqrt{m}$ can happen at most $O(\sqrt{m})$ times throughout the algorithm since there are at most m edges that can be deleted. By [GPP08] we have a bound $O(n)$ for one iteration, so we can bound the total number of symbolic steps in all iterations of this case by $O(n \cdot \sqrt{m})$.

It remains to bound the number of symbolic steps in Algorithm 6.1. Let C be the set returned by $\mathrm{Lock\text{-}Step\text{-}Search}(G,\ S,\ \emptyset,\ T_S)$. By Theorem 6.3 and the fact that $|T_S| < \sqrt{m}$, the number of symbolic steps in this call is bounded by $O(\sqrt{m} \cdot |C|)$, and the set C is not considered further in the algorithm after this call. Hence we can bound the total number of symbolic steps in all calls of the procedure by $O(n \cdot \sqrt{m})$. □

6.5 MDPs with Streett Objectives

In this section we present the basic and improved symbolic algorithms for model checking MDPs with Streett objectives.

6.5.1 Basic Symbolic Algorithm for MDPs with Streett Objectives

We refer to the basic symbolic algorithm for MDPs with Streett objectives as $\mathrm{StreettMDPbasic}$, and the pseudocode is given in Algorithm 6.6. The key differences compared to Algorithm 6.2 are as follows: (a) SCC computation is replaced by MEC computation; (b) along with the removal of bad vertices, their random attractor is also removed; and (c) removing the attractor ensures that the check required for trivial SCCs for graphs (Line 9) is not required any further.

To compute the almost-sure winning set for MDPs with Streett objectives, we first find all (maximal) good end-components and then solve almost-sure reachability with the union of the good end-components as target set as the last step of the algorithm. This is correct by Lemma 6.1. Towards finding all good end-components, the algorithm maintains two sets, the set goodEC of identified good end-components that is initially empty and the set $\mathcal{X}$ of end-components that are candidates for good end-components that is initialized with the MECs of the MDP. In each iteration of the while-loop one set S is removed from the set of candidates $\mathcal{X}$ and the set of bad vertices Bad(S) of S is determined. If Bad(S) is empty, then S is a good end-component and added to goodEC. Otherwise the random attractor of Bad(S) in $\mathscr{M}[S]$ is removed from S, which by Corollary 6.1 does not remove any vertices that are in a good end-component. The remaining vertices of S have no outgoing random edges and thus still induce a sub-MDP but the sub-MDP might not be strongly connected any more. Then the MECs of this sub-MDP are added to $\mathcal{X}$. These operations maintain the invariants that (i) each set in $\mathcal{X}$ is an end-component and (ii) each good end-component is a subset of one set in either goodEC or $\mathcal{X}$. By (i) a set in $\mathcal{X}$ is a (maximal) good end-component if it does not contain any bad vertices, i.e., in particular this holds for the sets added to goodEC (soundness). By (ii) and $\mathcal{X} = \emptyset$ at termination of the while-loop the algorithm identifies all (maximal) good end-components of the MDP (completeness). Since in each iteration of the while-loop either (1) a set is removed from $\mathcal{X}$ and added to goodEC or (2) bad vertices are removed from a set and not considered further by the algorithm, there can be at most $O(n)$ iterations of the while-loop. Furthermore, whenever bad vertices are removed, then the number of target pairs a given candidate set intersects is reduced by one. Thus each vertex is considered in at most $O(k)$ iterations of the while-loop. The most expensive operation in the while-loop is the computation of the MECs. Denoting the number of symbolic steps for the MEC computation with $O(\textsc{mec})$, the number of symbolic steps of Algorithm 6.6 is $O(\min(n,k) \cdot \textsc{mec})$ (assuming that the number of symbolic steps for the almost-sure reachability computation is lower than that).

Proposition 6.3. *Algorithm 6.6 correctly computes the almost-sure winning set in MDPs with Streett objectives and requires $O(n^2 \cdot \min(n,k))$ symbolic steps.*

Remark 6.1. The above bound uses the basic symbolic MEC decomposition algorithm. Using our improved symbolic MEC decomposition algorithm, the above bound could be improved to $O(n \cdot \sqrt{m} \cdot \min(n,k))$.

Algorithm 6.6: StreettMDPbasic: Basic Algorithm for MDPs with Streett Obj.

```
Input  : MDP $\mathcal{M} = ((V, E), (V_1, V_R), \delta)$ and pairs $\mathrm{SP} = \{(L_i, U_i) \mid 1 \le i \le k\}$
Output : $\langle\langle 1 \rangle\rangle_{as}(\mathcal{M}, \mathrm{Streett}(\mathrm{SP}))$
1  $\mathcal{X} \leftarrow$ allMECs($\mathcal{M}$); goodEC $\leftarrow \emptyset$
2  while $\mathcal{X} \neq \emptyset$ do
3      remove some $S \in \mathcal{X}$ from $\mathcal{X}$
4      $B \leftarrow \bigcup_{1 \le i \le k : U_i \cap S = \emptyset} (L_i \cap S)$
5      if $B \neq \emptyset$ then
6          $S \leftarrow S \setminus Attr_R(\mathcal{M}[S], B)$
7          $\mathcal{X} \leftarrow \mathcal{X} \cup$ allMECs($\mathcal{M}[S]$)
8      else
9          goodEC $\leftarrow$ goodEC $\cup \{S\}$
10 return $\langle\langle 1 \rangle\rangle_{as}\left(\mathcal{M}, Reach(\bigcup_{X \in \text{goodEC}} X)\right)$
```

6.5.2 Improved Symbolic Algorithm for MDPs with Streett Objectives

We refer to the improved symbolic algorithm for model checking MDPs with Streett objectives as StreettMDPimpr, and the algorithm is in Algorithm 6.7. First we present the main ideas for the improved symbolic algorithm. Then we explain the key differences compared to the improved symbolic algorithm for graphs.

Main Ideas.

1. First, we improve the algorithm by interleaving the symbolic MEC computation with the detection of bad vertices [CDHL16, Loi16]. This allows to replace the computation of MECs in each iteration of the while-loop with the computation of SCCs and an additional random attractor computation.

   a) *Intuition of interleaved computation.* Consider a candidate for a good end-component S after a random attractor to some bad vertices is removed from it. After the removal of the random attractor, the set S does not have random vertices with outgoing edges. Consider that further Bad(S) $= \emptyset$ holds. If S is strongly connected and contains an edge, then it is a good end-component. If S is not strongly connected, then $\mathcal{M}[S]$ contains at least two SCCs and some of them might have random vertices with outgoing edges. Since end-components are strongly connected and do not have random vertices with outgoing edges, we have that (1) every good end-component is completely contained in one of the SCCs of $\mathcal{M}[S]$ and (2) the random vertices of an SCC with outgoing edges and their random attractor do not intersect with any good end-component (see Lemma 6.2).

   b) *Modification from basic to improved algorithm.* We use these observations to modify the basic algorithm as follows: First, for the sets that are candidates for good end-components, we do not maintain the property that they are end-components, but only that they do not have random vertices with outgoing edges (it still holds that every maximal good end-component is either already identified or contained in one of the candidate sets). Second, for a candidate set S, we repeat the removal of bad vertices until Bad(S) $= \emptyset$ holds before we continue with the next step of the algorithm. This allows us to make progress after the removal of bad vertices by computing all SCCs (instead of MECs) of the remaining sub-MDP. If there is only one SCC, then this is a good end-component (if it contains at least one edge). Otherwise (a) we remove from each SCC the set of random vertices with outgoing edges and their random attractor and (b) add the remaining vertices of each SCC as a new candidate set.

2. Second, as for the improved symbolic algorithm for graphs, we use the symbolic lock-step search to quickly identify a top or bottom SCC every time a candidate has lost a small number of edges since the last time its superset was identified as being strongly connected. The symbolic lock-step search is described in detail in Section 6.2.

Differences to the Improved Graph Algorithm. Using interleaved MEC computation and lock-step search leads to a similar algorithmic structure for Algorithm 6.7 as for our improved symbolic algorithm for graphs (Algorithm 6.3). The key differences are as follows: First, the set of candidates for good end-components is initialized with the MECs of the input graph instead of the SCCs. Second, whenever bad vertices are removed from a candidate, also their random attractor is removed. Further, whenever a candidate is partitioned into its SCCs, for each SCC, the random attractor of the vertices with outgoing random edges is removed. Finally, whenever a candidate S is separated into C and $S \setminus C$ via symbolic lock-step search, the random attractor of the vertices with outgoing random edges is removed from C, and the random attractor of C is removed from S.

The following invariant is maintained throughout Algorithm 6.7 for the sets in goodEC and $\mathcal{X}$.

Invariant 6.5 (Maintained Sets). *The sets in $\mathcal{X} \cup$ goodEC are pairwise disjoint and for every good end-component C of G there exists a set $Y \supseteq C$ such that either $Y \in \mathcal{X}$ or $Y \in$ goodEC.*

Furthermore, the algorithm maintains the invariant that each candidate for a good end-component $S \in \mathcal{X}$ contains no random edges to vertices not in S.

Invariant 6.6 (No Random Outgoing Edges). *Given an MDP $\mathcal{M}$ and its underlying graph $G = (V, E)$, for each set $S \in \mathcal{X}$ there are no random vertices in S with edges to vertices in $V \setminus S$.*

Finally, for each candidate set $S \in \mathcal{X}$ the algorithm remembers sets H_S and T_S of vertices that have lost incoming resp. outgoing edges since the last time a superset of S was identified as being strongly connected. The algorithm maintains Invariant 6.1 and therefore it can use Algorithm 6.1 together with its correctness guarantee and bound on symbolic steps provided by Theorem 6.1.

Lemma 6.5 (Invariants of Improved Algorithm for MDPs). *Invariant 6.1, Invariant 6.5, and Invariant 6.6 are preserved throughout Algorithm 6.7, i.e., they hold before the first iteration, after each iteration, and after termination of the main while-loop. Further, Invariant 6.1 is preserved during each iteration of the main while-loop.*

Proof.

Invariant 6.1. The proof is a minor extension of the maintenance proof for Algorithm 6.3. In terms of strong connectivity of a candidate S and the maintenance of the sets H_S and

Algorithm 6.7: StreettMDPimpr: Improved Alg. for MDPs with Streett Obj.

```
Input  : MDP $\mathscr{M} = ((V,E),(V_1,V_R),\delta)$ and pairs $\mathrm{SP} = \{(L_i,U_i) \mid 1 \le i \le k\}$
Output : $\langle\langle 1\rangle\rangle_{as}(\mathscr{M}, \mathrm{Streett}(\mathrm{SP}))$
1  $\mathcal{X} \leftarrow$ allMECs($\mathscr{M}$); goodEC $\leftarrow \emptyset$
2  foreach $C \in \mathcal{X}$ do  $H_C \leftarrow \emptyset$; $T_C \leftarrow \emptyset$
3  while $\mathcal{X} \neq \emptyset$ do
4      remove some $S \in \mathcal{X}$ from $\mathcal{X}$
5      $B \leftarrow \bigcup_{1 \le i \le k : U_i \cap S = \emptyset}(L_i \cap S)$
6      while $B \neq \emptyset$ do
7          $A \leftarrow Attr_R(\mathscr{M}[S], B)$
8          $S \leftarrow S \setminus A$
9          $H_S \leftarrow (H_S \cup \mathsf{Post}(A)) \cap S$
10         $T_S \leftarrow (T_S \cup \mathsf{Pre}(A)) \cap S$
11         $B \leftarrow \bigcup_{1 \le i \le k : U_i \cap S = \emptyset}(L_i \cap S)$
12     if Post$(S) \cap S \neq \emptyset$ then                    // $\mathscr{M}[S]$ contains at least one edge
13         if $|H_S| + |T_S| = 0$ then  goodEC $\leftarrow$ goodEC $\cup \{S\}$
14         else if $|H_S| + |T_S| \ge \sqrt{m/\log n}$ then
15             delete $H_S$ and $T_S$
16             $\mathcal{C} \leftarrow$ allSCCs($\mathscr{M}[S]$)
17             if $|\mathcal{C}| = 1$ then  goodEC $\leftarrow$ goodEC $\cup \{S\}$
18             else
19                 foreach $C \in \mathcal{C}$ do
20                     $rout \leftarrow C \cap V_R \cap \mathsf{Pre}(S \setminus C)$
21                     $A \leftarrow Attr_R(\mathscr{M}[C], rout)$
22                     $C \leftarrow C \setminus A$
23                     $H_C \leftarrow \mathsf{Post}(A) \cap C$
24                     $T_C \leftarrow \mathsf{Pre}(A) \cap C$
25                     $\mathcal{X} \leftarrow \mathcal{X} \cup \{C\}$
26         else
27             $(C, H_S, T_S) \leftarrow$ Lock-Step-Search$(G, S, H_S, T_S)$
28             if $C = S$ then  goodEC $\leftarrow$ goodEC $\cup \{S\}$
29             else                                              // separate $C$ and $S \setminus C$
30                 $rout_C \leftarrow C \cap V_R \cap \mathsf{Pre}(S \setminus C)$          // empty if $C$ bottom SCC
31                 $A_C \leftarrow Attr_R(\mathscr{M}[C], rout_C)$                // $= Attr_R(\mathscr{M}[S], S \setminus C) \cap C$
32                 $A_S \leftarrow Attr_R(\mathscr{M}[S], C)$
33                 $C \leftarrow C \setminus A_C$
34                 $S \leftarrow S \setminus A_S$
35                 $H_C \leftarrow \mathsf{Post}(A_C) \cap C$
36                 $T_C \leftarrow \mathsf{Pre}(A_C) \cap C$
37                 $H_S \leftarrow (H_S \cup \mathsf{Post}(A_S)) \cap S$
38                 $T_S \leftarrow (T_S \cup \mathsf{Pre}(A_S)) \cap S$
39                 $\mathcal{X} \leftarrow \mathcal{X} \cup \{S\} \cup \{C\}$
40 return $\langle\langle 1\rangle\rangle_{as}\left(\mathscr{M}, Reach(\bigcup_{C \in \mathsf{goodEC}} C)\right)$
```

T_S, the only difference to the graph case is that after an SCC C is computed by allSCCs or Algorithm 6.1, another subset of vertices A (vertices with outgoing random edges and their random attractor) is removed from C. In this case the invariant is maintained by initializing H_C resp. T_C with the vertices of $C \setminus A$ with edges from resp. to vertices of A, i.e., $H_C \leftarrow \mathsf{Post}(A) \cap C$ and $T_C \leftarrow \mathsf{Pre}(A) \cap C$.

Invariant 6.5 – Disjointness. The sets in $\mathcal{X} \cup$ goodEC are pairwise disjoint at the initialization since goodEC is initialized as $\emptyset$. Furthermore, whenever a set S is added to goodEC in an iteration of the main while-loop, a superset $\tilde{S} \supseteq S$ is removed from $\mathcal{X}$ in the same iteration of the while-loop. Therefore by induction the disjointness of the sets in $\mathcal{X} \cup$ goodEC is preserved.

Invariant 6.5 – Containment of good end-components. At initialization, $\mathcal{X}$ contains all MECs of the input MDP $\mathscr{M} = (G = (V, E), (V_1, V_R), \delta)$. Each good end-component C of P is an end-component, so there exists a MEC $Y \supseteq C$ such that $Y \in \mathcal{X}$ for each good end-component C.

Consider a set $S \in \mathcal{X}$ that is removed from $\mathcal{X}$ at the beginning of an iteration of the main while-loop. Consider further a good end-component C of P such that $C \subseteq S$. We require that a set $Y \supseteq C$ is added to either $\mathcal{X}$ or goodEC in this iteration of the main while-loop.

First, whenever we remove $Attr_R(\mathscr{M}[S], \mathsf{Bad}(S))$ from S, by Corollary 6.1, we maintain the fact that $C \subseteq S$. Second, $P[S]$ contains an edge since $C \subseteq S$. Finally, one of the three cases happens:

Case (1): If $|H_S| + |T_S| = 0$, then the set $S \supseteq C$ is added to goodEC.

Case (2): If $|H_S| + |T_S| \geq \sqrt{m/\log n}$, then the algorithm computes the SCCs of $\mathscr{M}[S]$. If S itself is the (sole) SCC of $\mathscr{M}[S]$, then it is added to goodEC. Otherwise, since $C \subseteq S$ is strongly connected, it is completely contained in some SCC Y of $\mathscr{M}[S]$. Furthermore, since C has no outgoing random edges, by Lemma 6.2 it is contained in Y even after we remove $Attr_R(\mathscr{M}[Y], Y \cap V_R \cap \mathsf{Pre}(S \setminus Y))$ from it. Finally, Y is added to $\mathcal{X}$.

Case (3): If $0 < |H_S| + |T_S| < \sqrt{m/\log n}$, then the algorithm either adds $S \supseteq C$ to goodEC, or partitions S into $\tilde{S}$ and $S \setminus \tilde{S}$. Suppose the latter case happens, then by Theorem 6.1 we have that $\tilde{S}$ is an SCC of $\mathscr{M}[S]$. Further, since $C \subseteq S$ is strongly connected, it is completely contained in some SCC of $\mathscr{M}[S]$. Therefore either $C \subseteq \tilde{S}$ or $C \subseteq (S \setminus \tilde{S})$. If $C \subseteq \tilde{S}$, then by Lemma 6.2 after the removal of $Attr_R(\mathscr{M}[\tilde{S}], \tilde{S} \cap V_R \cap \mathsf{Pre}(S \setminus \tilde{S}))$ from $\tilde{S}$ we maintain that $C \subseteq \tilde{S}$. If $C \subseteq (S \setminus \tilde{S})$, then by Lemma 6.2 after the removal of $Attr_R(\mathscr{M}[S], \tilde{S})$ from $(S \setminus \tilde{S})$ we maintain that $C \subseteq (S \setminus \tilde{S})$. Finally, both $\tilde{S}$ and $S \setminus \tilde{S}$ are added to $\mathcal{X}$.

By the above case analysis we have that a set $Y \supseteq C$ is added to either $\mathcal{X}$ or goodEC in the iteration of the main while-loop.

Invariant 6.6. Given an MDP, the set $\mathcal{X}$ is initialized with the MECs of the MDP, and by definition they have no random outgoing edges. Therefore the invariant holds before the first iteration of the main while-loop.

Consider a candidate set $S \in \mathcal{X}$ in a given iteration of the main while-loop. By the induction hypothesis, S has no random vertices with edges to $V \setminus S$. First, some bad vertices can be iteratively removed from S. At each such removal, the random attractor to these vertices is removed from S as well. After the removal, by the definition of a random attractor, S has no random outgoing edges to the attractor, and therefore by induction has no random outgoing edges to $V \setminus S$. Second, S may be partitioned into at least two proper subsets. Then for each such subset C, the random attractor to random vertices in C with edges to $S \setminus C$ is removed from C. By induction and the definition of a random attractor, after the removal C contains no random outgoing edges to $V \setminus C$ and adding it to $\mathcal{X}$ preserves the invariant.

□

We are now ready to present the main result of this section, which establishes the properties of $\mathrm{StreettMDPimpr}$.

Theorem 6.5 (Improved Algorithm for MDPs)**.** *Algorithm 6.7 correctly computes the almost-sure winning set in MDPs with Streett objectives and requires $O(n \cdot \sqrt{m \log n})$ symbolic steps.*

Proof. Correctness. Whenever a candidate set S is added to goodEC, it contains an edge by the check at Line 12, Bad$(S) = \emptyset$ by the check at Line 6, and it has no outgoing random edges by Invariant 6.6 and the random attractor removal at Line 8. Furthermore, (a) at Line 13, S is strongly connected by Invariant 6.1, (b) at Line 17, S is strongly connected by the result of allSCCs, and (c) at Line 28, S is strongly connected by Theorem 6.1. Therefore we have that whenever a candidate set is added to goodEC, it is indeed a good end-component (soundness).

Finally, by soundness, Invariant 6.5, the termination of the algorithm (shown below), and the fact that $\mathcal{X} = \emptyset$ at the termination of the algorithm, we have that goodEC contains all good end-components of G (completeness).

Symbolic steps analysis. When using our improved symbolic algorithm for MEC decomposition, the initialization takes $O(n \cdot \sqrt{m})$ symbolic steps by Theorem 6.4.

In each iteration of the outer while-loop, a set S is removed from $\mathcal{X}$ and either (a) a set $S' \subseteq S$ is added to goodEC and no set is added to $\mathcal{X}$ or (b) at least two sets that are (subsets of) a partition of S are added to $\mathcal{X}$. Both can happen at most $O(n)$ times, thus there can be at most $O(n)$ iterations of the outer while-loop. The Pre and Post operations at Lines 12, 30, 35, 36, 37, and 38 can be charged to the iterations of the outer while-loop.

An iteration of the inner while-loop (Line 6) is executed only if some vertices B are removed from S; the vertices of B are then not considered further. Thus there can, in total, be at most $O(n)$ Post operations at Line 9 and Pre operations at Line 10 over all iterations of the inner while-loop.

Similarly, each CPre$_R$ operation executed as a part of a random attractor computation adds at least one vertex to the attractor, and the vertices of the attractor are then not considered any further in the algorithm. Therefore there can, in total, be at most $O(n)$ CPre$_R$ operations over all attractor computations at Lines 7, 21, 31, and 32.

Note that every vertex in each of H_S and T_S can be attributed to at least one unique implicit edge deletion since we only add vertices to H_S resp. T_S that are successors resp. predecessors of vertices that were separated from S (or deleted from the maintained graph). Whenever the case $|H_S| + |T_S| \geq \sqrt{m/\log n}$ occurs, for all subsets $C \subseteq S$ that are then added to $\mathcal{X}$, we initialize $H_C = T_C = \emptyset$. Therefore, the case $|H_S| + |T_S| \geq \sqrt{m/\log n}$ can happen at most $O(\sqrt{m \log n})$ times throughout the algorithm since there are at most m edges that can be deleted. In one iteration of this case, the number of symbolic steps executed by allSCCs together with symbolic steps executed at Lines 20, 23, and 24, is bounded by $O(n)$ [GPP08].

It remains to bound the number of symbolic steps in Algorithm 6.1. Let C be the set returned by the procedure; we charge the symbolic steps in this call of the procedure to the vertices of the smaller set of C and $S \setminus C$. By Theorem 6.1 we have either (a) $C = S$, the number of symbolic steps in this call is bounded by $O(\sqrt{m/\log n} \cdot |C|)$, and the set S is added to goodEC or (b) $\min(|C|, |S \setminus C|) \leq |S|/2$ and the number of symbolic steps in this call is bounded by $O(\sqrt{m/\log n} \cdot \min(|C|, |S \setminus C|))$. Case (a) can happen at most once for the vertices of C, and for case (b) note that the size of a set containing a specific vertex can be halved at most $O(\log n)$ times; thus we charge each vertex at most $O(\log n)$ times. Hence we can bound the total number of symbolic steps in all calls to the procedure by $O(n \cdot \sqrt{m \log n})$. □

6.6 Experiments

We present a basic prototype implementation of our algorithm and compare against the basic symbolic algorithm for graphs and MDPs with Streett objectives.

Models. We consider the academic benchmarks from the VLTS benchmark suite [CWI], which gives representative examples of systems with nondeterminism, and has been used in previous experimental evaluation (such as [BCvdP11, CHJS13]).

Specifications. We consider random linear-temporal-logic formulas and use the tool Rabinizer [KK14] to obtain deterministic Rabin automata. Then the negations of the formulas give us Streett automata, which we consider as the specifications.

Graphs. For the models of the academic benchmarks, we first compute SCCs, as all algorithms for Streett objectives compute SCCs as a preprocessing step. For SCCs of the model benchmarks we consider products with the specification Streett automata, to obtain graphs with Streett objectives, which are the benchmark examples for our experimental evaluation. The number of vertices in the benchmarks ranges from 50K to 200K, and the number of transitions ranges from 300K to 5Million.

MDPs. For MDPs, we consider the graphs obtained as above and consider a fraction of the vertices of the graph as random vertices, which is chosen uniformly at random. We consider 10%, 20%, and 50% of the vertices as random vertices for different experimental evaluation.

Experimental evaluation – symbolic steps. In the experimental evaluation we compare the number of symbolic steps (i.e., the number of Pre/Post operations[2]) executed by the algorithms. As the initial preprocessing step is the same for all the algorithms (computing all SCCs for graphs and all MECs for MDPs), the comparison presents the number of symbolic steps executed after the preprocessing. The experimental results for graphs are shown in Figure 6.2 and the experimental results for MDPs are shown in Figure 6.3 (in each figure the two lines represent equality and an order-of-magnitude improvement, respectively).

Discussion. Note that the lock-step search is the key reason for theoretical improvement, however, the improvement relies on a large number of Streett pairs. In the experimental evaluation, the linear-temporal-logic formulas generate Streett automata with small number of

[2]Recall that the basic set operations are cheaper to compute, and asymptotically at most the number of Pre/Post operations in all the presented algorithms.

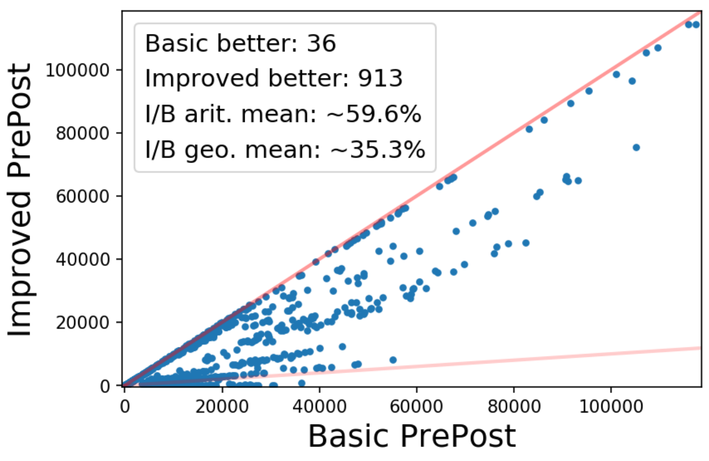
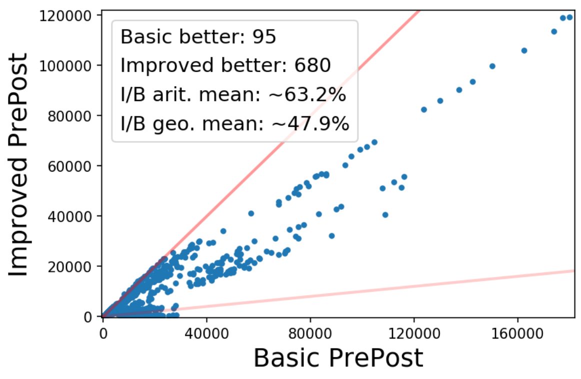
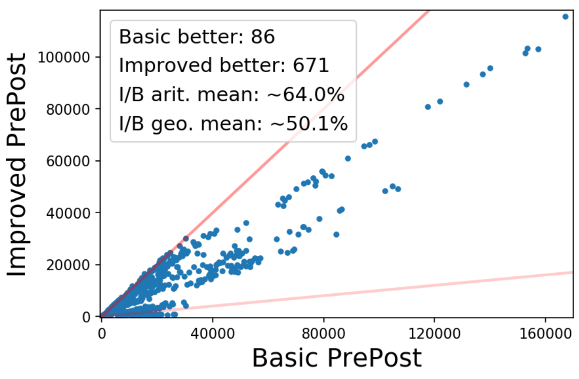


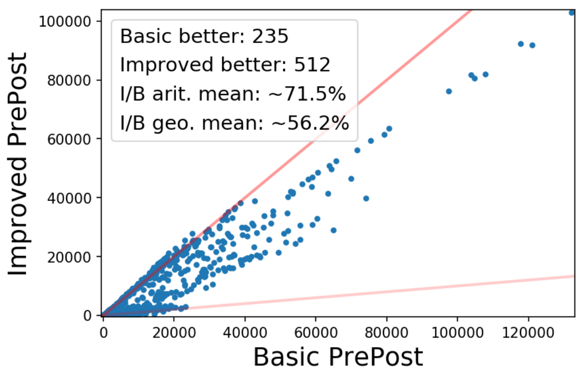
Figure 6.2: Comparison of symbolic steps for graphs with Streett objectives.



(a) 10% random vertices.



(b) 20% random vertices.



(c) 50% random vertices.

Figure 6.3: Comparison of symbolic steps for MDPs with Streett objectives.

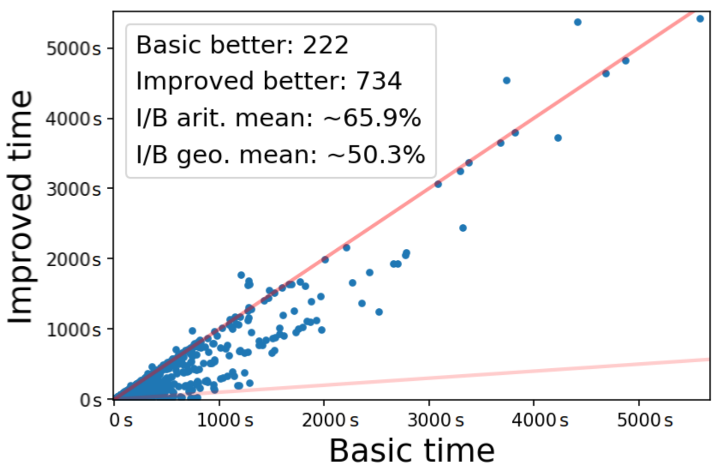


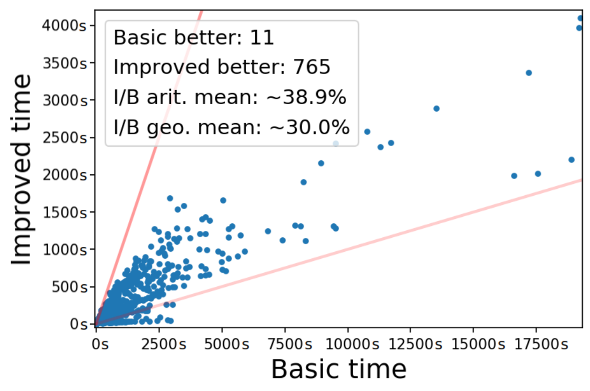
Figure 6.4: Comparison of time for graphs with Streett objectives.

pairs, which after the product with the model accounts for an even smaller fraction of pairs as compared to the size of the state space. This has two effects:

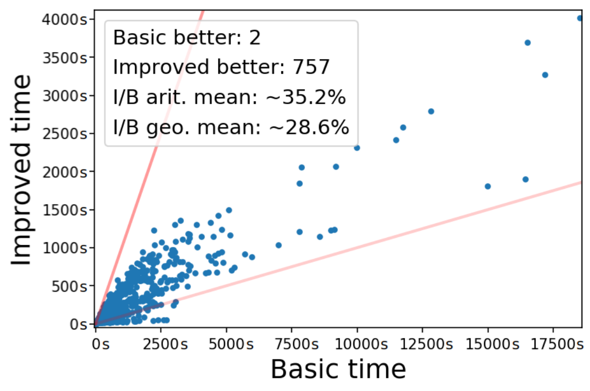
- In the experiments the lock-step search is performed for a much smaller parameter value ($O(\log n)$ instead of the theoretically optimal bound of $\sqrt{m/\log n}$), and leads to a small improvement.
- For large graphs, since the number of pairs is small as compared to the number of states, the improvement over the basic algorithm is minimal.

In contrast to graphs, in MDPs even with small number of pairs as compared to the state-space, the interleaved MEC computation has a notable effect on practical performance, and we observe performance improvement even in large MDPs.

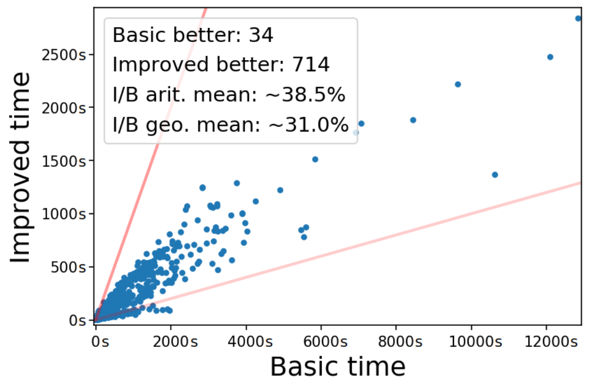
Experimental evaluation – runtime. We further present the results of the experimental evaluation when comparing based on the time. In all the figures, both axes plot the amount of seconds spent on the execution. Similar to the case of the symbolic steps, we begin the measurement after the initial preprocessing step (computing all SCCs for graphs and all MECs for MDPs) is finished. The comparison of running time yields similar results to the comparison of the number of symbolic steps; the results for graphs are shown in Figure 6.4 and the results for MDPs are shown in Figure 6.5.



(a) 10% random vertices.



(b) 20% random vertices.



(c) 50% random vertices.

Figure 6.5: Comparison of time for MDPs with Streett objectives.

CHAPTER 7

Conclusions

In this thesis we have presented several novel works for improved verification of concurrent systems. We have proposed new techniques for stateless model checking (SMC) of concurrent programs using dynamic partial order reduction (POR), and further we have presented new symbolic model-checking algorithms for graphs and Markov decision processes (MDPs) that are used to model finite-state concurrent systems.

In Chapter 3 we have introduced a new equivalence on concurrent traces, called the value-centric (VC) equivalence, which operates under the sequential consistency (SC) memory model and considers the values of trace-events in order to determine whether two traces are equivalent. We have shown that the VC equivalence is coarser than the standard Mazurkiewicz equivalence, while the corresponding consistency checking problem remains solvable in polynomial time. In fact, the coarsening of the VC equivalence can occur even when there are no concurrent write events. In addition, we have developed an SMC algorithm $\mathrm{VC\text{-}DPOR}$ that relies on the VC equivalence to partition the trace space into classes and explore each class efficiently. Our experiments show that, in a variety of benchmarks, the VC equivalence indeed produces smaller partitionings than those explored by alternative, state-of-the-art methods, which often leads to a large reduction in running times.

In Chapter 4 we have developed $\mathrm{RVF\text{-}SMC}$, a new SMC algorithm for the verification of concurrent programs under SC using a novel equivalence called *reads-value-from (RVF)*. On our way to $\mathrm{RVF\text{-}SMC}$, we have revisited the famous sequential-consistency checking problem [GK97]. Despite its NP-hardness, we have shown that the problem is parameterizable in $k + d$ (for k threads and d variables), and becomes even fixed-parameter tractable in d when k is constant. We have further developed practical heuristics that solve the problem efficiently in many practical settings. Our $\mathrm{RVF\text{-}SMC}$ algorithm couples our solution for the sequential-consistency checking to a novel exploration of the underlying RVF partitioning, and is able to model check many concurrent programs where previous approaches time-out. Our experimental evaluation reveals that RVF is very often the most effective equivalence, as the underlying partitioning is exponentially coarser than other approaches. Moreover, $\mathrm{RVF\text{-}SMC}$ generates representatives very efficiently, as the reduction in the partitioning is often met with significant speed-ups in the model checking task.

In Chapter 5 we have solved the consistency verification problem under a reads-from map for the total store order (TSO) and partial store order (PSO) relaxed memory models. Our algorithms scale as $O(k \cdot n^{k+1})$ for TSO, and as $O(k \cdot n^{k+1} \cdot \min(n^{k\cdot(k-1)}, 2^{k\cdot d}))$ for PSO, for n events,

k threads and d variables. Thus, they both become polynomial-time for a bounded number of threads, similar to the case for SC that was established recently [AAJ$^+$19, BE19]. In practice, our algorithms perform much better than the standard baseline methods, offering significant scalability improvements. Encouraged by these scalability improvements, we have used these algorithms to develop, for the first time, SMC under TSO and PSO using the reads-from (RF) equivalence, as opposed to the standard Shasha–Snir equivalence. Our experiments show that the underlying RF partitioning is often much coarser than the Shasha–Snir partitioning, which yields a significant speedup in the model checking task.

In Chapter 6 we have considered symbolic algorithms for graphs and MDPs with Streett objectives, as well as for maximal end-component (MEC) decomposition. With these algorithms we have established new superior algorithmic bounds for the respective problems, and our algorithmic bounds match for both graphs and MDPs. In contrast, while strongly connected components (SCCs) can be computed in linearly many symbolic steps, no such algorithm is known for MEC decomposition.

There are many exciting areas for future work on verification of concurrent systems. In SMC, interesting future work includes further improvements over consistency checking solutions in various settings and memory models, as well as extensions of coarse SMC (e.g., $\mathrm{RVF\text{-}SMC}$ of Chapter 4) to relaxed memory models. In symbolic algorithms, an interesting direction is to explore further improved symbolic algorithms for MEC decomposition, and further improved symbolic algorithms for graphs and MDPs with various classes of objectives.

Moreover, there is potential in exploring other areas that cross over with the topics explored in this thesis. As an example, we remark that consistency-verification algorithms have direct applications beyond SMC. In particular, most predictive dynamic analyses solve a consistency-verification problem in order to infer whether an erroneous execution can be generated by a concurrent system [SES$^+$12, KMV17, Pav19, MPV20, RGB20, MPV21]. Hence, novel consistency checking solutions allow to extend predictive analyses to new settings, e.g. TSO/PSO, in a scalable way that does not sacrifice precision. Further, combination of dynamic POR with symbolic techniques (such as symbolic execution) or static analyses (as in [HH17]) can lead to very practical verification solutions.

Bibliography

[AAA^{+}15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for tso and pso. In *TACAS*, 2015.

[AAdlB^{+}17] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 526–543, Cham, 2017. Springer International Publishing.

[AAJ^{+}19] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[AAJN18] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA):135:1–135:29, 2018.

[AAJS14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *POPL*, 2014.

[ABC^{+}19] Pranav Ashok, Tomáš Brázdil, Krishnendu Chatterjee, Jan Křetínský, Christoph H. Lampert, and Viktor Toman. Strategy representation by decision trees with linear classifiers. In *Quantitative Evaluation of Systems (QEST)*, 2019.

[ACP^{+}21] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless model checking under a reads-value-from equivalence. In *Computer Aided Verification (CAV)*, 2021.

[ACU17] Jade Alglave, Patrick Cousot, and Caterina Urban. Concurrency with Weak Memory Models (Dagstuhl Seminar 16471). *Dagstuhl Reports*, 6(11):108–128, 2017.

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, Dec 1996.

[AH04] R. Alur and T. A. Henzinger. Computer-aided verification. Unpublished, available at `http://www.cis.upenn.edu/group/cis673/`, 2004.

[AJLS18] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–248, Cham, 2018. Springer International Publishing.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, C-27(6):509–516, 1978.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, 2013.

[Alg10] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Paris Diderot University, 2010.

[AQR$^+$04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *CAV*, 2004.

[BBC$^+$06] Jiri Barnat, Lubos Brim, Ivana Cerná, Pavel Moravec, Petr Rockai, and Pavel Simecek. DiVinE - A tool for distributed verification. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 278–281. Springer, 2006.

[BBH$^+$13] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. Divine 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer, 2013.

[BCG$^+$21] Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. The reads-from equivalence for the TSO and PSO memory models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), 2021.

[BCKT18] Tomáš Brázdil, Krishnendu Chatterjee, Jan Křetínský, and Viktor Toman. Strategy representation by decision trees in reactive synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2018.

[BCvdP11] Jiri Barnat, Jakub Chaloupka, and Jaco van de Pol. Distributed algorithms for SCC decomposition. *J. Log. Comput.*, 21(1):23–44, 2011.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 533–553, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BE19] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA):165:1–165:28, 2019.

[BGS06] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. *Form. Methods Syst. Des.*, 28(1):37–56, 2006.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *Int. J. Softw. Tools Technol. Transf.*, 9(5-6):505–525, 2007.

[BK08] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[BL80] James Burns and Nancy A Lynch. Mutual exclusion using invisible reads and writes. In *In Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*. Citeseer, 1980.

[BMM11] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 428–440, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BMTZ21] Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of liveness properties for multithreaded shared-memory programs. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3. ACM, 2002.

[Bry85] R. E. Bryant. Symbolic manipulation of Boolean functions using a graphical representation. In *Conference on Design automation (DAC)*, pages 688–694, 1985.

[CB06] Frank Ciesinski and Christel Baier. LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *QEST*, pages 131–132, 2006.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, 2000.

[CCP$^+$17] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.*, 2(POPL):31:1–31:30, December 2017.

[CDHL16] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. Model and objective separation with conditional lower bounds: Disjunction is harder than conjunction. In *LICS*, pages 197–206, 2016.

[CDHL18] Krishnendu Chatterjee, Wolfgang Dvořák, Monika Henzinger, and Veronika Loitzenbauer. Lower bounds for symbolic computation on graphs: Strongly connected components, liveness, safety, and diameter. In *SODA*, pages 2341–2356, 2018.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2), 1986.

[CGK13] Krishnendu Chatterjee, Andreas Gaiser, and Jan Kretínský. Automata with generalized rabin pairs for probabilistic model checking and LTL synthesis. In *CAV*, pages 559–575, 2013.

[CGMP99] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.

[CGP99a] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CGP99b] E.M. Clarke, O. Grumberg, and D. Peled. Symbolic model checking. In *Model Checking*. MIT Press, 1999.

[CH11] K. Chatterjee and M. Henzinger. Faster and Dynamic Algorithms For Maximal End-Component Decomposition And Related Graph Problems In Probabilistic Verification. In *SODA*, pages 1318–1336, 2011.

[CH12] K. Chatterjee and M. Henzinger. An $O(n^2)$ Time Algorithm for Alternating Büchi Games. In *SODA*, pages 1386–1399, 2012.

[CH14] K. Chatterjee and M. Henzinger. Efficient and Dynamic Algorithms for Alternating Büchi Games and Maximal End-Component Decomposition. *Journal of the ACM*, 61(3):15, 2014.

[CHJS13] K. Chatterjee, M. Henzinger, M. Joglekar, and N. Shah. Symbolic algorithms for qualitative analysis of Markov decision processes with Büchi objectives. *Form. Methods Syst. Des.*, 42(3):301–327, 2013.

[CHL15] K. Chatterjee, M. Henzinger, and V. Loitzenbauer. Improved Algorithms for One-Pair and k-Pair Streett Objectives. In *LICS*, pages 269–280, 2015.

[CHL$^+$18] Krishnendu Chatterjee, Monika Henzinger, Veronika Loitzenbauer, Simin Oraee, and Viktor Toman. Symbolic algorithms for graphs and Markov decision processes with fairness objectives. In *Computer Aided Verification (CAV)*, 2018.

[CJH03] K. Chatterjee, M. Jurdziński, and T. A. Henzinger. Simple stochastic parity games. In *CSL*, pages 100–113, 2003.

[CKK$^+$17] Peter Chini, Jonathan Kolberg, Andreas Krebs, Roland Meyer, and Prakash Saivasan. On the Complexity of Bounded Context Switching. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CL73] Jean-Marie Cadiou and Jean-Jacques Lévy. Mechanizable proofs about parallel processes. In *SWAT*, 1973.

[CL02] Harold W. Cain and Mikko H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 153–154, New York, NY, USA, 2002. Association for Computing Machinery.

[CPT19] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. Value-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.

[CR16] Andreia Correia and Pedro Ramalhete. 2-thread software solutions for the mutual exclusion problem. `https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/cr2t-2016.pdf`, 2016.

[CS20] Peter Chini and Prakash Saivasan. A framework for consistency algorithms. In Nitin Saxena and Sunil Simon, editors, *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020, December 14-18, 2020, BITS Pilani, K K Birla Goa Campus, Goa, India (Virtual Conference)*, volume 182 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[CWI] CWI/SEN2 and INRIA/VASY. The VLTS Benchmark Suite.

[CYH$^+$09] Y. Chen, Yi Lv, W. Hu, T. Chen, Haihua Shen, Pengyu Wang, and Hong Pan. Fast complete memory consistency verification. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 381–392, 2009.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dij83] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 26(1):21–22, January 1983.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is coming: A modern probabilistic model checker. In *CAV*, pages 592–600, 2017.

[DL15] Brian Demsky and Patrick Lam. Satcheck: Sat-directed stateless model checking for sc and tso. In *OOPSLA*, pages 20–36, New York, NY, USA, 2015. ACM.

[EK14] Javier Esparza and Jan Kretínský. From LTL to deterministic automata: A safraless compositional approach. In *CAV*, pages 192–208, 2014.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[FK12] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *CAV*, 2012.

[FM09] Azadeh Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *TACAS*, 2009.

[FMSS15] Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embed. Comput. Syst.*, 14(4), September 2015.

[GHP95] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. *FMSD*, 7(3):227–241, 1995.

[GK97] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, August 1997.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Secaucus, NJ, USA, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, 1997.

[God05] Patrice Godefroid. Software model checking: The verisoft approach. *FMSD*, 26(2):77–101, 2005.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *CAV*, 1993.

[GPP03] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *SODA*, pages 573–582, 2003.

[GPP08] R. Gentilini, C. Piazza, and A. Policriti. Symbolic graphs: Linear solutions to connectivity related problems. *Algorithmica*, 50(1):120–158, 2008.

[HCC$^+$12] W. Hu, Y. Chen, T. Chen, C. Qian, and L. Li. Linear time memory consistency verification. *IEEE Transactions on Computers*, 61(4):502–516, 2012.

[HH16] Shiyou Huang and Jeff Huang. Maximal causality reduction for tso and pso. *SIGPLAN Not.*, 51(10):447–461, October 2016.

[HH17] Shiyou Huang and Jeff Huang. Speeding up maximal causality reduction with static dependency analysis. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pages 16:1–16:22, 2017.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[HT96] M. Henzinger and J. A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *SWAT*, pages 16–27, 1996.

[Hua15] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI*, 2015.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[Kes82] J. L. W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17(2):135–141, Jun 1982.

[KK14] Zuzana Komárková and Jan Kretínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In *ATVA*, pages 235–241, 2014.

[KLSV17] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for c/c++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL):17:1–17:32, December 2017.

[KMV17] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 157–170, New York, NY, USA, 2017. ACM.

[KNP11] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011.

[Knu66] Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9(5):321–322, May 1966.

[KP92] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theor. Comput. Sci.*, 101(2):337–359, 1992.

[KRV19a] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Effective lock handling in stateless model checking. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[KRV19b] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 96–110, New York, NY, USA, 2019. ACM.

[KSH12] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Using unfoldings in automated testing of multithreaded programs. In *ACSD*, 2012.

[KV20] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: model checking for hardware memory models. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1157–1171. ACM, 2020.

[KWG09] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 398–413, Berlin, Heidelberg, 2009. Springer-Verlag.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Techn. J.*, 38(4):985–999, 1959.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LKMA10] Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *FASE*, 2010.

[Loi16] V. Loitzenbauer. *Improved Algorithms and Conditional Lower Bounds for Problems in Formal Verification and Reactive Synthesis*. PhD thesis, University of Vienna, 2016.

[LR09] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *FMSD*, 35(1):73–97, 2009.

[LS20] Magnus Lång and Konstantinos Sagonas. Parallel graph-based stateless model checking. In Dang Van Hung and Oleg Sokolsky, editors, *ATVA*, volume 12302 of *Lecture Notes in Computer Science*, pages 377–393. Springer, 2020.

[LVK$^+$17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 618–632. ACM, 2017.

[Maz87] A Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324. Springer-Verlag New York, Inc., 1987.

[McM95] K. L. McMillan. A technique of state space search based on unfolding. *FMSD*, 6(1):45–65, 1995.

[MH06] C. Manovit and S. Hangal. Completely verifying memory consistency of test program executions. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 166–175, 2006.

[MM07] Tom Ball Madan Musuvathi, Shaz Qadeer. Chess: A systematic testing tool for concurrent software. Technical report, Microsoft Research, November 2007.

[MP96] Zohar Manna and Amir Pnueli. Temporal verification of reactive systems: Progress (draft). Unpublished, available at `http://theory.stanford.edu/~zm/tvors3.html`, 1996.

[MPV20] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. The complexity of dynamic data race prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 713–727, New York, NY, USA, 2020. Association for Computing Machinery.

[MPV21] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. Optimal prediction of synchronization-preserving races. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.

[MQB$^+$08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[ND13] Brian Norris and Brian Demsky. Cdschecker: checking concurrent data structures written with C/C++ atomics. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *OOPSLA*, pages 131–150. ACM, 2013.

[ND16] Brian Norris and Brian Demsky. A practical approach for model checking C/C++11 code. *ACM Trans. Program. Lang. Syst.*, 38(3):10:1–10:51, 2016.

[NRS$^+$18] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 354–371, 2018.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Pav19] Andreas Pavlogiannis. Fast, sound, and effectively complete dynamic race prediction. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[Pel93] Doron Peled. All from one, one for all: On model checking using representatives. In *CAV*, 1993.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12:115–116, 1981.

[PF77] Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 91–97, New York, NY, USA, 1977. ACM.

[PLV19] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, 2019.

[RBS00] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *FMCAD*, pages 143–160, 2000.

[RGB20] Jake Roemer, Kaan Genç, and Michael D. Bond. Smarttrack: Efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 747–762, New York, NY, USA, 2020. Association for Computing Machinery.

[RSSK15] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *CONCUR*, 2015.

[SA06] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *FASE*, 2006.

[SA07] Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *HVC*, 2007.

[Saf88] S. Safra. On the complexity of ω-automata. In *FOCS*, pages 319–327, 1988.

[SES$^+$12] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM.

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[SI94] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[SKH12] Olli Saarikivi, Kari Kahkonen, and Keijo Heljanko. Improving dynamic partial order reductions for concolic testing. In *ACSD*, 2012.

[Som15] F. Somenzi. CUDD: CU decision diagram package release 3.0.0, 2015.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

[SSO$^+$10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[Szy88] B. K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 621–626, New York, NY, USA, 1988. ACM.

[Tar72] R. E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[TKL$^+$12] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In *FMOODS/FORTE*, 2012.

[Tsa98] Yih-Kuen Tsay. Deriving a scalable algorithm for mutual exclusion. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, pages 393–407, London, UK, UK, 1998. Springer-Verlag.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *Petri Nets*, 1991.

[WYKG08] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS*, 2008.

[ZBEE19] Rachid Zennou, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Gradual consistency checking. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 267–285. Springer, 2019.

[ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *PLDI*, 2015.