Verification and Enforcement of Safe Schedules for Concurrent Programs

Metzler, Patrick

(2020)

Verification and Enforcement of Safe Schedules for Concurrent Programs

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des Grades
Doctor rerum naturalium (Dr. rer. nat.)
von

Patrick Metzler

Erstgutachterin:	Prof. Dr. rer. nat. Kirstin Peters
Zweitgutachter:	Prof. Dr. phil. Georg Weissenbacher

Disputation: 24. April 2020

Darmstadt, 2020
D 17

Veröffentlicht bei TUprints, E-Publishing-Service der TU Darmstadt.

URN: `urn:nbn:de:tuda-tuprints-134321`

URL: `https://tuprints.ulb.tu-darmstadt.de/id/eprint/13432`

Abstract

Automated software verification can prove the correctness of a program with respect to a given specification and may be a valuable support in the difficult task of ensuring the quality of large software systems. However, the automated verification of concurrent software can be particularly challenging due to the vast complexity that non-deterministic scheduling causes.

This thesis is concerned with techniques that reduce the complexity of concurrent programs in order to ease the verification task. We approach this problem from two orthogonal directions: state space reduction and reduction of non-determinism in executions of concurrent programs.

Following the former direction, we present an algorithm for dynamic partial-order reduction, a state space reduction technique that avoids the verification of redundant executions. Our algorithm, EPOR, eagerly creates schedules for program fragments. In comparison to other dynamic partial-order reduction algorithms, it avoids redundant race and dependency checks. Our experiments show that EPOR runs considerably faster than a state-of-the-art algorithm, which allows in several cases to analyze programs with a higher number of threads within a given timeout.

In the latter direction, we present a formal framework for using incomplete verification results to extract safe schedulers. As incomplete verification results do not need to proof the correctness of all possible executions of a program, their complexity can be significantly lower than complete verification results. Hence, they can be faster obtained. We constrain the scheduling of programs but not their inputs in order to preserve their full functionality. In our framework, executions under the scheduling constraints of an incomplete verification result are safe, deadlock-free, and fair. We instantiate our framework with the IMPACT model checking algorithm and find in our evaluation that it can be used to model check programs that are intractable for monolithic model checkers, synthesize synchronization via assume statements, and guarantee fair executions.

In order to safely execute a program within the set of executions covered by an incomplete verification, scheduling needs to be constrained. We discuss how to extract and encode schedules from incomplete verification results, for both finite and infinite executions, and how to efficiently enforce scheduling constraints, both in terms of reducing the time to look up permission of executing the next event and executing independent events concurrently (by applying partial-order reduction).

A drawback of enforcing scheduling constraints is a potential overhead in the execution time. However, in several cases, constrained executions turned out to be even faster than unconstrained executions. Our experimental results show that iteratively relaxing a schedule can significantly reduce this overhead. Hence, it is possible to adjust the incurred execution time overhead in order to find a sweet spot with respect to the amount of effort for creating schedules (i.e., the duration of verification). Interestingly, we found cases in which a much earlier reduction of execution time overhead is obtained by choosing favorable scheduling constraints, which suggests that execution time performance does not simply rely on the number of scheduling constraints but to a large extend also on their structure.

Zusammenfassung

Automatisierte Softwareverifikation erlaubt es, die Korrektheit eines Programms in Bezug auf eine gegebene Spezifikation zu beweisen und kann somit eine wertvolle Unterstützung bei der schwierigen Aufgabe der Qualitätssicherung großer Softwaresysteme sein. Jedoch stellt die zusätzliche Komplexität nichtdeterministischen Schedulings eine besondere Herausforderung für die automatisierte Verifikation nebenläufiger Software dar.

Diese Dissertation befasst sich mit Techniken zur Reduktion der Komplexität nebenläufiger Programme, um das Verifikationsproblem zu erleichtern. Wir erarbeiten dazu Lösungen aus zwei unterschiedlichen Richtungen: Zustandsraumreduktion und Reduktion von Nichtdeterminismus in Ausführungen nebenläufiger Programme.

Ersterer Richtung folgend stellen wir einen Algorithmus für Dynamic-Partial-Order-Reduction vor, einer Zustandsraumreduktion, die die Verifikation redundanter Ausführungen vermeidet. Unser Algorithmus, EPOR, erzeugt begierig (eager) Schedules für Programmfragmente. Im Vergleich zu anderen Algorithmen für Dydamic-Partial-Order-Reduction vermeidet er redundante Race- und Abhängigkeitstests. Unsere Experimente zeigen, dass EPOR deutlich schneller als ein bekannter und aktueller Algorithmus läuft, was in mehreren Fällen die Verifikation von Programmen mit mehr Threads innerhalb einer gegebenen Zeit erlaubt.

In letzterer Richtung stellen wir ein formales Framework zur Nutzung von unvollständigen Verifikationsergebnissen zur Erstellung von sicheren Schedules vor. Da unvollständige Verifikationsergebnisse nicht die Korrektheit aller möglichen Ausführungen eines Programms beweisen müssen, kann ihre Komplexität deutlich niedriger als bei vollständigen Verifikationsergebnissen sein. Dadurch können sie sehr viel schneller generiert werden. Wir schränken das Scheduling von Programmen ein, jedoch nicht ihre Eingabe, um ihre Funktionalität zu erhalten. In unserem Framework sind Ausführungen innerhalb den Schedulingbeschränkungen eines unvollständigen Verifikationsergebnisses sicher (safe), deadlockfrei und fair. Wir instantiieren unser Framework mit dem IMPACT Model-Checking-Algorithmus und zeigen in unserer Evaluation, dass es genutzt werden kann um Programme zu verifizieren, die nicht durch monolithisches Model-Checking handhabbar sind, um Synchronisation durch assume-Statements zu synthetisieren und um faire Ausführungen zu garantieren.

Um ein Programm sicher innerhalb der durch ein unvollständiges Verifikationsergebnisses abgedeckten Ausführungen auszuführen, muss das Scheduling beschränkt werden. Wir diskutieren, wie aus unvollständigen Verifikationsergebnissen Schedules extrahiert und encodiert werden können, sowohl für endliche als auch für unendliche Ausführungen. Zusätzlich diskutieren wir, wie Schedulingbeschränkungen effizient umgesetzt werden können, sowohl im Hinblick auf ein schnelles Nachschlagen der Erlaubnis, das nächste Event auszuführen, als auch auf die nebenläufige Ausführung unabhängiger Events (durch die Anwendung von Partial-Order-Reduction).

Ein Nachteil der Umsetzung von Schedulingbeschränkungen ist ein potenzieller Overhead in der Ausführungsdauer, jedoch erwiesen sich beschränkte Ausführungen in mehreren Fällen sogar als schneller als unbeschränkte Ausführungen. Unsere experimentellen Ergebnisse zeigen, dass das iterative Lockern von Schedulingbeschränkungen den Overhead in der Ausführungsdauer reduzieren kann. Daher ist es möglich, den erlittenen Overhead anzupassen, sodass ein optimaler Bereich im Hinblick auf die zur Erzeugung der Schedules nötigen Zeit (also die Dauer der Verifikation) gefunden wird. Interessanterweise zeigen sich Fälle, in denen durch die Wahl geeigneter Schedules der Overhead bereits viel früher reduziert werden kann, was nahelegt, dass die Ausführungsgeschwindigkeit nicht einfach nur von der Anzahl an Schedulingbeschränkungen abhängt, sondern zu einem großen Teil auch von deren Struktur.

Acknowledgments

While working on this thesis, I had the great fortune to work with and learn from numerous people. I am sincerely grateful for their technical and moral support.

First of all, I would like to thank Kirstin Peters for her support, advice, and helpful comments on this thesis. For supervising me during my stay in Vienna and afterwards, I am very grateful to Georg Weissenbacher. His remarks, comments, and advice have contributed a great deal to my research. Furthermore, I thank Neeraj Suri for giving me the opportunity to develop this thesis within the DEEDS group at TU Darmstadt. My gratitude goes to Kirstin and Georg, as well as Carsten Binnig, Christian Bischof, and Felix Wolf for serving on my defense committee.

I would also like to thank my colleagues Peter and Habib for constructive and helpful collaborations in the DEEDS FM group at TU Darmstadt. I very much enjoyed our time together, being it in Darmstadt, Budapest, or elsewhere. Thanks as well to all my other companions in the DEEDS group for making our time there so much more enjoyable. I am also much obliged that I had the pleasure to work with and supervise Asit Dhal, Robin Hesse, Timo Kalle, Jens Keim, Jisu Lee, Dinh-Van Vo, Nouri Al Nahawi, Sreeram Sadasivam, Augustin Wilberg, and Konstantin Wolf on their software projects as well as Bachelor's and Master's theses.

Most importantly, I am deeply indebted to my wife for her continuous support. No matter what I was struggling with, she found the right words that helped me to carry on. Last but not least, I am also truly thankful to my parents for their unconditional support at all times and particularly during all my years at the university.

Contents

Chapter 1

Introduction

Software verification can prove the correctness of a program with respect to a given specification, or property. This feature may be a valuable support in the difficult task of ensuring the quality of large software systems, especially when the verification is automated and carried out by a machine. However, limits on both the capabilities and scalability of software verification make it hard to implement feasible verification approaches. A particular challenge is the complexity of concurrent software that is executed non-deterministically.

This thesis is concerned with techniques that reduce the complexity of concurrent programs in order to ease the verification task. We approach this problem from two orthogonal directions: state space reduction and reduction of non-determinism in executions of concurrent programs.

The former direction, state space reduction, reduces the subset of states of a program that has to be checked during the verification process. The process of finding a solution to the verification problem is simplified, which enables potentially faster verification or verification of otherwise intractable programs.

The latter direction reduces the non-determinism in executions of concurrent programs, not only during the verification process but also during the usage of a program. By constraining the scheduling of a concurrent program, non-determinism is reduced or even completely eliminated. In contrast to state space reduction, the verification problem itself is simplified. The verification result gives fewer guarantees but may be easier obtained.

In our contributions, we focus on automated verification (rather than interactive theorem proving), model checking, verification of safety properties (rather than liveness properties), and an application to multi-threaded programs with shared memory (rather than actor programs or distributed systems).

This chapter introduces the background and motivation of our contributions.

1.1 Concurrency

Concurrent programs consist of several entities which interact through shared resources. Depending on the context, entities may be processes, threads, or actors. Shared resources may be shared memory, message passing communication, synchronization primitives, or inter-process communication. Concurrency may considerably increase a program's complexity and induce particular issues such as undesirable non-deterministic behavior, difficult allocation of shared resources, deadlocks, and starvation. Additionally, non-deterministic behavior may impede detection and reproduction of defects. Hence, concurrency makes support for software quality assurance more desirable and challenging at the same time.

In the context of concurrency, the state space explosion problem [Val96] describes the fact that the state space of a concurrent system grows exponentially with the number of processes (or actors) the system is comprised of and with the length of executions. State space explosion occurs when the ordering between processes is non-deterministic. For example, the scheduling of many general purpose operating systems is non-deterministic in that the order of memory accesses of different processes and threads (as well as messages between them) cannot be foreseen prior to an execution.

An *interleaving semantics* simplifies the behavior of a concurrent program such that in every execution, any two events cannot occur at the same time but happen before and after each other, respectively. Hence, executions in an interleaving semantics correspond to a linearly ordered sequence. A generalization of interleaving semantics without this simplification is *true concurrency.* In this thesis, we use an interleaving semantics to model concurrent programs under non-deterministic scheduling, as it provides a sufficient level of detail for general purpose multi-core architectures and is commonly used in related literature. A comparison of interleaving and true concurrency semantics is provided by Priese and Wimmel [PW98].

Under an interleaving semantics, a program with n threads that execute $m_1, \ldots, m_n$ events, respectively, has (a maximum of) $\frac{(\sum m_1,\ldots,m_n)!}{\prod m_1!,\ldots,m_n!}$ interleavings. Consider, for instance, a program with two threads. The number of interleavings grows already over 10^{29}, cf. Figure 1.1, when each thread executes only 50 events. Consequently, the state space can be prohibitively large even for a program of moderate size. Decades of research have been conducted on making verification of concurrent systems feasible, yet concurrency still poses a considerable hurdle.

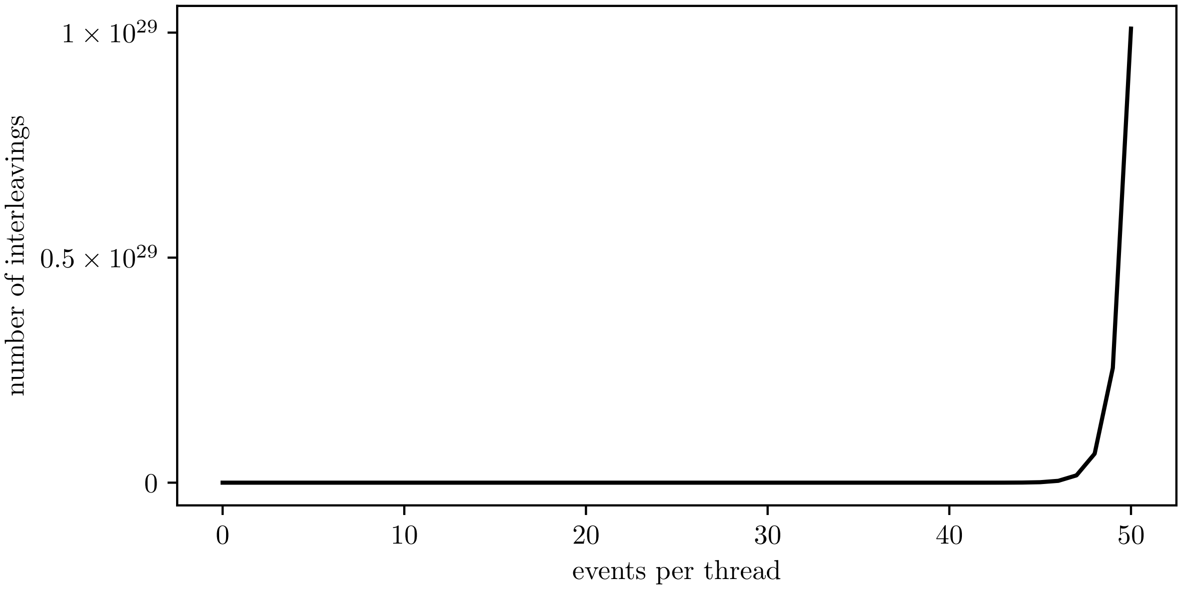

Figure 1.1: Number of interleavings of two threads

1.2 Model checking

A major technique for automated software verification is model checking, an approach to systematically explore all reachable states of a program in order to prove its correctness. While other techniques such as static analyses, abstract interpretation, type systems, and theorem provers may be valuable to prove the correctness of programs, model checking allows both a fully automated analysis and an expressiveness at least as high as other automated techniques. Nonetheless, model checking techniques may well combine state space exploration with other program analysis techniques. In this thesis, we deal with fully automated software verification and therefore focus on model checking.

Properties to be verified can be classified into *safety* and *liveness* properties. Intuitively, safety properties state that an undesirable state will never be reached and can be refuted by a finite execution leading to such a state. Hence, safety properties can be reduced to a reachability problem of an error state. Liveness properties, on the other hand, state that some desirable event will eventually happen and can only be refuted by an infinite execution. Many interesting properties of programs, such as memory safety, deadlock-freedom, exception-freedom, and satisfaction of all assertions in a program, are safety properties. In this thesis, we deal with the verification of safety properties, although we also consider the concept of *fairness*, i.e., a balanced allocation of resources to threads, which is a liveness property.

Since the introduction of *bounded (symbolic) model checking* (BMC) [BCCZ99, CBRZ01], state representations in propositional logic, ready to be processed by an automated satisfiability checker, or SAT solver, have been widely used for model checking. The combination of *abstraction refinement* [LPJ$^+$96, Par97, PH98] with the use of spurious counterexamples to guide the refinement yielded *counter example guided abstraction refinement* (CEGAR) [Kur94, BS93, CGJ$^+$03]. Intuitively, information extracted from a spurious counter example, which shows a property violation but does not correspond to an execution, is used to refine the current symbolic description without loosing too much abstraction.

Based on a SAT representation, McMillan introduced Craig interpolation [Cra57] in model checking [McM03] to find invariants describing safe, reachable states. Similar to CEGAR with interpolants, an approach for infinite state programs that uses interpolants of spurious counter examples, called *lazy abstraction with interpolants*, was introduced later [McM06].

Bradley introduced a SAT-based approach, *IC3* [Bra11], that, instead of unrolling the transition relation, incrementally strengthens the property to be proven. For the verification of software, the use of satisfiability modulo theories (SMT) [CGS10] instead of (or in combination with) a SAT solver contributed to transfers of model checking approaches for finite state systems (modeling hardware) to infinite state systems (modeling software), e.g., by Cimatti for IC3 [CG12].

As noted above, model checkers for concurrent programs have to handle, in addition to common hurdles with model checking for sequential programs, the state space explosion problem due to non-deterministic scheduling. Several techniques have been proposed for this purpose, many of which handle concurrency explicitly. Notable examples include extensions of lazy abstraction with interpolants [WKO13] and IC3 [GLW16] for concurrent programs. An alternative approach is *sequentialization* [LR09, LMP09], which transforms a concurrent program into a sequential program that simulates all (or a chosen subset) of the interleavings of the original program. After sequentialization, a model checker for sequential programs can be used. Most model checkers for concurrent programs apply state space reduction, as discussed below, in order to mitigate the state space explosion problem.

The complexity of the verification problem poses such a serious challenge that model checkers commonly fail to prove the correctness of a program, which typically manifests in a diverging run of the model checker. At this point, it is unclear whether the program under analysis is correct and the time spent on model checking is wasted. *Conditional model checking*, introduced by Beyer et al. [BHKW12], tries to make use of such failed verification attempts by reusing information from an incomplete verification result as a starting point for an other model checker. By repeatedly model checking the same program, each time with more initial information about already checked states, it is possible to verify programs which cannot not be handled by the same model checker monolithically. In the same spirit of making use of incomplete verification attempts, we propose an iterative model checking approach. Advancing even further in this direction, we show how such intermediate verification results can be useful even if no eventual complete verification result is produced.

1.3 State space reduction

State space reduction is a prominent approach to state space explosion due to concurrency. Its goal is to identify a subset of the reachable states to be explored such that correctness of all reachable states can still be proven. An early state space reduction approach was introduced by Lipton [Lip75]. Lipton's reduction tries to identify *left* and *right movers*, statements that can be executed as an atomic sequence of statements such that the program semantics is not changed with respect to an interesting property. Exploration of executions that interleave atomic statement sequences can be skipped.

Generalizing left and right movers, *trace theory* by Mazurkiewicz [Maz86, Maz95] laid a foundation for state space reductions with the concept of dependency between events and equivalence between executions. Equivalence classes are called *traces* and are characterized by a partial order between the events of their executions. Only if two events are dependent, the order of their execution can be observed, whence switching the order of two adjacent, independent events yields an equivalent execution.

Partial-order reduction (POR) [God96] is prominently used for verification of concurrent systems. It makes use of trace theory by focusing the exploration to one representative of each trace. Many enhancements and applications of POR have been presented, of which we mention only the most relevant for this thesis.

A major step in the development of POR techniques was the introduction of *dynamic partial-order reduction (DPOR)* [FG05]. Instead of a static dependency relation that is sound in all states of a program, a dynamic dependency relation between events is used which may depend on the current state. This finer-grained dependency relation may avoid the exploration of redundant executions. Besides many other POR approaches based on DPOR, *Cartesian partial-order reduction* [GFYS07] stands out in that in a single step of the exploration algorithm, a sequence, instead of a single event, is explored. Further notable POR approaches in the succession of DPOR include *source-DPOR* and *optimal-DPOR* [AAJS14]. The former algorithm has been shown to perform a considerably higher reduction than the original DPOR algorithm. The latter algorithm, optimal-DPOR, provides the guaranty that no more than one representative per trace is explored, however may be slower than source-DPOR because of additional bookkeeping of already explored event sequences.

While POR is widely adopted in model checking tools, it is not sufficient to eliminate the state space explosion problem for many interesting programs. In particular, the state space of a program may still be exponentially large after reduction. Thus, it is desirable to ease the verification problem beyond POR.

1.4 Restricting non-determinism of schedulers

Restricting a non-deterministic scheduler has been approached from two perspectives: with the goal of easing verification, at the cost of potentially missing incorrect executions, and with the goal of executing a concurrent program fully or partially deterministically. Context bounding [QR05] limits the number of context switches, i.e., switches between distinct threads, for each execution to a given bound. Executions with more context switches may occur but are not explored during verification. This focus on executions with few context switches has proven to be beneficial to find counter examples to the correctness of a program and has accordingly been used in several BMC approaches, e.g., [RG05, MQ07, FIP13]. However, none of these approaches enforces the context bound during the execution of a program, which prevents the verification from proving the correctness of a program under a non-deterministic scheduler.

Reducing the non-determinism of scheduling during execution, in contrast, is the goal of deterministic multi-threading (DMT) [OAA09] and related approaches. Several variants have been proposed that either guarantee a deterministic execution or that the execution will follow one of a small set of schedules. The obtained determinism or stability in occurring schedules eases concurrency testing, as failures are easier to reproduce and the probability of detecting a bug during testing is increased.

In case of full determinism, DMT guarantees that for every possible input, a deterministic schedule is executed. However, it is in general unknown whether executing a program under a given input will trigger a known or a new schedule. Indeed, it is possible to observe very different schedules for similar inputs and executing a program with deterministic scheduling may not even reduce the set of possible schedules. With the incentive that concurrency testing and verification benefit all the more from DMT if the set of possible schedules is small, *schedule memoization* by Cui et al. [CWTY10] was introduced by Cui, et al. Schedule memoization stores a set of schedules for selected inputs and attempts to keep schedules for new inputs similar to a stored schedule. Still, a guarantee that a similar schedule is possible for any new input cannot be given. A first step towards a set of a priori known schedules for all inputs was presented by Bergan, Ceze, and Grossman [BCG13]. Their approach generates a set of *input-covering schedules*, to which a program can be restricted to for any input. Some limitations still exist, for instance that the set of input-covering schedules may be much larger than necessary and that programs may not contain unprotected races. Nevertheless, we believe that this direction is promising and explore further opportunities in this thesis.

1.5 Contributions

The following contributions are contained within this thesis.

Efficiency for DPOR algorithms (Chapter 3) As described in Section 1.3, dependency between events of a concurrent program is a central concept for POR. Especially in DPOR algorithms, checks for dependency between two events in a given state are a common operation and responsible for a large share of the execution time of a DPOR algorithm. We show that avoiding redundant dependency checks may considerably accelerate a DPOR algorithm. We implement our approach by modifying the existing source DPOR algorithm (SDPOR) [AAJS14]. SDPOR has been shown to be more efficient than other proposed DPOR algorithms, i.e., for a given program, its execution time is shorter than that of other DPOR algorithms. Our experiments compare our algorithm, Eager POR (EPOR), to SDPOR and show that EPOR's time savings allow to verify programs that cannot be verified under SDPOR within a large time limit.

Iterative model checking and use of incomplete verification results (Chapter 4) As noted before, the state of the art in model checking for concurrent programs is unsatisfactory in that many interesting programs cannot be handled by current model checking tools. We propose an iterative model checking approach that makes use of incomplete verification results. Each iteration solves the given verification problem under certain scheduling constraints in order to reduce its complexity. The result of each iteration is a correctness proof under the current scheduling constraints or a counter example to the property. Consequently, from the first iteration result on, the verifier produces useful information, whereas a monolithic approach would waste resources if it fails to prove correctness under arbitrary scheduling.

We provide a formal framework within which we propose general requirements on useful, incomplete verification results. We design an iterative model checking algorithm and implement it in the IMPARA tool [WKO13]. Our experiments in several case studies show that our approach can be used to:

- model check programs that are intractable for monolithic model checkers
- safely execute a program, given the scheduling constraints of an incomplete verification result and even in the presence of unsafe executions
- synthesize synchronization given a specification on correct synchronization (via `assume` statements inserted into the program) and
- guarantee fair executions.

Enforcement of scheduling constraints for safe execution of concurrent programs (Chapter 5) In order to use scheduling constraints extracted from incomplete verification results to safely execute a program, it is necessary to enforce these constraints, e.g., by modifying the operating system scheduler or by modifying the program. Ideally, such modifications should not force the program into a strictly sequential execution, which would foil any benefit of concurrency. Furthermore, such modification may introduce a considerable execution time overhead, which may be crucial to the applicability of schedule enforcement.

We show that scheduling constraints extracted from incomplete verification results can be transformed into schedules that allow a concurrent execution of events. We design two types of schedules, for finite and infinite executions, respectively. Through applying POR on the given scheduling constraints, an ordering between events is only enforced where their dependency indicates that a different ordering may deviate from the program behavior described by an incomplete verification result. We implement both types of schedule enforcement and evaluate the execution time overhead on several benchmark programs. While the execution time overhead may be considerable, we propose several optimizations so that constrained executions show a much smaller overhead and in several cases are even faster than unconstrained executions.

1.6 Previous publications

Parts of this thesis have been published in the following articles. Material from these publications has been, partly verbatim, used in this thesis.

1. [MSB$^+$16]

   Patrick Metzler, Habib Saissi, Péter Bokor, Robin Hesse, and Neeraj Suri.

   Efficient verification of program fragments: Eager POR.

   Automated Technology for Verification and Analysis – 14th International Symposium, ATVA, 2016.

   The original publication is available at `http://link.springer.com/chapter/10.1007%2F978-3-319-46520-3_24`.

   This article presents the POR algorithm and related findings of Chapter 3.

2. [MSBS17]

   Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri.

   Quick verification of concurrent programs by iteratively relaxed scheduling.

   Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE, 2017.

   This article presents our concept of iteratively verifying and executing concurrent programs, shown here in Chapter 4.

3. [MSBS18]

   Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri.

   Safe execution of concurrent programs by enforcement of scheduling constraints.

   CoRR, abs/1809.01955, 2018, updated 2020, `https://arxiv.org/abs/1809.01955`.

   This article presents our approach to the enforcement of scheduling constraints for concurrent programs, shown here in Chapter 5.

4. [MSW19]

   Patrick Metzler, Neeraj Suri, and Georg Weissenbacher.

   Extracting Safe Thread Schedules from Incomplete Model Checking Results.

   Proceedings of the 26th International SPIN Symposium on Model Checking of Software, 2019.

   This article presents an iterative model checking approach for concurrent programs. Its findings occur in Chapter 4 as well as in Section 5.2.

5. [MSW20]

   Patrick Metzler, Neeraj Suri and Georg Weissenbacher.

   Extracting Safe Thread Schedules from Incomplete Model Checking Results.

   International Journal on Software Tools for Technology Transfer, STTT, 2020.

   This article is an extended version of our SPIN 2019 article [MSW19].

Chapter 2

System Model

We model multi-threaded programs with shared memory. Unless otherwise stated, program input may be non-deterministic and executions may be infinite. A *program P* consists of a set $\mathscr{T}$ of threads, a set S of states (including the initial state s_{init}), a set Q of (program) variables, a set L of local locations to describe control flow locations of threads, and the error location $\mathfrak{l}_{error}$:

Definition 1 (program)**.** *A* program *is a tuple $P = (\mathscr{T}, S, s_{init}, Q, L, \mathfrak{l}_{error})$, where S is the (potentially infinite) set of states, $s_{init} \in S$ is the unique initial state, Q is the set of variables, L is the set of local locations, $\mathfrak{l}_{error}$ is the unique error location, and $\mathscr{T}$ is a finite, totally-ordered set of threads T.*

States comprise an interpretation of the program variables and a global (control flow) location, which in turn consists of one local location per thread:

Definition 2 (location, state)**.** *The set of local locations is partitioned into a set of local locations L_T for each thread T. A global location $\mathfrak{l} \in L^{|\mathscr{T}|}$ is a tuple of one local location for each thread, i.e., $\mathfrak{l} \in L_{T_1} \times \ldots \times L_{T_n}$.*

A state $s \in S$ is composed of a global location $\mathfrak{l}(s)$ and an interpretation of the variables Q, which maps variables to values.

We assume that the location of the initial state is not the error location, i.e., $\mathfrak{l}(s_{init}) \neq \mathfrak{l}_{error}$. We write $s(v)$ for the value of variable v in state s. We write $\mathfrak{l}_T(s)$ for the local location of thread T at state s and $\mathfrak{l}_T$ for the local location of thread T in the global location $\mathfrak{l}$. Since local locations are disjoint, we also write $\mathfrak{l}(s)$ for $\mathfrak{l}_T(s)$ if T is clear from the context. The variables Q are partitioned into a set of global variables and a set of local variables for each thread. We write Q_T for the union of the global variables and the local variables of thread T.

```
1  initially:
2    empty buffer of size N
3    count = 0
4    mutex = 0
5  thread $T_1$:
6    while true:
7      produce()
8  thread $T_2$:
9    while true:
10     consume()
11 produce:
12   lock(mutex)
13   if count < N:
14     put item
15     count += 1
16   unlock(mutex)
17 consume:
18   lock(mutex)
19   if count > 0:
20     remove item
21     count −= 1
22   unlock(mutex)
```

Figure 2.1: Producer-consumer problem

Each thread is associated with a local transition relation between states, where it cannot change variable values or local locations of other threads. The global transition relation is the union of the local transition relations:

Definition 3 (global and local transition relation)**.** *The global transition relation R_G of a program is partitioned into a local transition relation $R_T \subseteq S \times S$ for each thread $T \in \mathscr{T}$. A thread may not change the local location and local variables of other threads, i.e., for all $q \in Q \setminus Q_T$, for all $s, s' \in S$ with $R_T(s, s')$, and for all threads $T' \neq T$, we require that $s(q) = s'(q)$ and $\mathfrak{l}_{T'}(s) = \mathfrak{l}_{T'}(s')$.*

A local transition relation consists of single transitions, which have a distinct local location as a source and a distinct local location as a destination. Intuitively, a control flow branching is modeled by multiple transitions and a statement without a control flow branching is modeled by a single transition. For example, lock(mutex) in line 12 in Figure 2.1 can be modeled by a single transition $R_{12,13}$ from location 12 to 13 and the following **if** statement can be modeled by two transitions $R_{13,14}$ and $R_{13,16}$, one from location 13 to 14 and one from 13 to 16.

The guard of a transition R, intuitively, is satisfied by a state s with the local source location of R if and only if there exists a state that can be reached from s through R:

Definition 4 (transition, guard)**.** *A thread's local transition relation R_T is partitioned into (local)* transitions *$R_{l,l'} \subseteq R_T$ such that for all states s, s':*

$$R_{l,l'}(s, s') \Leftrightarrow (R_T(s, s') \land l = \mathfrak{l}_T(s) \land l' = \mathfrak{l}_T(s'))$$

The guard *of a transition R, written $Guard(R)$, encodes the predicate $\exists s'.\, R(s, s')$.*

For example, the guard of transition $R_{13,14}$ from location 13 to 14 in Figure 2.1 is $Guard(R_{13,14}) = (\mathsf{count} < \mathsf{N})$.

We write $\mathscr{F}(Q)$ for the set of all first order formulae over the variables Q and optional additional interpreted symbols. A *state formula* is a formula $\phi \in \mathscr{F}(Q)$, i.e., encodes all states s in which $\phi(s)$ evaluates to true. We assume that each transition R can be represented by a *transition formula* $R_1 \wedge R_2$ such that $R_1 \in \mathscr{F}(Q)$ represents the guard $\mathit{Guard}(R)$ and $R_2 \in \mathscr{F}(Q \cup Q')$ represents the relation between state and successor state.

A transition R, respectively its thread T, is active at a given local location l, intuitively, if R models the statement at l. R, respectively T, is enabled at a given state s if R is active at $\mathfrak{l}_T(s)$ and s satisfies the guard of R.

Definition 5 (active and enabled transition)**.** *Let $R_{l,l'}$ be a transition of a thread T. $R_{l,l'}$ is* active *at local location $\mathfrak{l}$ (in states s with $\mathfrak{l}_T(s) = l$). $R_{l,l'}$ is* enabled *in those states s that satisfy $\mathfrak{l}_T(s) = l$ and $\mathit{Guard}(R_{l,l'})$. We require that there exists at most one enabled transition for a given thread and state.*

We write $\mathit{Transitions}(\mathfrak{l}_T) := \{R_{l,l'} \subseteq R_T : l = \mathfrak{l}_T\}$ for the set of active transitions of T at $\mathfrak{l}_T$. We write $\mathit{Next\text{-}Transition}(s, T)$ for the (unique) enabled transition of T in s if it exists and otherwise $\mathit{Next\text{-}Transition}(s, T) = \bot$. For a state s, we write $\mathit{enabled}(s) = \bigcup_{T \in \mathscr{T}} \mathit{Next\text{-}Transition}(s, T)$ for the set of enabled transitions of all threads in s.

While $\mathit{Next\text{-}Transition}(s, T) = R$ is unique (if it exists), there may exist multiple successor states s' with $R(s, s')$ and the program input decides which successor state to take.

Definition 6 (branching transition)**.** *A transition $R_{l,l'}$ is* branching *if there exist states s, s_1, s_2 such that $s_1 \neq s_2$, $R_{l,l'}(s, s_1)$, and $R_{l,l''}(s, s_2)$ for some transition $R_{l,l''}$ of the same thread.*

A branching transition may represent a control flow branching or, in the presence of program input, a non-deterministic assignment to a variable.

Executions are sequences of states, interleaved with threads. Safety properties can be encoded directly into a program (via the error location) so that executions satisfy the safety property if and only if they are safe. To simplify the presentation and without loss of generality, we assume that only a single error location exists. Multiple error locations can be modeled by a single error location and additional transitions to this location.

Definition 7 (execution)**.** *An* execution *τ of a program is a sequence $s_0, T_1, s_1, \ldots$ such that s_0 is the initial state of the program and for every adjacent triple (s_i, T_{i+1}, s_{i+1}) in the sequence, s_i and s_{i+1} are related by the local transition relation of T_{i+1}. If τ is finite, it is of the form $\tau = s_0, T_1, s_1, \ldots, T_n, s_n$ and additionally $\mathit{enabled}(s_n) = \emptyset$ holds. An execution is* safe *if it does not reach the error location, i.e., $\mathfrak{l}(s_i) \neq \mathfrak{l}_{error}$ for $0 \leq i \leq n$.*

An *execution prefix* is a finite prefix of an execution that has a state as its last element, written $\tau' < \tau$.

Deadlocks are states with active transitions but without enabled transitions, i.e., intuitively, the program has not terminated (as active transitions exist) but all threads are blocked.

Definition 8 (deadlock)**.** *A* deadlock *is a state s with $\bigcup_{T \in \mathscr{T}} \mathit{Transitions}(\mathfrak{l}_T(s)) \neq \emptyset \wedge \mathit{enabled}(s) = \emptyset$.*

As deadlocks are prominent issues in concurrent programming, we assume that deadlocks are always undesirable, whether or not they are explicitly marked as errors by the safety property. In order to simplify the presentation when discussing the fairness of executions, we assume in Chapter 4 that there are no finite, desired, executions, i.e., all finite executions lead to a deadlock. In other words, we assume that there exists an active (but not necessarily enabled) transition in all states (this transition can be a dummy transition if a program intentionally terminates). In contrast, in Chapter 3, we discuss only terminating programs whose executions can end in both deadlock and non-deadlock states but are always finite.

Deadlocks may only arise through blocking transitions such as a lock acquisition of an already taken lock.

Definition 9 (blocking transitions)**.** *T* may block *at a location $\mathfrak{l}_T$ if there exist states s, s' with this location $\mathfrak{l}_T = \mathfrak{l}_T(s) = \mathfrak{l}_T(s')$ such that T has an enabled transition at s but no enabled transition at s', i.e., $\mathit{Next\text{-}Transition}(s, T) \neq \bot \wedge \mathit{Next\text{-}Transition}(s', T) = \bot$.*

For example, Thread 1 blocks at line 12 in Figure 2.1 in states where the lock is already taken.

We assume that for each thread T, all locations at which T may block are marked with a predicate $\mathit{may\text{-}block}(\mathfrak{l}_T)$. It is permitted to overapproximate the predicate, at the expense of performance, i.e., we require that for all pairs $(\mathfrak{l}_T, T)$ which may block that $\mathit{may\text{-}block}(\mathfrak{l}_T)$ holds but not the converse. Furthermore, we assume that threads do not block at control flow branchings, i.e., for all threads T and locations l with $\mathit{may\text{-}block}(\mathfrak{l}_T)$, $|\mathit{Transitions}(\mathfrak{l}_T)| = 1$. This requirement can easily be satisfied by splitting transitions that model both a lock acquisition and a control flow branching into two separate transitions.

Fairness assures that in infinite executions, every thread has a chance to eventually make progress. Beyond deadlocks, we assume that unfair executions are undesirable. We use the concept of *(strong) fairness* [BK08].

Definition 10 (fair execution)**.** *An execution τ is* fair *if every thread that is enabled infinitely often along τ is scheduled infinitely often along τ. We write $\mathit{Fair\text{-}Executions}(P)$ for the fair executions of a program P.*

Intuitively, non-determinism can arise through scheduling of threads and through (non-deterministic) inputs, which can be modeled as multiple transitions of a thread that are enabled at a state. A *scheduler* can resolve the former kind of non-determinism. Intuitively, given an execution prefix $s_0, T_1, \ldots, T_n, s_n$, a scheduler chooses the thread that is to be executed in state s_n, unless no thread is enabled in s_n.

Definition 11 (scheduler). *A* scheduler ζ *of a program* P *is a function* $\zeta : (S \times \mathscr{T})^* \times S \to (\mathscr{T} \cup \{\bot\})$ *such that for all sequences* $\tau = s_0, T_1, \ldots, T_n, s_n$*,* ζ *chooses a thread that is enabled at* s_n *if such a thread exists, i.e.,* $\mathit{enabled}(s_n) \neq \emptyset \Rightarrow \mathit{Next\text{-}Transition}(s_n, \zeta(\tau)) \neq \bot$*.*

We write $\mathit{Schedulers}(P)$ for the set of all schedulers of P. We write $\mathit{Executions}(P, \zeta)$ for the set of all executions $\tau = s_0, T_1, s_1, \ldots$ of P such that for each adjacent triple (s_i, T_{i+1}, s_{i+1}), $T_{i+1} = \zeta(s_0, T_1, \ldots, s_i)$. If $\tau = s_0, T_1, \ldots, T_n, s_n$ is finite, additionally $\mathit{enabled}(s_n) = \emptyset$ must hold.

For example, consider an execution prefix of the program of Figure 2.1 where, beginning in the initial state, T_1 executes produce() and lock(mutex) followed by T_2, which executes consume() and lock(mutex) (we do not model **while** true by its own transition):

$$s_{init}, T_1, s_1, T_1, s_2, T_2, s_3, T_2, s_4$$

A scheduler must select T_1 rather than T_2 after this execution prefix since in s_4, the lock is held by T_1 and $\mathit{enabled}_{T_2}(s_5) = \emptyset$.

Definition 12 (deadlock-free and fair scheduler). *A scheduler* ζ *for a program* P *is* deadlock-free*, written deadlock-free*(ζ)*, if no execution in Executions*(P, ζ) *reaches a deadlock and* ζ *is* fair *if all executions in Executions*(P, ζ) *are fair, i.e., Executions*$(P, \zeta) \subseteq$ *Fair-Executions*(P)*.*

It is important to note that unless the program is in a terminal state (no transitions are enabled), a scheduler must schedule a thread that has an enabled transition in that state. A scheduler is deadlock-free if it can always find a deadlock-free execution, even if the program shows executions that reach a deadlock.

Program inputs, the latter source of non-determinism, are modeled as follows. A program has an input alphabet X and each input symbol $\iota \in X$ makes transitions deterministic, i.e., for each transition R, there exists a function $R^\iota : S \to S$ such that for all states s at which R is enabled, $R^\iota(s) = s'$ for some s' with $R(s, s')$. Non-deterministic input, or, more generally, influence from the environment, can be modeled by an *input* (as a dual concept to schedulers), defined as follows.

Definition 13 (input). *An* input *is a function* $\chi : (S \times \mathscr{T})^* \to X$*, which chooses an input symbol depending on the current execution prefix.*

In conjunction, an input and a scheduler render a program completely deterministic: the execution of P under input χ and scheduler ζ is the unique execution $s_0, T_1, s_1, \ldots \in \mathit{Executions}(P, \zeta)$ such that for all adjacent triples (s_i, T_{i+1}, s_{i+1}), for $\mathit{Next\text{-}Transition}(s_i, T) = a$, and for $\chi(s_0, T_1, \ldots, s_i, T_{i+1}) = \iota$, $s_{i+1} = a^\iota(s_i)$.

As transitions may occur repeatedly in a single execution, it is convenient to refer to the *occurrence* of a transition, called an *event*. An event consists of the thread that executes it and a counter that specifies the number of events the same thread has executed before in the same execution.

Definition 14 (event sequence)**.** *The* event sequence ρ *of an execution or execution prefix* $\tau = s_0, T_1, s_1, T_2, s_2, \ldots$ *is defined as* $\rho := (T_1, k_1)(T_2, k_2)\ldots$*, where* k_i *is the number of occurrences of* T_i *in* $T_1 T_2 \ldots$*. We also write* $\rho(\tau)$ *for* ρ*. If* τ *is an execution (rather than an execution prefix),* $\rho(\tau)$ *is* maximal*.*

We write *length*(ρ) for the length of an event sequence ρ. For $e_i = (T_i, k_i)$, we define $\mathit{tid}(e_i) = T_i$. The empty sequence is denoted by ε. Concatenation of a sequence ρ with another sequence ρ' or an element e is written as $\rho \cdot \rho'$ and $\rho \cdot e$, respectively.

POR uses Mazurkiewicz equivalence [Maz86, God95] to identify equivalent executions of which only one needs to be explored. Equivalence classes are called (Mazurkiewicz) traces. Mazurkiewicz equivalence is defined with respect to a dependency relation on transitions. Intuitively, a dependency relation is required to mark two transitions as dependent if their ordering in executions influences some local state or whether one of the two transitions is enabled. For example, a common approach is to mark two transitions as dependent if they are from the same thread or they access the same global variable and at least one of the transitions modifies its value. It is safe to overapproximate a dependency relation, i.e., marking transitions as dependent although they could safely marked as independent as well does not yield incorrect POR results. However, the induced equivalence classes may be smaller and may result in a less effective POR.

Definition 15 (dependency relation [God95])**.** *A dependency relation* $\nparallel$ *is a reflexive, symmetric relation on transitions such that for any two threads* T_1, T_2 *and any two transitions* $R_1 \in R_{T_1}$ *and* $R_2 \in R_{T_2}$*,* R_1 *and* R_2 *may only be independent if* R_1 *does neither enable nor disable* R_2 *and they are commutative, i.e.:*

- $\forall \iota_1, \iota_2 \in X. \forall s \in S. R_1, R_2 \in \mathit{enabled}(s) \Rightarrow R_1^{\iota_1}(R_2^{\iota_2}(s)) = R_2^{\iota_1}(R_1^{\iota_2}(s))$
- $\forall \iota \in X. \forall s \in S. R_1 \in \mathit{enabled}(s) \Rightarrow (R_2 \in \mathit{enabled}(s) \Leftrightarrow R_2 \in \mathit{enabled}(R_1^\iota(s)))$

We write $R_1 \nparallel R_2$ if R_1 and R_2 are dependent and $R_1 \parallel R_2$ if they are independent. We assume that for all programs, an arbitrary dependency relation $\nparallel$ is given. In practice, it is common to consider two transitions dependent if they are from distinct threads and access the same global variable such that at least one access modifies the variable. In order to simplify the detection of dependencies, the set of possibly accessed variables of a transition can be overapproximated without loosing correctness.

Given a dependency relation, we can define Mazurkiewicz equivalence. Intuitively, two executions τ, τ' are (Mazurkiewicz) equivalent if τ' can be obtained from τ by repeatedly swapping two adjacent, independent transitions.

Definition 16 (Mazurkiewicz equivalence)**.** *Mazurkiewicz equivalence is the smallest reflexive, symmetric, and transitive relation $\simeq$ on executions such that for all executions $\tau = s_0, T_1, s_1, T_2, s_2, \dots s_n$ and τ' of the form*

$$\tau' = s'_0, T_1, s'_1, \dots, s'_i, T_{i+2}, s'_{i+1}, T_{i+1}, s'_{i+2}, T_{i+3}, s'_{i+3}, \dots, s'_n$$

(T_{i+1} and T_{i+2} are swapped)

with Next-Transition$(s_i, T_{i+1}) \parallel$ Next-Transition(s_{i+1}, T_{i+2}), $\tau \simeq \tau'$.

Two event sequences $\rho(\tau)$, $\rho(\tau')$ are (Mazurkiewicz) equivalent if their corresponding executions τ, τ' are equivalent.

The states occurring in τ may differ from those in τ', however, it is guaranteed that under the same input, the values of local and global variables of each thread T, V_T, are equal in the states of τ and τ' [God96].

Terminating Programs For our contributions to dynamic partial-order reduction in Chapter 3, we assume that programs always terminate (all executions are finite) and all information about program input is given, intuitively, only in the initial state and executions do not depend on input symbols. This setting is common in related work on dynamic partial-order reduction [God97, FG05, AAJS14].

As executions do not depend on program inputs, all transitions are functions independent of input symbols. For an execution $\tau = s_0, T_1, s_1, \dots, T_n, s_n$ and its corresponding event sequence $\rho = (T_1, k_1) \dots (T_n, k_n)$, each event e_i corresponds to a unique, functional transition $R_\rho(e_i) = R_i$. The property of *branching* directly propagates from a transition to all its events: an event e_i is *branching* if $R_\rho(e_i)$ is branching. As transitions are functions independent of input, we write $R_i(s_{i-1}) = s_i$ or $s_{i-1} \xrightarrow{R_i} s_i$. Whenever $R_\rho(e_i) = R_{l,l'}$ we write $\mathfrak{l}_\rho(e_i) := l$ for the initial location of $R_{l,l'}$.

If there exist states $s_1, \dots, s_n$ and transitions $R_1, \dots, R_{n-1}$ such that $s_1 \xrightarrow{R_1} s_2 \dots s_{n-1} \xrightarrow{R_{n-1}} s_n$, and ρ is the corresponding event sequence, ρ is a *feasible* event sequence at s_1. If $s_1 = s_{init}$, s_n is denoted by s_ρ. The set of feasible event sequences at s_ρ is denoted by *feasible*(ρ).

The *program order* of a program models its control flow. We define the program order as a relation $PO \subseteq R_G \times R_G$ and require that $(R_{l_1,l_2}, R_{l_3,l_4}) \in PO$ whenever $l_2 = l_3$ (which implies that both transitions are from the same thread). For an event sequence $\rho = e_1 \dots e_n \in \mathit{feasible}(\varepsilon)$, we define that $e_i <^{\rho}_{PO} e_j$ whenever $i < j$ and $(R_\rho(e_1), R_\rho(e_2)) \in PO$ ($<^{\rho}_{PO}$ is a partial order on the events of ρ).

As we extend SDPOR by Abdulla et al [AAJS14] in Chapter 3, we adapt their definition of happens-before relations. A notable difference is that we define the ordering of events of the same thread separately as *program orders*.

Definition 17 (happens-before relation [AAJS14]). *For all execution prefixes τ, a happens-before relation assigns a partial order $<^\rho$ to the corresponding event sequence $\rho(\tau) = e_1, \dots, e_n$. $<^\rho$ is a partial order on the events of ρ such that*

- *$e_i <^\rho e_j$ only if $i < j$*
- *For any prefix $\tau' = s_0, T_1, \dots, s_i$ of τ and any transitions $e_j, e_k, j \leq i, k \leq i$, we have $e_j <^\rho e_k$ if and only if $e_j <^{\rho'} e_k$.*
- *Any event sequence ρ' which is a linearization of $<^\rho$ is assigned the same partial order $<^\rho = <^{\rho'}$.*
- *For any two execution prefixes $\tau = s_0, \dots, s_n$ and $\tau' = s'_0, \dots, s'_n$ with $<^{\rho(\tau)} = <^{\rho(\tau')}$, we have $s_n = s'_n$.*
- *For any sequences ρ, ρ', ρ'' such that $\rho \cdot \rho''$ is an event sequence, we have $<^\rho = <^{\rho'}$ if and only if $<^{\rho \cdot \rho''} = <^{\rho' \cdot \rho''}$.*
- *Let $\rho' = e_1, \dots, e_{n-1}, e', e''$ be an alternative event sequence. If $e_{n-1} <^\rho e_n$ and $e_{n-1} \not<^{\rho'} e'$ then $e_{n-1} <^{\rho'} e''$.*
- *Evens of the same thread are not related: $\mathit{tid}(e_i) = \mathit{tid}(e_j) \Rightarrow e_i \not<^\rho e_j$.*

A happens-before relation satisfies the requirements on dependency relations in the sense that for two transitions R_1 and R_2 of an execution with event sequence ρ such that e_1 and e_2 are the corresponding events and e_1 appears before e_2 in ρ, $R_1 \nparallel R_2 \Rightarrow e_1 <^\rho e_2 \vee e_1 <^\rho_{PO} e_2$. Correspondingly, in Chapter 3, we use happens-before relations as a basis for Mazurkiewicz equivalence so that for all event sequences ρ, ρ', we have $\rho \simeq \rho' \Leftrightarrow <^\rho = <^{\rho'} \wedge <^\rho_{PO} = <^{\rho'}_{PO}$.

For an event sequence $\rho \in \mathit{feasible}(\varepsilon)$ with postfix $\rho_2 = e_1 \dots e_n$ ($\rho = \rho_1 \cdot \rho_2$), we write $<^{\rho_1,\rho_2}$ for the happens-before relation of ρ_2 after ρ_1: $<^{\rho_1,\rho_2} = \{(e_1, e_2) \in <^\rho : e_1 \in \rho_2\}$. Similarly, we write $<^{\rho_1,\rho_2}_{PO}$ for the program order of ρ_2 after ρ_1: $<^{\rho_1,\rho_2}_{PO} = \{(e_1, e_2) \in <^\rho_{PO} : e_1 \in \rho_2\}$.

In addition to happens-before relations, we adapt the definition of reversible races by Abdulla et al. Intuitively, two events e, e' in an event sequence constitute a reversible race if there exists an equivalent sequence in which e and e' are adjacent and dependent.

Definition 18 (reversible race [AAJS14])**.** *Let $\rho = e_1 \dots e_n \in \mathit{feasible}(\varepsilon)$ be an event sequence. Two events e_i, e_j of ρ constitute a* reversible race*, written $e_i \precsim_\rho e_j$, if*

$$\begin{aligned} e_i <^\rho e_j \wedge \forall i < k < j.((e_i \not<^\rho e_k \wedge e_i \not<^\rho_{PO} e_k) \vee (e_k \not<^\rho e_j \wedge e_k \not<^\rho_{PO} e_j)) \\ \wedge R_\rho(e_j) \in \mathit{enabled}(e_1 \dots e_{i-1} e_{k_1} \dots e_{k_m}(s_{init})), \end{aligned}$$

where $e_{k_1} \dots e_{k_m}$ is the sequence $e_{i+1} \dots e_{j-1}$ with all events e removed that satisfy $e \not<^\rho e_j \wedge e \not<^\rho_{PO} e_j$.

Chapter 3

Using Program Sections for Efficient Dynamic Partial-Order Reduction

The effectiveness of POR approaches relies on the precision of the dependency relation. In the original POR approaches, dependencies are calculated statically leading to an inaccurate over-approximation. Dynamic partial order reduction approaches [FG05, GFYS07, AAJS14] tighten the precision of the dependency relation by considering only dependencies occurring at runtime, leading to a less redundant exploration.

While exploring the state space of a program, dynamic POR algorithms identify pairs of dependent events which additionally need to be explored in reversed order so that all Mazurkiewicz traces are covered. Such pairs of events constitute a *reversible race* [AAJS14]. In order to detect all reversible races of a program, a dynamic POR algorithm checks for each event whether it constitutes a race with any previous event in the current path. During each such race check, the algorithm needs (often multiple times) to check whether two events are dependent. Therefore, dependency checks constitute a large part of any dynamic POR algorithm's runtime overhead.

In this chapter, we present *Eager POR* (EPOR), an optimization of dynamic POR algorithms such as SDPOR [AAJS14] that significantly reduces the number of dependency checks. EPOR *eagerly* creates schedules to bundle dependency checks for sequences of events instead of checking dependencies in every visited state. These sequences, called *sections*, correspond to program fragments of one or more statements of each thread. By checking races in a section only once, many additional race checks and dependency checks can be avoided. A new constraint system-based representation of Mazurkiewicz traces ensures that all reversible races inside a section are explored in both orderings. As a result, EPOR

```
Thread 1:
    e1: write x
Thread 2:
    e2: read x
Thread 3:
    e3: read x
```

Figure 3.1: Readers-writers benchmark with one writer and two readers.

requires significantly fewer dependency checks compared to other DPOR algorithms where dependencies are checked after the execution of every event.

3.1 Motivating example

As a motivating example, consider the Readers-Writers benchmark in Figure 3.1 (also used in [AAJS14, FG05]). Thread 1 writes to the shared variable x (e_1), Threads 2 and 3 read from x (e_2 and e_3). The dynamic dependencies for all executions are

$$D = \{(e_1, e_2), (e_2, e_1), (e_1, e_3), (e_3, e_1)\},$$

as the operations e_2 and e_3 are commutative (do not constitute a race), while both e_1, e_2 and e_1, e_3 are non-commutative, (constitute a race).

Our approach is based on the observation that the set of all Mazurkiewicz traces of program fragments as in the Readers-writers example can be calculated without exploring any program states and checking for races between operations only once. The program of Figure 3.1 has 4 (Mazurkiewicz) traces and the dynamic POR algorithm SDPOR [AAJS14] explores one execution per trace. Each execution consists of 3 events, hence SDPOR performs 3 race checks per execution (each time an operation is appended to the current partial execution, a check is performed whether the current operation constitutes a race with any previous operation of the current partial execution). Each race check consists of several dependency checks (in order to decide whether e_1 and e_2 constitute a race, pairwise dependencies need to be determined for all events that occur between e_1 and e_2). In total, SDPOR performs 12 race checks and 25 dependency checks.

By exploiting the fact that all executions consist of the same operations and contain the same races, it is possible to reduce the number of race checks to 3 and the number of dependency checks to 8: after exploring an arbitrary execution of the program, we know that each execution consists of e_1, e_2, and e_3 and contains the races $(e_1, e_2), (e_1, e_3)$ (either in this or in reversed order), which can be determined using 3 race checks. We construct four partial orders $\{(e_1, e_2), (e_1, e_3)\}$, $\{(e_2, e_1), (e_1, e_3)\}$, $\{(e_1, e_2), (e_3, e_1)\}$, and $\{(e_2, e_1), (e_3, e_1)\}$, which correspond to the four traces of the program. By computing a linear

```
Thread 0:
    $e_{00}$: y := 0
    $e_{01}$: x[y] := 1
Thread 1:
    $e_{10}$: if x[0] = 0
    $e_{11}$: then z := 1
Thread 2:
    $e_{20}$: y := 1
```

Figure 3.2: A program with branchings.

extension of each partial order, we obtain an execution of each trace. In Section 3.2.1, we explain how to generalize this idea to systems with dynamic dependencies.

3.2 Constraint system-based POR

3.2.1 Exploring programs in sections

Requirements for Sections

As described in our motivating example (Section 3.1), EPOR requires only 3 instead of 12 race detections and only 8 instead of 25 dependency checks when exploring the Readers-Writers program. This reduction is possible because two conditions are met: every event sequence of maximal length feasible at the initial state of Readers-Writers contains the same events and dependencies do not depend on states (it is possible to precisely calculate all dependencies statically).

In order to generalize our approach to arbitrary programs, we identify program fragments called *sections* where a generalization of these two conditions hold:

(A) Every execution of the section contains the same set of events and these events correspond to the same program locations.

(B) Dependencies inside the section are the same among any execution of the section, modulo the ordering of dependent events (hence, it is possible to precisely calculate all dependencies of the section with the information given at the first state of the section).

Once all traces for a section are explored, EPOR performs the same race checks as SDPOR in order to find races between events inside the current section and events preceding the current section.

Throughout this section, we use the program of Figure 3.2 as an example to explain conditions (A) and (B). Here, three threads work on the shared variables x, y, and z, where x is an array of length two. Statements are labeled with events $e_{00}, e_{01}, e_{10}, e_{11}, e_{20}$, meaning that the execution of a statement is modeled by the event it is labeled with. Events e_{00}, e_{01}, and e_{10} constitute a section. Including e_{11} in the same section would violate condition (A) and including e_{20} would violate condition (B), as detailed below.

In order to meet condition (A), we have to ensure that the same set of events occurs, no matter how the section is executed. Different events may occur when a section contains a control flow branching so that both branches are reachable and yield a different number of events. Hence, we want to ensure that such two branches are assigned to separate sections. Formally, we require that for any branching event e in a section of some event sequence $\rho \in \mathit{feasible}(\varepsilon)$, all program order successor events e' ($e <_{PO}^{\rho} e'$) are not contained in the same section as e.

For example, in any event sequence $\rho \in \mathit{feasible}(\varepsilon)$ of the program in Figure 3.2, event e_{11} cannot be part of the same section as e_{10} because e_{10} is a branching event and e_{11} is a program order successor of e_{10}, i.e., $e_{10} <_{PO}^{\rho} e_{11}$.

As long as a section does not contain any branching event and one of its program order successors, condition (A) is satisfied. To illustrate this, assume that there exists an event sequence $\rho \in \mathit{feasible}(\varepsilon)$, in which we swap two events of different threads in the last section of ρ, yielding ρ'. Assume that ρ' does not correspond to an execution, i.e., there exists an event in ρ' that cannot be executed because no enabled transition is available for the given thread. Let e be the first of any such events and let R be the transition $R_{\rho}(e)$ that corresponds to e in ρ. Since threads may not block by assumption, R can only be disabled because no transition that enables it is executed in ρ'. Hence, there must exist a program predecessor e' of e such that e' occurs between the swapped events and e. Both e and e' lie in the same section, thus the above requirement is violated.

A section satisfies condition (B) if the dependencies inside the section can be determined at the first state of the section. This condition holds if swapping two dependent events inside a section does not influence whether following events are dependent. We characterize such a pair of dependent events that influences following dependencies as *hiding dependency* so that the absence of hiding dependencies implies (B). Let $\rho_0 \in \mathit{feasible}(\varepsilon)$ be an event sequence with a reversible race $e_1 \precsim_{\rho_0} e_2$. Then there exists an equivalent event sequence $\rho = \rho_1 \cdot e_1 \cdot e_2 \cdot \rho_2$. Let $\rho' = \rho_1 \cdot e_2 \cdot e_1 \cdot \rho_2$ be ρ with the race reversed. e_1 and e_2 form a *hiding dependency*, written $e_1 \xrightarrow{*}_{\rho} e_2$, if:

- ρ' is feasible at the initial state, i.e., $\rho' \in \mathit{feasible}(\varepsilon)$, and
- there exist e, e' such that e and e' have the same initial location in both ρ and ρ', i.e., $\mathfrak{l}_{\rho}(e) = \mathfrak{l}_{\rho'}(e)$ and $\mathfrak{l}_{\rho}(e') = \mathfrak{l}_{\rho'}(e')$, and
- e and e' are happens-before related in $e_1 \cdot e_2 \cdot \rho_2$ after ρ_1, i.e., $e <^{\rho_1, e_1 \cdot e_2 \cdot \rho_2} e'$, and
- e and e' are not happens-before related in $e_2 \cdot e_1 \cdot \rho_2$ after ρ_1, i.e., $e \not<^{\rho_1, e_2 \cdot e_1 \cdot \rho_2} e'$

In the example of Figure 3.2, event e_{20} cannot be in the same section as event e_{00} because they constitute a hiding dependency: the order in which e_{00} and e_{20} are executed influences the fact whether e_{01} and e_{10} are dependent and constitute a race.

A section which contains no hiding dependency trivially satisfies condition (B). Although dependencies inside of sections have to be independent of states inside the section, dynamic information about dependencies that is known at the beginning of a section can be accounted for. Therefore, EPOR makes use of all dynamic dependency information just as SDPOR.

Implementing Section Construction

In order to implement an algorithm that relies on sections, it is desirable to determine where the next section ends with only small overhead. Therefore, we use two static checks which detect branching transitions (in order to ensure condition (A)) and hiding dependencies (in order to ensure condition (B)).

A transition $R_{l,l'}$ is conservatively marked as branching whenever there exists another transition $R_{l,l''}$ with $l' \neq l''$. This classification corresponds to marking all transitions that model a branching statement as a branching transition, where a branching statement is a statement with multiple program order successors, e.g., a conditional jump, an if-then-else construct, or a loop. This over-approximates the set of all branching transitions (for example, a conditional jump with an unsatisfiable condition would still be classified as a branching transition).

We prepare the check whether two events may form a hiding dependency by a static dependency analysis. For each transition R, we calculate the set of program variables that can influence which variables are accessed by R. For each such variable, all transitions writing to the variable are marked as potentially influencing the set of variables accessed by R.

Constructing Mazurkiewicz Traces

Once the events and races of a section are known (e.g., by executing an arbitrary interleaving until the end of the current section), it is possible to calculate the Mazurkiewicz traces of all alternative executions of the section without calculating any further program states as follows. A Mazurkiewicz trace can be calculated by constructing a directed graph with events as nodes and an edge between two events e and e' whenever e should occur before e' in all representatives of the Mazurkiewicz trace. If the resulting graph is acyclic, it induces a partial order that directly corresponds to a Mazurkiewicz trace and any of its linear extensions is a representative of the Mazurkiewicz trace. Otherwise, the graph contains a cycle and there exists no execution that obeys the ordering of the graph.

For the example of Figure 3.2, we start by calculating a Mazurkiewicz trace of the section containing e_{00}, e_{01}, and e_{10}. We calculate the Mazurkiewicz trace where e_{01} occurs before e_{10} by defining the following graph:

$$e_{00} \xrightarrow{\text{po}} e_{01} \xrightarrow{\text{dep}} e_{10}$$

The edge (e_{00}, e_{01}) represents the program order of Thread 1 and the edge (e_{01}, e_{10}) represents the (only) race of the section. Because the graph is acyclic, there exists a linear extension of the induced partial order, $e_{00}e_{01}e_{10}$, and we found a Mazurkiewicz trace of the program. By swapping the direction of the edge (e_{01}, e_{10}), we obtain a graph for another Mazurkiewicz trace where the race $e_{01} \precsim_{e_{00}e_{01}e_{10}} e_{10}$ is reversed. We do not swap the edge (e_{00}, e_{01}) because it represents the program order, which is obeyed by all executions.

A linear extension of the induced partial order can be constructed in linear time w.r.t. the number of nodes by iteratively removing a minimal node (a node with no incoming edge) and all its outgoing edges [PR94]. If no minimal node is found, the graph is cyclic.

By calculating Mazurkiewicz traces as described, it is possible to construct representatives of all Mazurkiewicz traces "in advance", i.e., without performing any (typically expensive) program state computations.

3.2.2 Formal foundations of trace construction

This section formalizes the notions introduced in Section 3.2.1 and details how EPOR constructs Mazurkiewicz traces from a given event sequence.

Section 3.2.1 describes sections as program fragments and specifies two conditions (A) and (B) they have to satisfy in order to support our POR algorithm. We model a section as the set of event sequences that correspond to an execution of the program fragment of the section. We write *section*(ρ), where ρ is feasible at s_{init}, for the set of event sequences that are feasible at s_ρ and include exactly those events that model the statements of a section. Formally, *section*(ρ) includes all event sequences $\rho' = e_1 \dots e_k$ that are feasible at s_ρ and satisfy (where conditions (A) and (B) have been introduced informally in Section 3.2.1):

(A): For each branching event e in ρ', no event in program order with e follows e in ρ': $\forall 1 \leq i \leq k.\, \mathit{branching}(e_i) \Rightarrow \forall i < j \leq k.\, e_i \not<_{PO}^{\rho,\rho'} e_j$ and

(B): ρ' contains no hiding dependency: $\forall 1 \leq i \leq k.\, \forall i < j \leq k.\, \neg e_i \overset{*}{\rightarrow}_{\rho\cdot\rho'} e_j$ and

- maximality: There is no event e such that $\rho' \cdot e$ satisfies the above requirements.

For some *section*(ρ), a POR algorithm ideally explores only a subset *section-rep*$(\rho) \subseteq$ *section*(ρ) that contains exactly one representative of each Mazurkiewicz trace of the event sequences in *section*(ρ).

In order to formalize the generation of *section-rep*(ρ), we introduce *trace constraint systems.* Each satisfiable trace constraint system corresponds to a fragment of a Mazurkiewicz trace. The constraints of a trace constraint system in conjunction with the program order specify the fragment's partial order of events. By reversing those constraints, it is possible to reverse races and thereby generate all event sequences of *section-rep*(ρ) for some ρ.

Formally, a trace constraint system is a tuple $c = (A, C_{var}, C_{\mathit{fixed}})$ where

- $A = \{e_1, \ldots, e_k\}$ is a set of events.
- $C_{var} \in A \times A$ is a set of variable constraints of c.
- $C_{\mathit{fixed}} \in A \times A$ is a set of fixed constraints of c.

Whenever, for two event sequences $\rho \in \mathit{feasible}(\varepsilon)$, $\rho' = e_1 \ldots e_n \in \mathit{feasible}(\rho)$, we have

- $A = \{e_1, \ldots, e_n\}$,
- $C_{var} =<^{\rho,\rho'}$, and
- $C_{\mathit{fixed}} =<^{\rho,\rho'}_{PO}$,

we call c the *trace constraint system of* ρ' *at* s_ρ and write $c = \mathit{CS}(\rho, \rho')$.

Given a state s_ρ for some event sequence ρ, one can construct an event sequence ρ' from *section*(ρ) by starting with $\rho' = \varepsilon$ and iteratively adding events for enabled transitions at $s_{\rho\cdot\rho'}$ until adding another event would violate one of the conditions (A) and (B). All remaining event sequences of *section-rep*(ρ) can subsequently be constructed by the use of trace constraint systems as follows. First, the trace constraint system $\mathit{CS}(\rho, \rho')$ that corresponds to the trace of ρ' is constructed. Subsequently, all trace constraint systems which are equal to $\mathit{CS}(\rho, \rho')$ except for one or more reversed variable constraints are constructed. The set of these constraint systems is called *traces*(ρ) and defined as (given ρ' as described above)

$$\begin{aligned} \mathit{traces}(\rho) := \{(A, C_{var}, C_{\mathit{fixed}}) : A = \{e : e \in \rho'\} \\ \wedge\, C_{\mathit{fixed}} =<^{\rho,\rho'}_{PO} \\ \wedge\, \forall e_1, e_2 \in A.\, e_1 <^{\rho,\rho'} e_2 \Leftrightarrow (C_{var}(e_1, e_2) \vee C_{var}(e_2, e_1))\}. \end{aligned}$$

A solution ρ of a trace constraint system $c = (A, C_{var}, C_{\mathit{fixed}})$, written $\rho \in \mathit{solutions}(c)$, is an event sequence that (1) obeys the variable constraints in C_{var}, and (2) obeys the fixed constraints in C_{fixed}. Formally, we require for $\rho = e_1, \ldots, e_n$ that $\forall 1 \leq i \leq n.\, \forall i \leq j \leq n.\, \neg C_{var}(e_2, e_1) \wedge \neg C_{\mathit{fixed}}(e_2, e_1)$.

We call c *satisfiable* if a solution of c exists. A solution of a satisfiable trace constraint system c can be constructed in linear time with respect to the number of events that are contained in c. For example, create a linear extension of the partial order induced by the union of C_{var} and C_{fixed} in c. If this union contains cycles, c is not satisfiable, which is easily detected by a linear extension algorithm.

Using the notion of *traces*(ρ), one can construct *section-rep*(ρ) as a set that contains exactly one solution of each satisfiable trace constraint system in *traces*(ρ). As each trace constraint system in *traces*(ρ) is unique, only one representative of each trace of *section*(ρ) is constructed, enabling an optimal POR exploration. Correctness of section-based exploration is provided by the following theorem; given an event sequence $\rho' \in \mathit{section}(\rho)$, there exists a constraint system $c \in \mathit{traces}(\rho)$ whose solutions are equivalent to ρ'.

Theorem 1 (Correctness of section-based exploration)**.**
$\forall \rho \in \mathit{feasible}(\varepsilon).\, \forall \rho' \in \mathit{section}(\rho).\, \exists c \in \mathit{traces}(\rho).\, \forall \rho'' \in \mathit{solutions}(c).\, \rho'' \simeq \rho'$

Proof. Let $\rho \in \mathit{feasible}(\varepsilon), \rho', \rho'' \in \mathit{section}(\rho)$. Because of condition (A) in the definition of *section*(), ρ' and ρ'' correspond to the same transitions (correspond to the same control flow) (1). Because of condition (B) in the definition of *section*(), the same data dependencies appear in ρ' and ρ'' (2). Let *traces*(ρ) be calculated on the basis of $CS(\rho')$; by definition, all constraint systems in *traces*(ρ) contain exactly the events of ρ' and contain exactly one constraint for each dependency in $<^{\rho,\rho'}$. Additionally, there exists a constraint system in *traces*(ρ) for every ordering of races in ρ'. Hence, and because of (1) and (2), there exists some $c \in \mathit{traces}(\rho)$ whose constraints correspond to the ordering of races in ρ''. By the definition of *solutions*(), all event sequences $\rho''' \in \mathit{solutions}(c)$ are linear extensions of the partial order induced by the constraints of c and the program order for $\mathit{dom}(\rho')$. Hence, $\rho''' \simeq \rho''$. □

3.2.3 The *Eager POR* algorithm

This section introduces our dynamic POR algorithm EPOR. It is an extension of the SDPOR Algorithm [AAJS14]. Instead of exploring single events at each recursive call, EPOR creates schedules for sections of the program under analysis. If no schedule is currently present, EPOR creates new schedules for all event sequences in the section starting at the current state. If a schedule is present, it is used to guide the exploration. Checks for races inside a section are only performed once when schedules are created; checks for races between an event before the current section and an event inside the current section are still performed at every recursive call in order to ensure correctness.

As EPOR is based on SDPOR, we repeat basic definitions from SDPOR's pseudo code [AAJS14]. Let $\rho \in \mathit{feasible}(\varepsilon)$ be a feasible event sequence. The next event of a thread T at state s_ρ is denoted by $\mathit{next}_\rho(T)$ and $\rho \cdot T$ denotes $\rho \cdot \mathit{next}_\rho(T)$. For two threads T_1, T_2 with $e_1 = \mathit{next}_\rho(T_1), e_2 = \mathit{next}_\rho(T_2)$, we write $\rho \models T_1 \Diamond T_2$ to denote that e_1 and e_2 are independent af-

Algorithm 1: The EPOR algorithm

```
   Initially : Explore($\varepsilon$, $0$)
   Data: $sleep := \varnothing, backtrack := \varnothing, schedule := \lambda\rho.\,\varnothing$

1  Function Explore($\rho$, $sec\text{-}start$)
2      if $(enabled(\rho) \setminus sleep(\rho)) = \varnothing$ then
3          return
4      if $schedule(\rho) = \varnothing$ then
5          $sec\text{-}start := \mathrm{length}(\rho)$
6          Fill_Schedule($\rho$)
7      $Done := \varnothing$
8      while $\exists T \in (schedule(\rho) \setminus Done)$ do
9          Race_Detection($\rho$, $sec\text{-}start$, $T$)
10         $sleep(\rho) := \{T' \in sleep(\rho) : \rho \vDash T \Diamond T'\}$
11         Explore($\rho{\cdot}T$, $sec\text{-}start$)
12         add $T$ to $Done$
13         add $T$ to $sleep(\rho)$
14     while $\exists T \in (backtrack(\rho) \setminus sleep(\rho))$ do
15         $sec\text{-}start := \mathrm{length}(\rho)$
16         Race_Detection($\rho$, $sec\text{-}start$, $T$)
17         $sleep(\rho) := \{T' \in sleep(\rho) : \rho \vDash T \Diamond T'\}$
18         Explore($\rho{\cdot}T$, $sec\text{-}start$)
19         add $T$ to $sleep(\rho)$

20 Function Fill_Schedule($\rho$)
21     foreach $\rho_1 \in section\text{-}rep(\rho)$ do
22         foreach $prefix\ \rho_2 = e_1 \ldots e_n\ of\ \rho_1$ do
23             $T := tid(e_n)$
24             add $T$ to $schedule(\rho \cdot \rho_2)$
25             $sleep(\rho \cdot \rho_2) :=$
                       $\{T' \in sleep(\rho \cdot \rho_2) : \rho \vDash T \Diamond T'\}$

26 Function Race_Detection($\rho$, $sec\text{-}start$, $T$)
27     let $\rho'$ be the prefix of $\rho$ of length $sec\text{-}start$
28     foreach $e \in \rho'\ with\ e \precsim_{\rho\cdot T} next_\rho(T)$ do
29         $\rho_1 := pre(\rho, e)$
30         $\rho_2 := notdep(\rho, e) \cdot T$
31         if $I_{\rho_1}(\rho_2) \cap backtrack(\rho_1) = \varnothing$ then
32             add some $T' \in I_{\rho_1}(\rho_2)$ to $backtrack(\rho_1)$
```

ter ρ, i.e., $e_1 \not<^{\rho\cdot e_1\cdot e_2} e_2 \wedge e_1 \not<_{PO}^{\rho\cdot e_1\cdot e_2,} e_2$. Overloading the notation *enabled*(), we define $enabled(\rho) = \{T : Next\text{-}Transition(s_\rho, T) \neq \bot\}$. For $\rho' \in feasible(\rho)$, we define $T \in I_\rho(\rho') \Leftrightarrow \exists\rho''.\, \rho \cdot \rho' \simeq \rho \cdot T \cdot \rho''$. For event e in ρ, $pre(\rho, e)$ denotes the prefix of ρ up to but not including e and $notdep(\rho, e)$ denotes the subsequence of ρ that contains all events that occur after e in ρ but are not dependent with e in ρ.

The EPOR algorithm is shown as Algorithm 1. The main routine `Explore(`ρ, *sec-start*`)` takes as arguments an event sequence ρ that identifies the current state of the program and an integer *sec-start* that identifies the index in ρ at which the last section of ρ starts. The initial call is `Explore(`ε, 0`)` so that the exploration starts at the initial state. EPOR uses three global variables *sleep*, *backtrack*, and *schedule*, which map an event sequence to a set of threads. For some event sequence ρ feasible at the initial state, $sleep(\rho)$ corresponds to the sleep set at state s_ρ, $backtrack(\rho)$ holds threads whose events need to be explored at state s_ρ in order to reverse races between two events of different sections, and $schedule(\rho)$ holds threads which are scheduled at state s_ρ in order to explore a section.

At some call `Explore(`ρ, *sec-start*`)`, EPOR first checks whether a deadlock is reached or ρ is sleep set-blocked (line 2). Subsequently, if no schedule for the current state is present, the subroutine `Fill_Schedule()` calculates $section\text{-}rep(\rho)$ (as described in Section 3.2.2) and corresponding schedules (lines 4–6).

The loop in lines 8–13 explores any events of threads that are scheduled for the current state in order to explore a section. The subroutine `Race_Detection()` checks whether there are reversible races between an event before the start of the current section (as specified in variable *sec-start*) and an event inside the current section. This avoids race checks between two events that are both inside the current section. For every reversible race that is found, the reversed race is scheduled for later exploration just as in the SDPOR algorithm.

Finally, the loop in lines 14–19 explores any events of threads that have been scheduled for the current state in order to reverse a race. Before the race check, the marker for the start of the current section is updated so that all reversible races in the current event sequence are found.

Correctness

EPOR is correct in the sense that for every execution of a given program, it explores a representative of the corresponding Mazurkiewicz trace.

Theorem 2 (Correctness of EPOR)**.** *$\forall \rho \in feasible(\varepsilon).\, \forall \rho_1 \in feasible(\rho).\, \rho_1$ is maximal $\Rightarrow$ $\exists \rho_2.\, \rho_2 \simeq \rho_1 \wedge$* `Explore(`$\rho$, *length*$(\rho)$`)` *calls* `Explore(`$\rho \cdot \rho_2$, $\cdot$`)`*, i.e., ρ_2 is explored*

Proof. In this proof, we use the ordering $\propto$, where $\rho_1 \propto \rho_2$ if `Explore(`ρ_1, $\cdot$`)` returned before `Explore(`ρ_2, $\cdot$`)` (as in [AAJS14]).

We have to prove that for all $\rho \in feasible(\varepsilon)$ and all maximal $\rho_1 \in feasible(\rho)$, there exists $\rho_2 \simeq \rho_1$ such that `Explore(`ρ, *length*(ρ)`)` calls `Explore(`ρ_2, $\cdot$`)`. Proof by induction on ρ, ordered by $\propto$.

Base case: `Explore(`ρ, $\cdot$`)` does not recursively call `Explore(`$\cdot$, $\cdot$`)`. `Explore(`ρ, $\cdot$`)` returns either in line 2 or at the end of the function. In the former case, $enabled(\rho) \setminus sleep(\rho) = \varnothing$. By the correctness of SDPOR, no further event needs to be explored. In the latter case, $schedule(\rho) \setminus Done = \varnothing$. As no recursive call is performed, the loop body is never executed and `Fill_Schedule(`ρ`)` has been executed but has not added any threads to $schedule(\rho)$. Hence, $feasible(\rho) = \varnothing$.

Inductive step: induction hypothesis: $\forall \rho' \in feasible(\varepsilon).\, \rho' \propto \rho \Rightarrow \forall \rho'_1 \in feasible(\rho').$
ρ'_1 is maximal $\Rightarrow \exists \rho'_2.\, \rho'_2 \simeq \rho'_1 \wedge \rho'_2$ is explored

Proof by contradiction. Let *explored* be the set of all explored threads at ρ, i.e., $explored := \{T :$ `Explore(`$\rho \cdot T$, $\cdot$`)` has been called by `Explore(`ρ, $\cdot$`)`$\}$.

Assume that there exists a maximal $\rho_1 \in feasible(\rho)$ such that for all $T \in explored$ and for all $\rho_2 \in feasible(\rho \cdot next_\rho(T))$ that are explored, $\rho \cdot \rho_1 \not\simeq \rho \cdot T \cdot \rho_2$, i.e., no event sequence equivalent to $\rho \cdot \rho_1$ is explored. Then there exists a race $e_i \precsim_{\rho \cdot T \cdot \rho_2} e_j$ that distinguishes $\rho \cdot \rho_1$ and $\rho \cdot T \cdot \rho_2$ (both e_i and e_j do not occur in ρ). By the induction hypothesis, e_i cannot occur in ρ_2, hence, $e_i = next_\rho(T)$.

Case (1): e_i and e_j belong to different sections. Hence, a recursive call with $\rho \cdot T$ has been made by `Explore(`ρ, $\cdot$`)` and we have $\rho \cdot T \propto \rho$. When e_j was selected for exploration, the race detection in lines 9 or 16 must have checked e_i and e_j for a race, as they lie in different sections (`Race_Detection(`$\rho \cdot T \cdot \ldots$, *sec-start*, T'`)` has been called with $sec\text{-}start \geq length(\rho \cdot T)$ and $T' = tid(e_j)$). By the correctness of

SDPOR and as EPOR uses the same race detection in case (1), there exists $T'' \in \mathit{backtrack}(\rho) \setminus \mathit{sleep}(\rho)$ that reverses the race $e_i \precsim_{\rho \cdot T \cdot \rho_2} e_j$ by the end of the race detection. All threads in $\mathit{backtrack}(\rho) \setminus \mathit{sleep}(\rho)$ are explored in `Explore(`ρ`, ·)`. Contradiction.

Case (2): e_i and e_j belong to the same section $\mathit{section}(\rho')$ for some ρ'. By the definition of `Fill_Schedule()`, $\mathit{section\text{-}rep}(\rho')$ is explored. By Theorem 1, $\mathit{section\text{-}rep}(\rho')$ contains a representative of every trace in $\mathit{section}(\rho')$ and the race $e_i \precsim_{\rho \cdot T \cdot \rho_2} e_j$ is reversed. Contradiction. □

3.3 Implementation and evaluation

We implemented EPOR and SDPOR in the Python programming language and ran it on multiple benchmark programs that are written in a simple imperative programming language where threads communicate over shared memory. We used sequential consistency as a memory model, which corresponds to total program orders. Two events are data dependent if one of the events writes to a memory location the other event either reads from or writes to. All experiments were run on 8 Intel i7-4790 CPUs at 3.60GHz with 16 GB main memory. Software material for reproduction of these experiments is available [Met20].

We use the runtime and the number of dependency checks as main metrics for the comparison of EPOR and SDPOR. A dependency check determines whether two events are in the dynamic dependency relation of the current program and is often performed several times in order to determine whether two events constitute a reversible race. The complete results can be found in Appendix A. A missing runtime indicates that the corresponding algorithm did not terminate for the given benchmark configuration within 35000 seconds ($\sim$ 9.7 hours) or required more than 16 GB of memory.

In Table 3.1, we present results for four benchmarks which have previously been used to evaluate dynamic POR algorithms. The Readers-Writers, Indexer, and Last Zero benchmarks are used in [AAJS14] to evaluate SDPOR; the Shared Pointer benchmark is borrowed from [GFYS07]. The Readers-Writers (N) benchmark contains a single writer and $N - 1$ readers. The Indexer (N) benchmark consists of N threads that write to a shared hash table. It is the only benchmark presented here that contains hiding dependencies. The scheduling of an execution influences the control flow behavior. The parameter of the Indexer benchmark specifies the number of threads. The Last Zero (N) benchmark consists of $N - 1$ threads that update a shared array and an additional threads that reads the same array. Again, the scheduling of an execution influences the control flow behavior. The Shared Pointer (N) benchmark consists of two equal threads which execute a loop N times, followed by an update of the respective other's threads pointer.

In all four benchmarks, EPOR shows a speed-up over SDPOR for the highest parameter. The number

Table 3.1: Comparison of EPOR and SDPOR on four well-known benchmarks.

Benchmark	Algorithm	Time (s)	Traces	Dep. Checks	Speedup(%)
Readers-Writers (9)	SDPOR	0.668	256	60885	—
Readers-Writers (9)	EPOR	0.400	256	3204	40.1
Readers-Writers (20)	SDPOR	6874.472	524288	1570045995	—
Readers-Writers (20)	EPOR	2728.742	524288	17827145	60.3
Indexer (12)	SDPOR	0.413	8	27072	—
Indexer (12)	EPOR	0.284	8	19325	31.2
Indexer (16)	SDPOR	13060.033	32768	1345407904	—
Indexer (16)	EPOR	7998.984	32805	466384458	38.8
Last Zero (6)	SDPOR	0.911	96	66384	—
Last Zero (6)	EPOR	0.724	96	29570	20.5
Last Zero (16)	SDPOR	*not terminating within 35000 s / 16 GB*			
Last Zero (16)	EPOR	18408.671	262144	7232899654	—
Shared Pointer (50)	SDPOR	32.529	101	14074966	—
Shared Pointer (50)	EPOR	17.398	101	11459539	46.5
Shared Pointer (100)	SDPOR	238.968	201	192707828	—
Shared Pointer (100)	EPOR	170.762	201	154590222	28.5

```
Thread TID:
  x[(TID+1)%l] := x[TID]
```

Figure 3.3 (a) Ring

```
Thread TID:
  if x[TID] == 0 then
    x[(TID+1)%l] := 1
  if x[TID] == 0 then
    x[(TID+1)%l] := 1
```

Figure 3.3 (b) Branching

```
Thread TID:
  x[(TID+1)%l] := x[TID]
  x[(TID+1)%l] := x[TID]
```

Figure 3.3 (c) Ring Extended

Figure 3.3: Three artificial benchmarks (x is a global array of length l, a is a local variable. Each program statement is executed atomically.)

of dependency checks is always lower for EPOR than for SDPOR (except for Indexer (11), where no races occur), while the number of explored maximal event sequences is equal between EPOR and SDPOR for all configurations.

In order to investigate the performance of EPOR in special cases, we have designed two artificial benchmarks Ring and Branching, which are depicted in Figure 3.3b and 3.3a. They loosely resemble the communication of threads which communicate in a ring, for example as in a ring election protocol. Every line is executed atomically. The Branching benchmark consists of two branching statements and two assignments; whether the assignments are executed depends on the scheduling of a particular execution. In the Ring benchmark, each thread likewise communicates with its next thread, but without control flow branchings. The Ring benchmark is similar to the Readers-Writers benchmark, but shows a higher number of dependencies, as each thread is both reading and writing. Selected results for these two benchmarks are depicted in Table 3.2.

Table 3.2: Comparison of EPOR and SDPOR on two simple benchmarks.

Benchmark	Algorithm	Time (s)	Traces	Dep. Checks	Speedup(%)
Ring (17)	SDPOR	5984.174	131070	734642101	—
Ring (17)	EPOR	538.031	131070	2096753	91.0
Ring (19)	SDPOR	*not terminating within 35000 s / 16 GB*			
Ring (19)	EPOR	2884.695	524286	8653144	—
Branching (5)	SDPOR	1.180	311	145186	—
Branching (5)	EPOR	1.045	311	114640	11.4
Branching (11)	SDPOR	19068.490	318363	2200202598	—
Branching (11)	EPOR	8220.448	318978	1343673801	56.9

For the Ring and Branching benchmarks, EPOR requires considerably less dependency checks than SDPOR for all configurations. The number of explored traces is equal for EPOR and SDPOR except for the Branching benchmark with 9 to 11 threads. The speed-up of EPOR over SDPOR is very prominent for the Ring benchmark; SDPOR does not terminate for 19 threads. Equally significantly, EPOR requires several orders of magnitude less dependency checks than SDPOR. For the Branching benchmark, EPOR still shows a considerable speed-up over SDPOR, however, the saving in terms of dependency checks is lower than for the Ring benchmark.

Less Unsatisfiable Trace Constraint Systems

Interestingly, EPOR shows a much higher runtime overhead than SDPOR for a slightly changed Ring benchmark as depicted in Figure 3.3c (Ring Extended). Here, each thread repeats its assignment so that the program order is not empty as opposed to the Ring benchmark.

As will be detailed later, EPOR (in its original form) does not scale as well for this benchmark as for the benchmarks previously presented. We explain this by the fact that EPOR generates at most 2 unsatisfiable trace constraint systems for the previous benchmarks while the number of unsatisfiable trace constraint systems for the Ring Extended benchmark increases with the number of threads. These additional unsatisfiable constraint systems occur due to the dependency structure of the Ring Extended benchmark. Each thread consists of two transitions, which model its two assignments. Each of these transitions depends on both transitions of the previous thread and additionally on both transitions of the next thread. Consequently, when combining the constraints of a trace constraint system for the Ring Extended benchmark with the program order between the two transitions of each thread, a cycle occurs with considerably higher probability than it is the case for the Ring benchmark.

For program fragments with dense dependencies as in the Ring Extended benchmark, we propose an

Table 3.3: Comparison of EPOR, EPOR-SH (short sections), and SDPOR on the Ring Extended benchmark.

Benchmark	Algorithm	Time (s)	Traces	Dep. Checks	Unsat. TCS	Speedup(%)
Ring Extended (6)	SDPOR	70.729	38466	7537485	0	—
Ring Extended (6)	EPOR	3412.561	38466	144095	16738750	-4724.8
Ring Extended (6)	EPOR-SH	72.869	38466	6747840	126	-3.0
Ring Extended (8)	SDPOR	6552.194	1548546	806537903	0	—
Ring Extended (8)	EPOR		*not terminating within 35000 s / 16 GB*			
Ring Extended (8)	EPOR-SH	5061.882	1548546	720212287	510	22.7

alternative definition of sections in order to reduce the generation of unsatisfiable trace constraint systems. Specifically, sections are shortened so that no trace constraint systems are generated whose constraints show cycles due to a combination with the program order. We call these adapted sections *short sections*. Cycles due to the program order can be avoided by permitting only one dependent event per thread inside a single short section. Formally, we define short sections by adding the following constraint to the definition of sections given in Section 3.2.2 such that all event sequences $\rho' = e_1 \ldots e_k \in \mathit{section}(\rho)$ additionally satisfy $\forall e_i, e_j, e_m, e_n \in \rho.\, e_i <^{\rho,\rho'} e_j \wedge e_m <^{\rho,\rho'} e_n \wedge \mathit{tid}(e_i) = \mathit{tid}(e_m) \Rightarrow e_i = e_m$.

We have implemented the EPOR algorithm with short sections instead of sections, denoted by EPOR-SH, and compare it with EPOR and SDPOR on the Ring Extended benchmark. The observed numbers are shown in Table 3.3. For 6 threads, EPOR-SH still shows a considerable number of unsatisfiable constraint systems but reduces this number by more than 99% in comparison to EPOR with original sections. While EPOR is more than 47 times slower than SDPOR for 6 threads and does not terminate for 8 threads, EPOR-SH is only slightly slower than SDPOR for 6 threads and more than 22% faster than SDPOR for 8 threads. Hence, the overhead of generating the remaining unsatisfiable trace constraint systems is still small enough so that EPOR-SH outperforms SDPOR. Appendix A shows the performance of EPOR-SH on our remaining benchmarks.

In order to increase the robustness of EPOR, it is perceivable to dynamically adapt the section length to the dependency structure of the program. Additionally, we expect that the number of generated unsatisfiable trace constraint systems can be reduced by exploiting information about the infeasibility of a constraint system to prevent the generation of further trace constraint systems that contain the same cycle (with or without program order). Such optimizations would further improve the performance of EPOR and EPOR-SH.

Chapter 4

Generating Safe Scheduling Constraints by Iterative Model Checking

This chapter presents the concept and formal foundation of our approach to guarantee safe executions with only incomplete verification results. We investigate the necessary and desirable properties of scheduling constraints that can be used for iterative model checking and safe executions of concurrent programs. Furthermore, we introduce an iterative model checking approach that generates scheduling constraints from incomplete verification results. The enforcement of scheduling constraints is discussed in Chapter 5.

Under the premise that a complete verification of many realistic concurrent programs is infeasible, or at least too slow to be carried out before the deployment of a program, we are interested in how incomplete verification results can be used to safely execute a program at least under scheduling constraints. If we accomplish to translate an incomplete verification result to scheduling constraints which guide the scheduler so that unsafe states are avoided, we trade freedom of scheduling for a facilitated verification process. Clearly, arbitrary scheduling constraints are not suitable, as, for example, forcing a program into a deadlock may be safe but prevents further usage of the program.

Our framework is designed to make the amount of non-determinism and thereby the complexity of the verification task adjustable by using incomplete verification results and reducing non-determinism by dynamically constrained scheduling. In particular, instead of waiting for a complete verification,

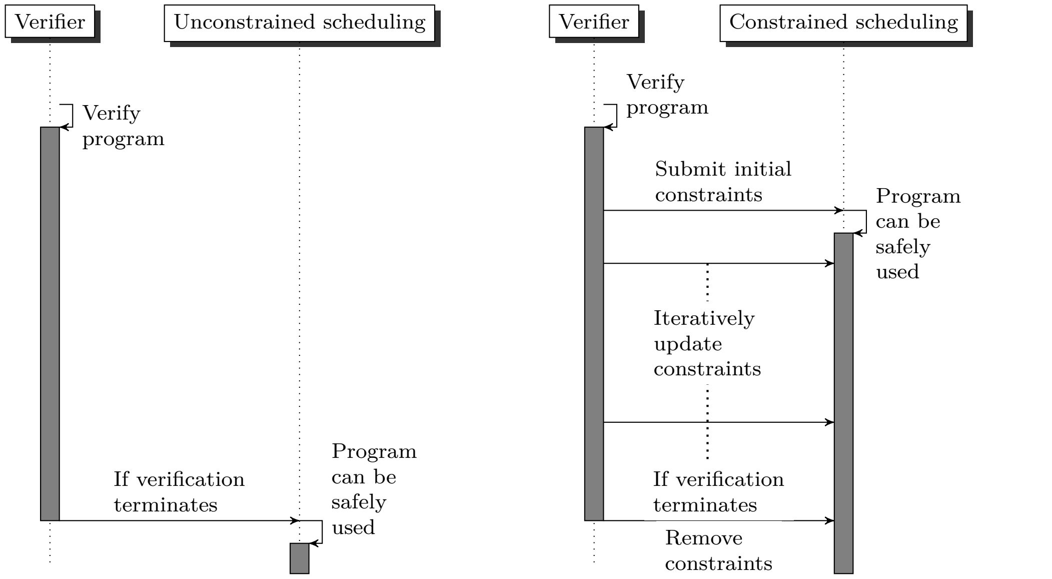
Verifier
Unconstrained scheduling
Verify program
If verification terminates
Program can be safely used
Verifier
Constrained scheduling
Verify program
Submit initial constraints
Program can be safely used
Iteratively update constraints
If verification terminates
Remove constraints

Figure 4.1 (a) Conventional

Figure 4.1 (b) With scheduling constraints

Figure 4.1: The program verification process (sequence diagram)

we propose to use incomplete verification results that guarantee program correctness under scheduling constraints.

The conventional verification process can be summarized as follows:

1. Develop or update a program.

2. Verify the program.

3. In case the verification is successful, the program can be safely used under a non-deterministic scheduler.

In case verification is successful, correctness is ensured for all feasible schedules of the program. This guaranty comes at the price of a typically large verification delay because of exponentially many schedules when unconstrained (non-deterministic) scheduling is used.

Instead of waiting until the program is completely verified, we propose to use a program under scheduling constraints induced by an incomplete verification result. Specifically, our approach proceeds as follows.

1. Develop or update a program.

2. Start the verification of the program.
3. As soon as a suitable incomplete verification result is available, the program can be safely used under the induced scheduling constraints.
4. If another suitable incomplete verification result becomes available, the scheduling constraints can be updated.
5. If the verification completes, the scheduling constraints can be removed. Otherwise, the program can still be safely used under the scheduling constraints.

An important requirement on incomplete verification results to be suitable for constrained scheduling is that all program inputs must be covered, i.e., that the extracted scheduling constraints permit to execute the program safely and without deadlock regardless of the current input. Meaningful intermediate verification results either show a counter example for program correctness or guarantee correctness under some feasible scheduling constraints. No additional constraints should be necessary such as constraints about program inputs or execution length, as a program may not be fully operational under such constraints. In particular, a correct schedule has to be known for each possible program input, even if inputs are given interactively (during a program execution). We formally investigate such requirements on incomplete verification results in Section 4.2.

The difference between conventional verification and IRS is illustrated in Figure 4.1. Conventionally, the usage of a program is delayed until after the complete verification, which may even be infeasible. Suitable incomplete verification results enable to reduce this delay at the expense of freedom in scheduling. In case a complete verification of a program is infeasible, scheduling constraints extracted from incomplete verification results enable a verification of the program in the first place. The enforcement of scheduling constraints presumably incurs an overhead in execution time. In this case, the combination of several incomplete verification results might be used to reduce this overhead. For example, the scheduling constraints of several incomplete verification results can be compared and the one with the least overhead can be used. If the scheduling constraints of several incomplete verification results overlap, the scheduling constraints can be relaxed, which may reduce the overhead. When the overhead of enforcing scheduling constraints can be reduced, it is possible to exploit the sweet spot between a short verification delay, where only few incomplete verification results are necessary, and a low overhead, for which many incomplete verification results might be necessary. In other words, our goal is to find *as much* incomplete verification results as necessary for an acceptable execution time performance and *no more* incomplete verification results than necessary in order to limit the verification delay.

Besides finding an execution that can be enforced with a small execution time overhead, aspects such as fairness may be important as well and scheduling constraints can be used to guarantee a safe and fair execution (if such an execution is found by the verifier), so that no thread starves.

Furthermore, an interesting feature of scheduling constraints from incomplete verification results is that an unsafe program can be safely used, as long as the verifier finds scheduling constraints that hide the defect. The defect may either be corrected or left unchanged with scheduling constraints ensuring that the defect does not manifest. In contrast, with conventional verification it is necessary to correct the program before it can be safely used and the verification process is required to be restarted and completed successfully.

Indeed, using incomplete verification results implies that the answer to the verification problem is not either *safe* or *unsafe* anymore but may also be *partially safe*.

Examples for applications of scheduling constraints extracted from incomplete verification results include:

1. A program that is infeasible to verify completely because of concurrency and the involved state space complexity can be safely used.

2. Concurrent programs can be used in safety-critical environments, where a successful verification is mandatory and prevents the use of arbitrary concurrent programs.

3. If a program update introduces a defect that occurs only under certain thread interleavings, the program can be safely used under scheduling constraints that hide the defect.

4. More generally than 3, IVRs can be used to safely execute unsafe programs which are safe under at least one scheduler. E.g., instead of programming synchronization explicitly, our model checking algorithm can be used to synthesize synchronization so that all executions are safe. Information on which synchronization is valid is specified via the property to be verified, e.g., by assertion statements in the program.

5. The verification process can be stopped after a given time budget is exhausted. The best scheduling constraints that are found until then are used.

6. Under a given budget of execution time performance (e.g., maximum execution time overhead or responsiveness), scheduling constraints of incomplete verification results can continuously be tested for their execution time performance. Once scheduling constraints are found under which the program can be executed fast enough, verification can be stopped.

```
initially:
  empty buffer of size N
  count = 0
  mutex = 0

thread $T_1$:
  while true:
    produce()

thread $T_2$:
  while true:
    consume()

produce:
  lock(mutex)
  if count < N:
    put item
    count += 1
  else:
    error (overflow)
  unlock(mutex)

consume:
  lock(mutex)
  if count > 0:
    remove item
    count −= 1
  else:
    error (underflow)
  unlock(mutex)
```

Figure 4.2: An erroneous version of the producer-consumer problem

7. In addition to 6, the verification can be continued after the program is used. When faster scheduling constraints are found, they can replace the current scheduling constraints under which the program is used. With a suitable implementation, it is not necessary to update the program itself.

Our iterative model checking approach provides safety verification of potentially non-terminating programs with a bounded number of threads, non-deterministic input, non-deterministic scheduling, and shared memory. Each iteration produces an *incomplete verification result* (IVR) to prove the safety of a program under a (semi-)deterministic scheduler. The scheduling constraints contained in an IVR allow to safely execute the program under analysis, as discussed in Chapter 5, even if the underlying operating system scheduler is non-deterministic.

We use the producer-consumer example from Figure 4.2 to explain our approach. The verifier analyses an initial schedule, e.g., where threads T_1 and T_2 produce and consume in turns, and emits an IVR $\mathcal{R}_1$, guaranteeing safe executions under this schedule. With its second IVR, the verifier might verify the correctness of producing two items in a row and the scheduling constraints can be relaxed accordingly. When the verifier hits an unsafe execution (the producer causes an overflow or the consumer causes an underflow), it emits an unsafe IVR for debugging. If the verifier accomplishes to analyze all possible executions of the program, it will report the final result *partially safe*, as the program can be used safely under all inputs but unsafe executions exist. Had there been no unsafe or safe IVR, the final result would be *safe* or *unsafe*, respectively.

This chapter shows how to instantiate our approach by answering the following questions:

1. Which state space abstractions are suitable for iterative model checking?

Algorithm 2: IMC and IRS for a program P

```
Data: $\mathcal{R}$ – the current IVR, initially $\tau \mapsto \varnothing$ (no execution is permitted)
$G$ – the current state of the verifier

Verifier:
    while not $finished_G$ (verification is not complete) do
        $\mathcal{R}' \leftarrow$ Model_Checking_Iteration($\mathcal{R}$)
        if $\mathcal{R}'$ contains an error path then
            yield error path for debugging
        if $\mathcal{R}'$ is suitable for IRS then
            update $\mathcal{R}$ based on $\mathcal{R}'$

Execution environment:
    set the current partial execution $\tau$ to the empty sequence
    while $P$ has not terminated do
        choose some thread $T$ from $\mathcal{R}(\tau)$
        execute the next event of $T$
        append $T$ and the current state of $P$ to $\tau$
```

The abstraction should be able to represent non-terminating executions and facilitate the extraction of schedules.

2. How to formalize and represent suitable IVRs?

   IVRs should be as small as possible in order to allow short iterations, while they must be large enough to guarantee fully functional executions under all possible program inputs. More precisely, for every possible program input, an IVR must cover a program execution.

3. What are suitable model checking algorithms that can be adapted to produce IVRs?

   A suitable algorithm should easily allow to select schedules for exploration.

4.1 Framework

We aim for a framework of *iterative model checking (IMC)* and *iteratively relaxed scheduling (IRS)* that allows to the previously described reduction of verification complexity and execution under reduced scheduling non-determinism. Algorithm 2 illustrates the composition of our framework. A *verifier* performs IMC and reports incomplete verification results (IVRs). The scheduling constraints contained in the current IVR are enforced by the *execution environment*, which performs IRS. In addition to reporting IVRs for admissible executions, the verifier may report error paths for debugging purposes. After an error path has been reported, the verification can be continued (even before the defect is repaired) as long as the verifier ensures that no error path is contained in the current IVR used by the execution environment. In order to prevent unnecessary assumptions on the verifier, we do not require the use of a specific data structure such as a state graph. Instead, we only require that the verifier maintains an internal state G that contains information on safe parts of the state space. We write $\mathit{finished}_G$ for the condition that verification is complete. If no error path is found and $\mathit{finished}_G$ holds, a program is safe without scheduling constraints. Later, we will also use the predicate $\mathit{safe}_G()$, defined such that for all states s, $\mathit{safe}_G(s)$ guarantees that the error location is not reachable from s $(\forall s, s' \in S.\, \mathit{safe}_G(s) \wedge R_G(s, s') \Rightarrow \mathfrak{l}(s') \neq \mathfrak{l}_{error})$. In other words, $\mathit{safe}_G(s)$ holds if the verification for the subset of executions that start at s is complete and no error has been found among these executions.

During execution of a program, the IRS execution environment maintains that the current partial execution τ adheres to the scheduling constraints represented by IVR $\mathcal{R}$. An IVR is a function which maps an execution prefix τ to a set of admissible threads.

In Algorithm 2, verifier and execution environment are executed concurrently. The execution environment can be executed several times during a single run of the verifier. For example, it must be possible to use the program, i.e., execute many steps of the execution environment after the verifier has completed a single iteration and is not able to produce another IVR.

The execution environment of Algorithm 2 produces an interleaving of an admissible execution. For an efficient enforcement of scheduling constraints with a low execution time overhead, it is possible to relax this strict interleaving. Please refer to Section 5.1 for a discussion and a solution to this issue.

Algorithm 2 in line 6 checks whether the latest produced $\mathcal{R}'$ is *suitable* for IRS. To define this suitability is a main concern of this chapter and discussed in Section 4.2. In broad terms, an IVR should either show an error path or permit only executions that are

- safe,
- deadlock-free, and
- in case of infinite executions, fair.

To permit only executions that are proven to be safe is our main goal. Therefore, only executions that are explicitly permitted may be executed. Deadlock-freedom, however, is important as well in order to ensure that a program can be fully used and IRS does not introduce new deadlocks, for example, because a particular input has not been considered during verification and the execution environment does not know how to safely continue the execution when this input occurs. Finally, fairness is important to make use of all threads and avoid the starvation of a thread. How to generate IVRs is discussed in Section 4.4.

If it is possible to reduce the execution time overhead by relaxing scheduling constraints or by finding new scheduling constraints that are faster to enforce, the overhead incurred by IRS can be adjusted: the more schedulings are verified, the less overhead will occur. In this case, the sweet spot between a short verification delay and a small execution time overhead can be found by continuously testing the

```
initially:
  empty buffer of size N
  count = 0
  mutex = 0
thread $T_1$:
  while true:
    produce()
thread $T_2$:
  while true:
    consume()
produce:
  lock(mutex)
  if count < N:
    put item
    count += 1
  unlock(mutex)
consume:
  lock(mutex)
  if count > 0:
    remove item
    count −= 1
  unlock(mutex)
```

Figure 4.3: Producer-consumer problem

execution time overhead with the current set of schedules found to be safe. As soon as the execution time overhead is small enough (i.e., a "sufficient amount of non-determinism" is used), the program can be used and verification can be stopped (i.e., no more than the "necessary amount of non-determinism" is used). The issue of which scheduling constraints can be enforced with a low overhead and how an execution environment can appropriately represent IVRs is discussed in Chapter 5.

4.2 Requirements on incomplete verification results

Our goal is to ease the verification task by producing incomplete verification results (IVRs) which prove the program safety under reduced non-determinism, i.e., only for a certain scheduler. We only allow "legitimate" restrictions of the scheduler that do not introduce deadlocks or exclude threads. Inputs must not be restricted, since this might reduce functionality and result in unhandled inputs.

Hence, we define an IVR to be a function $\mathcal{R}$ that maps execution prefixes to sets of threads, representing scheduling constraints. An IVR for the program from Figure 4.3, for instance, may output $\{T_1\}$ in states with an empty buffer, meaning that only thread T_1 may be scheduled here, and $\{T_2\}$ otherwise, so that an item is produced if and only if the buffer is empty.

Definition 19 (incomplete verification result)**.** *An* incomplete verification result *(IVR) for a program P is a function $\mathcal{R} : (S \times \mathscr{T})^* \times S \to \mathcal{P}(\mathscr{T})$ that maps execution prefixes to sets of threads.*

An IVR represents scheduling constraints. We write $\mathit{Schedulers}(P_{\mathcal{R}})$ for the set of schedulers that enforce these scheduling constraints: for all $\zeta_{\mathcal{R}} \in \mathit{Schedulers}(P_{\mathcal{R}})$ and for all execution prefixes $\tau = s_o, T_1, s_1, \ldots, s_n$, we have $\zeta_{\mathcal{R}}(\tau) \in \mathcal{R}(\tau)$, i.e., $\mathcal{R}(\tau)$ specifies a set of threads that are permitted to be scheduled after τ, according to the scheduling constraints.

A scheduler *enforces* (the scheduling constraints of) an IVR $\mathcal{R}$ if $\zeta_{\mathcal{R}}(\tau) \in \mathcal{R}(\tau)$ for all execution

$\mathcal{R}_{\mathscr{A}}$ is safe
$\Uparrow$
$\mathcal{R}_{\mathscr{A}}$ is realizable
$\Uparrow$
$\mathcal{R}_{\mathscr{A}}$ is deadlock-free
$\Uparrow$
$\mathcal{R}_{\mathscr{A}}$ admits fairness
$\Uparrow$
$\mathcal{R}_{\mathscr{A}}$ is fair

Figure 4.4: Properties of IVRs and their logical relation. $\Uparrow$ denotes logical implication.

prefixes τ. IVR $\mathcal{R}$ *permits* all executions possible under a scheduler that enforces $\mathcal{R}$ and we write $\mathit{Executions}(\mathcal{R}) := \bigcup_{\zeta_{\mathcal{R}} \in \mathit{Schedulers}(P_{\mathcal{R}})} \mathit{Executions}(P, \zeta_{\mathcal{R}})$.

In order to describe useful IVRs, we define *safe*, *realizable*, *deadlock-free*, *fairness-admitting*, and *fair* IVRs, where each property is implied by the following, cf. Figure 4.4.

Safety. An IVR $\mathcal{R}$ can either expose a defect in a program or guarantee that all permitted executions are safe. Here, we are mainly concerned with the latter case. An IVR $\mathcal{R}$ is *safe* if all executions permitted by $\mathcal{R}$ are safe.

Definition 20 (safe incomplete verification result)**.** *An IVR $\mathcal{R}$ is* safe *if all executions in Executions($\mathcal{R}$) are safe.*

An unsafe IVR permits an unsafe execution and is called a *counterexample*.

Completeness. To reduce the work for the model checker, a safe IVR $\mathcal{R}$ should ideally have to prove the correctness of as few executions as possible. At the same time, it should cover sufficiently many executions so that the program can be used without functional restrictions. For instance, the IVR $\mathcal{R}(\tau) := \emptyset$, for all τ, is safe but not useful, as it does not permit any execution. Consequently, $\mathcal{R}$ should permit at least one enabled transition, in all non-deadlock states, which is done by *realizable* IVRs.

Definition 21 (realizable incomplete verification result)**.** *A safe IVR is* realizable *if there exists a scheduler that enforces $\mathcal{R}$, i.e., $\mathit{Schedulers}(P_{\mathcal{R}}) \neq \emptyset$.*

Furthermore, an IVR should never introduce a deadlock.

Definition 22 (deadlock-free incomplete verification result)**.** *A realizable IVR $\mathcal{R}$ is* deadlock-free *if all schedulers which enforce $\mathcal{R}$ are deadlock-free, i.e., $\mathit{Schedulers}(P_{\mathcal{R}}) \neq \emptyset \wedge \forall \tau \in \mathit{Executions}(P_{\mathcal{R}}).\ \mathit{deadlock\text{-}free}(\tau)$.*

Fairness. In general, we deem only fair executions desirable. The IVR $\mathcal{R}(\tau) := \{T_1\}$, for instance, is deadlock-free for the program of Figure 4.3 but useless, as no item is consumed. If a fair execution of the program is possible under the constraints of a deadlock-free IVR, it *admits fairness.*

Definition 23 (incomplete verification result that admits fairness)**.** *A deadlock-free IVR $\mathcal{R}$* admits fairness *if there exists a fair scheduler $\zeta \in$ Schedulers$(P_{\mathcal{R}})$.*

If a scheduler permits both fair and unfair executions, it might be difficult to guarantee fairness at runtime. In such cases, a *fair* IVR can be used: A deadlock-free IVR $\mathcal{R}$ is *fair* if all schedulers enforcing $\mathcal{R}$ are fair.

Definition 24 (fair incomplete verification result)**.** *A deadlock-free IVR $\mathcal{R}$ is* fair *if all schedulers $\zeta \in$ Schedulers$(P_{\mathcal{R}})$ are fair.*

Equivalently, the requirements on $\mathcal{R}$ can be defined by the following game: there are two players, the scheduler player and the input player. Configurations are prefixes τ of executions. If τ is of the form $s_0, T_1, s_1, \ldots, T_n, s_n$, the scheduler player appends a thread T such that $\mathit{Next\text{-}Transition}(s_n, T) \neq \bot$. If τ is of the form $s_0, T_1, s_1, \ldots, T_n$, the input player appends a state s_n such that $R(s_{n-1}, s_n)$, where $R = \mathit{Next\text{-}Transition}(s_{n-1}, T)$. The scheduler player wins if an error-free terminal state that is not a deadlock is reached or if the resulting execution is infinite and fair. The input player wins if an error state or a deadlock is reached. $\mathcal{R}$ is fair if it corresponds to a winning strategy of the scheduler player.

4.3 Abstract reachability trees as incomplete verification results

In this section, we instantiate the notion of IVRs using abstract reachability trees (ARTs), which underlie a range of software model checking tools [HJMS02, McM06, KW11, BK11] and have recently been used for concurrent programs [WKO13]. We introduce criteria that identify ARTs which satisfy the requirements on useful IVRs. An overview of the properties of ARTs and their relation to properties of IVRs is given in Figure 4.5.

Due to the explicit representation of scheduling choices from the beginning of an execution up to an (abstract) state, ARTs are well-suited to represent IVRs. Model checking algorithms based on ARTs perform a path-wise exploration of program executions and represent the current state of the exploration using a tree in which each node v corresponds to a set of states at a global location $\mathfrak{l}(v)$. These states, represented by a predicate $\phi(v)$, (safely) over-approximate the states reachable via the program path

$$\begin{tikzcd}
\text{ART:} & & \text{IVR:} \\
\mathscr{A}\text{ is safe} \arrow[rr, Rightarrow] & & \mathcal{R}_{\mathscr{A}}\text{ is safe} \\
& & \mathcal{R}_{\mathscr{A}}\text{ is realizable} \arrow[u, Rightarrow] \\
\mathscr{A}\text{ is deadlock-free} \arrow[uu, Rightarrow] \arrow[rr, Rightarrow] & & \mathcal{R}_{\mathscr{A}}\text{ is deadlock-free} \arrow[u, Rightarrow] \\
\mathscr{A}\text{ admits fairness} \arrow[u, Rightarrow] \arrow[rr, Rightarrow] & & \mathcal{R}_{\mathscr{A}}\text{ admits fairness} \arrow[u, Rightarrow] \\
\mathscr{A}\text{ is fair} \arrow[u, Rightarrow] \arrow[rr, Rightarrow] & & \mathcal{R}_{\mathscr{A}}\text{ is fair} \arrow[u, Rightarrow]
\end{tikzcd}$$

Figure 4.5: Overview on the relationship between properties of IVRs and ARTs. $\Rightarrow$ and $\Uparrow$ denote logical implication.

from the root of the ART (ϵ) to v. Edges expanded at v correspond to transitions starting at $\mathfrak{l}(v)$. A node w may *cover* v (written $v \rhd w$) if the states at w include all states at v ($\phi(v) \Rightarrow \phi(w)$); in this case, v is covered ($\mathit{covered}(v)$) and its successors need not be further explored. (Intuitively, executions reaching v are continued from w.) Formally, an ART is defined as follows:

Definition 25 (abstract reachability tree [McM06, WKO13])**.** *An* abstract reachability tree *(ART) is a tuple $\mathscr{A} = (V, \epsilon, \rightarrow, \rhd)$, where $(V, \rightarrow)$ is a finite tree with root $\epsilon \in V$ and $\rhd \subseteq V \times V$ is a covering relation. Nodes v are labeled with global control locations and state formulas, written $\mathfrak{l}(v)$ and $\phi(v)$, respectively. Edges $(v, w) \in \rightarrow$ are labeled with a thread and a transition, written $v \xrightarrow{T,R} w$.*

For ease of notation, we do not distinguish between a transition and its transition formula. An ART is *well-labeled* if:

- $\phi(\epsilon)$ represents the initial state,
- for all states s, s' and for all edges $v \xrightarrow{T,R_{l,l'}} w$ in $\mathscr{A}$: $R_{l,l'} \in R_T \wedge (\phi(v)(s) \wedge R_{l,l'}(s, s')) \Rightarrow \phi(w)(s')$, and
- for every v, w with $v \rhd w$: $\phi(v) \Rightarrow \phi(w)$ and $\neg\mathit{covered}(w)$.

An incomplete ART $\mathscr{A}_{\text{p-c}}$ for the producer-consumer problem of Figure 4.2 is shown in Figure 4.6. Nodes show the state formulas and edges are labeled with the thread and statement corresponding to the transition. The dashed edge is a $\rhd$-edge.

ART-induced schedulers. A well-labeled ART $\mathscr{A}$ directly corresponds to an IVR $\mathcal{R}_{\mathscr{A}}$ that simulates an execution by traversing $\mathscr{A}$. Before we define $\mathcal{R}_{\mathscr{A}}$, we introduce a correspondence relation between executions and paths in $\mathscr{A}$.

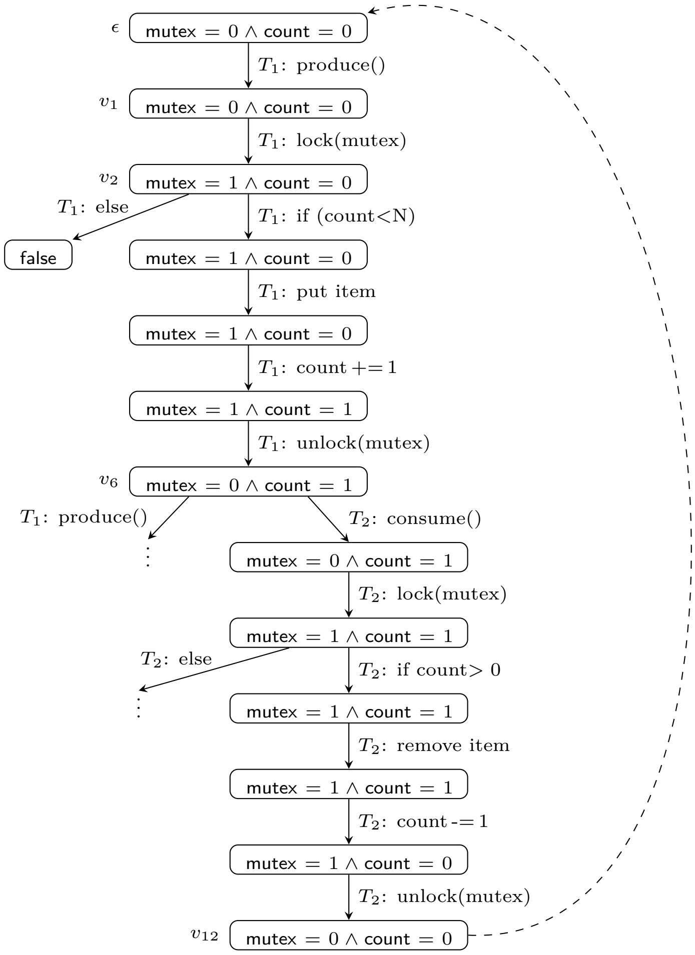

Figure 4.6: Extract of an ART for the program of Fig 4.2

When a path reaches a covered node, we continue the path at the covering node, so that we can match infinite executions. For a direct correspondence of an execution to such a path, we skip covered nodes. Formally, for a path π along $\rightarrow_{\mathscr{A}}$ and $\rhd$ edges, we write $\hat{\pi}$, for the unique path along $\rightarrow_{\mathscr{A}}$ edges that results from replacing all $v_1 \xrightarrow{T,R}_{\mathscr{A}} v_2 \rhd v_3$ edges in π by $v_1 \xrightarrow{T,R}_{\mathscr{A}} v_3$. NB, $\hat{\pi}$ is not necessarily a path in $\mathscr{A}$.

An execution $\tau = s_0, T_1, s_1, \ldots$ *corresponds* to a path π in $\mathscr{A}$, written $\tau \sim \pi$, if for $\hat{\pi} = v_0, T'_1, R_1, v_1, \ldots$ and for all $i > 0$: $v_0 = \epsilon$ and $T_i = T'_i$ and $s_i \vDash \phi(v_i)$ and $R_i(s_{i-1}, s_i)$. For example, the execution prefix

$$\tau = s_0,\ \underbrace{T_1, s_1, \ldots, T_1,}_{T_1 \text{ scheduled 6 times}}\ s_6,\ \underbrace{T_2, s_7, \ldots, T_2,}_{T_2 \text{ scheduled 6 times}}\ s_0$$

corresponds to the path in $\mathscr{A}_{\text{p-c}}$ from ϵ over $v_1, \ldots, v_{12}$ back to ϵ. As only T_1 is expanded at ϵ, $\mathcal{R}_{\mathscr{A}\text{p-c}}$ allows only $\{T_1\}$ after τ.

Based on this correspondence relation, we define $\mathcal{R}_{\mathscr{A}}$.

Definition 26 (incomplete verification result induced by an abstract reachability tree)**.** *The IVR $\mathcal{R}_{\mathscr{A}}$ represented by a well-labeled abstract reachability tree $\mathscr{A}$ is defined as follows. Let $\tau = s_0, T_1, s_1, \ldots, s_n$ be an execution prefix. If $\mathscr{A}$ contains no path that corresponds to τ, $\mathcal{R}_{\mathscr{A}}(\tau) := \mathscr{T}$ ($\mathcal{R}_{\mathscr{A}}$ leaves the schedules for this execution unconstrained). Otherwise, let $\pi = v_0, T'_1, R_1, v_1, \ldots, v_n$ be the path in $\mathscr{A}$ that corresponds to τ. $\mathcal{R}_{\mathscr{A}}(\tau) := \mathscr{T}' \subseteq \mathscr{T}$ such that $\mathscr{T}'$ is the set of threads that are expanded at v_n (in case v_n is covered by some node w, $\mathscr{T}'$ is the set of threads that are expanded at w).*

Safety. An ART $\mathscr{A}$ is *safe* if whenever $\mathfrak{l}_T(v)$, for a node v of $\mathscr{A}$, is the error location then $\phi(v) = \mathit{false}$. As only safe executions may correspond to a path in a safe ART (cf. Theorem 3.3 of [WKO13]), $\mathcal{R}_{\mathscr{A}}$ is a safe IVR.

Completeness. In order to derive a deadlock-free IVR from a well-labeled ART $\mathscr{A}$, we have to fully expand at least one thread T at each node v that represents reachable states (where T is fully expanded at v if v has an outgoing edge for every active transition of T at $\mathfrak{l}_T(v)$). However, there may exist reachable states s represented by $\phi(v)$ for which no transition of T is enabled (i.e., $\mathit{enabled}_T(s) = \emptyset$). If T is the only thread expanded at v, $\mathcal{R}_{\mathscr{A}}$ is not realizable. This situation can arise for locations l at which T may block (marked with $\mathit{may\text{-}block}(\mathfrak{l}_T)$).

Consequently, we introduce *deadlock-free* ARTs and require that whenever $\mathit{may\text{-}block}(\mathfrak{l}_T(v))$, $\phi(v)$ is strong enough to entail that the transitions R of T expanded at v (or at the node covering v, respectively) are enabled. For instance, $\phi(v_1)$ in Figure 4.6 proves the enabledness of

T_1 at v_1, as $\phi(v_1) \Rightarrow \text{mutex} = 0$ and $\mathsf{lock(mutex)}$ is enabled if $\text{mutex} = 0$.

Definition 27 (deadlock-free ART)**.** *A well-labeled, safe ART $\mathscr{A}$ is* deadlock-free *if each node v of $\mathscr{A}$ is either covered or one thread T is fully expanded at v and for all edges $w \xrightarrow{T,R}_{\mathscr{A}} w'$ in $\mathscr{A}$ where R may block, $\phi(w)$ can prove the feasibility of R, i.e.:*

$$\begin{aligned}&(\forall v \in V_{\mathscr{A}}.\, \mathit{covered}(v) \vee \exists T \in \mathscr{T}.\forall R \in \mathit{Transitions}(\mathfrak{l}_T(v)).\, \exists w \in V_{\mathscr{A}}.\, v \xrightarrow{T,R}_{\mathscr{A}} w)\\ &\wedge(\forall v, w \in V_{\mathscr{A}}.\forall T \in \mathscr{T}.\forall R \in \mathit{Transitions}(\mathfrak{l}_T(v)).\, ((v \xrightarrow{T,R}_{\mathscr{A}} w \wedge \mathit{may\text{-}block}(\mathfrak{l}_T))\\ &\quad\Rightarrow (\phi(v) \Rightarrow \mathit{Guard}(R))))\end{aligned}$$

The requirement $\phi(v) \Rightarrow \mathit{Guard}(R)$, i.e., to require that $\phi(v)$ can prove the feasibility of R, may seem strong and an ART that satisfies this constraint difficult to construct. To limit the associated cost, we require this feasibility check only for transitions that may block. Furthermore, in Appendix B.1, we argue that such an ART is easy to construct for “reasonable” programs.

Lemma 1 (Deadlock-free ARTs and IVRs)**.** *For all deadlock-free ARTs $\mathscr{A}$, $\mathcal{R}_{\mathscr{A}}$ is a deadlock-free verification result.*

Proof. Let $\mathcal{R}_{\mathscr{A}}$ be the IVR of a deadlock-free ART $\mathscr{A}$. First, we construct a scheduler that enforces $\mathcal{R}_{\mathscr{A}}$, which proves that $\mathcal{R}_{\mathscr{A}}$ is realizable. Second, we show that all schedulers that enforce $\mathcal{R}_{\mathscr{A}}$ are deadlock-free, which concludes the proof that $\mathcal{R}_{\mathscr{A}}$ is deadlock-free.

For arbitrary execution prefixes of the form $\tau = s_0, T_1, s_1, \ldots, s_n$, let $\mathscr{T}'(\tau) = \mathcal{R}_{\mathscr{A}}(\tau) \cap \{T \in \mathscr{T} : \mathit{Next\text{-}Transition}(s_n, T) \neq \bot\}$. Let $\zeta : (S \times \mathscr{T})^* \times S \to \mathscr{T}$ be an arbitrary function such that $\forall \tau.\, \zeta(\tau) \subseteq \mathscr{T}'(\tau)$ whenever $\mathscr{T}'(\tau)$ is not empty. (A description of how ζ can be constructed is given by the definition of $\mathcal{R}_{\mathscr{A}}$.) By construction, ζ enforces $\mathcal{R}_{\mathscr{A}}$ if ζ is a scheduler. We show that ζ is a scheduler by contradiction. Assume that ζ is not a scheduler. Then there exists an execution prefix $\tau = s_0, T_1, s_1, \ldots, s_n$ such that $\zeta(\tau) = T$, $\mathit{Next\text{-}Transition}(s_n, T) = \bot$ and $\mathit{enabled}(s_n) \neq \emptyset$.

case τ does not correspond to a path in $\mathscr{A}$: By the definition of $\mathcal{R}_{\mathscr{A}}$, $\mathcal{R}_{\mathscr{A}}(\tau) = \mathscr{T}$. By assumption $\mathit{enabled}(s_n) \neq \emptyset$, $\mathscr{T}'$ is not empty. By the construction of ζ, $T \in \mathscr{T}'$. Contradiction to $\mathit{Next\text{-}Transition}(s_n, T) = \bot$.

case τ corresponds to a path $\pi = v_0, T_1, R_1, v_1, \ldots, v_n$ in $\mathscr{A}$: By the construction of $\mathcal{R}_{\mathscr{A}}$, T is expanded at v_n.

case $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$: By the definition of *may block*, T has exactly one transition R active at $\mathfrak{l}_T(v_n)$. As $\mathscr{A}$ is deadlock-free, $\phi(v_n) \Rightarrow \mathit{Guard}(R)$. By assumption $\tau \sim \pi$, $s_n \vDash \phi(v_n)$. Hence,

$s_n \models \mathit{Guard}(R)$ and $R \in \mathit{enabled}(s_n)$. Contradiction to $\mathit{enabled}(s_n) = \emptyset$.

case not $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$**:** By the definition of *may block*, $\mathit{Next\text{-}Transition}(s_n, T) = R \neq \bot$ for some transition R. Contradiction to $\mathit{Next\text{-}Transition}(s_n, T) = \bot$.

It remains to show that all schedulers that enforce $\mathcal{R}_{\mathscr{A}}$ are deadlock-free. Let ζ be an arbitrary scheduler that enforces $\mathcal{R}_{\mathscr{A}}$. Assume that ζ is not deadlock-free. Then there exists an execution $\tau = s_0, T_1, s_1, \ldots, s_n \in \mathit{Executions}(P, \zeta)$ such that s_n is a deadlock, i.e., $\mathit{enabled}(s_n) = \emptyset \wedge \exists T \in \mathscr{T}.\, \mathit{Transitions}(\mathfrak{l}_T(s_n)) \neq \emptyset$. As $\tau \in \mathit{Executions}(\mathcal{R}_{\mathscr{A}})$, τ corresponds to a path $\pi = v_0, T_1, R_1, v_1, \ldots, v_n$ in $\mathscr{A}$. Let $T = \zeta(\tau)$. By choice of ζ, T is expanded at v_n. With the same argument as above, in case $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$, we have $\phi(v_n) \Rightarrow \mathit{Guard}(R)$ for some transition $R \in \mathit{Transitions}(\mathfrak{l}_T(v_n))$ and a contradiction to $\mathit{enabled}(s_n) = \emptyset$ and in case not $\mathit{may\text{-}block}(\mathfrak{l}_T(v_n))$, we have $\mathit{Next\text{-}Transition}(s_n, T) \neq \bot$ and a contradiction to $\mathit{Next\text{-}Transition}(s_n, T) = \bot$. □

Fairness. IVRs derived from deadlock-free ARTs do not necessarily admit fairness if the underlying ART contains cycles (across $\rhd$ and $\rightarrow$ edges) that represent unfair executions. In order to make sure a deadlock-free ART *admits fairness*, we implement a scheduler that allows $\mathscr{A}$ to schedule each thread infinitely often (whenever it is enabled infinitely often) by requiring that every $(\rhd \cup \rightarrow)$-cycle is “fair”, defined below. A $(\rhd \cup \rightarrow_{\mathscr{A}})$-cycle is a simple cycle in the graph $(V_{\mathscr{A}}, \rhd \cup \rightarrow_{\mathscr{A}})$, i.e., a finite sequence of nodes $v_1, \ldots, v_n$ such that:

- $v_1 = v_n$ and, if $n > 2$, $v_1 \neq v_2$
- $v_i \neq v_j$ for all $i, j \in \{2, \ldots, n-1\}, i \neq j$
- $v_i \rhd v_{i+1}$ or $v_i \rightarrow_{\mathscr{A}} v_{i+1}$ for all $i \in \{1, \ldots, n-1\}$

A deadlock-free ART *admits fairness* if every $(\rhd \cup \rightarrow)$-cycle contains, for every thread T that is active at a node of the cycle, a node v such that T is expanded at v.

Definition 28 (ART admitting fairness)**.** *Let $\mathscr{A} = (V_{\mathscr{A}}, \rhd, \epsilon, \rightarrow_{\mathscr{A}})$ be a deadlock-free ART. $\mathscr{A}$* admits fairness *if for every $(\rhd \cup \rightarrow_{\mathscr{A}})$-cycle c:*

$$\forall T \in \mathscr{T}.\, (\exists v' \in c.\, \mathit{Transitions}(\mathfrak{l}_T(v')) \neq \emptyset$$
$$\Rightarrow \exists v \in c.\, \exists w \in V_{\mathscr{A}}.\, \forall R \in \mathit{Transitions}(\mathfrak{l}_T(v)).\, v \xrightarrow{T,R}_{\mathscr{A}} w)$$

The following lemma shows that ARTs that admit fairness indeed fulfill the requirements of IVRs that admit fairness.

Lemma 2 (Fair ARTs and IVRs)**.** *For all ARTs $\mathscr{A}$ that admit fairness, $\mathcal{R}_{\mathscr{A}}$ is an incomplete verification result that admits fairness.*

Proof. We need to show that there exists a fair scheduler ζ that enforces an arbitrary ART $\mathscr{A}$ that admits fairness. After constructing ζ, we show that ζ is fair by contradiction.

Let $\tau = s_0, T_1, s_1, \ldots, s_n$ be an execution prefix and let π be a path such that τ corresponds to $\pi = v_0, T_1, \ldots, v_n$. By $\gamma(T)$, we denote the number of occurrences of T in π. Let $\mathscr{T}'$ be the set of threads that is both enabled at s_n and permitted by $\mathscr{A}$, i.e., $\mathscr{T}' = \mathcal{R}_{\mathscr{A}}(\tau) \cap \{T : \textit{Next-Transition}(s_n, T) \neq \bot\}$. We let ζ schedule an arbitrary thread $T \in \mathscr{T}'$ such that no other thread in $\mathscr{T}'$ occurs less often in π, i.e., $\zeta(\tau) = T \in \mathscr{T}'$ such that $\forall T' \in \mathscr{T}' . \gamma(T) \leq \gamma(T')$. By Lemma 1 and as $\mathscr{A}$ admits fairness, ζ is indeed a scheduler ($\mathscr{T}'$ is only empty when $\textit{enabled}(s_n)$ is empty).

It remains to show that ζ is fair, i.e., that every execution scheduled by ζ is fair. Let τ be an execution that is scheduled by ζ (τ is of the form $\tau = s_{init}, \zeta(s_{init}), s_1, \ldots$). If τ is finite, it is trivially fair. Otherwise, assume that τ is not fair. Then there exists a thread T that is infinitely often enabled in τ but does not occur in τ after some prefix of τ. Let π be a path in $\mathscr{A}$ such that τ corresponds to π. Let v_T be a node at which T is enabled and that occurs infinitely often in π. As $\mathscr{A}$ is finite and by Lemma 5 (p. 119), there exists a cycle that contains v_T such that π visits all nodes in this cycle infinitely often. As $\mathscr{A}$ admits fairness, there exists $v \xrightarrow{T,R}_{\mathscr{A}} v'$ such that v is in this cycle and $R \in \textit{enabled}(s)$ for all states s that correspond to v. As T is not scheduled in τ after some finite number i of steps, there exist one or more other threads $T' \neq T$ with $v \xrightarrow{T'}_{\mathscr{A}} w$ for some $w \neq v'$ which are scheduled at v for all steps $k > i$. Let t be the set of those threads T'. By the construction of the scheduler, $\gamma(T') \leq \gamma(T)$ for all $T' \in t$. After only finitely many steps l, $\gamma(T) < \gamma(T')$ for all $T' \in t$ (e.g., take l to be the product of the maximum path length from v to v and the number $\sum_{T' \in t} 1 + \gamma(T) - \gamma(T')$ of required visits of v). Hence, there exists a prefix of π of length $l' \geq l$ in which $v \xrightarrow{T}_{\mathscr{A}} v'$ is the last step, i.e., T has been scheduled. Contradiction to the assumption that T is not scheduled after i steps in π. □

Note that the expansion of a thread T at a node in a cycle of an ART that admits fairness does not guarantee that the transition is part of the cycle. A slight modification of the fairness condition for ARTs leads to a sufficient condition for ARTs as fair IVRs, as the following definition and lemma show. The difference in the fairness condition is that all enabled threads are expanded *within* each $(\rhd \cup \rightarrow)$-cycle c, which we denote by $\textit{fair}(c)$. The $(\rhd \cup \rightarrow)$-cycle shown in Figure 4.7, for instance, is fair.

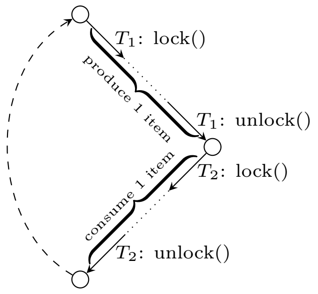

Figure 4.7: A fair cycle for the program of Fig 4.2

Definition 29 (fair ART). *Let $\mathscr{A} = (V_{\mathscr{A}}, \rhd, \epsilon, \rightarrow_{\mathscr{A}})$ be a deadlock-free ART. $\mathscr{A}$ is* fair *if for every $(\rhd \cup \rightarrow_{\mathscr{A}})$-cycle c, $fair(c)$, where:*

$$fair(c) \equiv \forall T \in \mathscr{T}. (\exists v' \in c.\, Transitions(\mathfrak{l}(v'))T \neq \emptyset$$
$$\Rightarrow \exists v \in c.\, \underline{\exists w \in c}. \forall R \in Transitions(\mathfrak{l}(v))T.\, v \xrightarrow{T,R}_{\mathscr{A}} w)$$

Note the difference between an ART that admits fairness and a fair ART (highlighted in the formula above): the successor node w of v that guarantees that a thread can be scheduled is required to be within the given cycle for fair ARTs.

Lemma 3 (Fair ARTs and IVRs). *For all fair ARTs $\mathscr{A}$, $\mathcal{R}_{\mathscr{A}}$ is a fair verification result.*

Proof. Let $\mathscr{A}$ be a fair ART. By Lemma 1 and as $\mathscr{A}$ is deadlock-free, there exists a scheduler ζ that enforces $\mathscr{A}$. It remains to show that ζ is fair, which we prove by contradiction. Suppose that an unfair execution τ is possible under ζ. There exists a thread T that is enabled infinitely often in τ but does not occur in τ after a finite prefix. Let π be a path through $\mathscr{A}$ such that τ corresponds to π. As $V_{\mathscr{A}}$ is finite, there exists a node v that occurs infinitely often in π and at which T is enabled. By Lemma 5 (p. 119), v is part of a cycle of which all nodes occur infinitely often in π. By fairness, one edge in this cycle is labeled with T. By the definition of ARTs ($(V_{\mathscr{A}}, \rightarrow_{\mathscr{A}})$ is a tree), this edge occurs infinitely often in π. Contradiction. □

Given an ART $\mathscr{A}$ that admits fairness, one can generate a fair ART $\mathscr{A}'$ such that $Executions(\mathcal{R}_{\mathscr{A}'}) \subseteq Executions(\mathcal{R}_{\mathscr{A}})$. An algorithm that generates $\mathscr{A}'$ is given as Algorithm 6 in Appendix B.3.

Algorithm 3: Iterative IMPACT for concurrent programs: main procedure (based on [WKO13])

```
input                   : Program with threads $\mathscr{T}$
intermediate outputs    : fair ARTs $\mathscr{A}_1 \subseteq \mathscr{A}_2 \subseteq \ldots \subseteq \mathscr{A}_n$ and unsafe ARTs
output                  : safe, partially safe, or unsafe

Data: $\mathscr{A} = (V, \epsilon, \rightarrow, \rhd) := (\{\epsilon\}, \epsilon, \emptyset, \emptyset)$, $W := \{\epsilon\}$, $I := \{\}$

Function Main()
    while true do
        status := Iteration()
        if $status = no\ progress$ then
            break
        else if $status = counterexample$ then
            yield $\mathscr{A}$ as an unsafe IVR
        else
            $\mathscr{A}'$ := Remove_Error_Paths($\mathscr{A}$)
            yield $\mathscr{A}'$ as a safe IVR
    if $\mathscr{A}$ is safe then
        return safe
    else if Remove_Error_Paths($\mathscr{A}$) admits fairness then
        return partially-safe
    else
        return unsafe

Function Iteration()
    $W$ := New_Schedule_Start()
    if $W = \emptyset$ then
        return no progress
    while $W \neq \emptyset$ do
        select and remove $v$ from $W$
        Close($v$)
        if $v$ not covered then
            status := Refine ($v$)
            if $status = counterexample$ then
                return counterexample
            status := Check_Enabledness($v$)
            if $status = no\ progress$ then
                return no progress
            Expand ($v$)
    return progress

Function Expand($v$)
    $T$ := Schedule_Thread ($v$)
    Expand_Thread ($T$, $v$)

Function Check_Enabledness($v$)
    $\pi := v_0 \xrightarrow{T_1, R_1} v_1 \ldots \xrightarrow{T_n, R_n} v_n$
            path from $\epsilon$ to $v$
    if not may-block($\mathfrak{l}(v_{n-1})$) T_n then
        return progress
    if $R_1 \wedge \ldots \wedge R_{n-1} \wedge \neg Guard(R_n)$ is unsat then
        $\phi(v) := \phi(v) \wedge Guard(R_n)$
    else
        return Backtrack($v$)

Function Close($v$)
    for all uncovered nodes $w$ that have been created
            before $v$ do
        if $\mathfrak{l}(w) = \mathfrak{l}(v) \wedge (\phi(v) \Rightarrow \phi(w))$
                $\wedge \forall c \in C_{\mathscr{A}}(v, w).\, fair(c)$ then
            $\rhd := \rhd \cup \{(v, w)\}$
            $\rhd := \rhd \setminus \{(x, y) : y \text{ is a descendant of } v\}$
            for $T$ with $v \xrightarrow{T} v'$ and not $w \xrightarrow{T} w'$ do
                add $(v, T)$ to $I$

Function Backtrack($v$)
    $\pi := v_0 \xrightarrow{T_1, R_1} v_1 \ldots \xrightarrow{T_n, R_n} v_n$ path from $\epsilon$ to $v$
    $i := n - 1$
    while $i \geq 0$ do
        if $\exists T, v_i'.\, v_i \xrightarrow{T} v_i' \notin \mathscr{A}$
                $\wedge$(Skip($v_i$, $T$) = false) then
            add $v_i \xrightarrow{T} v_i'$ to $\mathscr{A}$
            $W := W \cup \{v_i'\}$
            prune $\xrightarrow{T_{i+2}, R_{i+2}} v_{i+3} \cdots$
                $\ldots \xrightarrow{T_n, R_n} v_n$ from $\mathscr{A}$
            $\phi(v_{i+1}) := false$
            return progress
        $i := i - 1$
    return no progress
```

4.4 Iterative model checking

A suitable algorithm for our framework must generate fair IVRs. We use model checking based on ARTs (cf. Section 4.3), which allows us to check infinite executions and explicitly represent scheduling. Nevertheless, other program analysis techniques such as symbolic execution are also suitable to generate IVRs. Our algorithm (Algorithm 3) constitutes an iterative extension of the IMPACT algorithm [McM06] for concurrent programs [WKO13]. We choose IMPACT as a base for our algorithm because it has an available implementation for multi-threaded programs, which we use to evaluate our approach in Section 4.5.

IMPACT generates an ART by path-wise unwinding the transitions of a program. Once an error location is reached at a node v, IMPACT checks whether the path π from the ART's root to v corresponds to a feasible execution. If this is the case, a property violation is reported; otherwise, the node labeling is strengthened via interpolation. Thereby, a well-labeled ART is maintained. Once the ART is complete, its node labeling provides a safety proof for the program.

To build an ART as in the producer-consumer example of Figure 4.6, IMPACT starts by constructing the root node ϵ with $\phi(\epsilon) = \text{true}$ and $\mathfrak{l}(\epsilon) = (8,\ 12)$, where we indicate locations by line numbers in Figure 4.2. Initially, $\mathsf{mutex} = 0$, $\mathsf{count} = 0$, and the buffer size is bound by an arbitrary constant $\mathsf{N} > 0$. Thread T_1 is expanded by adding a node v_1 with $\phi(v_1) = \text{true}$ and $\mathfrak{l}(v_1) = (14,\ 12)$. From v_1, thread T_1 is expanded repeatedly until node v_6 with $\phi(v_6) = \text{true}$ and $\mathfrak{l}(v_6) = (8,\ 12)$ is produced. At this point, all statements of the $\mathsf{produce()}$ procedure have been expanded once. As v_6 has the same global location as ϵ and $\phi(v_6) \Rightarrow \phi(\epsilon)$, a covering $v_6 \rhd \epsilon$ can be inserted. However, when the else branch of thread T_1 at node v_1 is expanded, a node v_{error} labeled with the error location is added. In order to check the feasibility of the error path $\epsilon \to v_1 \to v_2 \to v_{\text{error}}$, IMPACT tries to find a sequence interpolant for:

$$\begin{aligned}
&\mathsf{count} = 0 \wedge \mathsf{mutex} = 0,\\
&\mathsf{mutex}' = 1,\\
&\mathsf{count} \geq \mathsf{N}
\end{aligned}$$

As we assume that the buffer is never of size 0, i.e., $\mathsf{N} > 0$, $\bigwedge \mathcal{U}$ is unsatisfiable and a possible sequence interpolant is:

$$\begin{aligned}
I_0 &\equiv \text{true}\\
I_1 &\equiv \mathsf{count} = 0 \wedge \mathsf{mutex} = 0\\
I_2 &\equiv \mathsf{count} = 0 \wedge \mathsf{mutex}' = 1\\
I_3 &\equiv \text{false}
\end{aligned}$$

with:

$$\begin{aligned}
&I_0 \wedge \mathsf{count} = 0 \wedge \mathsf{mutex} = 0 \Rightarrow I_1\\
&I_1 \wedge \mathsf{mutex}' = 1 \Rightarrow I_2\\
&I_2 \wedge \mathsf{count} \geq \mathsf{N} \Rightarrow I_3
\end{aligned}$$

Hence, v_{error} can be labeled with false, so that the ART remains safe, and the preceding labels can be updated to $\phi(\epsilon) = \phi(v_1) = \mathsf{count} = 0 \wedge \mathsf{mutex} = 0$ and $\phi(v_2) = \mathsf{count} = 0 \wedge \mathsf{mutex} = 1$. Due to the relabeling, the covering $v_6 \rhd \epsilon$ has to be removed and v_6 has to be expanded.

When T_2 has been expanded six times beginning at v_6, a node v_{12} is added with $\mathfrak{l}(v_{12}) = (8,\ 12)$. IMPACT applies a heuristic that attempts to introduce coverings eagerly, which results in a label $\phi(v_{12}) = \mathsf{mutex} = 0 \wedge \mathsf{count} = 0$ and a covering $v_{12} \rhd \epsilon$ can be added. With this covering, the current ART is fair and can be used as an IVR. In contrast, IMPACT for concurrent programs would then continue to explore additional interleavings by expanding, e.g., T_2 at ϵ. A complete ART is found when both error paths and all interleavings of produce() and consume() that respect the available buffer size N are explored. IMPACT for concurrent programs does not terminate until such a complete ART is found and would not terminate at all if the buffer size is unbounded. Our algorithm, however, is able to yield an fair IVR each time a new interleaving has been explored.

In each iteration, our extended algorithm yields an IVR which is either unsafe (a counterexample) or fair (can be used as scheduling constraints). If the algorithm terminates, it outputs “safe”, “partially safe”, or “unsafe”, depending on whether the program is safe under all, some, or no schedulers. Procedure *Main()* repeatedly calls *Iteration()* (line 3), which, intuitively, corresponds to an execution of the original algorithm of [WKO13] under a deterministic scheduler. *Iteration()* (potentially) extends the ART $\mathscr{A}$. If no progress is made ($\mathscr{A}$ is unchanged), the algorithm terminates and reports “safe”, “partially safe”, or “unsafe” (lines 12, 14, and 16). If *Iteration()* produces a counterexample $\mathscr{A}$, the ART is yielded as an intermediate output (line 7). Otherwise, *Iteration()* has found a new IVR, which is yielded as an intermediate output (line 10). This IVR corresponds to $\mathscr{A}$ with all previously found counterexamples removed, i.e., the largest fair ART that is a subgraph of $\mathscr{A}$, denoted by *Remove_Error_Paths()*.

Iteration() maintains a work list W of nodes v to be explored via *Close(v)*, which tries to find (as in [WKO13]) a node that covers v. In addition to the covering check of [WKO13], we check fairness, i.e., a covering is only added if no unfair cycle arises, where $C_{\mathscr{A}}(v, w)$ denotes all cycles that would be closed by adding the edge $v \rhd w$ (line 46). If such a node w is found, any thread T that is expanded at v but not at w (line 49) must not be skipped at w by POR. Instead of expanding T instantaneously at w (as in [WKO13]), which would result in the exploration of two schedules in the same iteration, T is added to the set I so that it can be explored in a subsequent iteration (for a different schedule). If no covering node for v is found, the same refinement procedure as in [WKO13], extended with a return value *counterexample* representing feasible error paths, is called (line 25). If the path to v is not a feasible error path (line 28 of Algorithm 3), *Check_Enabledness()* performs a deadlock check by testing whether the last transition that leads to v is enabled in all states represented by the predecessor node. If not, deadlock-freedom is not guaranteed and *Backtrack()* tries to find a substitute node where exploration can continue.

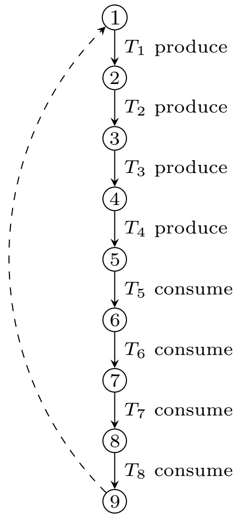

Figure 4.8: First IVR for the producer-consumer problem (simplified)

The deterministic scheduler of *Iteration()* is controlled by *New_Schedule_Start()* and *Schedule_-Thread()*. The former selects a set of initial nodes for the exploration (line 18 of Algorithm 3); the latter decides which thread to expand at a given node (line 34). We use a simple heuristic that selects the first (in breadth-first order) node which is not yet fully expanded and use a round-robin scheduler for *Schedule_Thread* that switches to the next thread once a back jump occurs (e.g., the end of a loop body is reached). Additionally, *Schedule_Thread* returns only threads that are necessary to expand at the given node after POR (cf. *Skip()* [WKO13]). More elaborate heuristics are conceivable but out of the scope of this thesis. An extended presentation of our algorithm is provided in Appendix B.4.

The correctness of Algorithm 3 w.r.t. safety follows from the correctness of [McM06] and [WKO13]. Additionally, Algorithm 3 is also fair:

Theorem 3 (Fairness of iterative Impact)**.** *Whenever Algorithm 3 yields a safe ART $\mathscr{A}$, $\mathscr{A}$ is fair.*

Proof. By contradiction. Assume that Algorithm 3 returns a safe ART $\mathscr{A} = (V_{\mathscr{A}}, \epsilon, \rightarrow_{\mathscr{A}}, \triangleright)$ that is not fair. By definition 29, $\mathscr{A}$ contains a $(\triangleright \cup \rightarrow_{\mathscr{A}})$-cycle c that does not satisfy $\mathit{fair}(c)$. As $(V_{\mathscr{A}}, \rightarrow_{\mathscr{A}})$ is a tree, the cycle contains a $\triangleright$ edge. However, Algorithm 3 checks, in line 46, whether the candidate covering would produce an unfair cycle. A $\triangleright$ edge is only added if the resulting cycle is fair. Contradiction. □

```
initially:
  empty buffer of size 1000
  count = 0
  mutex = 0

thread $T_{1\ldots4}$:
  while true:
    lock()
    if count != 1000:
      int return_value = produce()
      assert(return_value != OVERFLOW);
    unlock()

thread $T_{5\ldots8}$:
  while true:
    lock()
    if top > 0:
      return_value = consume();
      assert(return_value != UNDERFLOW);
    unlock()
```

Figure 4.9: A correct program for the producer-consumer problem with four producers and four consumers

4.5 Evaluation

In five case studies, we evaluate our iterative model checking algorithm and scheduling based on IVRs. We use the Impara model checker [WKO13], as it is the only available implementation of model checking for non-terminating, multi-threaded programs based on a forward analysis on ARTs we have found. Impara uses lazy abstraction with interpolants based on weakest preconditions. We extend the tool by implementing our algorithm presented in Section 4.4. Impara accepts C programs as inputs, however, some language features are not supported and we have rewritten programs accordingly.[1] We refer to the (non-iterative) Impara tool as Impara-C (for complete verification) and to our extension of Impara with iterative model checking as Impara-IMC. All experiments have been executed on a 4-core Intel Core i5-6500 CPU at 3.2 GHz.

4.5.1 Infeasible complete verification

Even for a moderate number of threads, complete verification, i.e., verification of a program under all possible schedules and inputs, may be infeasible. In particular, Impara-C times out (after 72 h) on a corrected variant of the producer-consumer problem (Figure 4.9) with four producers and four

[1] E.g., Pthreads mutexes, some uses of the address-of operator, and reuse of the same function by several threads are not supported. We solve these issues by rewriting our benchmark programs so that Impara handles them correctly and their semantics is not changed. Our modifications to Impara, including two bug fixes, are available in the software material published with this thesis [Met20].

```
Thread T1:
  while true:
    lock(mutex1)
    lock(mutex2)
    execute_critical_section()
    unlock(mutex2)
    unlock(mutex1)

Thread T2:
  while true:
    lock(mutex2)
    lock(mutex1)
    execute_critical_section()
    unlock(mutex2)
    unlock(mutex1)
```

Figure 4.10: A program with a deadlock

consumers. Impara-IMC produces the first IVR $\mathcal{R}_1$ after 4:29:53 hours. A simplification of $\mathcal{R}_1$ is depicted in Figure 4.8; it covers all executions in which the threads appear to execute their loop bodies atomically in the order $T_1, T_2, \ldots, T_8$. While the main bottleneck for Impara-C is state explosion and finding many coverings for different schedules, we observe that the main issue to produce $\mathcal{R}_1$ is to find a single covering that comprises all threads, i.e., to find a fair cycle. The essential predicates that lead to a fair cycle are:

$\mathsf{count} > 0$, $\mathsf{count} + 1 > 0$, $\mathsf{count} + 2 > 0$, $\mathsf{count} + 3 > 0$,
$\mathsf{count} \neq 1000$, $\mathsf{count} \neq 999$, $\mathsf{count} \neq 998$, $\mathsf{count} \neq 997$

The subsequent IVRs $\mathcal{R}_2, \ldots, \mathcal{R}_8$ are found much faster than the first IVR, after 19:31, 12:3, 6:13, 28:0, 9:25, 8:27, and 8:40 minutes. We stop the model checker after eight IVRs. According to our implementation of *New_Schedule_Start()* in Algorithm 3, IVR $\mathcal{R}_i$ permits, in addition to all executions permitted by $\mathcal{R}_{i-1}$, those executions in which the threads appear in the order $T_i, T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_8$. Hence, $\mathcal{R}_8$ gives the scheduler more freedom than $\mathcal{R}_1$, which may result in a better execution performance, e.g., because a producer which has its item available earlier does not have to wait for all previous producers.

4.5.2 Deadlocks

A common issue with multi-threaded programs are deadlocks, which may occur when multiple mutexes are acquired in a wrong order, as in the program in Figure 4.10, in which two threads use two mutexes to protect their critical sections. A deadlock is reached, e.g., when T_2 acquires mutex2 directly after T_1 has acquired mutex1. A monolithic verification approach would try to verify one or more executions and, as soon as a deadlock is found, report the execution that leads to the deadlock as a counterexample. With manual intervention, this counterexample can be inspected in order to identify and fix the bug.

In contrast, Impara-IMC logs both safe and unsafe IVRs. The first IVR found in this example covers all executions in which Threads 1 and 2 execute their loop bodies in turns, with Thread 1 beginning. The corresponding program schedule consists of a single section schedule depicted in Figure 4.11. As

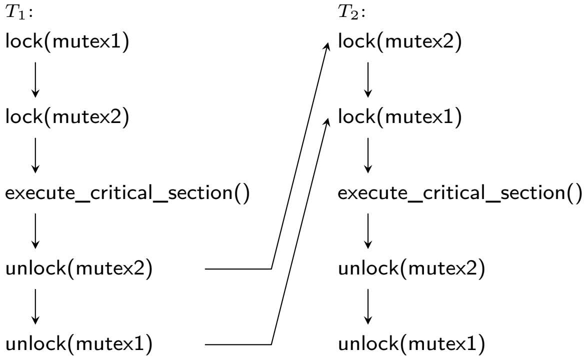

Figure 4.11: Section schedule for the program of Figure 4.10

```
Threads
  T1: while true: produce()
  T2: while true: produce()
  T3: while true: consume()
  T4: while true: consume()

produce:
  if buffer_is_not_full():
    lock()
    assert buffer_is_not_full()
    add_item()
    unlock()

consume:
  if buffer_is_not_empty():
    lock()
    assert buffer_is_not_empty()
    remove_item()
    unlock()
```

Figure 4.12: The producer-consumer problem with a race condition

expected, executing the program with enforcing the first program schedule never leads to a deadlock. Executing the uninstrumented program (without scheduling constraints) leads to a deadlock after only a few hundred loop iterations. Hence, IMC enables to safely use the program deadlock-free and without manual intervention.

4.5.3 Race conditions through erroneous synchronization

The program in Figure 4.12 shows a variant of the producer-consumer problem with two producers and two consumers which uses erroneous synchronization: both the produce and consume procedures check the amount of free space without acquiring the mutex first. For example, a buffer underflow occurs if the buffer contains only one item and the two consumers concurrently find that the buffer is not empty; although the buffer becomes empty after the first consumer has removed the last item, the second consumer tries to remove another item.

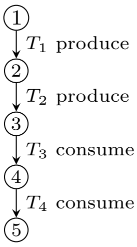

Figure 4.13: First IVR (simplified)

The first IVR found by Impara-IMC is depicted simplified in Figure 4.13. The simplification merges all individual edges of a procedure into a single edge, which is possible as Impara-IMC does not apply

```
Variables:
  int block
  boolean busy
  boolean inode
  mutex m_inode
  mutex m_busy
Initially: inode = busy

Thread T1:
  while true:
    lock(m_inode)
    if not inode:
      lock(m_busy)
      busy := true
      unlock(m_busy)
      inode := true
    block := 1
    unlock(m_inode)

Thread T2:
  while true:
    lock(m_busy)
    if not busy:
      block := 0
    unlock (m_busy)

Thread T3:
  while true:
    lock(m_inode)
    lock(m_busy)
    inode := false
    busy := false
    unlock(m_inode)
    unlock(m_busy)
```

Figure 4.14: The file system benchmark

```
Thread T1:
  while true:
    if not inode:
      busy := true
      inode := true
    atomic−begin
    assume inode and busy
    block := 1
    atomic−end

Thread T2:
  while true:
    if not busy:
      atomic−begin
      assume not busy
      block := 0
      atomic−end

Thread T3:
  while true:
    atomic−begin
    assume inode = busy
    inode := false
    busy := false
    atomic−end
```

Figure 4.15 (a) The file system benchmark with synchronization constraints in assume statements

```
Thread T2':
  while true:
    atomic−begin
    assume not busy
    block := 0
    atomic−end
```

Figure 4.15 (b) Thread T_2': the **if** statement is omitted

context switches inside of procedures during the first iteration. Since both procedures appear to be executed atomically, no assertion violation is found during the first iteration. We ran the program with a program schedule corresponding to the first IVR. As expected, we have not observed any assertion violations.

4.5.4 Declarative synchronization

Figure 4.14 shows an extension of a benchmark used in [FFQ02], which is a simplified extract of the multi-threaded Frangipani file system. The program uses a time-varying mutex: depending on the current value of the busy bit, a disk block is protected by m_busy or m_inode. We want to evaluate whether we can use Impara-IMC to generate safe program schedules even if all mutexes are (intentionally) removed from the program.

For this purpose, we use a variant of the file system benchmark where all mutexes are removed and synchronization constraints are declared as assume statements, shown in Figure 4.15a. It is sufficient to assure for T_1 that the block is written only if it is allocated, i.e., both inode and busy are true. For T_2, it is sufficient to assure that the block is only reset if it is not busy, i.e., busy = false. Finally, for T_3, it is necessary to assure that the block is deallocated only if it is already deallocated or fully allocated, i.e., inode = busy.

Table 4.1: Experimental results (rounded to full seconds, TO: timeout)

	Model checking	
Benchmark	Time 1st IVR	Impara-C
prod.-cons. 1p 1c 1000b	**2 m 0 s**	TO (72h)
prod.-cons. 2p 2c 1000b	**23 m 47 s**	TO (72h)
prod.-cons. 4p 4c 1000b	**4 h 29 m 53 s**	TO (72h)
prod.-cons. 1p 1c 5b	**2 s**	2 m 28 s
prod.-cons. 2p 2c 5b	**18 s**	1 m 16 s
prod.-cons. 4p 4c 5b	**2 m 41 s**	9 m 44 s
double lock 1 ms	0 s	0 s
file system	0 s	0 s
barrier	**1 s**	4 m 14 s

Running Impara-IMC on the file system benchmark without mutexes yields a first program schedule that schedules T_1, T_2, T_3 repeatedly in this order, according to our simple heuristic for an initial IVR. However, although all executions permitted by this schedule are fair, the if-condition of T_2 always evaluates to false and T_2 never performs useful work. To obtain a more useful schedule, we inform the model checker that the (omitted) else-branch of Thread T_2 is not useful. We encode this information by inserting **else**: assume false. After simplifying the code, we obtain T_2' as depicted in Figure 4.15b. For the updated code, Impara-IMC yields a first scheduler that schedules T_3 before T_2 before T_1, so that all threads perform useful work.

4.5.5 Verification time

We evaluate the verification time by running Impara-IMC and Impara-C on four correct benchmark programs. For Impara-IMC, we report the time necessary to generate the first IVR. We use a timeout of 72 hours. As Table 4.1 shows, Impara-IMC finds the first IVR often much faster than or at least as fast as it takes Impara-C for complete model checking; it can produce an IVR even for our largest benchmarks, where Impara-C times out. For a buffer size of 5, Impara-C can verify the producer-consumer benchmark even with eight threads but again, Impara-IMC is considerably faster in finding the first IVR. Subsequent IVRs were generated considerably faster than the first IVR, which might be caused by caching of facts in the model checker.

Chapter 5

Safe Execution of Multi-Threaded Programs by Enforcement of Scheduling Constraints

After Chapter 4 introduced the concept of iterative model checking and iteratively-relaxed scheduling (IRS), and investigated how to generate suitable incomplete verification results (IVRs), i.e., the verifier part of Algorithm 2, this chapter investigates the execution environment part, i.e., how to represent and enforce scheduling constraints.

Initial experiments have shown that constraining scheduling may introduce considerable execution time overhead. We reduce this overhead

- directly, by optimizing the enforcement so that a program is executed faster than under a naive enforcement of the same scheduling constraints
- indirectly, by investigating how scheduling constraints can be relaxed (which would require to verify additional states) to make their enforcement faster.

An IRS execution environment may be realized inside an application program or by modifying the operating system. For example, in the former case, the program may be instrumented so that a thread waits before memory accesses that are not yet permitted to occur, according to the scheduling constraints. Even if the scheduler of the operating system is non-deterministic, the scheduling constraints are enforced. In the latter case, it is conceivable to directly constraint the scheduler of the operating system to obtain an IRS execution environment and enforce schedules.

As in the related field of DMT, synchronization in addition to existing synchronization in a program (e.g., mutexes, condition variables, barriers) is necessary to enforce scheduling constraints. Our experiments confirm that constraining scheduling may introduce a considerable execution time overhead, in extreme cases a 44-fold slowdown. A main concern for the practicality of IRS is to limit this overhead depending on the requirements of a use case. We try to design IRS with a low overhead by addressing several aspects: the amount of additional synchronization for schedule enforcement, storage and look-up of scheduling constraints, and the effect of relaxing constraints on the execution time overhead. Optimizations of schedule enforcement in these aspects provide potential to execute a program even faster than with unconstrained scheduling and conventional synchronization, as our experiments show as well.

When implementing IRS, it is desirable to efficiently maintain and enforce scheduling constraints in order to incur as little overhead as feasible over conventional program execution. We are aiming for an suitable data structure to store, look up, and enforce scheduling constraints, that is, a *schedule*. We try to optimize schedules with respect to:

- Low space requirement
- Fast look up
- Few and fast synchronization between threads for the enforcement of scheduling constraints

At the same time, the following issues should be considered:

- It should be possible to update scheduling constraints, e.g., to relax constraints after additional states have been verified.
- Infinite executions and non-deterministic inputs should be supported.
- Where possible POR should be applied.

Several possibilities for storing scheduling constraints are possible: a set of schedules can be maintained where each schedule describes a permissible execution; in order to avoid storing redundant schedules, POR may be used to store one schedule for each permissible Mazurkiewicz equivalence class. However, a suitable formulation of POR that supports non-deterministic inputs has to be found. Furthermore, it may be advantageous to store all scheduling constraints in a single data structure. Nevertheless, care has to be taken that the overall size of scheduling constraints is feasible, as even after POR, the number of schedules may be exponentially large. A particular challenge is to represent schedules for infinite executions, especially when using POR, as all applications of POR are, to the best of our knowledge, designed for finite executions.

In order to permit a quick look-up of which events may be scheduled next during an execution, a set of unordered schedules seems inefficient. An ordering based on common prefixes seems advantageous, potentially merging all schedules into a single schedule valid for multiple Mazurkiewicz equivalence classes.

Besides reducing the space requirement of scheduling constraints, POR may also help to avoid superfluous synchronization when enforcing a schedule. For example, constraints can be stored in a vector clock [Mat89]. Finally, the implementation of synchronization has a large influence on the execution time performance of a program under schedule enforcement. For example, busy-waiting is typically much faster than lock-based synchronization, however only as long as the number of simultaneously waiting threads does not exceed the number of available hardware cores.

We proceed stepwise to design an enforcement scheme for scheduling constraints: Section 5.1 discusses schedule enforcement of finite executions or execution fragments. Section 5.2 extends this approach to infinite executions.

5.1 Finite executions

5.1.1 Symbolic traces for terminating executions

The general IRS algorithm from Section 4.1 (Algorithm 2) maintains a set of admissible traces and controls the scheduling of a given program such that at any time, the current partial execution adheres to some admissible trace. As more and more schedules or symbolic traces are proven to be correct, they are added to the set of admissible traces. This representation of scheduling constraints has an exponential space requirement and it seems impractical to store all symbolic traces for large programs. Similarly, when permission for an event is checked, the look-up time is exponential if no further structure is given to the set of admissible traces. *Unfoldings* have been applied for model checking both Petri nets [McM92] and concurrent programs [KSH12, RSSK15, SRDK17]. By unfoldings, it is possible to represent all executions of a concurrent program in a single data structure, which is more space-efficient than storing a set of all symbolic traces since each event occurs only once in an unfolding. Looking up an event in an unfolding is faster than searching in an unstructured set of symbolic traces, as well. However, the size of an unfolding can still grow quickly (exponentially in the worst case) with an increasing number of threads [KSH15]. The space efficiency of verification based on a depth-first search is lost. Hence, unfoldings are not directly suitable to store scheduling constraints for practical programs. In order to implement IRS, we address the problem of space complexity by using *trace prefixes*. If all admissible Mazurkiewicz traces or executions are stored in order to express scheduling constraints, so that each time a new execution has been verified and is permitted, more space is required. In contrast, trace prefixes can be used as scheduling constraints such that when new executions are permitted, constraints may be removed and *less* space is required. However, the use of trace prefixes requires the verifier to explore symbolic traces in a depth-first manner. More freedom can be given to the verifier by extending trace prefixes to partial unfoldings, at the price of a higher space requirement.

Our tests of several IRS implementations confirmed that as expected, inter-thread synchronization incurs a major part of execution time overhead of IRS over unconstrained scheduling. In order to reduce the execution time overhead caused by synchronization between threads, it is crucial to omit such synchronization in case an event needs not to be scheduled after an event from an other thread. The IRS algorithm presented in this section achieves this by executing several events without intermediate synchronization, as is detailed below. Besides reducing the amount of inter-thread synchronization, execution time overhead can be considerably reduced by reducing the duration of a single synchronization, for example by using lock-free synchronization instead of locks. We discuss this matter in Section 5.1.5.

In the following, we state our system model and present the IRS algorithm, proving correctness and deadlock-freedom of the algorithm.

5.1.2 Generalizing Mazurkiewicz equivalence

Given a dependency relation on events, Mazurkiewicz equivalence [Maz86, Maz95] guarantees that all equivalent executions reach the same final state and visit the same intermediate local states [God96]. However, a Mazurkiewicz equivalence class, or *Mazurkiewicz trace*, is not directly suitable to encode a schedule for IRS: two executions that follow different control flow paths or receive different (non-deterministic) inputs could be non-equivalent, while we aim for a representation that describes all executions permitted by an IVR (cf. Section 4.2). At the same time, we would like to use POR, hence Mazurkiewicz equivalence, to avoid unnecessary synchronization between events. Our solution is a generalization of Mazurkiewicz traces, called *symbolic traces*, defined below.

The happens-before relation of one or more executions is represented by a *symbolic trace graph* as a triple $o = (E_o, C_o, \rightarrow_o)$ such that

- E_o is a set of events,
- $C_o \subseteq \mathscr{F}(Q)$ is a set of state predicates, and

- $\rightarrow_o \subseteq E_o \times C_o \times E_o$ is a partial order labeled with path constraints, which expresses a happens-before relation.

As an auxiliary function, we introduce *remove*(e, o), which removes event e from symbolic trace graph o. Formally, *remove*$(e, (E_o, C_o, \rightarrow_o)) = (E'_o, C'_o, \rightarrow'_o)$ such that

- $E'_o = E_o \setminus e$,
- $\rightarrow'_o = \{(e_1, c, e_2) \in \rightarrow_o : e_1 \neq e \wedge e_2 \neq e\}$, and
- $C'_o = \{c : (\_, c, \_) \in \rightarrow'_o\}$.

A finite execution $\tau = s_0 T_1 s_1 \ldots T_n s_n$ with event sequence $\rho = e_1 \ldots e_n$ adheres to the happens-before relation of a symbolic trace graph $o = (E_o, C_o, \rightarrow_o)$, written $\tau \preccurlyeq o$, if ρ is empty, or

- $e_1 \in E_o$,
- $\forall (e, c, e') \in \rightarrow_o . \, e' = e_1 \Rightarrow s_0 \nvDash c$ and
- $(s_1 T_2 s_2 \ldots s_n) \preccurlyeq$ *remove*(e_1, o).

If ρ additionally contains exactly the events of E_o ($E_o = \{e : e \in \rho\}$), we write $\tau \approx o$. Execution τ is called a *linearization* of o. A symbolic trace graph o *corresponds* to (the scheduling constraints of) an IVR $\mathcal{R}$ (cf. Section 4.2), if the linearizations of o are exactly the executions permitted by $\mathcal{R}$, i.e., *Executions*$(\mathcal{R}) = \{\tau \in$ *Executions*$(P) : \tau \approx o\}$.

Based on symbolic trace graphs and their correspondence to happens-before relations of executions, we define *symbolic traces*, as a generalization of Mazurkiewicz traces. Intuitively, a symbolic trace contains scheduling information for all possible program inputs and represents all executions of a program with matching scheduling.

Definition 30. *A* symbolic trace *is a symbolic trace graph o that corresponds to some deadlock-free IVR.*

A *trace prefix* is a symbolic trace, except that we do not require its linearizations to be complete executions. Formally, a symbolic trace graph $o_1 = (E_1, C_1, \rightarrow_1)$ is a trace prefix of a symbolic trace $o_2 = (E_2, C_2, \rightarrow_2)$, written $o_1 < o_2$, if $E_1 \subsetneq E_2 \wedge \rightarrow_1 = \{(e_1, c, e_2) \in \rightarrow_2 : e_1, e_2 \in E_1\} \wedge \forall e_1 \rightarrow_2 e_2 . \, e_1 \notin E_1 \Rightarrow e_2 \notin E_1$.

Example 1. *A symbolic trace for the program of Figure 5.1, is given in Figure 5.2. The program consists of two threads, T_1 and T_2. Each thread is given a pointer as input and increments the value at*

```
1 T1:
2     input: int *x
3     local: int a
4     a := *x
5     *x := a + 1
6     assert *x == a + 1

7 T2:
8     input: int *y
9     local: int b
10    b := *y
11    *y := b + 1
```

Figure 5.1: Example program.

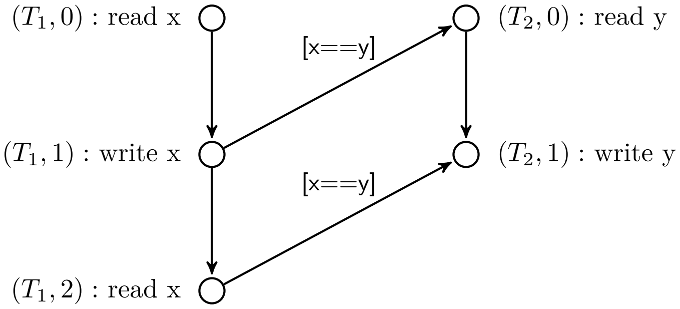

Figure 5.2: Example symbolic trace for the program of Figure 5.1. In all executions that adhere to this symbolic trace, the assertion in line 6 of thread T_1 is not violated. (Transitive edges are omitted.)

the pointer's target, mistakenly without synchronization. Thread T_1 asserts that the target of x *indeed holds the intended value. In case the pointers* x *and* y *point to different memory locations, the threads do not interfere with each other and the assertion holds. Otherwise, dependent accesses occur and the assertion does not hold under every possible ordering of events. The symbolic trace in Figure 5.2 ensures that the assertion holds in all executions that adhere to the symbolic trace. Nodes correspond to events and are labeled with the corresponding memory access for clarity. Edges between events of the same thread represent the thread's program order; edges between events of different threads represent scheduling constraints. Since dependencies between T_1 and T_2 occur only if the pointer targets match,* x==y*, the scheduling constraints are labeled with this condition.*

Whether an execution $\tau = s_0 T_1 s_1 \ldots T_n s_n$ adheres to the scheduling constraints of a symbolic trace graph, $\tau \preccurlyeq o$, can be checked as follows. Let $\rho = e_1 \ldots e_n$ be the event sequence of τ. If ρ is empty, $\tau \preccurlyeq o$ holds. Intuitively, that ρ is empty means that it does not contain any events that can violate any constraint given by o. If ρ is not empty, check whether e_1 has an incoming edge in o whose state predicate is satisfied by s_0. If this is the case, $\tau \preccurlyeq o$ is not satisfied. Otherwise, e_1 can be safely executed. Recursively check whether $s_1 \ldots T_n s_n \preccurlyeq \mathit{remove}(e_1, o)$.

Algorithm 4: IRS with trace prefixes and execution of sequences without synchronization

```
Data: $o_{adm}$ – the current admissible trace prefix

Initialization:
    initialize $o_{adm}$ to an arbitrary, deadlock-free admissible trace prefix for program $P$
    initialize internal verification status $G$ such that $safe_G(o_{adm})$

Verifier:
    while not $finished_G$ do
        do next verification step and update $G$
        if $\exists o' < o_{adm}.\ deadlock\text{-}free_G(o')$ then
            $o_{adm} \leftarrow o'$

Execution environment:
    set the current execution $\tau$ to the empty sequence
    while $P$ has not terminated do
        choose some sequence $\tau'$ from $free(\tau, o_{adm})$
        execute $\tau'$
        append $\tau'$ to $\tau$
```

5.1.3 Algorithm

The general IRS algorithm from Section 4.1 requires a synchronization between individual threads and the IRS execution environment after each event in order to check compliance of the current execution with a previously verified trace. Additionally, it stores all current admissible traces explicitly, which increases space requirements and look-up times as the verification advances. With Algorithm 4, we present an IRS algorithm that can be efficiently implemented. It addresses both previously described issues by the use of trace prefixes as scheduling constraints and allowing threads to run uninterrupted for multiple memory events whenever scheduling constraints do not require synchronization.

In order to simplify the presentation, it is assumed that the IRS execution environment enforces sequential consistency independently from scheduling constraints. For platforms where this incurs a considerable slowdown, scheduling of events of the same thread can be relaxed by considering intra-thread scheduling constraints.

Do not synchronize already reversed races. By using trace prefixes as scheduling constraints, it is possible to avoid synchronization before events when every possible continuation of the current execution is proven to be error-free. The corresponding part in an admissible trace does not have to be enforced and scheduling constraints can be removed.

Instead of managing a set of admissible traces, Algorithm 4 uses a single trace as the current admissible trace prefix. Every event that occurs in this prefix has to be executed according to its partial order, however every additional event may be executed without synchronization. Once the verifier has collected enough information about correct executions of the program, the admissible trace prefix is updated.

As soon as all events of a safe admissible trace prefix have been executed via an execution prefix, all continuations of τ must not reach an unsafe state. Formally, we define safety for admissible trace prefixes as possible.

Definition 31 (safe admissible trace prefix)**.** *Given the verification status G of a verifier, a trace prefix o is a* safe admissible trace prefix*, written $safe_G(o)$, if for all executions $\tau = s_0T_1s_1 \ldots s_n$, $(\tau \approx o \Rightarrow safe_G(s_n)) \wedge (\tau \preccurlyeq o \Rightarrow \mathfrak{l}(s_n) \neq \mathfrak{l}_{error})$.*

Similarly, we require a guaranty about the absence of deadlocks for admissible trace prefixes. Whenever an execution prefix is covered by a safe admissible trace prefix, no deadlock should be reachable from this execution prefix. Whenever an execution prefix adheres to a safe admissible trace prefix, there should exist an event that does not need to wait for an other event.

Definition 32 (deadlock-free admissible trace prefix)**.** *Given the verification status G of a verifier, a safe admissible trace prefix is* deadlock-free*, written deadlock-free$_G(o)$, if for all execution prefixes $\tau = s_0T_1s_1 \ldots s_n$, $\tau \approx o$ implies that no deadlock can be reached from s_n and $\tau \preccurlyeq o$ implies that $\exists e \in o.\, \forall e' \xrightarrow{c}_o e.\, s_n \models c \Rightarrow tid(e') = tid(e)$.*

The current admissible trace prefix o_{adm} is updated by shortening it, i.e., by removing constraints at the end of its happens-before relation. Formally, a new admissible trace prefix o' is required to satisfy $o' < o_{adm}$. On a more abstract level, the verifier finds an initial, complete, and correct symbolic trace o of the program and generates a sequence $o > o_1 > \ldots > o_n$ of subsequent trace prefixes such that for all $1 \leq i \leq n$, $safe_G(o_i)$.

A verifier can update a trace prefix o as follows. Each edge (e_1, e_2) with $tid(e_1) \neq tid(e_2)$ in o is interpreted as a scheduling constraint that requires e_2 to be executed after e_1. Updates of trace prefixes remove scheduling constraints. Let o' be o with e_1, e_2, and all their successors (w.r.t. the happens-before relation) removed. It is safe to remove the scheduling constraint (e_1, e_2) if all states s that are reachable by a linearization of o' are safe, i.e., $safe_G(s)$. Depending on the verification approach used, it may be more efficient to delay the removal of (e_1, e_2) until it occurs at an end of o, w.r.t. the happens-before relation, i.e., no event happens after e_2 that has an incoming or outgoing edge with an event from an other thread.

In the worst case, even if scheduling constraint (e_1, e_2) is at the end of a trace prefix, the verifier has to prove safety for exponentially many states before (e_1, e_2) can be safely removed. On the one hand, this complexity is a general limitation of IRS. On the other hand, the duty of the verifier can be reduced exponentially by adding only one scheduling constraint, which may reduce the verification delay considerably.

In order to use the optimization of trace prefixes, we extend the definition of adherence to a symbolic trace graph as follows. A finite execution $\tau = s_0T_1s_1 \dots T_ns_n$ with event sequence $\rho = e_1 \dots e_n$ is covered by a symbolic trace graph o, written $\tau \precsim o$, if:

- $\tau \preccurlyeq o$ or
- $\exists \tau' < \tau.\, \tau' \approx o$

In other words, if a prefix $\tau' < \tau$ adheres to o and contains exactly the events of o, all continuations of τ' are covered by o. By our safety requirement for trace prefixes, it is safe to use $\precsim$ instead of $\preccurlyeq$.

Do not preempt minimal events. In addition to the use of trace prefixes, Algorithm 4 omits synchronization before events that do not have to occur second in a race, i.e., events that do not have a predecessor in o_{adm} from a different thread.

The execution environment of Algorithm 4 reduces the number of synchronizations by permitting a sequence of events, potentially from multiple threads, between two synchronizations. This sequence is chosen from the set $\mathit{free}(\tau, o)$ as a continuation of the current execution τ that adheres to o or contains only synchronization-free events. To simplify the presentation of Algorithm 4, we include the previously described optimization using trace prefixes in the definition of $\mathit{free}(\tau, o)$.

Definition 33. *Given an execution prefix $\tau_1 = s_0T_1s_1 \dots T_ns_n$ and a symbolic trace graph, the set of synchronization-free event sequences, $\mathit{free}(\tau_1, o)$, is defined as those execution fragments τ_2 that induce a (feasible) execution prefix $\tau_1 \cdot \tau_2$ and either is covered by o or consist only of unconstrained events, i.e., $\mathit{free}(\tau_1, o) := \{\tau_2 : \exists \tau \in \mathit{Executions}(P).\, \tau_1 \cdot \tau_2 < \tau \wedge (\tau_1 \cdot \tau_2 \precsim o \vee \forall e \in \rho(\tau_2).\, \forall e' \in o.\, e' \xrightarrow{c}_o e \wedge s_n \models c \Rightarrow \mathit{tid}(e') = \mathit{tid}(e))\}$.*

Nota bene, as described in Section 4.1, verifier and execution environment are executed concurrently such that the execution environment can be executed several times during a single run of the verifier.

5.1.4 Correctness and deadlock-freedom

An IRS algorithm is correct if only safe executions can occur under its execution environment. The following theorem provides correctness of Algorithm 4.

Theorem 4 (Correctness of IRS)**.** *Whenever an execution $\tau = s_0T_1s_1 \dots T_ns_n$ has been executed by Algorithm 4, all visited states are error-free, i.e., $\forall 0 \leq i \leq n.\, \mathit{error\_free}(s_i)$.*

Proof. Let $\tau = s_0 T_1 s_1 \dots T_n s_n$ be an execution or execution prefix executed by Algorithm 4, let o_{adm} be the current admissible trace prefix, and G the current verifier state. Induction on the number of steps of the execution environment, i.e., the number of times a sequence from *free*() has been executed. Base case: zero steps have been executed. As the initial state is always safe, all visited states are error-free. Inductive case: n steps have been executed. The result of the previous $n-1$ steps is an execution prefix τ_1 Let s_k be the state reached after executing τ_1. By induction hypothesis, all states visited by τ_1 are error-free. If the admissible trace prefix o_{adm} has been updated by some o'_{adm} since the last execution step, this also holds for o'_{adm} since $o'_{adm} < o_{adm}$ is required. It remains to show that the sequence τ_2 from $\mathit{free}(\tau_1, o)$, such that $\tau = \tau_1 \cdot \tau_2$, does not visit an error state.

Case distinction according to the definition of *free*().

1. There exists a prefix $\tau' = s_0 T_1 s_1 \dots T_k s_k$ of τ such that $\tau' \approx o_{adm}$.

   Algorithm 4 in line 7 ensures that $\mathit{safe}_G(o_{adm})$. The verifier guarantees that $\mathit{safe}_G(s_k)$, hence all states visited by τ_2 are error-free.

2. All events of $\rho(\tau_2)$ are synchronization-free, i.e., $\forall e \in \rho(\tau_2). \forall e' \in o.\, e' \xrightarrow{c}_o e \wedge s_n \vDash c \Rightarrow \mathit{tid}(e') = \mathit{tid}(e)$.

   $\tau_1 \cdot \tau_2 \preccurlyeq o$, hence the verifier guarantees that all states visited by τ_2 are error-free.

□

In addition to correctness, an important requirement is that a program is never completely blocked by scheduling constraints (provided that at least one correct execution exists). The following deadlock-freedom theorem guarantees that this cannot happen with Algorithm 4.

Theorem 5 (Deadlock-freedom of IRS)**.** *Whenever an execution or execution prefix τ has been executed by Algorithm 4 with a deadlock-free admissible trace prefix o_{adm}, either the program has terminated or $\mathit{free}(\tau, o_{adm})$ is not empty.*

Proof. Analogous to the correctness proof, the definition of deadlock-free admissible trace prefixes guarantees that no deadlock can be reached (for $\tau \precsim o_{adm}$) or that $\mathit{free}(\tau, o_{adm})$ is not empty (for $\tau \preccurlyeq o_{adm}$). □

```
1 %20 = call i32 @getThreadId(%"class.benchmark::WorkerThread"* %this)
2 %21 = alloca i32
3 store i32 %20, i32* %21
4 %22 = load i32, i32* %21
5 %23 = bitcast i32* %17 to i8*
6 call void @before_memory_access(i32 %22, i8* %23, i64 4, i32 1)
7 %24 = cmpxchg i32* %17, i32 0, i32 %19 seq_cst seq_cst
8 call void @after_memory_access(i32 %22)
```

Listing 5.1: A global memory access (**cmpxchg**) after inserting callbacks directly before and after.

5.1.5 Experimental evaluation

Implementation

We have implemented Algorithm 4 in an IRS prototype. This prototype handles C and C++ programs translated to LLVM-IR. The LLVM-IR code is instrumented via the LLVM compiler infrastructure [LLV] in order to enforce an admissible trace prefix whenever the program is executed. The IRS execution environment is realized completely inside the instrumented application program and does not depend on any modifications of the operating system or assumptions on the used scheduler. Via a standard dependency analysis the prototype identifies all *dependent* memory accesses, which are memory accesses that either directly access global memory or may influence the result of an other global memory access. Scheduling constraints are enforced by callbacks directly before each dependent memory access that check whether this memory access is currently permitted. Callbacks directly after each dependent memory access communicate to other threads that the memory access has been performed. Memory fences inside these callbacks ensure sequential consistency, as assumed by our presentation in Section 5.1.3. Before each instrumented memory access, a thread checks whether an event of an other thread has to occur before its own upcoming event via a look-up in a global vector clock. Busy waiting is performed until the current thread is permitted to continue. After the memory access, the callback signals that the memory access is completed by updating the global vector clock. In contrast to earlier versions of our prototype, no thread is added to the program.

Listing 5.1 shows an example application of our instrumentation. Identifiers have been renamed for easier readability. Only line 7 (containing the compare-and-swap instruction **cmpxchg**) is contained in the original program. All additional lines are added by our instrumentation. Initially, a custom, deterministic thread ID is obtained. Afterwards, thread ID, memory location and whether the access can modify the memory are reported by callback before_memory_access to the library, where the event is recorded. After the memory access, callback after_memory_access signals that the memory access is completed.

When testing several alternatives of implementing schedule enforcement, we observed that, as ex-

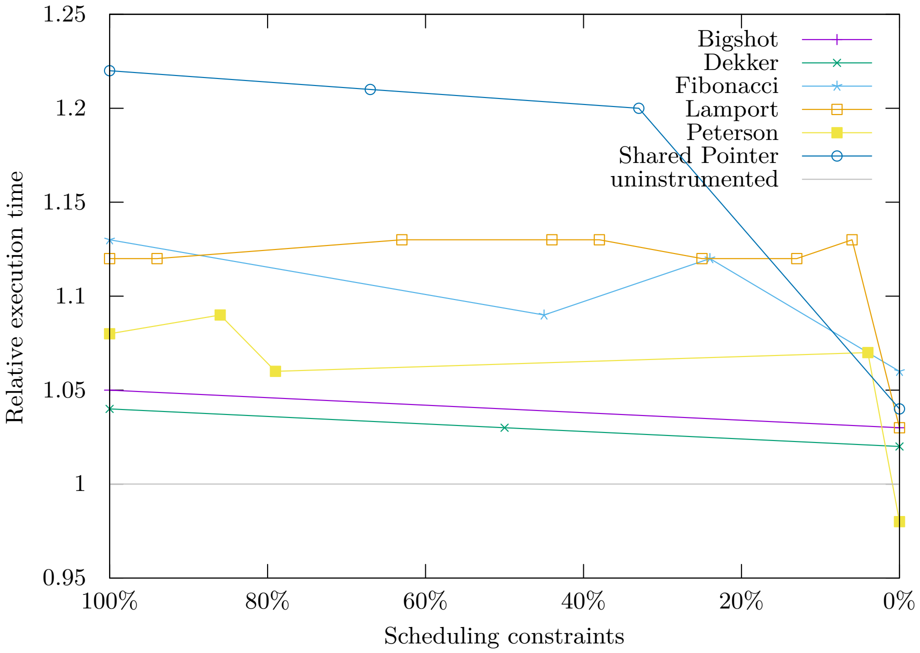

Figure 5.3: Execution time overhead of IRS relative to uninstrumented benchmarks for decreasing numbers of scheduling constraints (two-threaded benchmarks)

pected, lock-based implementations of waiting for other threads' events is much slower than busy waiting. A disadvantage of busy waiting is CPU consumption during waiting, which can reduce performance when more threads are active than hardware cores are available. We expect that improvements over our current, simple scheme of busy-waiting for permissions can be made by the use of a more advanced combination of busy waiting with lock-based synchronization or scheduler interaction (e.g., the POSIX sched_yield() system call). Additionally, we tested an implementation that uses a loadable kernel module to communicate with the Linux scheduler. Whenever an event is not yet permitted to be executed, the corresponding task's state is set to TASK_WAIT and only restored once the event is permitted. This design circumvents the additional CPU consumption of busy waiting. However, additional overhead appears because the current program counter of each thread has to be communicated to the loadable kernel module. In our tests, this design showed only an advantage if most events were constrained, i.e., the likelihood that an event has to wait is high.

Evaluation

Enforcing scheduling constraints in order to disable schedules outside of a given admissible trace is likely to incur execution time overhead (here: simply *overhead*) in comparison to plain program executions (without IRS). A crucial factor for the applicability of IRS in practice is how scheduling constraints in IRS influence this overhead, which we evaluate on several benchmark programs. The main goal of this evaluation is to investigate whether, for a given admissible trace and induced scheduling constraints, relaxing those constraints reduces the overhead and, if this is the case, how fast. Additionally, we investigate whether the selection of the initial and following admissible traces, i.e., the structure of the admissible trace prefix, influences the overhead. Software material for reproduction of these experiments is available [Met20].

Setup. All experiments have been conducted with our IRS implementation described in Section 5.1.5. The hardware used is an Intel Core i5-6500 CPU at 3.20GHz with four cores running Linux 4.8.0. Each benchmark is run with and without instrumentation by our prototype. The instrumented version is run in several configurations, with a decreasing amount of scheduling constraints. The initial number of scheduling constraints and the number of scheduling constraints that can be removed in one step, and thereby the number of configurations per benchmark, vary as the number of conflicting memory accesses varies among benchmarks. Each configuration is run 1000 times. We report the median execution time and overhead relative to the unmodified benchmark. Detailed measurement results are shown in Appendix C.

Benchmark set 1. The first set of benchmarks are concurrent programs from the SV-COMP benchmark suite [SVC] and the POR literature (Shared Pointer, [GFYS07]). We chose these benchmarks because they are well-studied verification problems and contain a high amount of concurrent interaction, which is expected to highlight performance issues of IRS. All benchmarks contain two threads. The corresponding results are shown in Figure 5.3. For these benchmarks, IRS produces a maximum overhead of 22%, which is much less than we expected and might be already an acceptable overhead for certain applications. For all benchmarks, the overhead is reduced by relaxing scheduling constraints, albeit in some cases, a significant reduction occurs only at the last reduction step. In some cases, the overhead is negative, i.e., the instrumented version of a benchmark executed faster than the plain benchmark. We conjecture that both measurement noise as well as improved timing of cache operations due to a different interleaving of memory operations may be relevant for this effect, as already noted by Olszewski et al. [OAA09]. Similarly, an increased overhead after removing scheduling constraints could be caused in such a way. Overall, both the initial overhead and the amount of reductions are lower than we expected.

Benchmark set 2. Since we expected a higher overhead, we conduct the same experiment on two benchmarks from the POR literature (Indexer [FG05] with 15 threads and Last Zero [AAJS14] with 16 threads), where we expect a higher overhead as a larger amount of threads and dependencies result in a higher amount of scheduling constraints. Figure 5.4 shows the corresponding results. Indeed, for

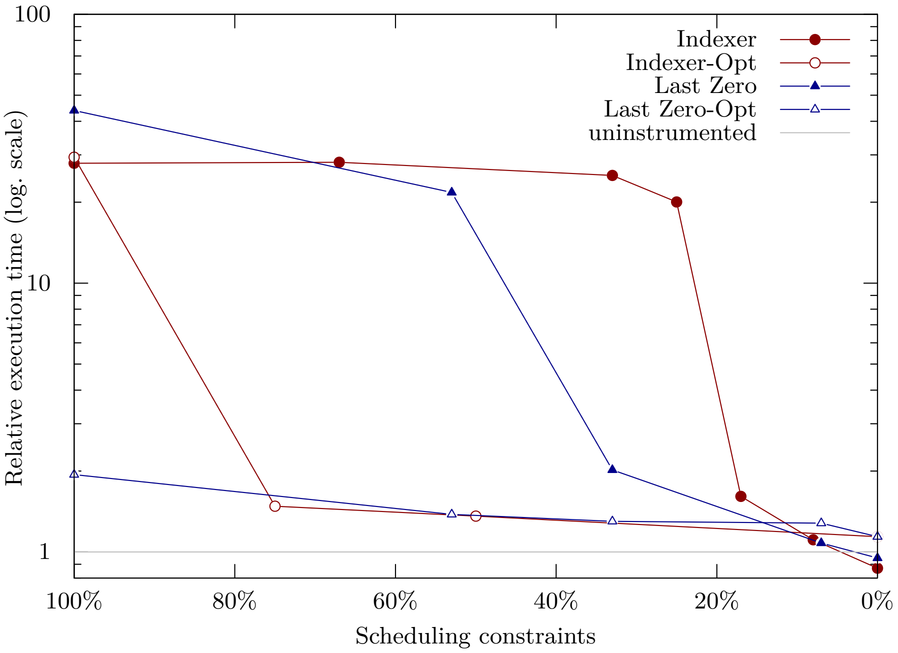

Figure 5.4: Execution time overhead of IRS relative to uninstrumented benchmarks for decreasing numbers of scheduling constraints (many-threaded benchmarks)

the Indexer and Last Zero benchmarks, the overhead is much higher. Interestingly, the overhead for Indexer abruptly decreases from 1904% to 61% at the transition from 3 to 2 scheduling constraints. We explain this observation by the fact that the permitted trace prefix with 3 scheduling constraints requires 3 threads to wait, while after removing 1 scheduling constraint, only 2 threads have to wait. Since our implementation uses busy waiting, many concurrently waiting threads may prevent threads that are not required to wait from quickly proceeding.

Structure of scheduling constraints. An interesting question is whether the overhead can be reduced by choosing a different trace prefix with roughly the same amount of scheduling constraints. Interestingly, we have found optimized trace prefixes for both Indexer and Last Zero that indeed show a drastically reduced overhead with the same or even more scheduling constraints. The corresponding results are depicted as Indexer-Opt and Last Zero-Opt in Figure 5.4. For Indexer, we found that choosing a trace prefix that requires less threads to wait can be executed faster. Figure 5.5 shows two alternative trace prefixes for Indexer. Nodes represent events and edges a happens-before relation. The nodes of even-indexed threads are shown in gray and events of the same thread are arranged vertically one below the other. Figure 5.5a shows one of the slower trace prefixes, where many threads wait rarely, and Figure 5.5b shows one of the faster (optimized) trace prefixes, where few threads wait often. For

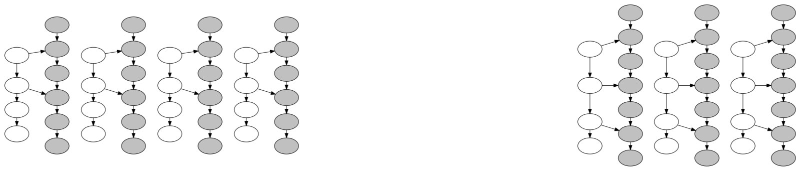

Figure 5.5 (a) Many threads wait rarely

Figure 5.5 (b) Few threads wait often (optimized)

Figure 5.5: Trace prefixes for Indexer (threads with only conflict-free events are omitted)

the former trace prefix, 16 Mazurkiewicz traces, for the latter trace prefix, only 8 Mazurkiewicz traces have to be verified. Although more scheduling constraints are enforced, the program execution is faster with the latter trace prefix. While we optimize trace prefixes manually, it is conceivable that verifiers can prioritize faster trace prefixes automatically, e.g., by applying a heuristic or by testing few traces and comparing their overhead. Such a prioritization resembles the effects of ordering heuristics on the performance of POR algorithms studied by Lauterburg et al. [LKMA10]. For Last Zero, our original trace prefixes require the second event of a worker thread to wait for the first event of the next worker thread. By letting threads wait already before their first events, the program execution is drastically accelerated already for 100% scheduling constraints, i.e., when only a single Mazurkiewicz trace is verified.

Summary. Our results show that relaxing scheduling constraints can reduce the overhead for all benchmarks. For example, after verifying only 8 of 4096 Mazurkiewicz traces of Indexer, the overhead is reduced from 2841% to 48%. However, in other cases, the execution time may not decrease considerably until a large part of all scheduling constraints have been removed. In yet other cases, the overhead is reduced considerably by removing a single scheduling constraint, while it does not change considerably before and after this step. Besides the number of scheduling constraints, the choice of the permissible trace prefix, i.e., the structure of the induced scheduling constraints, may have a large influence on the overhead. These observations suggest that a sensible selection of an initial symbolic trace during verification can considerably improve the execution time performance of a program that is executed with IRS. Comparing our current results for Indexer and Last Zero to earlier experiments with a less optimized schedule enforcement [MSBS17], we see a considerable speed-up when optimized trace prefixes are used.

5.2 Infinite executions

5.2.1 Program schedules for non-terminating executions

In order to support non-terminating programs, we extend our approach from Section 5.1 to schedules for infinite executions. Based on ARTs, we introduce finite schedules that represent infinite executions by the use of the covering relation found by the model checker. Therefore, our approach is suitable for verifiers that produce IVRs based on ARTs (cf. Section 4.3), while IVRs of a different format may have to be adapted.

The scheduling constraints of a safe IVR $\mathcal{R}_{\mathscr{A}}$ can be enforced by a scheduler that traverses the nodes of $\mathscr{A}$ according to the current state of execution and schedules some thread that owns an edge at the current node. However, such a naive enforcement would result in a strictly sequential execution of transitions and would foil any benefit of concurrency. In addition to the support of infinite executions, we aim for few necessary synchronizations between threads.

To enable parallel executions, we introduce *program schedules* that relax the scheduling constraints by means of partial-order reduction (POR). Note that this application of POR concerns the enforcement of scheduling constraints and occurs in addition to POR applied by our model checking algorithm when constructing an ART (cf. Section 4.4). Nevertheless, dependency information that is used for POR during model checking can be reused so that redundant computations are avoided.

Our goal is to permit the parallel execution of independent transitions (in different threads) whose order does not affect the outcome of the execution represented by $\mathscr{A}$ (i.e., the resulting executions are Mazurkiewicz-equivalent). Using traditional POR to construct such scheduling constraints poses two challenges: 1. Executions may be infinite, but we need a finite representation of scheduling constraints. 2. The control flow of an execution may be unpredictable, i.e., it is a priori unclear which scheduling constraints will apply. We solve issue 1 by partitioning ARTs into *sections* and associate a finite schedule with every section. To address issue 2, we require that sections do not contain branchings (control flow and non-deterministic transitions).

Consider the program and corresponding ART in Figure 5.6a. The **if** statement of T_1 is modeled as a separate read transition followed by a branching at node v_4. We define three section paths:

$$\pi_1 := \epsilon \to v_1 \to v_2 \to v_3 \to v_4$$

$$\pi_2 := v_4 \to v_5 \to v_7 \to \epsilon$$

$$\pi_3 := v_4 \to v_6 \to \epsilon$$

```
Variables:
  int x, y, z
Thread T1:
  while true:
    x := 1
    if z = 0:
      y := 1
Thread T2:
  while true:
    y := 0
    x := 0
```

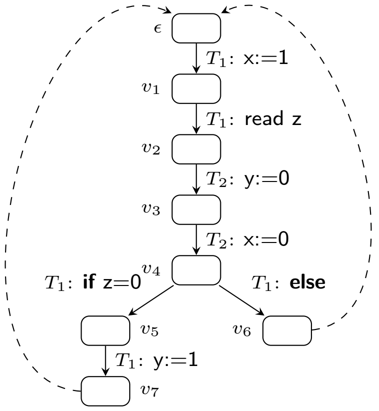

Figure 5.6 (a) A Program with a fair ART

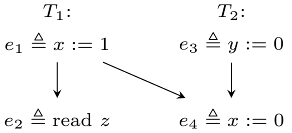

Figure 5.6 (b) The section schedule for the section path π_1 from ϵ to v_4

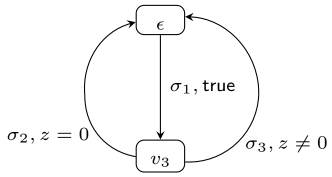

Figure 5.6 (c) A corresponding program schedule

After π_1 has been executed, a scheduler can distinguish the cases $y = 0$ and $y \neq 0$ and schedule π_2 or π_3 accordingly.

Formally, we define *branching nodes* and *section paths* as follows.

Definition 34 (branching node)**.** *A node v in an ART $\mathscr{A}$ represents a branching, written branching(v), if a single thread has at least two outgoing edges at v, i.e., $\exists T \in \mathscr{T}.\, \exists w, w' \in V_{\mathscr{A}}.\, w \neq w' \wedge v \xrightarrow{T} w \wedge v \xrightarrow{T} w'$.*

For an optimized partial-order reduction, successor nodes v with $\phi(v) \equiv$ *false* can be discarded.

A section path $v_1 \xrightarrow{R_1} \ldots \xrightarrow{R_n} v_{n+1}$ corresponds to a branching-free path in an ART whose first transition may be guarded. A section path follows $\rightarrow_{\mathscr{A}}$ edges, skipping covering edges $\triangleright$.

Definition 35 (section path)**.** *Given a node v of an ART $\mathscr{A}$, a* section path from v *is a finite sequence $v_0 T_0 R_0 v_1 \ldots v_{n-1} T_{n-1} R_{n-1} v_n$ such that*

- $v_0 = v$
- $\forall i \in \{0, \ldots, n-1\}.\, (v_i \xrightarrow{T_i, R_i}_{\mathscr{A}} v_{i+1} \vee (\exists v'.\, v_i \xrightarrow{T_i}_{\mathscr{A}} v' \wedge (v', v_{i+1}) \in \triangleright))$
- $\forall i \in \{1, \ldots, n-1\}.\, \neg branching(v_i)$

We write $v \xrightarrow{\pi}_{\mathscr{A}} w$ if there exists a section path π from v to w in $\mathscr{A}$. Analogous to event sequences of executions, we define event sequences of section paths.

Definition 36 (event sequence of a section path)**.** *The* event sequence *of a section path $v_0 T_0 R_0 v_1 \ldots$ $\ldots v_{n-1} T_{n-1} R_{n-1} v_n$ is defined as $(T_0, k_0) \ldots (T_{n-1}, k_{n-1})$, where k_i is the number of occurrences of T_i in $T_0 \ldots T_{i-1}$.*

The *Section schedule* of a section path describes the Mazurkiewicz equivalence class of all executions that follow the section path and hence the same control flow.

Definition 37 (section schedule)**.** *The* section schedule *of a section path $v_0 T_0 R_0 v_1 \ldots v_{n-1} T_{n-1} R_{n-1} v_n$ with event sequence $e_0 \ldots e_{n-1}$ is the smallest partial order $\sigma = (V_\sigma, \rightarrow_\sigma)$ such that $V_\sigma = \{e_0, \ldots, e_{n-1}\}$ and $\rightarrow_\sigma \supseteq \{(e_i, e_j) : i < j \wedge R_i \nparallel R_j\}$.*

We write *PO($\mathscr{A}$)* for the set of section schedules of $\mathscr{A}$. Given a node v in an ART that admits fairness and a section path π that starts at v, we write $\sigma(\pi)$ for the (unique) section schedule of π.

The section schedule $\sigma(\pi_1)$ of π_1 is depicted in Figure 5.6b. It consists of four events $e_1 \triangleq T_1 : \mathsf{x{:=}1}$, $e_2 \triangleq T_1 : \mathsf{read\ z}$, $e_3 \triangleq T_2 : \mathsf{y{:=}0}$, and $e_4 \triangleq T_2 : \mathsf{x{:=}0}$. An arrow $e \rightarrow e'$ indicates that $\sigma(\pi_1)$ requires e to occur before e'. Events of the same thread are ordered according to the program order of the respective thread. Events e_1 and e_3 are from different threads and write to the same variable, whence they are dependent and the section schedule needs to specify an ordering: e_1 must occur before e_3. Accordingly, the complete section schedule is $(\{e_1, e_2, e_3, e_4\}, \{(e_1, e_2), (e_3, e_4), (e_1, e_3)\})$.

By the following lemma, an execution from a state corresponding to the first node of a section and scheduled according to the respective section schedule will always lead to a state corresponding to the last node of the section. For instance, the following execution fragments both lead from the initial state to a state represented by v_4 $(s_4, s'_4 \models \phi(v_4))$, as e_1 and e_3 are independent and can be swapped:

$$s_{init}, T_1, s_1, T_2, s_2, T_1, s_3, T_2, s_4 \leftrightsquigarrow e_1, e_3, e_2, e_4$$

$$s_{init}, T_2, s'_1, T_1, s'_2, T_1, s'_3, T_2, s'_4 \leftrightsquigarrow e_3, e_1, e_2, e_4$$

Lemma 4 (Correctness of section schedules)**.** *Let $\mathscr{A}$ be a deadlock-free ART. Let $\tau = \tau_1 \cdot \tau_2$ be an execution prefix such that τ_1 corresponds to a path π_1 in $\mathscr{A}$ to the first node of a section path π in $\mathscr{A}$ and τ_2 corresponds to a linear extension of a section schedule $\sigma(\pi)$ for π. τ is equivalent to an execution $\tau' = \tau_1 \cdot \tau'_2$ that corresponds to $\pi_1 \cdot \pi$.*

Proof. Let $\mathscr{A}$ be a deadlock-free ART. Let $\tau = \tau_1 \cdot \tau_2$ be an execution prefix such that τ_1 corresponds to a path π_1 in $\mathscr{A}$ to the first node of a section path π in $\mathscr{A}$ and τ_2 corresponds to a linear extension of a section schedule $\sigma(\pi)$ for π.

By the definition of section paths, all linear extensions of $\sigma(\pi)$ follow the same control flow and contain the same transitions. By the definition of section schedules, $\sigma(\pi)$ is the partial order that describes the Mazurkiewicz trace of all executions that correspond to π. Hence, τ_2 and all other execution infixes τ_2' that follow τ_1 ($\tau_1 \cdot \tau_2'$ is an execution prefix) and correspond to a linear extension of $\sigma(\pi)$ are equivalent to each other [God96]. In particular, τ_2 is equivalent to the execution infix τ_2' that corresponds to π. Hence, $\tau' := \tau_1 \cdot \tau_2'$ corresponds to $\pi_1 \cdot \pi$ and is equivalent to τ. □

While section schedules represent scheduling constraints for execution fragments, we obtain scheduling constraints for complete executions by connecting several section schedules into a *program schedule.* A program schedule Σ is a labeled graph $(V_\Sigma, \rightarrow_\Sigma)$. Each node $v \in V_\Sigma$ is a node from $\mathscr{A}$ and the start of a section path π in $\mathscr{A}$. Each edge is labeled with the section schedule of π and the guard $\mathit{Guard}(R)$ of the first transition R in π.

In order to guarantee that each execution that corresponds to a path in an ART $\mathscr{A}$ adheres to the scheduling constraints of a program schedule Σ for $\mathscr{A}$, we require that Σ contains a section schedule for at least the initial node and all branching nodes of $\mathscr{A}$. Furthermore, as $\mathscr{A}$ is deadlock-free, there exists a thread T which is fully expanded at v in $\mathscr{A}$ and we require that Σ likewise has outgoing edges at v labeled with T for each transition of T at v. Figure 5.6c shows a program schedule for our example program.

Definition 38 (program schedule)**.** *Given an ART $\mathscr{A}$ that admits fairness with root ϵ, the* program schedule *of $\mathscr{A}$ is a labeled graph $\Sigma = (V_\Sigma, \rightarrow_\Sigma)$ such that:*

- $V_\Sigma \subseteq V_{\mathscr{A}}$ *(Σ's nodes are a subset of $\mathscr{A}$'s nodes)*
- $\epsilon \in V_\Sigma \wedge \{v \in V_{\mathscr{A}} : \mathit{branching}(v)\} \subseteq V_\Sigma$ *(Σ contains $\mathscr{A}$'s initial node and all branching nodes)*
- $\rightarrow_\Sigma \subseteq V_\Sigma \times \mathit{PO}(\mathscr{A}) \times \mathscr{T} \times \mathscr{F}(Q) \times V_\Sigma$ *(edges are labeled with a section schedule, a thread, and a transition)*
- $\forall v \in V_\Sigma.\, \exists T \in \mathscr{T}.\, \forall R \in \mathit{Transitions}(\mathfrak{l}_T(v)).\, \exists u \in V_{\mathscr{A}}.\, v \xrightarrow{T,R}_{\mathscr{A}} u \wedge \exists w \in V_\Sigma.\, \exists \sigma \in \mathit{PO}(\mathscr{A}).\, v \xrightarrow{\sigma,T,R}_\Sigma w$ *(every node v has an outgoing edge for each transition of a thread T expanded at v in $\mathscr{A}$ and T is fully expanded in $\mathscr{A}$)*
- $\forall (v, \sigma, T, R, w) \in \rightarrow_\Sigma.\, \exists \pi.\, \pi = v_0 T_0 R_0 v_1 \dots v_{n-1} T_{n-1} R_{n-1} v_n \wedge v \xrightarrow{\pi}_{\mathscr{A}} w \wedge \sigma(\pi) = \sigma \wedge T = T_0 \wedge R = R_0$ *(every edge corresponds to a section path in $\mathscr{A}$ that starts with the thread and transition of the edge)*

Similar to schedulers induced by IVRs, a scheduler can enforce the scheduling constraints of a program schedule by looking up a section schedule that matches the current execution prefix and scheduling an event whose predecessors (according to the section schedule) have already been executed. Hence, all independent events in a section can be executed concurrently without synchronization. All events of a section schedule have to appear before the first event of the next section schedule is executed, so that the states reached between sections correspond to nodes of the program schedule. For example, the event $T_1 : y := 1$ from section π_2 must not occur in between events T_1 : read z and $T_2 : y := 0$ from section π_1. We formalize this requirement as follows.

Definition 39 (execution schedule)**.** *Given a (possibly infinite) path* $v_1 \xrightarrow{\sigma_1, T_1, R_1} v_2 \xrightarrow{\sigma_2, T_2, R_2} \ldots$ *with* $\sigma_i = (V_i, \rightarrow_i), i \geq 1$ *in a program schedule, we define its* execution schedule *as the partial order* $(V_1 \uplus V_2 \uplus \ldots, (\rightarrow_1 \uplus \rightarrow_2 \uplus \ldots) \cup V_1 \times V_2 \cup V_2 \times V_3 \cup \ldots)$*, where* $\uplus$ *denotes a disjoint union.*

An execution τ adheres to the scheduling constraints of a program schedule Σ if τ is a linear extension of the execution schedule of some path in Σ.

Definition 40 (semantics of program schedules)**.** *The semantics of a program schedule* Σ *is defined as the set of all executions that are a linear extension of an execution schedule of a path in* Σ*.*

Hence, a program schedule of an ART $\mathscr{A}$ that admits fairness permits exactly those executions that correspond to a path in $\mathscr{A}$ (modulo Mazurkiewicz equivalence). In particular, as Mazurkiewicz equivalence preserves safety properties [God96], only safe executions are permitted.

Theorem 6 (correctness of program schedules)**.** *Let* $\mathscr{A}$ *be an ART that admits fairness and* Σ *a program schedule for* $\mathscr{A}$*. All program executions that adhere to the scheduling constraints of* Σ *are equivalent to an execution that corresponds to a path in* $\mathscr{A}$*.*

Proof. Let $\mathscr{A}$ be an ART that admits fairness, Σ a program schedule for $\mathscr{A}$, and τ be an execution that adheres to the scheduling constraints of Σ, i.e., all finite prefixes τ' of τ correspond to a path $\pi_{\tau'} = v_0 \xrightarrow{\sigma_0(\pi_0)}_{\Sigma} \ldots v_n \xrightarrow{\sigma_n(\pi_n)}_{\Sigma} v_{n+1}$ in Σ. We show a slightly stronger statement: all finite prefixes τ' of τ are equivalent to an execution prefix that corresponds to the path $\pi_0 \ldots \pi_n$ in $\mathscr{A}$.

Induction on the length of τ'.

case τ' is empty: τ' corresponds to the empty path in $\mathscr{A}$.

inductive case: Let $\pi_{\tau'} = v_0 \xrightarrow{\sigma_0(\pi_0)}_{\Sigma} \ldots v_n \xrightarrow{\sigma_n(\pi_n)}_{\Sigma} v_{n+1}$ be the path in Σ that τ' corresponds to.

Let $\tau' = x_1 x_2$ be partitioned so that x_1 corresponds to the prefix $v_0 \ldots v_n$ in that path. Such a

Algorithm 5: IVR induced by a program schedule Σ

```
input  : $s_0 T_0 \dots s_n$, an execution prefix
output : a set of threads that is permitted to execute after the given execution prefix

Data: $\Sigma$, a program schedule
Data: $\sigma$, initially some section schedule such that $\epsilon \xrightarrow{\sigma}_{\Sigma} w$ for some $w$
Data: $v := w$
Data: $i := 0$

Function $R(s_0 T_0 \dots s_n)$
    if $n - i = |\sigma|$ then
        $i := n$
        choose $\sigma, w$ s.t. $v \xrightarrow{\sigma,T,R}_{\Sigma} w$ and $s_n$ satisfies the guard of transition $R$
        $v := w$
    $\sigma' := \sigma$ with the events of $T_i, \dots, T_{n-1}$ removed
    return $\min(\sigma')$ (all threads that have no predecessors in $\sigma$)
```

partitioning exists, as by Definition 39, an event must occur after all events from the previous section schedule and before all events from the following section schedule.

By the induction hypothesis, there exists an execution $x_1^{\approx}$ that is equivalent to x_1 and corresponds to the path $\pi_0 \dots \pi_{n-1}$ in $\mathscr{A}$. x_2 is a linear extension of $\sigma(\pi_n)$. By Lemma 4, there exists $x_2^{\approx}$ such that $x_1^{\approx} \cdot x_2^{\approx}$ is equivalent to $x_1 \cdot x_2$ and corresponds to $\pi_0 \dots \pi_n$.

□

Algorithm 5 shows how an IVR can be derived from a program schedule. The algorithm requires a program schedule for the program under execution and maintains a current section schedule σ. Given an execution prefix τ, it checks whether there are still events in σ that are not yet executed. If this is not the case, the current section is reset to a section that is feasible in the current state. Afterwards, those events that have already been executed are temporarily removed from σ and a thread is scheduled that has no predecessors in σ after this removal.

5.2.2 Experimental evaluation

Implementation

To evaluate the enforcement of program schedules for infinite executions, we implement a custom (user space) scheduler, similar to our implementation for finite executions.

In a first step, we automatically translate ARTs constructed by Impara-IMC to program schedules encoded as vector clocks [Mat89]. To omit sections in the generated program schedule that would never be executed and thereby reduce the size of the program schedule, we discard all paths in the ART that lead only to nodes labeled with *false*. As we use only deadlock-free ARTs, an alternative, feasible path, always exists. A given ART is traversed from the root. Recursively, we build section paths by traversing the graph until a branching node is reached. At the branching node, a fully expanded thread T is chosen. The next sections are started at all child nodes of the branching node that are reached by a transition of T. For each section, the section schedule is generated based on the dependency information of memory accesses. Section schedules are represented by vector clocks. Additionally, each section schedule contains a link to all possible successor sections, i.e., those sections that start at a direct successor node of the current section. If there exist nodes v, w such that all possible paths between v and w are section paths and correspond to pairwise equivalent executions, a single section path between v and w with relaxed scheduling constraints is sufficient. In this case, no dependencies between memory events need to be enforced. However, we use only the first IVR in our experiments (produced in a single iteration of Algorithm 3), whence we do not evaluate this case.

Once all section schedules for the given ART are generated by enumerating all section schedules, including link information about successor sections, and marking the initial section.

Second, we instrument the source code of benchmark programs manually with callbacks to our user space scheduler and code for time measurement. The user space scheduler is implemented in C++11 and uses the C++ standard library for atomic memory operations. Program schedules are included as header files. Every access to a non-thread-local, global variable (shared variable) is replaced by a C++ preprocessor macro that calls the user space scheduler, executes the original statement, and calls the user space scheduler to notify that the statement has been executed. In our selection of benchmark programs, we had to instrument assignments and if-then-else statements. In the case of control flow branchings that depend on a shared variable, i.e., an if-then-else statement where the branching expression depends on a shared variable, additional callbacks are necessary to notify the scheduler of the taken control flow path.

To ensure that memory accesses enclosed by callbacks are indeed executed after the preceding callback and before the succeeding callback, memory fences are used.

The result of steps one and two is a multi-threaded program that executes concurrent memory accesses according to a given program schedule. Every execution of this program can be generated by Algorithm 5. Nevertheless, threads are executed concurrently and only forced to execute sequentially where required by the program schedule. Each time a thread T enters the callback preceding a memory access, T looks up the current section schedule and program counters of the other threads. If the vector clock of the section schedule, at the position of the current event of T, shows an event of an other thread that has to occur first, T waits until this event has been executed. If no more events are required to occur before the current event of T by the section schedule, T executes the current memory access and, in the succeeding callback, updates its program counter so that the other threads are notified that T

has executed another event.

In case all events of the current section have already been executed, T chooses the successor section associated to its current event. Waiting for all threads to completely execute the current section before switching to a successor section ensures that the program, at the end of each section, reaches a state that is represented by a node in the program schedule (and thereby, in the ART generated by the model checker). In case T has no successor section associated to its current event, T waits for an other thread to choose the next section. In case the last node of the current section is a branching node, only the thread with a control flow branching chooses the next section. In case T has a control flow branching at the end of the last section, T chooses the successor section based on the taken control flow branch.

Thirdly, we instrument the benchmark programs with code for time measurement. Each thread executes in an indefinite loop. Each time a thread has accomplished useful work in the current loop iteration, e.g., producing or consuming an item, writing a block or inode, or executing the critical section, it increments its *performance counter*. The main thread sleeps for 2 seconds, the time out duration, and subsequently prints the sum of the performance counters of all threads and terminates the program. Such a single run of a benchmark program is executed five times and we report the respective median value of performance counter sums. All experiments have been executed on a 4-core Intel Core i5-6500 CPU at 3.2 GHz.

While we manually instrumented the benchmark source code, an automated instrumentation is well conceivable. Main tasks of such an automated instrumentation are to identify shared variables and all points in the program, where dependent expressions are accessed. Relevant shared variables can be either overapproximated so that all shared or global variables are included or found by a static dependency analysis. Even if the variables to be instrumented are overapproximated, the expected additional execution time overhead is small, as our experiments show: a callback to our scheduler is fast if the current thread does not have to wait for other threads before executing the next variable access. Expressions that depend on a shared variable can likewise be found by a static dependency analysis. The automated instrumentation may of course be implemented on the level on the intermediate representation of a compiler and does not have to be conducted on the source code level. Software material for reproduction of these experiments is available [Met20].

Evaluation

Table 5.1 shows the performance impact of enforcing IVRs on several correct programs. Each program is model-checked once until the first IVR (verification times are reported in Table 4.1). As a baseline, the program is run without schedule enforcement (unconstrained). The first IVR is enforced without (Opt0),

Table 5.1: Experimental results
Performance is measured in number of useful loop iterations (e.g., with a successful concurrent access such as a produced item) within a time limit of 2 seconds.

	Performance (higher is better)			
Benchmark	Opt0	Opt1	Opt2	Unconstr.
prod.-cons. 1p 1c 1000b	4 864 489	7 466 093	**11 370 258**	8 199 202
prod.-cons. 2p 2c 1000b	3 400 187	5 959 041	8 428 598	**11 643 208**
prod.-cons. 4p 4c 1000b	1 327 063	2 576 695	3 676 876	**7 210 796**
prod.-cons. 1p 1c 5b	4 945 116	7 075 596	**12 372 817**	7 915 465
prod.-cons. 2p 2c 5b	3 194 019	5 514 429	**9 271 859**	6 933 172
prod.-cons. 4p 4c 5b	1 345 991	2 465 108	**3 392 111**	3 240 136
double lock 1 ms	1 845	1 834	**3 217**	1 797
file system	3 667	4 877 035	6 705 672	**23 822 129**
barrier	1 238 720	8 285 228	**14 586 849**	1 077 907

and with optimizations (Opt1, Opt2). Opt1 applies POR and omits operations on synchronization objects (mutexes, barriers).[1] Opt2 uses, in addition to Opt1, longer section schedules (by replicating a section eight times) and stronger partial-order reduction that identifies independent accesses to distinct indices of an array. Additionally, for the producer-consumer benchmark, we apply a compiler-like optimization, removing and reordering events to reduce the number of constraints.[2] Both Opt1 and Opt2 enable the concurrent execution of more memory accesses, e.g., because the beginning of a critical section can already be executed before a thread arrives at a constrained access that has to wait. The schedules for each benchmark (Opt0–Opt2) are obtained from the first IVR. As all benchmarks use unbounded loops, we measure the execution time performance by counting useful (i.e., with a successful concurrent access such as a produced item) loop iterations and terminating the execution after 2 seconds.

At the example of a section schedule of the producer-consumer benchmark with two threads, Figure 5.7a–5.7b illustrates the difference between optimizations. Figure 5.7a shows a section schedule for Opt0. All shared memory events are executed strictly sequentially, as it is the case with unconstrained executions: only the thread holding the lock is allowed to access shared memory. Opt1 removes the lock operations while maintaining the same ordering of events. Opt2, cf. Figure 5.7b, relaxes the original ordering, subsumes eight loop executions of both threads, and eliminates the redundant read event of count.

In Figure 5.7b, when the consumer executes the scheduler callback before its first event (read count), it looks up the constraint $e_{01} \rightarrow e_{10}$ and waits for the producer to finish event e_{01}. When the producer in

[1] As enforcing an IVR is redundant to synchronization over existing mutexes and barriers, omitting them is safe.

[2] Opt2 follows a general algorithm, however, we do not automate our implementation of Opt2, as it would be a large effort to implement compiler optimizations. Our implementation of Opt1 is automated.

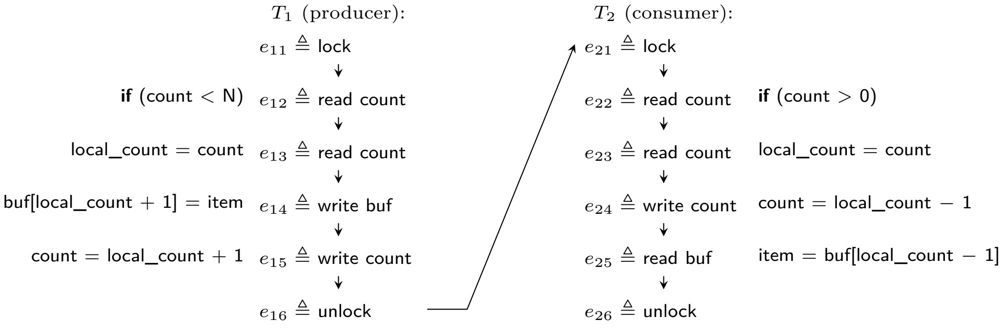
T_1 (producer):
T_2 (consumer):
$e_{11} \triangleq$ lock
if (count < N) $e_{12} \triangleq$ read count
local_count = count $e_{13} \triangleq$ read count
buf[local_count + 1] = item $e_{14} \triangleq$ write buf
count = local_count + 1 $e_{15} \triangleq$ write count
$e_{16} \triangleq$ unlock
$e_{21} \triangleq$ lock
$e_{22} \triangleq$ read count if (count > 0)
$e_{23} \triangleq$ read count local_count = count
$e_{24} \triangleq$ write count count = local_count − 1
$e_{25} \triangleq$ read buf item = buf[local_count − 1]
$e_{26} \triangleq$ unlock

Figure 5.7 (a) Section schedule for the producer-consumer benchmark (Opt0)

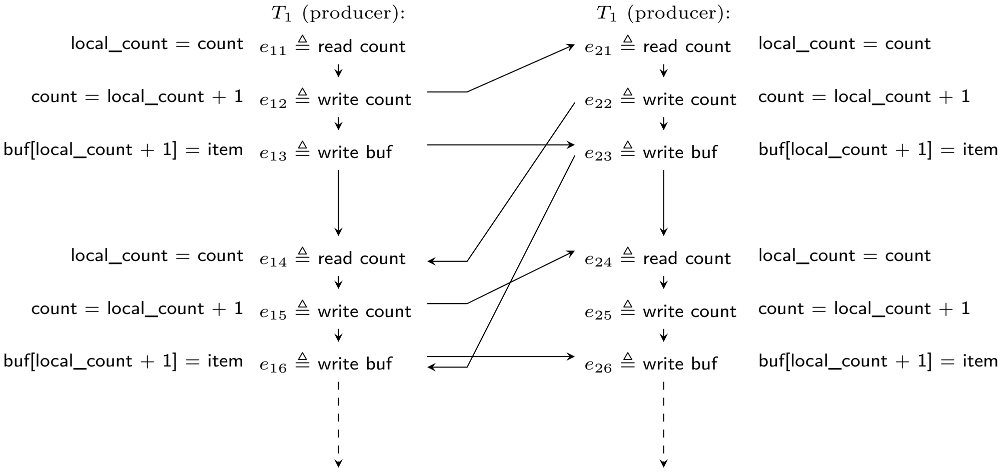
T_1 (producer):
T_1 (producer):
local_count = count $e_{11} \triangleq$ read count
count = local_count + 1 $e_{12} \triangleq$ write count
buf[local_count + 1] = item $e_{13} \triangleq$ write buf
local_count = count $e_{14} \triangleq$ read count
count = local_count + 1 $e_{15} \triangleq$ write count
buf[local_count + 1] = item $e_{16} \triangleq$ write buf
$e_{21} \triangleq$ read count local_count = count
$e_{22} \triangleq$ write count count = local_count + 1
$e_{23} \triangleq$ write buf buf[local_count + 1] = item
$e_{24} \triangleq$ read count local_count = count
$e_{25} \triangleq$ write count count = local_count + 1
$e_{26} \triangleq$ write buf buf[local_count + 1] = item

Figure 5.7 (b) Section schedule for the producer-consumer benchmark (Opt2) – only the first two of eight loop iterations are shown the callback after e_{01} has notified that e_{01} has been executed, the consumer continues and executes e_{10}. Similarly, the producer is permitted to execute e_{03} before e_{12} has been executed. Thus, the constrained execution under the optimized schedule permits “more” concurrency (i.e., more events to be executed concurrently) than the unconstrained execution with locks. For instance, the consumer is allowed to read the counter already after the producer has written it and does not have to wait for the producer to also write an item to the buffer.

We use the producer-consumer implementation (with correct synchronization and buffer sizes 5 and 1000) from SV-COMP [SVC] (stack_safe), modified with an unbounded loop and with 1, 2, and 4 producers and consumers. The double lock benchmark is a corrected version (lock operations in T_2 reversed) of the deadlock benchmark (Section 4.5.2), where the critical section is simulated by sleeping for 1 ms; the uncorrected version reached a deadlock after only 172 loop iterations. The file system benchmark from SV-COMP (time_var_mutex_safe) is extended with a third thread and again with unbounded loops as in Section 4.5.4. The barrier benchmark uses two barriers to implement ring communication between threads.

Somewhat surprisingly, some benchmarks are slower when executed unconstrained. We conjecture that this is caused by more memory accesses being executed in parallel under Opt2. In all but one cases, Opt2 is considerably faster than Opt1, which is considerably faster than Opt0. The highest overhead is observed for the file system benchmark, where Opt2 is about 3.5 times slower than the unconstrained execution. We conjecture that the high overhead here stems from an unequal distribution of loop iterations among threads, when executed unconstrained: the loop body of T_2 was executed nearly 100 times more frequently than T_1, while it is shorter and probably faster. Opt0–Opt2 execute all threads nearly balanced. In addition to the Pthreads barriers used in the barrier benchmark, we tried a variant with busy waiting barriers, where the unconstrained execution showed a performance of 13 567 135, which is still slower than Opt2.

Comparing the results for the producer-consumer benchmark with a buffer size of 1000 to those for a buffer size of 5, we observe that there is no considerable effect on Opt0–Opt2 but on most of the unconstrained executions. This observation is comprehensible, as the first IVR does not make use of more than at most four cells in the buffer (in case of four producers). The performance of unconstrained executions decreases with a smaller buffer as the chance that the buffer is full and a producer has to wait is higher. For all three configurations with a buffer size of 5, Opt2 shows the highest execution time performance.

Even in repeated executions of the experiment, the unconstrained variant of double lock showed only “starving” executions in the sense that the second thread was never able to acquire the mutexes before

Table 5.2: Experimental performance results for pfscan

	Execution time (s)		
Schedule	Constrained	Unconstrained	Relative
S1	3.34	3.25	1.03
S2	3.34	3.25	1.03
S3	3.6	3.25	1.10
S4	3.57	3.25	1.10

the timeout of 2 seconds. Hence, the constrained executions improve on the operating system scheduler in terms of a balanced execution of all threads.

In order to compare to the enforcement of *input-covering schedules* [BCG13] (explained in Section 6), we measure the overhead of our scheduler implementation on the pfscan benchmark used there. Pfscan is a parallel implementation of grep and uses 1 producer and 2 consumer threads to distribute tasks, consisting of reading and searching a file for a given query. As input, we use 8 files with 100MB of random content each. We evaluate 4 different schedules[3], which show an overhead between 3% and 10% (with Opt2). Hence, IVRs can perform much better than input-covering schedules (60% overhead reported in [BCG13]).

Table 5.2 contains our experimental results for the pfscan benchmark. We use two worker threads in addition to the main thread. The benchmark is executed with scheduling constraints of several program schedules S1–4 (column two) and unconstrained (column three). Execution times are given in seconds. The fourth column gives the relative execution time (overhead). In all constrained configurations, operations on synchronization objects have been omitted (Opt1). S1, S2, and S3 are program schedules as they can be produced during the first iteration of our model checking algorithm. Program schedule S4 allows any interleaving of critical sections so that all executions of the unconstrained program are matched. S1 and S2 contain sections that comprise both worker threads, while S3 and S4 contain only single-threaded sections. S1 and S2 differ in the ordering of the worker threads.

S3 causes an overhead of 10% with respect to the unconstrained execution. Although S4 allows any interleaving of critical sections, there remains an overhead of 10% caused by looking up section schedules during the execution. S1 and S2 show only a small overhead of 3%. We conjecture that the lower number of section schedule look-ups (compared to S3 and S4) is responsible for the considerably lower overhead.

[3] As IMPARA cannot handle several features used by pfscan (such as condition variables, structs, and standard output), we manually generate initial IVRs.

Chapter 6

Related work

Partial order reduction Partial order reduction (POR) (refer to, e.g., Godefroid [God96], Baier [BK08], or Clarke [CGP01] for a basic overview) identifies equivalent executions based on the dependencies between concurrent events. Properties that are compatible with POR, such as state reachability, can be checked by exploring one representative of each execution equivalence class, which reduces the verification complexity. This reduction does not restrict scheduling and is orthogonal to our approach of reducing the verification complexity via scheduling constraints. Our first contribution addresses the efficiency of POR algorithms by avoiding redundant dependencies. Furthermore, we combine both the generation of scheduling constraints (cf. Chapter 4) and their enforcement (cf. Chapter 5) with POR to increase the efficiency of our approach.

Static POR techniques use a static approximation of dependencies [Val89, GP93, Pel93, BKSS11]. While both static and dynamic POR algorithms can be augmented with section-based exploration as in EPOR, we focus on dynamic dependency calculation, which drastically increases the state space reduction for benchmarks such as Indexer.

Dynamic POR has been introduced by Flanagan and Godefroid [FG05]. Their algorithm DPOR computes a *persistent set* of events to explore in every visited state. Like many POR algorithms, DPOR has been combined with the *sleep set* technique [God90]. For every visited state, the corresponding sleep set contains events whose exploration would be redundant and is avoided.

Abdulla, Aronis, Jonsson, and Sagonas have proposed two model checking algorithms based on DPOR [AAJS14], named SDPOR and ODPOR, replacing persistent sets with *source sets*. In some cases, the source set of a state is smaller than the smallest persistent set of this state, which improves the state graph reduction. EPOR uses source sets in order to reverse races between sections but avoids redundant race checks and source set calculations inside of sections.

The ODPOR algorithm is an extension of SDPOR that can increase the amount of state space reduction for certain benchmarks, however adding runtime overhead that is not always compensated by a higher state space reduction: for many benchmarks, SDPOR is faster than ODPOR due to less runtime overhead [AAJS14]. Consequently, we compare our algorithm EPOR to SDPOR instead of ODPOR in order to investigate whether even the lower runtime overhead of SDPOR can be reduced.

CDPOR by Gueta, Flanagan, Yahav, and Sagiv [GFYS07] handles sequences of events, similar to EPOR and unlike DPOR, SDPOR, and ODPOR. However, CDPOR explores only events of a single thread at once, while EPOR handles events sequences of all threads and of varying length.

POR approaches for relaxed memory models have been proposed as well, e.g., [ZKW15]. Our system model handles programs with relaxed memory models by using partial program orders. Symbolic model checking (both bounded and unbounded) using POR has been addressed, e.g., in [KWG09, WKO13]. We present EPOR as an improvement of dependency calculation in explicit-state dynamic POR algorithms but do not see any fundamental difficulty in using it for symbolic POR.

Model checking Unbounded model checking [HJM04, WKO13, NFLP16, GLSW17] is a technique to verify the correctness of potentially non-terminating programs. In our setting, we use algorithms that represent the already explored state space and schedules by abstract reachability trees (ARTs) [HJMS02, McM06, WKO13] and perform this exploration in a forward manner. Instead of discarding an ART after an unsuccessful attempt to verify a program, we use the ART to extract safe schedules.

Conditional model checking [BHKW12] reuses arbitrary intermediate verification results. A subsequent verification attempt focuses on states that are not yet proven to be safe by the intermediate verification result. For example, multiple algorithms or configurations of algorithms can be used in subsequent verification attempts to combine their strengths. In contrast to our approach, intermediate verification results in the framework of conditional model checking are not guaranteed to prove the safety of a program that is functional under all inputs and does not enforce the preconditions (e.g., scheduling constraints) of the intermediate result.

Reducing the complexity of scheduling Program analyses that use context bounding [QW04, QR05, MQ07] consider only those executions of a program which contain only up to k context switches between threads, for a typically small bound k. As with our model checking approach, the model checking problem is eased, however, context bounding is limited to finite executions. While reachability for concurrent, recursive programs is undecidable [Ram00], additionally bounding the number of context switches makes the problem decidable [QR05]. Context bounding may be used within our iterative model checking algorithm (as long as executions are finite) as a policy that selects the next thread to be expanded when constructing an ART. In this sense, context bounding is a special case of an exploration policy for our algorithm. Similar to context bounding, a generally undecidable model checking problem may become tractable when handled by our algorithm: if an IVR is found, the program can be safely used even if the reachability problem is undecidable under unconstrained scheduling.

When applied with bounded model checking (BMC) for concurrency bug finding [RG05, CF11, MQ07, LR09, ITF$^+$14], context bounding focuses the search for erroneous schedules to those with few context switches. Consequently, potential bugs are missed that manifest themselves only after more context switches than the current bound. However, based on empirical results, Musuvathi and Qadeer argue that a low context bound is sufficient to find many interesting bugs [MQ07]. They propose iterative context bounding (ICB) as an extension to BMC: a program is iteratively checked with an increasing context bound, similar to increasing the bound on execution lengths in BMC. Given limited resources (that usually do not allow to search the complete state space of a program), ICB prioritizes schedules with few context switches. As mentioned above, this search strategy of ICB could be used within our iterative model checking algorithm. However, in contrast to bug finding based on BMC, our goal is a sound program analysis (under scheduling constraints), i.e., a safety proof for complete, unbounded program executions, which is not given, in general, by BMC. Another difference between context bounding in bug finding and our model checking approach are guarantees about scheduling: when searching for erroneous schedules, bug finding may use assumptions about the likelihood of schedules in order to guide the search. However, any such assumptions are not enforced. Consequently, bug finding may miss feasible executions of a program that contain, e.g., a bug that has not been found under context bounding. In contrast, our algorithm produces enforceable scheduling constraints so that only checked executions occur.

Sequentialized programs [QW04, LR09, FIP13, ITF$^+$14, NFLP16, NSF$^+$17] emulate the semantics of a multi-threaded program, allowing tools for sequential programs to be used. The amount of possible schedules is either not reduced at all or similar to context bounding.

Nguyen et al. [NSF$^+$17] transform a concurrent program into several instances that show only a reduced number of schedules, respectively. The technique of dividing a program into instances is based on lazy sequentialization for BMC [FIP13]. The scheduling constraints for each instance follow a fixed schema and need not necessarily be feasible. Hence, this approach of generating scheduling constraints is not well suited to find a single feasible schedule, as in our approach, but rather are intended to cover all schedulings up to given context bound. Each scheduling-constrained instance is checked individually by BMC with a context bound. Similar to our model checking approach, this decreases the complexity of the model checking problem and improves bug finding. As only executions with a bounded length and a bounded number context switches are checked, this approach is unsuitable for verification.

Quasi-static scheduling [LM87, CKL$^+$05, DGTY10] has been proposed to find a static schedule that is feasible for any dynamic program input, e.g., in the context of real-time scheduling. For example, the shorter branch of a control flow branching can be padded to the length of the other branch so that the length, or number of events, is the same regardless of the dynamic control flow branching. A static schedule for the program fragment consisting of this control flow branching can allow the statically known number of events so that it is feasible regardless of the dynamic choice. Such a schedule is denoted as quasi-static. The problem of finding a quasi-static schedule can be extended with additional requirements such as validity (a schedule must not prevent a program from terminating) and regularity (the language of permitted executions must be regular, in the sense that each processor may occur only a bounded number of times before all other processors occur) [DGTY10]. Such requirements are similar to our requirement on IVRs (cf. Section 4.2). Quasi-static scheduling has been discussed for several models, including processes communicating over a complete graph of buffers [DGTY10] and Petri nets [CKL$^+$05]. We use similar requirements on schedules, but for multi-threaded programs with shared memory. Instead of discussing the problem of quasi-static schedulability (i.e., the existence of a quasi-static schedule), we are interested in finding a concrete schedule that can both be verified and enforced, representing schedules (even for infinite executions), and enforcing schedules with concurrent computation (instead of sequentializing the events of multiple threads). We are not aware of any combination of quasi-static scheduling with model checking or other verification techniques.

Reducing the non-determinism due to weak memory models In addition to scheduling, a source of non-determinism are relaxed memory models in modern architectures. Automated fence insertion [FLM03, KVY10, AAC$^+$12, AAC$^+$13, LW13] transforms a program that is safe under sequential consistency to a program that is also safe under weaker memory models. While the amount of non-determinism in the ordering of events is reduced, non-determinism due to scheduling cannot be influenced. The approach of Burckhardt and Musuvathi [BM08] monitors executions for violations of sequential consistency but does not enforce it. Fang et al. [FLM03] present an automated memory fence insertion technique to enforce SC using instrumentation at the source code level. In both cases, the program can be safely verified under the assumption that SC holds with a reduced state space. Similarly to our model checking approach, these approaches restrict the amount of non-determinism. However, in contrast to our model checking approach, they are not able to dynamically adapt the amount of non-determinism and are restricted to non-determinism due to relaxed memory access.

Enforcement of (partly) deterministic scheduling Synchronization synthesis, for example presented by Gupta et al. [GHR+15], automatically inserts locks and other synchronization primitives that are more powerful than fences in that also scheduler-related non-determinism can be eliminated. In contrast to our approach of generating scheduling constraints, their synthesis cannot enforce arbitrary scheduling constraints generated by a verifier. Additionally, their approach may introduce deadlocks into a program [GHR+15], hence it is unsuitable for our model checking approach, where we have to rely on the fact that a verified schedule does not limit the program's functionality.

Deterministic multi-threading (DMT) [AWHF10, BAD+10, BCG13, CWG+11, CSL+13, LCB11, MAB12, OAA09] limits the amount of non-determinism due to scheduling for multi-threaded programs. Dthreads by Liu et al. [LCB11] adapts the interface of the multi-threading library Pthreads and guarantees, for any given input, a deterministic execution. Dthreads interleaves parallel phases (in which threads write only to a local copy of the shared memory) and sequential phases (in which the local copies are merged). Dthreads cannot handle programs that bypass the Pthreads library by synchronizing directly over shared memory [LCB11]. Such coarse-grained schedules that are based on ordering constraints between synchronization primitives (locks, for instance), sometimes called *sync-schedules*, can be enforced with a moderate execution time overhead, as demonstrated by Dthreads. However, determinism is only guaranteed if a program is known to be race-free, i.e., does not contain shared memory accesses without explicit synchronization. Since race conditions are a common defect in concurrent programs, it is interesting to enforce determinism outside of explicit synchronization as well. Potentially racy programs can be scheduled deterministically with fine-grained schedules on individual memory accesses, sometimes called *mem-schedules*, however, with a considerably higher execution time overhead []. Our program schedules are an instance of the latter class, hence executions are deterministic even if a program is not race-free.

Cui et al. propose Peregrine [CWG+11], which reduces the high execution time overhead of mem-schedules while scheduling even racy programs deterministically. *Hybrid schedules* are a combination of sync-schedules and mem-schedules and use coarse-grained scheduling of synchronization primitives when possible. Our implementation of IRS does not make use of such an optimization but can be easily extended to avoid instrumentation in program fragments that are known to be protected by synchronization. Our program schedules do already contain the information that events protected by synchronization do not need additional scheduling, as this information is obtained from the dependency analysis of POR. Optimizing our implementation and reducing the amount of instrumentation is likely to further reduce the execution time overhead of IRS.

In the subsequently presented Parrot framework [CSL$^+$13], Cui et al. propose to combine DMT with a model checker for bug-finding. Parts of a program that are manually marked as performance-critical are executed non-deterministically and model checked to increase the confidence about their correctness. Only the remaining parts of the program are executed deterministically, so that the overhead of additional synchronization is reduced.

Instead of creating schedules ad hoc, as done by Dthreads, *Schedule memoization* [CWTY10] records an initial set of executions and enforces schedules of these initial executions during subsequent executions where these schedules are compatible. Schedules may be incompatible if an input is seen that leads to a different schedule, in which case a new schedule is created ad hoc. Cui et al. argue that using similar schedules for similar inputs is more valuable than completely deterministic schedules which may vary greatly between two similar inputs. Our approach of IMC generates schedules via model checking and guarantees that all possible inputs are covered. Hence, the stability of schedule memoization can be provided by IMC as well, by using the same heuristic for all inputs.

In contrast to IRS, the above described DMT approaches do not provide any guarantees about *which* schedule is enforced, for a particular input. Using these approaches to simplify program verification is therefore impractical if many program inputs need to be covered. While we conjecture that some of the former techniques can be extended to communicate a general scheduling policy that guides a verifier, it is not directly clear how to do so. In contrast, IRS provides a formal interface that uses admissible traces to communicate scheduling constraints. Additionally, the above described DMT approaches do not allow to relax scheduling constraints during runtime, in contrast to IRS, which enables to iteratively relax scheduling constraints and, provided that the program is eventually proven safe, remove all scheduling constraints. On the implementation level, the approaches of [OAA09, LCB11, CSL$^+$13] (but not [BAD$^+$10, CWG$^+$11]) synchronize only at library calls (such as uses of Pthreads locks), which improves execution time performance but may result in non-deterministic executions when global memory is accessed (perhaps accidentally) directly, e.g., without lock protection. In contrast, our IRS implementation schedules all accesses to shared variables, which we consider to be important, as the task of verification is to guaranty a safe executions without the assumption that all memory accesses are protected by synchronization.

We are aware of only one DMT approach that supports symbolic inputs [BCG13]. Similar to our *sections*, *bounded epochs* describe infinite schedules as permutations of finite schedules. Via symbolic execution, an *input-covering* set of schedules is generated, which contains a schedule for each permutation of bounded epochs. As all permutations need to be analyzed (even if they are infeasible), state space explosion through concurrency is only partially avoided; indeed, the experimental evaluation shows that the analysis is infeasible even for five threads when the program has many such permutations. In contrast, we do not require race-freedom, use model checking, sections may contain multiple threads, omit infeasible schedules, and allow a safe execution from the first schedule on, i.e., an IVR can be considerably smaller than an input-covering set of schedules.

Deterministic concurrency requires a program to be deterministic regardless of scheduling. In [RVY13], a deterministic variant of a concurrent program is synthesized based on constraints on conflicts learned by abstract interpretation. In contrast to many DMT approaches, symbolic inputs are supported, however, no verification of general safety properties is done and the degree of non-determinism is not adjustable, in contrast to IVRs.

Chapter 7

Conclusion

This thesis presents a state space exploration algorithm for POR that improves the efficiency of dependency checks by eager generation of schedules. Furthermore, we investigate iterative model checking; we show how to generate incomplete verification results (IVRs) that guarantee safe executions for a subset of possible schedulings. A framework to extract safe schedules from IVRs and enforce such schedules during the execution of concurrent programs is presented in Chapter 3. We show how deterministic fragments of programs and their executions, named *sections* can be identified, and use them both for eager schedule creation in our state space exploration algorithm and to construct schedules from IVRs for infinite executions.

Our POR algorithm, EPOR, eagerly creates schedules for sections, i.e., program fragments. In comparison to known dynamic POR algorithms, it avoids redundant race and dependency checks. Our experiments compare our algorithm to the most efficient POR algorithm we are aware of, SDPOR, and show that EPOR runs considerably faster than SDPOR, which allows in several cases to analyze programs with a higher number of threads within a given timeout.

In Chapter 4, we present a formal framework for using IVRs to extract safe schedulers. We discuss why it is legitimate to constrain scheduling (in contrast to inputs) and formulate general requirements a model checker has to satisfy in our framework. Executions under the scheduling constraints of an IVR are safe, deadlock-free, and fair. We instantiate our framework with the IMPACT model checking algorithm and find in our evaluation that it can be used to model check programs that are intractable for monolithic model checkers, synthesize synchronization via assume statements, and guarantee fair executions.

Iteratively Relaxed Scheduling (cf. Chapter 5) enables to enforce the scheduling constraints of an

IVR. We discuss how to extract and encode schedules from IVRs, for both finite and infinite executions, and how to efficiently implement the enforcement of scheduling constraints, both in terms of reducing the time to look up permission of executing the next event and executing independent events concurrently (by applying POR).

A drawback of enforcing IVRs is a potential execution time overhead, however, in several cases, constrained executions turned out to be even faster than unconstrained executions. Our experimental results show that iteratively relaxing a schedule can reduce execution time overhead. Thereby, we give evidence that IRS indeed allows to adjust the incurred execution time overhead in order to find a sweet spot with respect to the amount of effort for creating schedules (i.e., the duration of verification). Interestingly, we found cases in which a much earlier reduction of execution time overhead is obtained by choosing favorable scheduling constraints, which suggests that execution time performance does not simply rely on the number of scheduling constraints but to a large extend also on their structure.

Future directions We deem several aspects of our work worth being explored beyond the scope of this thesis.

Concerning section-based exploration for POR, we use sections of maximal length, i.e., if the current section can be extended by another event such that the section criteria (the section contains no branching event and its successor and no hiding dependency), it is extended. However, not adding an event to a section does not violate correctness. Using sections of variable length allows to choose the next thread to explore more freely. Potentially, this additional flexibility leads to a faster exploration and there exists a tradeoff between long sections and flexibility in choosing the next thread.

We expect that IMC and IRS can be especially useful to allow the use of concurrent programs in safety-critical situations that currently allow only sequential programs because concurrent programs with conventional, non-deterministic scheduling are too complex to be verified. Hence, an interesting question is to evaluate IMC and IRS in such a scenario and investigate the amount of additional concurrency and reduced execution time in comparison to a sequential program.

Our setup for evaluating IMC can be improved by extending the C programming language subset that is supported by the model checker. Furthermore, the model checker could be extended to use interpolants other than weakest precondition interpolants. With this optimization, interpolants are expected to be found faster and hence, fair cycles could be generated faster.

As described in Section 5.1.5, we tested an IRS implementation that schedules threads directly from within the kernel. However, system calls from the threads of the scheduled program to the kernel module are necessary to notify the IRS scheduler of the current program counter values of threads. An IRS scheduler that obtains this information directly, without system calls, is expected to incur a considerably lower execution time overhead.

A further improvement of IRS, similar to *hybrid schedules* [CWG$^+$11], is conceivable to reduce the execution time overhead for programs with existing synchronization such as locks. Whenever an access of shared memory is statically known to be never required to wait for an other memory access, its instrumentation can be omitted. For example, memory accesses between a lock acquire and a lock release operation of the same lock may be amenable to this optimization.

Bibliography

[AAC$^+$12] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 204–219. Springer, 2012.

[AAC$^+$13] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 530–536. Springer, 2013.

[AAJS14] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014.

[AWHF10] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 193–206. USENIX Association, 2010.

[BAD$^+$10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In James C. Hoe and Vikram S. Adve, editors, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 53–64. ACM, 2010.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCG13] Tom Bergan, Luis Ceze, and Dan Grossman. Input-covering schedules for multithreaded programs. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 677–692. ACM, 2013.

[BHKW12] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: a technique to pass information between verifiers. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, SIGSOFT/FSE, Cary, NC, USA - November 11 - 16, 2012*, page 57. ACM, 2012.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[BKSS11] Péter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. Supporting domain-specific state space reductions through local partial-order reduction. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering, ASE, Lawrence, KS, USA, November 6-10, 2011*, pages 113–122. IEEE Computer Society, 2011.

[BM08] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 107–120. Springer, 2008.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[BS93] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 1993.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CF11] Lucas C. Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 331–340, 2011.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking.* MIT Press, 2001.

[CGS10] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic*, 12(1):7:1–7:54, 2010.

[CKL+05] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone, and Yosinori Watanabe. Quasi-static scheduling of independent tasks for reactive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1492–1514, 2005.

[Cra57] William Craig. Linear reasoning. a new form of the herbrand-genzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[CSL+13] Heming Cui, Jirí Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP, Farmington, PA, USA, November 3-6, 2013*, pages 388–405. ACM, 2013.

[CWG+11] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP, Cascais, Portugal, October 23-26, 2011*, pages 337–351. ACM, 2011.

[CWTY10] Heming Cui, Jingyue Wu, Chia-che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 207–221. USENIX Association, 2010.

[DGTY10] Phillipe Darondeau, Blaise Genest, P. S. Thiagarajan, and Shaofa Yang. Quasi-static scheduling of communicating tasks. *Information and Computation*, 208(10):1154–1168, 2010.

[FFQ02] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Long Beach, California, USA, January 12-14, 2005, pages 110–121. ACM, 2005.

[FIP13] Bernd Fischer, Omar Inverso, and Gennaro Parlato. Cseq: A concurrency pre-processor for sequential C verification tools. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE, Silicon Valley, CA, USA, November 11-15, 2013*, pages 710–713. IEEE, 2013.

[FLM03] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In Utpal Banerjee, Kyle Gallivan, and Antonio González, editors, *Proceedings of the 17th Annual International Conference on Supercomputing, ICS, San Francisco, CA, USA, June 23-26, 2003*, pages 285–294. ACM, 2003.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In Dragan Bosnacki and Stefan Edelkamp, editors, *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, volume 4595 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2007.

[GHR$^+$15] Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. Succinct representation of concurrent trace sets. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Mumbai, India, January 15-17, 2015*, pages 433–444. ACM, 2015.

[GLSW17] Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, pages 246–265. Springer, 2017.

[GLW16] Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna verification tool: IC3 for parallel software - (competition contribution). In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 954–957. Springer, 2016.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.

[God95] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* PhD thesis, Université de Liège, 1995.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science.* Springer, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 174–186. ACM Press, 1997.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer, 1993.

[HJM04] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI, Washington, DC, USA, June 9-11, 2004*, pages 1–13. ACM, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Portland, OR, USA, January 16-18, 2002*, pages 58–70. ACM, 2002.

[ITF$^+$14] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International*

Conference, CAV, Held as Part of the Vienna Summer of Logic, VSL, Vienna, Austria, July 18-22, 2014. Proceedings, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2014.

[KSH12] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Using unfoldings in automated testing of multithreaded programs. In Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors, *IEEE/ACM International Conference on Automated Software Engineering, ASE, Essen, Germany, September 3-7, 2012*, pages 150–159. ACM, 2012.

[KSH15] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Unfolding based automated testing of multithreaded programs. *Automated Software Engineering*, 22(4):475–515, 2015.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ., 1994.

[KVY10] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD, Lugano, Switzerland, October 20-23*, pages 111–119. IEEE, 2010.

[KW11] Daniel Kroening and Georg Weissenbacher. Interpolation-based software verification with wolverine. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 573–578. Springer, 2011.

[KWG09] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2009.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP, Cascais, Portugal, October 23-26, 2011*, pages 327–336. ACM, 2011.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the 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 David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.

[LLV] The LLVM compiler infrastructure. `https://llvm.org`, accessed 23 June 2020.

[LM87] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[LMP09] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2009.

[LPJ$^+$96] Woohyuk Lee, Abelardo Pardo, Jae-Young Jang, Gary D. Hachtel, and Fabio Somenzi. Tearing based automatic abstraction for CTL model checking. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD, San Jose, CA, USA, November 10-14, 1996*, pages 76–81. IEEE Computer Society / ACM, 1996.

[LR09] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[LW13] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in PSO memory systems. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2013.

[MAB12] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Detlock: Portable and efficient deterministic execution for shared memory multicore systems. In *2012 SC Companion: High Performance*

Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012, pages 721–730. IEEE Computer Society, 2012.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. *edited version, originally published in the Proceedings of the International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, October 1988*, 1989.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.

[Maz95] Antoni W. Mazurkiewicz. Introduction to trace theory. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995.

[McM92] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1992.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[Met20] Patrick Metzler. Verification and enforcement of safe schedules for concurrent programs – software material. `https://doi.org/10.5281/zenodo.3992500`, 2020.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI, San Diego, California, USA, June 10-13, 2007*, pages 446–455. ACM, 2007.

[MSB$^+$16] Patrick Metzler, Habib Saissi, Péter Bokor, Robin Hesse, and Neeraj Suri. Efficient verification of program fragments: Eager POR. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 375–391, 2016.

[MSBS17] Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri. Quick verification of concurrent programs by iteratively relaxed scheduling. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE, Urbana, IL, USA, October 30 - November 03, 2017*, pages 776–781. IEEE Computer Society, 2017.

[MSBS18] Patrick Metzler, Habib Saissi, Péter Bokor, and Neeraj Suri. Safe execution of concurrent programs by enforcement of scheduling constraints, updated on april 14, 2020. *CoRR*, abs/1809.01955, `https://arxiv.org/abs/1809.01955`, 2018.

[MSW19] Patrick Metzler, Neeraj Suri, and Georg Weissenbacher. Extracting safe thread schedules from incomplete model checking results. In Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay, editors, *Model Checking Software - 26th International Symposium, SPIN, Beijing, China, July 15-16, 2019, Proceedings*, volume 11636 of *Lecture Notes in Computer Science*, pages 153–171. Springer, 2019.

[MSW20] Patrick Metzler, Neeraj Suri, and Georg Weissenbacher. Extracting safe thread schedules from incomplete model checking results. *International Journal on Software Tools for Technology Transfer, STTT*, 22:565–581, 2020.

[NFLP16] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 174–191, 2016.

[NSF$^+$17] Truc L. Nguyen, Peter Schrammel, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Parallel bug-finding in concurrent programs via reduced interleaving instances. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE, Urbana, IL, USA, October 30 - November 03, 2017*, pages 753–764. IEEE Computer Society, 2017.

[OAA09] Marek Olszewski, Jason Ansel, and Saman P. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, Washington, DC, USA, March 7-11, 2009*, pages 97–108, 2009.

[Par97] Abelardo Pardo. *Automatic Abstraction Techniques For Formal Verification Of Digital Systems*. PhD thesis, Dept. of Computer Science, University of Colorado at Boulder, 1997.

[Pel93] Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[PH98] Abelardo Pardo and Gary D. Hachtel. Incremental CTL model checking using BDD subsetting. In Basant R. Chawla, Randal E. Bryant, and Jan M. Rabaey, editors, *Proceedings of the 35th Conference on Design Automation, Moscone center, San Francico, California, USA, June 15-19, 1998.*, pages 457–462. ACM Press, 1998.

[PR94] Gara Pruesse and Frank Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, 1994.

[PW98] Lutz Priese and Harro Wimmel. A uniform approach to true-concurrency and interleaving semantics for petri nets. *Theoretical Compututer Science*, 206(1-2):219–256, 1998.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[QW04] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI, Washington, DC, USA, June 9-11, 2004*, pages 14–24. ACM, 2004.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

[RG05] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.

[RSSK15] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 456–469. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[RVY13] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Automatic synthesis of deterministic concurrency. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2013.

[SRDK17] Marcelo Sousa, César Rodríguez, Vijay D'Silva, and Daniel Kroening. Abstract interpretation with unfoldings. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2017.

[SVC] Benchmark suite of the competition on software verification (SV-COMP). `https://github.com/sosy-lab/sv-benchmarks`, accessed 23 June 2020.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[Val96] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996.

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design, FMCAD, Portland, OR, USA, October 20-23, 2013*, pages 210–217. IEEE, 2013.

[ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, PLDI, OR, USA, June 15-17, 2015*, pages 250–259. ACM, 2015.

Appendix A

Eager POR: Detailed Experimental Results

The following table shows our complete experiment results for detailed reference. All benchmarks are parametric, where the parameter specifies the number of threads, except for the Shared Pointer benchmark, where it specifies the number of loop iterations. EPOR and EPOR-SH refer to our algorithm with sections as defined in Section 3.2.2 and short sections as defined in 3.3. Column *Unsat. TCS* refers to the number of unsatisfiable trace constraint systems generated by EPOR and EPOR-SH; column *Speedup* refers to the percentage-wise time saving over SDPOR.

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Readers-Writers (2)	SDPOR	0.001	2	3	2	0	0
Readers-Writers (2)	EPOR-SH	0.001	2	2	1	0	0.0
Readers-Writers (2)	EPOR	0.001	2	2	1	0	0.0
Readers-Writers (3)	SDPOR	0.002	4	28	12	0	0
Readers-Writers (3)	EPOR-SH	0.002	4	10	3	0	0.0
Readers-Writers (3)	EPOR	0.002	4	10	3	0	0.0
Readers-Writers (4)	SDPOR	0.005	8	148	47	0	0
Readers-Writers (4)	EPOR-SH	0.005	8	33	6	0	0.0
Readers-Writers (4)	EPOR	0.005	8	33	6	0	0.0
Readers-Writers (5)	SDPOR	0.015	16	607	153	0	0
Readers-Writers (5)	EPOR-SH	0.012	16	92	10	0	20.0
Readers-Writers (5)	EPOR	0.012	16	92	10	0	20.0
Readers-Writers (6)	SDPOR	0.041	32	2155	449	0	0
Readers-Writers (6)	EPOR-SH	0.030	32	236	15	0	26.8
Readers-Writers (6)	EPOR	0.030	32	236	15	0	26.8

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Readers-Writers (7)	SDPOR	0.109	64	6969	1233	0	0
Readers-Writers (7)	EPOR-SH	0.072	64	578	21	0	33.9
Readers-Writers (7)	EPOR	0.072	64	578	21	0	33.9
Readers-Writers (8)	SDPOR	0.274	128	21107	3233	0	0
Readers-Writers (8)	EPOR-SH	0.172	128	1375	28	0	37.2
Readers-Writers (8)	EPOR	0.170	128	1375	28	0	38.0
Readers-Writers (9)	SDPOR	0.668	256	60885	8193	0	0
Readers-Writers (9)	EPOR-SH	0.403	256	3204	36	0	39.7
Readers-Writers (9)	EPOR	0.400	256	3204	36	0	40.1
Readers-Writers (10)	SDPOR	1.627	512	169111	20225	0	0
Readers-Writers (10)	EPOR-SH	0.934	512	7346	45	0	42.6
Readers-Writers (10)	EPOR	0.936	512	7346	45	0	42.5
Readers-Writers (11)	SDPOR	3.907	1024	455705	48897	0	0
Readers-Writers (11)	EPOR-SH	2.145	1024	16618	55	0	45.1
Readers-Writers (11)	EPOR	2.125	1024	16618	55	0	45.6
Readers-Writers (12)	SDPOR	9.231	2048	1197851	116225	0	0
Readers-Writers (12)	EPOR-SH	4.853	2048	37165	66	0	47.4
Readers-Writers (12)	EPOR	4.799	2048	37165	66	0	48.0
Readers-Writers (13)	SDPOR	21.675	4096	3083805	272385	0	0
Readers-Writers (13)	EPOR-SH	10.840	4096	82300	78	0	50.0
Readers-Writers (13)	EPOR	10.741	4096	82300	78	0	50.4
Readers-Writers (14)	SDPOR	50.985	8192	7799839	630785	0	0
Readers-Writers (14)	EPOR-SH	24.221	8192	180696	91	0	52.5
Readers-Writers (14)	EPOR	24.299	8192	180696	91	0	52.3
Readers-Writers (15)	SDPOR	116.479	16384	19429409	1445889	0	0
Readers-Writers (15)	EPOR-SH	54.318	16384	393794	105	0	53.4
Readers-Writers (15)	EPOR	54.015	16384	393794	105	0	53.6
Readers-Writers (16)	SDPOR	268.414	32768	47759395	3284993	0	0
Readers-Writers (16)	EPOR-SH	121.130	32768	852667	120	0	54.9
Readers-Writers (16)	EPOR	119.901	32768	852667	120	0	55.3
Readers-Writers (17)	SDPOR	608.308	65536	116031525	7405569	0	0
Readers-Writers (17)	EPOR-SH	264.130	65536	1835844	136	0	56.6
Readers-Writers (17)	EPOR	262.993	65536	1835844	136	0	56.8
Readers-Writers (18)	SDPOR	1361.840	131072	278986791	16580609	0	0
Readers-Writers (18)	EPOR-SH	582.379	131072	3933150	153	0	57.2
Readers-Writers (18)	EPOR	579.521	131072	3933150	153	0	57.4
Readers-Writers (19)	SDPOR	3076.191	262144	664600617	36896769	0	0
Readers-Writers (19)	EPOR-SH	1264.264	262144	8389770	171	0	58.9
Readers-Writers (19)	EPOR	1256.383	262144	8389770	171	0	59.2
Readers-Writers (20)	SDPOR	6874.472	524288	1570045995	81657857	0	0
Readers-Writers (20)	EPOR-SH	2738.353	524288	17827145	190	0	60.2
Readers-Writers (20)	EPOR	2728.742	524288	17827145	190	0	60.3
Indexer (11)	SDPOR	0.015	1	880	946	0	0
Indexer (11)	EPOR-SH	0.025	1	880	946	0	-66.7
Indexer (11)	EPOR	0.026	1	880	946	0	-73.3

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Indexer (12)	SDPOR	0.413	8	27072	12825	0	0
Indexer (12)	EPOR-SH	0.274	8	19325	7961	0	33.7
Indexer (12)	EPOR	0.284	8	19325	7961	0	31.2
Indexer (13)	SDPOR	4.181	64	485600	106214	0	0
Indexer (13)	EPOR-SH	3.367	64	239590	74980	0	19.5
Indexer (13)	EPOR	3.506	64	239590	74980	0	16.1
Indexer (14)	SDPOR	49.120	512	5279831	1177634	0	0
Indexer (14)	EPOR-SH	42.644	512	2812237	795788	0	13.2
Indexer (14)	EPOR	44.144	512	2812237	795788	0	10.1
Indexer (15)	SDPOR	766.280	4096	79436769	16007293	0	0
Indexer (15)	EPOR-SH	556.283	4096	35103635	9347279	0	27.4
Indexer (15)	EPOR	576.093	4096	35103635	9347279	0	24.8
Indexer (16)	SDPOR	13060.033	32768	1345407904	251890633	0	0
Indexer (16)	EPOR-SH	7485.608	32805	466384458	116349641	0	42.7
Indexer (16)	EPOR	7998.984	32805	466384458	116349641	0	38.8
Last Zero (2)	SDPOR	0.002	2	9	13	0	0
Last Zero (2)	EPOR-SH	0.003	2	13	13	0	-50.0
Last Zero (2)	EPOR	0.003	2	8	10	0	-50.0
Last Zero (3)	SDPOR	0.013	6	197	128	0	0
Last Zero (3)	EPOR-SH	0.024	6	125	116	0	-84.6
Last Zero (3)	EPOR	0.012	6	80	84	0	7.7
Last Zero (4)	SDPOR	0.068	16	2065	709	0	0
Last Zero (4)	EPOR-SH	0.044	16	800	579	0	35.3
Last Zero (4)	EPOR	0.070	16	676	479	0	-2.9
Last Zero (5)	SDPOR	0.255	40	13613	2791	0	0
Last Zero (5)	EPOR-SH	0.173	40	4279	2371	0	32.2
Last Zero (5)	EPOR	0.195	40	4976	2120	0	23.5
Last Zero (6)	SDPOR	0.911	96	66384	10275	0	0
Last Zero (6)	EPOR-SH	0.633	96	19645	8480	0	30.5
Last Zero (6)	EPOR	0.724	96	29570	7885	0	20.5
Last Zero (7)	SDPOR	3.018	224	274999	33881	0	0
Last Zero (7)	EPOR-SH	2.142	224	79578	27720	0	29.0
Last Zero (7)	EPOR	2.517	224	147844	26234	0	16.6
Last Zero (8)	SDPOR	9.206	512	1109904	97439	0	0
Last Zero (8)	EPOR-SH	6.975	512	294877	85185	0	24.2
Last Zero (8)	EPOR	8.339	512	647298	80647	0	9.4
Last Zero (9)	SDPOR	22.350	1152	3836659	306046	0	0
Last Zero (9)	EPOR-SH	33.547	1152	1464128	314042	0	-50.1
Last Zero (9)	EPOR	33.950	1152	2884130	310058	0	-51.9
Last Zero (10)	SDPOR	108.007	2560	15149844	1160330	0	0
Last Zero (10)	EPOR-SH	94.648	2560	5405445	923038	0	12.4
Last Zero (10)	EPOR	95.582	2578	11544604	1015493	0	11.5
Last Zero (11)	SDPOR	264.036	5632	51558504	3325567	0	0
Last Zero (11)	EPOR-SH	197.799	5632	16019928	2410338	0	25.1
Last Zero (11)	EPOR	257.922	5632	40368624	2649056	0	2.3

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Last Zero (12)	SDPOR	821.374	12288	175535648	9951180	0	0
Last Zero (12)	EPOR-SH	480.859	12288	41678637	5885987	0	41.5
Last Zero (12)	EPOR	705.437	12288	125302898	5950551	0	14.1
Last Zero (13)	SDPOR	2160.776	26624	565002531	29044732	0	0
Last Zero (13)	EPOR-SH	1361.417	26624	111575184	14917085	0	37.0
Last Zero (13)	EPOR	1441.852	26624	347226642	11989526	0	33.3
Last Zero (14)	SDPOR	8138.822	57344	1744754931	78289802	0	0
Last Zero (14)	EPOR-SH	3372.409	57344	300987594	37479306	0	58.6
Last Zero (14)	EPOR	3421.276	57344	1005154306	29966707	0	58.0
Last Zero (15)	SDPOR	17441.597	122880	4019531983	230194076	0	0
Last Zero (15)	EPOR-SH	6026.374	122880	514821851	93547034	0	65.4
Last Zero (15)	EPOR	6703.371	122880	1896719286	73740996	0	61.6
Last Zero (16)	SDPOR						
Last Zero (16)	EPOR-SH	19144.029	262144	1934932782	239409835	0	—
Last Zero (16)	EPOR	18408.671	262144	7232899654	179027187	0	—
Shared Pointer (10)	SDPOR	0.480	21	80395	32777	0	0
Shared Pointer (10)	EPOR-SH	0.896	21	61207	33655	0	-86.7
Shared Pointer (10)	EPOR	0.535	21	60546	33025	0	-11.5
Shared Pointer (20)	SDPOR	2.123	41	661981	225737	0	0
Shared Pointer (20)	EPOR-SH	4.226	41	528044	229295	0	-99.1
Shared Pointer (20)	EPOR	2.968	41	525351	226835	0	-39.8
Shared Pointer (30)	SDPOR	7.837	61	2374011	722897	0	0
Shared Pointer (30)	EPOR-SH	14.770	61	1932212	730935	0	-88.5
Shared Pointer (30)	EPOR	8.047	61	1923801	725445	0	-2.7
Shared Pointer (40)	SDPOR	17.013	81	6201931	1668257	0	0
Shared Pointer (40)	EPOR-SH	37.533	81	5060976	1682575	0	-120.6
Shared Pointer (40)	EPOR	13.508	81	5042257	1672855	0	20.6
Shared Pointer (50)	SDPOR	32.529	101	14074966	3205817	0	0
Shared Pointer (50)	EPOR-SH	125.372	101	11494347	3228215	0	-285.4
Shared Pointer (50)	EPOR	17.398	101	11459539	3213065	0	46.5
Shared Pointer (60)	SDPOR	52.435	121	27575051	5479577	0	0
Shared Pointer (60)	EPOR-SH	219.720	121	22323086	5511855	0	-319.0
Shared Pointer (60)	EPOR	43.751	121	22263258	5490075	0	16.6
Shared Pointer (70)	SDPOR	84.797	141	49302287	8633537	0	0
Shared Pointer (70)	EPOR-SH	370.194	141	39524860	8677495	0	-336.6
Shared Pointer (70)	EPOR	64.530	141	39430039	8647885	0	23.9
Shared Pointer (80)	SDPOR	84.948	161	83360055	12811697	0	0
Shared Pointer (80)	EPOR-SH	458.459	161	66218755	12869135	0	-439.7
Shared Pointer (80)	EPOR	95.521	161	66076608	12830495	0	-12.4
Shared Pointer (90)	SDPOR	143.694	181	128693768	18158057	0	0
Shared Pointer (90)	EPOR-SH	919.317	181	102871367	18230775	0	-539.8
Shared Pointer (90)	EPOR	132.781	181	102676446	18181905	0	7.6
Shared Pointer (100)	SDPOR	238.968	201	192707828	24816617	0	0
Shared Pointer (100)	EPOR-SH	1531.204	201	154847568	24906415	0	-540.8

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Shared Pointer (100)	EPOR	170.762	201	154590222	24846115	0	28.5
Ring (2)	SDPOR	0.002	2	3	2	0	0
Ring (2)	EPOR-SH	0.001	2	2	1	0	50.0
Ring (2)	EPOR	0.001	2	2	1	0	50.0
Ring (3)	SDPOR	0.008	6	39	18	0	0
Ring (3)	EPOR-SH	0.005	6	11	3	2	37.5
Ring (3)	EPOR	0.005	6	11	3	2	37.5
Ring (4)	SDPOR	0.018	14	247	80	0	0
Ring (4)	EPOR-SH	0.022	14	43	6	2	-22.2
Ring (4)	EPOR	0.017	14	43	6	2	5.6
Ring (5)	SDPOR	0.064	30	1231	275	0	0
Ring (5)	EPOR-SH	0.045	30	139	10	2	29.7
Ring (5)	EPOR	0.047	30	139	10	2	26.6
Ring (6)	SDPOR	0.168	62	4932	813	0	0
Ring (6)	EPOR-SH	0.118	62	397	15	2	29.8
Ring (6)	EPOR	0.121	62	397	15	2	28.0
Ring (7)	SDPOR	0.459	126	17742	2283	0	0
Ring (7)	EPOR-SH	0.226	126	1038	21	2	50.8
Ring (7)	EPOR	0.298	126	1038	21	2	35.1
Ring (8)	SDPOR	1.297	254	59947	6275	0	0
Ring (8)	EPOR-SH	0.382	254	2540	28	2	70.5
Ring (8)	EPOR	0.710	254	2540	28	2	45.3
Ring (9)	SDPOR	3.530	510	191381	17288	0	0
Ring (9)	EPOR-SH	0.877	510	5577	36	2	75.2
Ring (9)	EPOR	1.635	510	5577	36	2	53.7
Ring (10)	SDPOR	8.967	1022	543438	44107	0	0
Ring (10)	EPOR-SH	3.418	1022	12281	45	2	61.9
Ring (10)	EPOR	2.919	1022	12281	45	2	67.4
Ring (11)	SDPOR	23.903	2046	1551020	116202	0	0
Ring (11)	EPOR-SH	8.452	2046	27769	55	2	64.6
Ring (11)	EPOR	6.020	2046	27769	55	2	74.8
Ring (12)	SDPOR	57.755	4094	4498596	299602	0	0
Ring (12)	EPOR-SH	18.373	4094	61507	66	2	68.2
Ring (12)	EPOR	17.331	4094	61507	66	2	70.0
Ring (13)	SDPOR	153.056	8190	12342751	752788	0	0
Ring (13)	EPOR-SH	34.668	8190	127345	78	2	77.3
Ring (13)	EPOR	40.175	8190	127345	78	2	73.8
Ring (14)	SDPOR	307.406	16382	36655573	2172569	0	0
Ring (14)	EPOR-SH	65.806	16382	261835	91	2	78.6
Ring (14)	EPOR	60.154	16382	261835	91	2	80.4
Ring (15)	SDPOR	731.446	32766	105588804	5623429	0	0
Ring (15)	EPOR-SH	143.513	32766	534423	105	2	80.4
Ring (15)	EPOR	145.635	32766	534423	105	2	80.1
Ring (16)	SDPOR	1782.207	65534	278381118	13318473	0	0
Ring (16)	EPOR-SH	327.465	65534	1084045	120	2	81.6

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Ring (16)	EPOR	327.977	65534	1084045	120	2	81.6
Ring (17)	SDPOR	5984.174	131070	734642101	35656128	0	0
Ring (17)	EPOR-SH	708.740	131070	2096753	136	2	88.2
Ring (17)	EPOR	538.031	131070	2096753	136	2	91.0
Ring (18)	SDPOR						
Ring (18)	EPOR-SH	1542.738	262142	4167297	153	2	—
Ring (18)	EPOR	1062.553	262142	4167297	153	2	—
Ring (19)	SDPOR						
Ring (19)	EPOR-SH	3359.111	524286	8653144	171	2	—
Ring (19)	EPOR	2884.695	524286	8653144	171	2	—
Ring (20)	SDPOR						
Ring (20)	EPOR-SH	4454.283	1048574	9495364	190	2	—
Ring (20)	EPOR	4442.308	1048574	9495364	190	2	—
Ring (21)	SDPOR						
Ring (21)	EPOR-SH	13158.802	2097150	28329284	210	2	—
Ring (21)	EPOR	13084.234	2097150	28329284	210	2	—
Branching (2)	SDPOR	0.009	11	181	155	0	0
Branching (2)	EPOR-SH	0.009	11	174	147	0	0.0
Branching (2)	EPOR	0.008	11	142	124	1	11.1
Branching (3)	SDPOR	0.046	28	3169	1105	0	0
Branching (3)	EPOR-SH	0.055	28	2679	1124	0	-19.6
Branching (3)	EPOR	0.046	28	2206	943	1	0.0
Branching (4)	SDPOR	0.268	103	24945	6933	0	0
Branching (4)	EPOR-SH	0.308	103	21967	6960	0	-14.9
Branching (4)	EPOR	0.233	103	17296	5617	1	13.1
Branching (5)	SDPOR	1.180	311	145186	32384	0	0
Branching (5)	EPOR-SH	1.458	311	143461	34068	0	-23.6
Branching (5)	EPOR	1.045	311	114640	26926	1	11.4
Branching (6)	SDPOR	5.600	1010	796033	155629	0	0
Branching (6)	EPOR-SH	6.679	1010	809098	156745	0	-19.3
Branching (6)	EPOR	4.512	1010	645243	120540	1	19.4
Branching (7)	SDPOR	23.737	3165	3963738	665731	0	0
Branching (7)	EPOR-SH	29.320	3165	4153755	677854	0	-23.5
Branching (7)	EPOR	18.819	3165	3332731	505448	1	20.7
Branching (8)	SDPOR	111.485	10063	19677616	3051999	0	0
Branching (8)	EPOR-SH	124.574	10063	19995225	2827886	0	-11.7
Branching (8)	EPOR	76.783	10063	16091273	2042519	1	31.1
Branching (9)	SDPOR	588.386	31780	102640823	15619776	0	0
Branching (9)	EPOR-SH	835.651	31775	106250930	17043326	0	-42.0
Branching (9)	EPOR	444.051	30921	68635810	11463305	1	24.5
Branching (10)	SDPOR	3107.106	100651	516099474	79852841	0	0
Branching (10)	EPOR-SH	3832.897	100327	530295199	73161559	0	-23.4
Branching (10)	EPOR	1964.219	99920	325828401	48463434	1	36.8
Branching (11)	SDPOR	19068.490	318363	2200202598	358100829	0	0
Branching (11)	EPOR-SH	21970.231	316881	2091377423	284175909	0	-15.2

Continued on next page

Continued from previous page

Benchmark	Algorithm	Time(s)	Traces	Dep. Checks	Race Checks	Unsat. TCS	Speedup(%)
Branching (11)	EPOR	8220.448	318978	1343673801	179170034	1	56.9
Ring Extended (2)	SDPOR	0.003	6	41	34	0	0
Ring Extended (2)	EPOR-SH	0.004	6	38	29	0	-33.3
Ring Extended (2)	EPOR	0.004	6	9	6	10	-33.3
Ring Extended (3)	SDPOR	0.050	90	2264	1029	0	0
Ring Extended (3)	EPOR-SH	0.047	72	1553	663	14	6.0
Ring Extended (3)	EPOR	0.365	90	126	15	4006	-630.0
Ring Extended (4)	SDPOR	0.692	786	44477	14734	0	0
Ring Extended (4)	EPOR-SH	0.737	786	39708	12722	30	-6.5
Ring Extended (4)	EPOR	7.826	786	1632	28	64750	-1030.9
Ring Extended (5)	SDPOR	7.497	5730	631224	156322	0	0
Ring Extended (5)	EPOR-SH	7.754	5730	565678	138590	62	-3.4
Ring Extended (5)	EPOR	164.094	5730	16734	45	1042846	-2088.8
Ring Extended (6)	SDPOR	70.729	38466	7537485	1427204	0	0
Ring Extended (6)	EPOR-SH	72.869	38466	6747840	1285045	126	-3.0
Ring Extended (6)	EPOR	3412.561	38466	144095	66	16738750	-4724.8
Ring Extended (7)	SDPOR	608.836	247170	81503018	11900225	0	0
Ring Extended (7)	EPOR-SH	622.568	247170	72416459	10706749	254	-2.3
Ring Extended (7)	EPOR						
Ring Extended (8)	SDPOR	6552.194	1548546	806537903	94539059	0	0
Ring Extended (8)	EPOR-SH	5061.882	1548546	720212287	83761394	510	22.7
Ring Extended (8)	EPOR						

Appendix B

IMC: Additional Material

B.1 Enabled threads

An ART $\mathscr{A}$ may contain an edge $v \xrightarrow{T,R} w$ such that transition R is enabled in some state $s \vDash \phi(v) \wedge \mathfrak{l}()(s) = \mathfrak{l}()(v)$ but disabled in some state $s' \vDash \phi(v) \wedge \mathfrak{l}()(s) = \mathfrak{l}()(v)$. This may pose a problem when the goal is to construct an ART that admits fairness, as Definition 28 requires an edge $v \xrightarrow{T,R} w$ that is enabled in all states that correspond to v, for every cycle and thread that is enabled in that cycle. We argue that the situation above cannot occur, even when constructing an ART with the conventional Impact algorithm for concurrent programs [WKO13], if programs make only "reasonable" use of locks, as described below.

We restrict programs such that whether a transition is enabled in a state s may only depend on the global location $\mathfrak{l}(s)$ of s but not on the variable valuation of s. Formally:

$$\forall T \in \mathscr{T}. \exists f : (\mathscr{T} \times L) \to \{0,1\}. \forall s. (\textit{Next-Transition}(s,T) \neq \bot \Leftrightarrow f(\mathfrak{l}(s),T) = 1)$$

For such programs, every transition R with an edge $v \xrightarrow{T,R} w$ in a well-labeled ART is trivially enabled in all states $s \vDash \phi(v) \wedge \mathfrak{l}()(s) = \mathfrak{l}()(v)$. In the following, we argue that such programs are sufficient to express "reasonable" uses of locks.

We assume that there exists a synchronization primitive $\mathsf{lock(l)}$ that acquires the lock l if it is free and otherwise lets the executing thread wait until l is free. A thread is *disabled* when its next statement is $\mathsf{lock(l)}$ for a lock l that is not free. Furthermore, we assume that lock is the only primitive in the targeted programming language that can disable threads (other synchronization constructs can be built using lock).

Consider a program P that, for every lock statement stmt that occurs in P, always executes stmt with the same lock. P satisfies (B.1). On the other hand, consider a program P' that maintains an array of locks locks = [l_1, l_2, \dotsc, l_n] and contains a statement lock(locks[*]) that tries to acquire a non-deterministically chosen lock. P' does not satisfy (B.1).

A pattern that violates (B.1) may be translated so that a unique lock is used at a given program location as follows. A program fragment (where l is a local variable)

```
l = locks[*];
lock(l);
critical_section();
unlock(l);
```

is translated to:

```
l = locks[*];
switch l:
  case l_1:
    lock(l_1);
    critical_section_1();
    unlock(l_1);
  case l_2:
    lock(l_2);
    critical_section_2();
    unlock(l_2);
  ...
  case l_n:
    lock(l_n);
    critical_section_n();
    unlock(l_n);
```

This transformation leads to a linear blow up in program size. However, we assume that practical programs which violate (B.1) are rare and call programs on which above transformation does not critically increase program size *programs with a "reasonable" use of locks.* For such programs, an ART that is an incomplete product of the conventional Impact algorithm for concurrent programs can be easily extended to an ART that admits fairness.

B.2 Auxiliary lemmas

The following lemma is used in the proofs of Lemma 2 and 3. It states that for every node v of a finite graph that is visited infinitely often in a path, this path also visits infinitely often all nodes of a cycle that contains v.

Lemma 5 (Completely visited cycles). *Let $G = (V, \rightarrow)$ be a directed, finite graph. For all infinite paths $\pi \in V^\omega$ through G and for all nodes $v \in V$ that occur infinitely often in π, there exists a projection $\pi' \subseteq \pi$ such that $\pi' = \pi_v^\omega$ and π_v is a cycle that contains v, i.e., there exists a cycle π_v in G that contains v such that by removing nodes from π, we obtain a path $\pi' = \pi_v^\omega$ that visits all nodes of π_v infinitely often.*

Proof. Let G, π, v be as in the lemma. π has the form $\pi_0 \circ v \circ \pi_1 \circ v \circ \pi_2 \circ v \cdots$ with $v \notin \pi_i, i \geq 0$. For all $i \geq 1$, $v \circ \pi_i \circ v$ is a closed walk, which can be shortened to a cycle $v \circ \pi_i' \circ v$. As there are only finitely many cycles in G (V is finite), there exists a cycle $v \circ \pi_{i_1}' \circ v$ that is repeated infinitely often in the sequence $\pi_0 \circ v \circ \pi_1' \circ v \circ \pi_2' \circ v \cdots$, i.e., $\pi_{i_1}' = \pi_{i_2}' = \pi_{i_3}' = \cdots$ for an infinite sequence of indices $i_1 < i_2 < i_3 < \cdots$. Let $\pi_v = v \circ \pi_{i_1}$. We have that $\pi' = \pi_v^\omega$ is a projection of π and π_v is a cycle that contains v. □

B.3 Transformation of fair ARTs to independently fair ARTs

Given a fair, well-labeled, safe ART $\mathscr{A}$, Algorithm 6 generates an independently fair ART $\mathscr{A}'$ such that $\mathit{Executions}(\mathcal{R}_{\mathscr{A}'}) \subseteq \mathit{Executions}(\mathcal{R}_{\mathscr{A}})$.

Algorithm 6: Transformation of fair ARTs to independently fair ARTs

```
input  : fair ART $\mathscr{A}$
output : independently-fair ART $\mathscr{A}'$ with $\mathit{Executions}(\mathcal{R}_{\mathscr{A}'}) \subseteq \mathit{Executions}(\mathcal{R}_{\mathscr{A}})$

Data: $\mathscr{A}' := \emptyset$, $W := \epsilon$

while $\exists v \in W$ do
    remove $v$ from $W$
    if $v$ can be independently-fair covered by some node $w \in \mathscr{A}'$ then
        add $\{v \rhd w\}$ to $\mathscr{A}'$
        continue
    else if $v$ is part of a $(\rhd_{\mathscr{A}} \cup \rightarrow_{\mathscr{A}})$-cycle $v \dots wv$ that is not independently fair then
        add $\{v \rightarrow \dots \rightarrow w\}$ as fresh nodes to $\mathscr{A}'$
        set $v$ to an exit node of the cycle that has not yet been expanded
        add $\{w \rightarrow \dots \rightarrow v\}$ to $\mathscr{A}'$
    add $\{v \rightarrow v' : v \rightarrow_{\mathscr{A}} v'\}$ to $\mathscr{A}'$
    add $\{v \rhd v' : v \rhd_{\mathscr{A}} v'\}$ to $\mathscr{A}'$
    add $\{v' : v \rightarrow_{\mathscr{A}} v' \vee v \rhd_{\mathscr{A}} v'\}$ to $W$
```

B.4 Iterative Impact for concurrent programs

This section provides additional details for our iterative model checking algorithm presented in Section 4.4.

Algorithm 7: Iterative Impact for concurrent programs (additional functions)

```
continued :
Function Refine(v)
    if $\mathfrak{l}(v) \neq \mathfrak{l}_{error}$ or $\phi(v) \equiv false$ then
        return
    $\pi := v_0, \ldots, v_n$ path from $\epsilon$ to $v$
    if path formula of $\pi$ has interpolant $A_0, \ldots, A_n$ then
        for $i = 0 \ldots n$ do
            $\phi := A_i^{-i}$
            if $\phi(v_i) \nvDash \phi$ then
                $W := W \cup \{w : w \rhd v_i\}$
                $\rhd := \rhd \setminus \{(w, v_i) : w \rhd v_i\}$
                $\phi(v_i) := \phi(v_i) \wedge \phi$
        for $w \in V$ such that $v$ is a descendant of $w$ do
            Close $(w)$
    else
        return counterexample

Function Expand_Thread(T, v)
    for $R_{l,l'} \in Transitions(\mathfrak{l}_T(v))$ do
        $w :=$ fresh node
        $\mathfrak{l}(w) := \mathfrak{l}(v)[T \mapsto \mathfrak{l}']$
        $\phi(w) :=$ True
        $W := W \cup \{w\}$
        $V := V \cup \{w\}$
        $\rightarrow := \rightarrow \cup \{(v, T, R, w)\}$

Function Skip(v, T)
    if $(v, T) \in I$ then
        return false
    else
        choose unique $T', R'$ such that $u \xrightarrow{T', R'} v$
        return $(T < T' \wedge (Transitions(v)T \parallel \{a'\})) \wedge \neg Loop(u, T')$

Function Schedule_Thread (v)
    let $R_n$ be the transition of thread $T_n$ by which $v$ is reached
    if $R_n$ represents a back jump then
        $T := T_n + 1 \mod |\mathscr{T}|$
    else
        $T := T_n$
    while Skip (v, T) do
        $T := T + 1 \mod |\mathscr{T}|$
    return $T$
```

In order to represent a path of length n as a formula, we define n fresh copies of the set of variable symbols, denoted by $Q_1, \ldots, Q_n$, such that Q_1 is equal to the previously defined copy Q' for transition formulae. The *path formula* of a path $\pi = v_0 \xrightarrow{T_0, R_0}_{\mathscr{A}} \ldots \xrightarrow{T_{n-1}, R_{n-1}}_{\mathscr{A}} v_n$ in an ART $\mathscr{A}$ is the formula $\phi(v_0) \wedge R_0 \wedge R_1^1 \wedge \cdots \wedge R_{n-1}^{n-1} \in \mathscr{F}(Q \cup Q' \cup Q_1 \cup \cdots \cup Q_n)$, where $R_i^i, 1 \leq i \leq (n-1)$ is obtained from $R_i \in \mathscr{F}(Q \cup Q')$ by substituting the variable symbols in Q and Q' with their corresponding copies of Q_i and Q_{i+1}, respectively. We write A^{-i} for some formula A to reverse this substitution, i.e., $A = (A^i)^{-i}$. This notation is used in Algorithm 7 to construct a sequent interpolant $A_0, \ldots, A_n \in \mathscr{F}(Q \cup Q' \cup Q_1 \cup \cdots \cup Q_n)$ for a path formula and extract state formulas $A_i^{-i} \in \mathscr{F}(Q)$.

We use the set I to record cases in which POR would hide a transition $w \xrightarrow{T} w'$ after adding a covering $(v, w) \in \mathit{covering}()$. The original approach by Wachter et al. [WKO13] of immediately expanding a thread after adding such a covering does not suite our iterative variant of the algorithm, as this approach could explore more than one schedule in a single iteration. Instead of immediately exploring $w \xrightarrow{T} w'$, we record this transition in I and prevent the procedure *Skip()* from skipping it (i.e., applying POR).

Appendix C

IRS: Detailed Experimental Results

The following table shows our detailed measurement results. The columns contain the benchmark name (-opt means with optimized trace prefixes), the number of constraints in the respective trace prefix, the mean execution time in μs and the execution time overhead compared to the uninstrumented benchmark version.

Benchmark	Constraints	Time	Overhead%
bigshot	1	124	5%
bigshot	0	121	3%
dekker	2	115	4%
dekker	1	114	3%
dekker	0	113	2%
fibonacci	98	176	13%
fibonacci	44	169	9%
fibonacci	24	181	12%
fibonacci	0	166	6%
lamport	16	123	12%
lamport	15	123	12%
lamport	10	124	13%
lamport	7	124	13%

Continued on next page

Continued from previous page

Benchmark	Constraints	Time	Overhead%
lamport	6	124	13%
lamport	4	123	12%
lamport	2	123	12%
lamport	1	124	13%
lamport	0	113	3%
peterson	28	124	8%
peterson	24	125	9%
peterson	22	122	6%
peterson	1	123	7%
peterson	0	113	-2%
shared pointer	3	135	22%
shared pointer	2	134	21%
shared pointer	1	133	20%
shared pointer	0	115	4%
indexer(15)	12	7538	2692%
indexer(15)	8	7603	2716%
indexer(15)	4	6793	2416%
indexer(15)	3	5412	1904%
indexer(15)	2	435	61%
indexer(15)	1	299	11%
indexer(15)	0	235	-13%
last zero(16)	15	10664	4288%
last zero(16)	8	5286	2075%
last zero(16)	5	492	102%
last zero(16)	1	263	8%
last zero(16)	0	230	-5%
indexer(15)-opt	12	5558	2841%
indexer(15)-opt	9	279	48%

Continued on next page

Continued from previous page

Benchmark	Constraints	Time	Overhead%
indexer(15)-opt	6	257	36%
indexer(15)-opt	0	215	14%
last zero(16)-opt	15	378	94%
last zero(16)-opt	8	269	38%
last zero(16)-opt	5	253	30%
last zero(16)-opt	1	250	28%
last zero(16)-opt	0	223	14%