Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction

PAROSH AZIZ ABDULLA, STAVROS ARONIS, BENGT JONSSON,
and KONSTANTINOS SAGONAS, Uppsala University

Stateless model checking is a powerful method for program verification that, however, suffers from an exponential growth in the number of explored executions. A successful technique for reducing this number, while still maintaining complete coverage, is Dynamic Partial Order Reduction (DPOR), an algorithm originally introduced by Flanagan and Godefroid in 2005 and since then not only used as a point of reference but also extended by various researchers. In this article, we present a new DPOR algorithm, which is the first to be provably optimal in that it always explores the minimal number of executions. It is based on a novel class of sets, called *source sets*, that replace the role of persistent sets in previous algorithms. We begin by showing how to modify the original DPOR algorithm to work with source sets, resulting in an efficient and simple-to-implement algorithm, called *source-DPOR*. Subsequently, we enhance this algorithm with a novel mechanism, called *wakeup trees*, that allows the resulting algorithm, called *optimal-DPOR*, to achieve optimality. Both algorithms are then extended to computational models where processes may disable each other, for example, via locks. Finally, we discuss tradeoffs of the source- and optimal-DPOR algorithm and present programs that illustrate significant time and space performance differences between them. We have implemented both algorithms in a publicly available stateless model checking tool for Erlang programs, while the source-DPOR algorithm is at the core of a publicly available stateless model checking tool for C/pthread programs running on machines with relaxed memory models. Experiments show that source sets significantly increase the performance of stateless model checking compared to using the original DPOR algorithm and that wakeup trees incur only a small overhead in both time and space in practice.

CCS Concepts: • **Theory of computation → Verification by model checking**; Logic and verification; • **Software and its engineering** → *Formal software verification*; *Software testing and debugging*;

Additional Key Words and Phrases: Dynamic partial order reduction, software model checking, systematic testing, concurrency, source sets, wakeup trees

ACM Reference Format:
Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM* 64, 4, Article 25 (August 2017), 49 pages.
https://doi.org/10.1145/3073408

This is a revised and extended version of an article titled "Optimal Dynamic Partial Order Reduction" that appeared in the *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, ACM, New York, NY, 373–384. DOI: http://doi.acm.org/10.1145/2535838.2535845

This work was carried out within the Linnaeus centre of excellence UPMARC (Uppsala Programming for Multicore Architectures Research Center) and was supported in part by the EU FP7 STREP project RELEASE (287510) and the Swedish Research Council.

Authors' addresses: P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden; emails: {Firstname.Lastname}@it.uu.se.

https://doi.org/10.1145/3073408

1 INTRODUCTION

Verification and testing of concurrent programs is difficult, since one must consider all the different ways in which processes/threads can interact. *Model checking* (Clarke et al. 1983; Queille and Sifakis 1982) addresses this problem by systematically exploring the state space of a given program and verifying that each reachable state satisfies a given property. Applying model checking to realistic programs is problematic, however, since it requires capturing and storing a large number of global states. *Stateless model checking* (Godefroid 1997) avoids this problem by exploring the state space of the program without explicitly storing global states. A special runtime scheduler drives the program execution, making decisions on scheduling whenever such decisions may affect the interaction between processes. Stateless model checking has been successfully implemented in tools, such as VeriSoft (Godefroid 2005), Chess (Musuvathi et al. 2008), and Concuerror (Christakis et al. 2013), the tool we use in this article.

While stateless model checking is applicable to realistic programs, it suffers from combinatorial explosion, as the number of possible interleavings grows exponentially with the length of program execution. There are several approaches that limit the number of explored interleavings, such as depth bounding (Russell and Norvig 2009) and context bounding (Musuvathi and Qadeer 2007). Among them, *partial order reduction* (POR) (Clarke et al. 1999; Godefroid 1996; Peled 1993; Valmari 1991) stands out, as it provides full coverage of all behaviours that can occur in *any* interleaving, even though it explores only a representative subset. POR is based on the observation that two interleavings can be regarded as equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (independent) execution steps. In each such equivalence class (called a *Mazurkiewicz trace* (Mazurkiewicz 1987)), POR explores at least one interleaving. This is sufficient for checking most interesting safety properties, including race freedom, absence of global deadlocks, and absence of assertion violations (Clarke et al. 1999; Godefroid 1996; Valmari 1991).

Existing partial order reduction approaches are essentially based on two techniques, both of which reduce the set of process steps that are explored at each scheduling point:

- The *persistent set* technique, which explores only a provably sufficient subset of the enabled processes. This set is called a *persistent set* (Godefroid 1996); variations are *stubborn sets* (Valmari 1991) and *ample sets* (Clarke et al. 1999).
- The *sleep set* technique (Godefroid 1996), which maintains information about the past exploration in a so-called *sleep set*, which contains processes whose exploration would be provably redundant.

These two techniques are independent and complementary and can be combined to obtain increased reduction.

The construction of persistent sets is based on information about possible future conflicts between threads. Early approaches analyzed such conflicts statically, leading to over-approximations and therefore limiting the achievable reduction. *Dynamic Partial Order Reduction* (DPOR) (Flanagan and Godefroid 2005b) improves the precision by recording actually occurring conflicts during the exploration and using this information to construct persistent sets on-the-fly, "by need." DPOR guarantees the exploration of at least one interleaving in each Mazurkiewicz trace when the explored state space is acyclic and finite. This is the case in stateless model checking in which only executions of bounded length are analyzed (Flanagan and Godefroid 2005b; Godefroid 2005; Musuvathi et al. 2008).

Challenge. Since DPOR is excellently suited as a reduction technique, several variants, improvements, and adaptations for different computation models have appeared in the literature (Flanagan and Godefroid 2005b; Kastenberg and Rensink 2008; Lei and Carver 2006; Saarikivi et al. 2012; Sen and Agha 2007; Tasharofi et al. 2012). The obtained reduction can, however, vary significantly depending on several factors, for example, the order in which processes are explored at each point of scheduling. For a particular implementation of DPOR, up to an order of magnitude of difference in the number of explored interleavings has been observed when different strategies are used (Tasharofi et al. 2012). Heuristics for choosing which next process to explore have been investigated without conclusive results (Lauterburg et al. 2010). It has also been shown that for specific communication models, specializations of DPOR algorithms, obtained by carefully defining when execution steps are in conflict (for example, exploiting the transitivity of the dependency relation in actor systems (Tasharofi et al. 2012)), can achieve better reduction.

Let us explain one fundamental reason for the above variation in obtained reduction. In DPOR, the combination of persistent set and sleep set techniques guarantees to explore at least one complete interleaving in each Mazurkiewicz trace. Moreover, it has already been proven that the use of sleep sets is sufficient to prevent the complete exploration of two different but equivalent interleavings (Godefroid et al. 1995). At first sight, this seems to imply that sleep sets can give optimal reduction. What it actually implies, however, is that when the algorithm initiates the exploration of an interleaving that is equivalent to an already explored one, the exploration will begin but it will be blocked sooner or later by the sleep sets in what we call a *sleep-set-blocked* exploration. When only sleep sets are used for reduction, the exploration effort will include an arbitrary number of sleep-set-blocked explorations. It is here where persistent sets enter the picture and limit the number of initiated explorations. Computation of smaller persistent sets leads to fewer sleep-set-blocked explorations. However, as we show in this article, persistent sets are not powerful enough to completely prevent sleep-set-blocked exploration.

Another reason for the observed non-optimal reduction is that most published algorithms define conflicts as occurring when two statements access the same shared object, and at least one of them modifies the object. However, in many cases more reduction can be achieved by using a more refined definition of conflicts. For instance, two writes to a shared variable need not conflict if they write the same value. As another example, a send statement and a receive statement that access the same message queue need not be in conflict, unless the two statements concern the same message. Thus, a reduction technique would benefit from a refined definition of dependencies, which considers not only the accessed shared object but also the actual effect of each program statement in a given execution. Such a refined definition would be particularly beneficial for programs that employ message passing communication.

In view of these variations, a fundamental challenge is to develop an *optimal* DPOR algorithm that (i) always explores the minimum number of interleavings, regardless of scheduling decisions, (ii) can be efficiently implemented, and (iii) is applicable to a variety of computation models, including concurrency via message passing, communication via shared variables protected by locks, or in more general forms of interprocess communication. Such an optimal DPOR algorithm could be standardly implemented in stateless model checkers and test generation tools for concurrent systems. It would also attain the maximal reduction that can be achieved using information about conflicts between processes, implying that any further reductions must be obtained by other means, for example, by symbolic techniques, SAT-based approaches such as the recently proposed SATCheck technique (Demsky and Lam 2015), or by techniques that take alternative notions of causality into account such as Maximal Causality Reduction (Huang 2015).

Contributions. In this article, we present a fundamentally new DPOR technique that is based on a new theoretical foundation for partial order reduction, in which persistent sets are replaced by a novel class of sets, called *source sets.* Source sets subsume persistent sets (i.e., any persistent set is also a source set), and source sets are often smaller than persistent sets. Moreover, source sets are provably minimal, in the sense that the set of explored processes from some scheduling point must be a source set in order to guarantee exploration of all Mazurkiewicz traces. When a minimal persistent set contains more elements than the corresponding source set, the additional elements will always initiate sleep-set-blocked explorations. This implies that a necessary and sufficient condition for the correctness of *any* DPOR algorithm is that the set of explored process steps is a source set. For instance, correctness of persistent-set-based DPOR algorithms is often proved by establishing that the set of explored process steps is always a persistent set. The results of this article imply that in such proofs, it is enough to show the weaker property that this set is always a source set. We thus claim that source sets are the “right” conceptual foundation for developing DPOR techniques.

To show the power of source sets, we develop a simple DPOR algorithm, called *source-DPOR*, which is based on source sets. It is derived by modifying the classical persistent-set-based DPOR algorithm by Flanagan and Godefroid (2005b) so persistent sets are replaced by source sets. The modification only consists in a small change to a single test in the algorithm. The power of source sets can be observed by noting that with this small modification, *source-DPOR* achieves significantly better reduction in the number of explored interleavings than the classical DPOR algorithm. In fact, *source-DPOR* explores the minimal number of interleavings for a large number of our benchmarks in Section 11.

To further demonstrate the power of source sets, we use them to develop a provably optimal DPOR algorithm. This is done by combining source sets with a novel mechanism, called *wakeup trees*, thereby deriving the algorithm *optimal-DPOR*. Wakeup trees control the initial steps of future explorations, implying that *optimal-DPOR* never encounters any sleep-set-blocked (i.e., redundant) exploration. An important feature of wakeup trees is that they are simple data structures that are constructed from already-explored interleavings, hence they do not increase the amount of exploration. On the other hand, they allow us to reduce the number of explored executions. In our benchmarks, maintenance of the wakeup trees reduces total exploration time when *source-DPOR* encounters sleep-set-blocked explorations, and, furthermore, it never requires more than 10% of additional time in the cases where there are none or only a few sleep-set-blocked explorations. Memory consumption is practically always the same between our two DPOR algorithms, and in our experience the space cost of maintaining wakeup trees is very small. Still, as we will show, one can construct programs where the size of wakeup trees grows exponentially, and, consequently, the memory requirements of optimal-DPOR can be considerably worse than those of the source-DPOR algorithm. However, each branch in the wakeup tree is a prefix of some actually explored execution, and hence the size of the wakeup trees can never be larger than the size of all explored executions. Therefore, memory consumption is a problem only in situations where any DPOR-based algorithm needs to explore an exponential set of traces and does not seem to affect the time performance of the optimal algorithm; see Section 9.

We show the applicability of our algorithms to a wide range of computation models, including shared variables and message passing, by formulating them in a general setting, which only assumes that we can compute a *happens-before* relation (also called a causal ordering) between the events in an execution. For systems with shared variables, the happens-before relation can be based on the variables that are accessed or modified by events. For message passing systems, the happens-before relation can be based on correlating the transmission of a message with the corresponding reception. Our approach allows us to make finer distinctions, leading to better reduction, than many other approaches that define a happens-before relation that is based on program statements, possibly taking into account the local state in which they are executed (Clarke et al. 1999; Flanagan and Godefroid 2005b; Godefroid 1996, 2005; Lauterburg et al. 2010; Tasharofi et al.

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

Fig. 1. Writer-readers code excerpt.

2012; Valmari 1991). For instance, we allow a send transition to be dependent with another send transition only if the order in which the two messages are received is significant. Similarly to preceding articles on DPOR, we assume, for simplicity of exposition, that threads are deterministic. Our techniques can be extended to control nondeterminism without introducing fundamentally new concepts, whereas data nondeterminism typically requires additional symbolic techniques.

We have implemented both *source-DPOR* and *optimal-DPOR* as extensions for Concuerror (Christakis et al. 2013), a stateless model checking tool for Erlang programs. Erlang's concurrency model focuses primarily on message passing, but it is also possible to write programs that manipulate shared data structures. Our evaluation shows that on a wide selection of benchmarks, including benchmarks from the DPOR literature, but more importantly on real Erlang applications of considerable size, we obtain optimal or very close to optimal reduction in the number of interleavings even with *source-DPOR*, therefore significantly outperforming the original DPOR algorithm not only in number of explored interleavings but in total execution time as well.

Organization. In the next section, we illustrate the basic new ideas of our techniques. We introduce our computational model and formulation of the partial order framework in Section 3. To simplify presentation, we initially assume that processes do not disable each other. In Section 4, we introduce source sets and establish that source sets are necessary and sufficient for soundness of any DPOR algorithm. The *source-DPOR* algorithm is described in Section 5. We formalize the concept of wakeup trees in Section 6, before describing the *optimal-DPOR* algorithm in Section 7. We then extend the algorithms to computational models where processes may disable each other, for example, via locks, in Section 8. Section 9 discusses some tradeoffs in computational cost between source-DPOR and optimal-DPOR. Implementation of the algorithms is described in Section 10, and experimental evaluation in Section 11. The article ends by surveying related work and offering some concluding remarks.

2 BASIC IDEAS

In this section, we give an informal introduction to the concepts of source sets and wakeup trees, and their improvement over existing approaches, using some small examples.

Source Sets. In Figure 1, the three processes p, q, and r perform dependent accesses to the shared variable `x`. In this example, let us consider two accesses as dependent if they access the same variable and one of them is a write. Since there are no writes to `y` and `z` here, the accesses to `y` and `z` are not dependent with anything else. For this program, there are four Mazurkiewicz traces (i.e., equivalence classes of executions), each characterized by its sequence of accesses to `x` (three accesses can be ordered in six ways, but two pairs of orderings are equivalent, since they differ only in the ordering of adjacent reads, which are not dependent).

Any POR method selects some subset of $\{p, q, r\}$ to perform some first step in the set of explored executions. It is not enough to select only p, since then executions where some read access happens before the write access of p will not be explored. In DPOR, assume that the first execution to be explored is $p.q.q.r.r$ (we denote executions by the dotted sequence of scheduled process steps). A DPOR algorithm will detect the dependency between step `(1)` by p and step `(2)` by q and note that it seems necessary to explore sequences that start with a step of q. The DPOR algorithm will also detect the dependency between `(1)` and `(3)` and possibly note that it is necessary to explore sequences that start with a step of r.

Existing DPOR methods guarantee that the set of processes explored from the initial state is a *persistent set*. In short, a set P of processes is persistent in the initial state if, in any execution from the initial state, the first step that is dependent with the first step of some process in P must be taken by some process in P. In this example, the only persistent set that contains p in the initial state is $\{p, q, r\}$. To see this, suppose that, for example, r is not in the persistent set P, that is, $P = \{p, q\}$. Then the execution $r.r$ contains no step from a process in P, but its second step is dependent with the first step of p, which is in P. In a similar way, one can see that also q must be in P.

In contrast, our source set-based algorithms allow $S = \{p, q\}$ as the set of processes explored from the initial state. The set S is sufficient, since any execution that starts with a step of r is equivalent to some execution that starts with the first (local) step of q. The set S is not a persistent set, but it is a *source set*. Intuitively, a set S of processes is a source set if for each execution E from the initial state there is some process $proc$ in S such that the first step in E that is dependent with $proc$ is taken by $proc$ itself, that is, $proc$ is not preceded by any process that is dependent with $proc$. To see that $\{p, q\}$ is a source set, note that when E is $r.r$, then we can choose q as $proc$, noting that $r.r$ is not dependent with the first step of q. Any persistent set is also a source set, but, as shown by this example, the converse is not true.

Our algorithm *source-DPOR* combines source sets with sleep sets, and will explore exactly four interleavings, whereas any algorithm based on persistent sets will explore at least five (if the first explored execution starts with p), some of which will be sleep set blocked if sleep sets are used. If we extend the example to include n reading processes instead of just two, then the number of sleep-set-blocked explorations increases significantly (see Table 4 in Section 10).

Sleep Sets. To make this section self-contained, we here briefly review the concept of *sleep sets*. Sleep sets (Godefroid 1996; Godefroid and Wolper 1991) use information about past explorations to prevent redundant future explorations. For each prefix E of the execution that is currently being explored, a sleep set is maintained. The sleep set contains a set of processes, whose exploration after E would be redundant for the reason that an equivalent execution has already been explored by the DPOR algorithm. The sleep set at each prefix E is manipulated as follows: (i) after exploring the interleavings that extend E with some process p, the process p is added to the sleep set at E, and (ii) when exploring executions that extend $E.p$, the sleep set at $E.p$ is initially obtained as the sleep set at E with all processes that are dependent with p removed. The effect is that the algorithm need never explore a step of a process in the sleep set. We illustrate this on the program of Figure 1. After having explored executions starting with p, the process p is added to the sleep set at the empty execution, following rule (i). When initiating the exploration of executions that start with q, the process p is in the sleep set at q, according to rule (ii). Therefore, p should not be explored after q, since executions that start with $q.p$ are equivalent to executions that start with $p.q$, and such executions have already been explored.

Wakeup Trees. As mentioned, by utilizing source sets, *source-DPOR* will explore a minimal number of executions for the program of Figure 1. There are cases, however, where *source-DPOR* encounters sleep-set-blocked explorations.

We illustrate this by the example in Figure 2, a program with four processes, p, q, r, s. Two events are dependent if they access the same shared variable, that is, `x`, `y`, or `z`. Variables `m`,`n`,`l` are local. Each statement accessing a global variable has a unique label; for example, process s has three such statements labeled `(5)`, `(6)`, and `(7)`. Statements that operate on local variables are assumed to be part of the previous labeled statement. For example, label `(6)` marks the read of the value of `y`, together with the assignment to `l`, and the condition check on `n`. If the value of `n` is 1, then the

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

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

Fig. 2. Program with control flow.

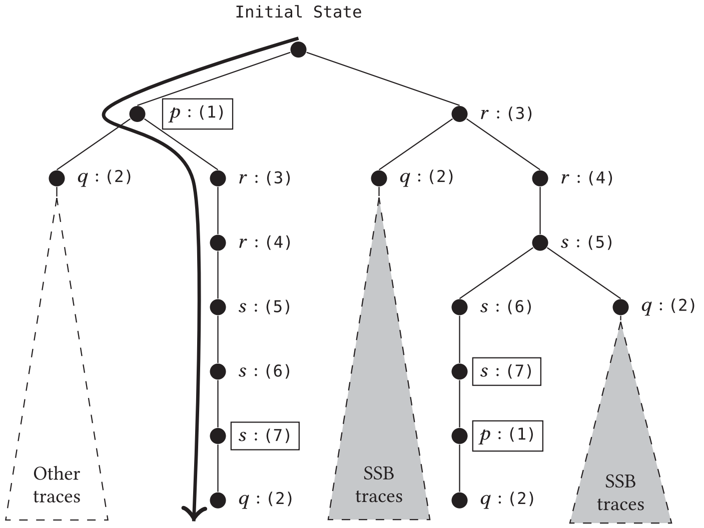

Fig. 3. Explored interleavings for the program of Figure 2.

condition check on l is also part of (6), which ends just before the assignment to x that has the label (7). Similar assumptions are made for the other local statements.

Consider a DPOR algorithm that starts the exploration with p, explores the interleaving $p.r.r.s.s.s$ (marked in Figure 3 with an arrow from top to bottom), and then detects the race between events (1) and (7). It must then explore some interleaving in which the race is reversed, that is, the event (7) occurs before the event (1). Note that event (7) will occur only if it is preceded by the sequence (3)–(4)–(5)–(6) and not preceded by a step of process q. Thus, an interleaving that reverses this race must start with the sequence $r.r.s.s$. Such an interleaving is shown in Figure 3 between the two chunks labeled "SSB traces" (Sleep Set Blocked traces).

Having detected the race in $p.r.r.s.s.s$, *source-DPOR* adds r to the source set at the initial state. However, it does not "remember" that r must be followed by $r.s.s$ to reverse the race. After exploring r, it may therefore continue with q. However, after $r.q$ any exploration is doomed to encounter sleep set blocking, meaning that the exploration reaches a state in which all enabled processes are in the sleep set. To see this, note that p is in the sleep set when exploring r, and will remain there forever in any sequence that starts with $r.q$ (as explained above, it is removed only after the sequence $r.r.s.s.s$). This corresponds to the left chunk of "SSB traces" in Figure 3.

Optimal-DPOR solves this problem by replacing the backtrack set with a structure called a *wakeup tree*. This tree contains initial fragments of executions that are guaranteed not to encounter sleep set blocking. In the example, the Optimal-DPOR algorithm will handle the race between `(1)` and `(7)` by adding the sequence $r.r.s.s.s$ to the wakeup tree. The point is that after $r.r.s.s.s$, the process p has been removed from the sleep set, and so sleep set blocking is avoided.

Events versus Transitions. A majority of approaches to partial order reduction define dependencies as a relation between syntactic statements in the program (Clarke et al. 1999; Flanagan and Godefroid 2005b; Godefroid 1996, 2005; Lauterburg et al. 2010; Tasharofi et al. 2012; Valmari 1991), often referred to as *transitions*. The notion of transition is sometimes refined by including the local state from which it is executed. There are also also approaches where dependencies are defined between occurrences of transitions in a given execution (Lei and Carver 2006; Sen and Agha 2007), sometimes referred to as *events*. In general, events allow finer distinctions when defining dependencies than transitions. For instance, two send operations need not be dependent unless the order in which the two messages are received is significant. This information may not be available if the send operations are regarded as syntactic statements, even if the local states of the sending process is considered. On the other hand, if the send operations are considered in the context of a particular execution, then the definition of dependencies can be appropriately refined. Thus, a reduction technique in which the definition of dependencies is based on events can achieve better reduction than one based on transitions, for instance, when applied to programs that employ message passing.

We illustrate that events can be more suitable than transitions on a small example.

```
p:                  |   q:
send(q, first);     | receive m1;
send(q, second);    | receive m2;
                    |   L := append(append(L, m1), m2);
```

Here, process p sends two messages to process q, and process q receives two messages and appends them to the list `L`. Messages are received in FIFO order. For this example, we assume that `receive` statements must return a message but are non-blocking; if a `receive` is ever attempted by a process when its incoming message queue is empty, then the process dies.

For this example, there are three possible outcomes of executing the program, characterized by the number of messages received by q. Thus, we would want to define a definition of dependencies, which gives rise to three different Mazurkiewicz traces: $p.p.q.q$, $p.q.q.p$, and $q.p.p$. We note that a transition-based definition, which regards any `send` statement as dependent with any `receive` statement will give rise to four Mazurkiewicz traces. To reduce this number, we must define the transmission of the second message by p to be independent of the reception of the first message by q, thereby making the executions $p.p.q.q$ and $p.q.p.q$ equivalent. This is possible if the definition of dependencies is based on events, since that allows us to take the context of the execution into account. In this particular example, this difference allows us to reduce the number of explored transition sequences from four to three. In larger programs, where message queues may contain several messages, the reduction in explored interleavings can be substantial.

3 FRAMEWORK

In this section, we introduce the technical background material. First, we present the general model of concurrent systems for which the algorithms are formulated, thereafter the assumptions on the happens-before relation, and, finally, the notions of independence and races.

3.1 Abstract Computation Model

We consider a concurrent system composed of a finite set of *processes* (or threads). Each process executes a deterministic program, whereby statements act on the *state* of the system, which is made up of the local states of each process and the shared state of the system (*state* will refer to the global state, unless explicitly specified otherwise). We do not restrict the system to a specific mode of process interaction, allowing instead the use of shared variables, messages, and so on. We assume that the state space does not contain cycles and that executions have bounded length. This means that executions must terminate by themselves within a bounded number of steps. Our algorithms, as presented in this article, are unsound if executions are simply truncated beyond some given bound. Soundness, in the sense of covering all executions up to a given length, can be restored at the expense of refining the control over execution scheduling and exploring some executions beyond the imposed bound; the details of this are beyond the scope of this article.

Let Σ be the set of states of the system. The system has a unique *initial state* $s_0 \in \Sigma$. We assume that the program executed by a process p can be represented as a partial function $execute_p : \Sigma \mapsto \Sigma$ that moves the system from one state to a subsequent state. Each such application of the function $execute_p$ represents an atomic *execution step* of process p, which may depend on and affect the state. We let each execution step (or just *step* for short) represent the combined effect of some global statement together with the following finite sequence of local statements (that only access and affect the local state of the process), ending just before the next global statement. This avoids consideration of interleavings of local statements of different processes in the analysis. Such an optimization is common in tools (e.g., Verisoft (Godefroid 1997)).

An *execution sequence* E of a system is a finite sequence of execution steps of its processes that is performed from the initial state s_0. Since each execution step is deterministic, an execution sequence E is uniquely characterized by the sequence of processes that perform steps in E. For instance, $p.p.q$ denotes the execution sequence where first p performs two steps, followed by a step of q. The sequence of processes that perform steps in E also uniquely determine the state of the system after E, which is denoted $s_{[E]}$. The execution of a process is said to *block* in some state s if the process cannot continue (i.e., $execute_p(s)$ is undefined); for example, trying to receive a message in a state where the message queue is empty. For a state s, let $enabled(s)$ denote the set of processes p that are enabled in s (i.e., for which $execute_p(s)$ is defined). We say that E is *maximal* if $enabled(s_{[E]}) = \emptyset$, that is, no process is enabled after E. We use . to denote concatenation of sequences of processes. Thus, if p is not blocked after E, then $E.p$ is an execution sequence. We use $w, w', \ldots$ to range over arbitrary sequences of processes.

An *event* of E is a particular occurrence of a process in E. More precisely, an event is a pair $\langle p, i \rangle$, representing the ith occurrence of process p in the execution sequence. We use $e, e', \ldots$ to range over events, as well as:

- $E \vdash w$ to denote that $E.w$ is an execution sequence.
- $w \setminus p$ to denote the sequence w with its first occurrence of p removed.
- $dom(E)$ to denote the set of events $\langle p, i \rangle$ that are in E, that is, $\langle p, i \rangle \in dom(E)$ iff E contains at least i steps of p.
- $dom_{[E]}(w)$, where $E.w$ is an execution sequence, to denote $dom(E.w) \setminus dom(E)$, that is, the events in $E.w$ which are in w.
- $next_{[E]}(p)$ to denote $dom_{[E]}(p)$ as a special case.
- $\widehat{e}$ to denote the process p of an event $e = \langle p, i \rangle$.
- $op_{[E]}(e)$ to denote the operation performed by event e in execution sequence E.
- $<_E$ to denote the total order between events in E, that is, $e <_E e'$ denotes that e occurs before e' in E.
- $E' \leq E$ to denote that the sequence E' is a prefix of the sequence E.

To simplify the presentation, we first, in Section 3 to 7, make the assumption that a process does not disable another process, that is, if p is enabled and another process q performs a step, then

p is still enabled. We formalize this as the following assumption, so it can be referenced in the subsequent exposition.

Assumption 3.1. *If $E \vdash p$ and $E \vdash q$ with $p \neq q$, then $E \vdash p.q$.*

Assumption 3.1 is valid for concurrent programs that communicate via message passing; that is, it is valid for programs that follow the actor model of concurrency and Erlang programs in particular. However, note that even in such programs a process can disable itself, for example, after a step such that the next statement is a `receive` statement. In Section 8, we thereafter show how our DPOR techniques can be extended to the situation where this assumption is not made, and processes can disable each other, for example, by using locks or await statements.

3.2 Event Dependencies

A central concept in DPOR algorithms is that of a *happens-before relation* (Lamport 1978) between events in an execution sequence (also called a *causal relation* (Sen and Agha 2007)). We denote the happens-before relation in the execution sequence E by $\rightarrow_E$. Intuitively, for an execution sequence E, and two events e and e' in $dom(E)$, $e \rightarrow_E e'$ means that e "causally precedes" e' or that the ordering between e and e' may influence the outcome of the execution. For instance, e can be the transmission of a message that is received by e', e can be a write operation to a shared variable that is accessed by e', or e and e' can both write to the same shared variable.

Our algorithms assume a function (called a *happens-before assignment*), which assigns a "happens-before" relation to any execution sequence. In order not to restrict to a specific computation model, we take a general approach, where the happens-before assignment is only required to satisfy a set of natural properties, which are collected in Definition 3.2. As long as it satisfies these properties, its precision can vary. For instance, the happens-before assignment can let any transmission to a certain message buffer be causally related with a reception from the same buffer. However, better reduction can be attained if the assignment does not make the transmission of a message dependent with the reception of a different message.

In practice, the happens-before assignment function is implemented by relating accesses to the same variables, transmissions, and receptions of the same messages, and so on, typically using *vector clocks* (Mattern 1989). In Section 10, we describe such an assignment, suitable for Erlang programs.

Definition 3.2 (Properties of valid happens-before relations). A happens-before assignment, which assigns a unique happens-before relation $\rightarrow_E$ to any execution sequence E, is *valid* if it satisfies the following properties for all execution sequences E:

(1) $\rightarrow_E$ is an irreflexive partial order on $dom(E)$, which is included in $<_E$.
(2) The execution steps of each process are totally ordered, that is, $\langle p, i\rangle \rightarrow_E \langle p, i+1\rangle$ whenever $\langle p, i+1\rangle \in dom(E)$,
(3) If $E' \leq E$, then $\rightarrow_E$ and $\rightarrow_{E'}$ are the same on $dom(E')$.
(4) Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence that has exactly the same "happens-before" relation $\rightarrow_{E'}$ as $\rightarrow_E$. This means that the relation $\rightarrow_E$ induces a set of equivalent execution sequences, all containing the same set of events and with the same "happens-before" relation. We use
    - $E \simeq E'$ to denote that $dom(E) = dom(E')$ and that E and E' are linearizations of the same "happens-before" relation and
    - $[E]_\simeq$ to denote the equivalence class of E.
(5) If $E \simeq E'$, then $s_{[E]} = s_{[E']}$.

(6) For any sequences E, E', and w, such that $E.w$ is an execution sequence, we have $E \simeq E'$ iff $E.w \simeq E'.w$.
(7) If p, q, and r are different processes, then if $\mathit{next}_{[E]}(p) \to_{E.p.r} \mathit{next}_{[E.p]}(r)$ and $\mathit{next}_{[E]}(p) \not\to_{E.p.q} \mathit{next}_{[E.p]}(q)$, then $\mathit{next}_{[E]}(p) \to_{E.p.q.r} \mathit{next}_{[E.p.q]}(r)$.

The first six properties should be obvious for any reasonable happens-before relation; the only non-obvious one would be the last one. Intuitively, Property (7) states that if, after some sequence E, the next step of p happens before the next step of r, then the step of p still happens before the step of r even when some step of another process, which is not dependent with p, is inserted between p and r. This property holds for all computation models in which the happens-before assignment is based on whether events access shared objects in potentially conflicting ways. As an example, p and q can be reading a shared variable that is written by r. Another example is when p sends a message that is received by r. If an intervening process q is independent with p, then it cannot affect this message, and therefore r still receives the same message. It also holds for several more refined happens-before assignments, which are based on actual values involved in accesses to shared objects. An example is provided by the program in Figure 9 (Section 8).

Properties (4) and (5) of Definition 3.2 together imply, as a special case, that if e and e' are two consecutive events in E with $e \not\to_E e'$, then they can be swapped, and the state after the two events remains the same.

3.3 Independence and Races

We now define independence between events of a computation. If $E.p$ and $E.w$ are both execution sequences, then $E \vdash p \diamond w$ denotes that $E.p.w$ is an execution sequence such that $\mathit{next}_{[E]}(p) \not\to E.p.we$ for any $e \in \mathit{dom}_{[E.p]}(w)$. In other words, $E \vdash p \diamond w$ states that the next event of p would not "happen before" any event in w in the execution sequence $E.p.w$. Intuitively, it means that p is independent with w after E. In the special case when w contains only one process q, then $E \vdash p \diamond q$ denotes that the next steps of p and q are independent after E. We use $E \nvdash p \diamond w$ to denote that $E.p.w$ is an execution sequence for which $E \vdash p \diamond w$ does not hold.

For an execution sequence E and an event $e \in \mathit{dom}(E)$, let

- $\mathit{pre}(E, e)$ denote the prefix of E up to, but not including, the event e,
- $\mathit{notdep}(e, E)$ denote the sub-sequence of E consisting of the events that occur after e but do not "happen after" e (i.e., the events e' that occur after e such that $e \not\to Ee'$).

A central concept in most DPOR algorithms is that of a race. Let e and e' be two events in $\mathit{dom}(E)$, where $e <_E e'$. We say that

- *e is in a race with e'*, denoted $e \lessdot_E e'$ if $\widehat{e} \neq \widehat{e'}$ (i.e., these are events from different processes) and $e \to_E e'$ and there is no event $e'' \in \mathit{dom}(E)$, different from e' and e, such that $e \to_E e'' \to_E e'$,
- *e is in a reversible race with e'*, denoted $e \precsim_E e'$, if $e \lessdot_E e'$ and in any equivalent execution sequence $E' \simeq E$ where e occurs immediately before e', then $\widehat{e'}$ was not blocked before the occurrence of e.

Intuitively, $e \lessdot_E e'$ denotes that e and e' are co-enabled, that is, there is an equivalent execution sequence $E' \simeq E$ in which e and e' are adjacent. Moreover, $e \precsim_E e'$ denotes that e does not enable e', so the order of e and e' can be reversed.

Whenever a DPOR algorithm detects a race, then it will check whether the events in the race can be executed in the reverse order. Since the events are related by the happens-before relation,

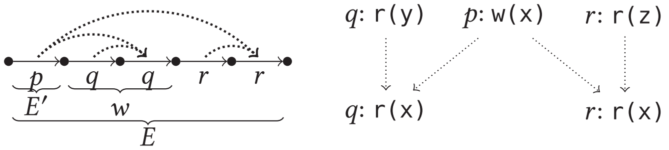

Fig. 4. A sample run of the program in Figure 1 is shown to the left. This run is annotated by a happens-before relation (the dotted arrows). To the right, the happens-before relation is shown as a partial order. Notice that $E' \vdash q \diamond r$, since q and r are not happens-before related in $E'.r.q$. We also observe that $I_{[E']}(w) = \{q\}$, as q is the only process occurring in w and its first occurrence has no predecessor in the dotted relation in w. Furthermore, $WI_{[E']}(w) = \{q, r\}$, since r is not happens-before related to any event in w.

this may lead to a different state: Therefore, the algorithm must try to explore a corresponding execution sequence.

In Figure 4, there are two pairs of events e, e' such that $e \lessdot_E e'$, namely $\langle p, 1\rangle, \langle q, 2\rangle$ and $\langle p, 1\rangle, \langle r, 2\rangle$. It also holds for both these pairs that $e \precsim_E e'$, since both q and r are enabled before $\langle p, 1\rangle$. In other words, both the races in the program are reversible.

4 SOURCE SETS

In this section, we define the new concept of source sets. Intuitively, source sets are sets of processes that perform "first steps" in the possible future execution sequences. A set of processes is a source set if any possible future execution sequence contains a "first step" of some process in the source set. We first define two related notions of possible "first steps" in a sequence.

Definition 4.1 (Initials and Weak Initials). For an execution sequence $E.w$, the set $I_{[E]}(w)$ of processes that are *initials* and the set $WI_{[E]}(w)$ of processes that are *weak initials* are defined as follows:

(1) $p \in I_{[E]}(w)$ iff there is a sequence w' such that $E.w \simeq E.p.w'$
(2) $p \in WI_{[E]}(w)$ iff there are sequences w' and v such that $E.w.v \simeq E.p.w'$

The following lemma gives alternative characterizations of the sets $I_{[E]}(w)$ and $WI_{[E]}(w)$.

Lemma 4.2. *For an execution sequence $E.w$ and a process p, we have*

(1) $p \in I_{[E]}(w)$ iff $p \in w$ and there is no other event $e \in dom_{[E]}(w)$ with $e \rightarrow_{E.w} next_{[E]}(p)$,
(2) $p \in WI_{[E]}(w)$ iff either $p \in I_{[E]}(w)$ or $p \in enabled(s_{[E]})$ and $E \vdash p \diamond w$.

Intuitively, a process in $I_{[E]}(w)$ or $WI_{[E]}(w)$ has no "happens-before" predecessors in $dom_{[E]}(w)$ and is in $I_{[E]}(w)$ if it actually occurs in w.

Proof. We prove each of the cases separately.

(1) (Forward direction) From $E.w \simeq E.p.w'$, using Property (4) of Definition 3.2, we infer $dom_{[E]}(w) = dom_{[E]}(p.w')$, which implies $p \in w$. Furthermore, the fact that p can be moved to the beginning of w implies that p cannot have any predecessors in $dom_{[E]}(w)$, that is, there is no other event $e \in dom_{[E]}(w)$ with $e \rightarrow_{E.w} next_{[E]}(p)$.
(Reverse direction) If $p \in w$ and there is no other event $e \in dom_{[E]}(w)$ with $e \rightarrow_{E.w} next_{[E]}(p)$, then we can linearize $dom_{[E]}(w)$ in such a way that p occurs first, that is, there is a w' with $E.w \simeq E.p.w'$.
(2) (Forward direction) From $E.w.v \simeq E.p.w'$, using Property (4) of Definition 3.2, we infer $dom_{[E]}(w.v) = dom_{[E]}(p.w')$. If $p \in w$, then the events in v can be removed from w' to obtain $dom_{[E]}(w) = dom_{[E]}(p.w'')$ for some w'', implying $p \in I_{[E]}(w)$. If $p \notin w$, then $p \in v$. Since p can be moved before w', it cannot happen-after any event in w; since p occurs

the after w, it cannot happen-before any event in w, and hence $E \vdash p \diamond w$. Since p can be performed directly after E, we have $p \in enabled(s_{[E]})$.
(Reverse direction) If $p \in I_{[E]}(w)$, then $p \in WI_{[E]}(w)$ follows directly (letting v be the empty sequence). If $p \in enabled(s_{[E]})$ and $E \vdash p \diamond w$, then p is not related with w in the happens-before relation, and hence $E.w.p \simeq E.p.w$, implying $p \in WI_{[E]}(w)$. □

Definition 4.3 (Source Sets). Let E be an execution sequence, and let W be a set of sequences w, such that $E \vdash w$ for each $w \in W$. A set P of processes is *a source set for W after E* if for each $w \in W$ we have $WI_{[E]}(w) \cap P \neq \emptyset$.

The key property is that if P is a source set for W after E, then for each execution sequence of form $E.w$ with $w \in W$, there is a process $p \in P$ and a sequence w' such that $E.p.w' \simeq E.w.v$ for some sequence v. Therefore, when an exploration algorithm intends to cover all suffixes in W after E, the set of processes that are chosen for exploration from $s_{[E]}$ must be a source set for W after E. We formulate this observation as a theorem.

Theorem 4.4 (Key Property of Source Sets). *Let E be an execution sequence, let W be a set of continuations of E, and let W' be a subset of W such that for each $w \in W$ there is a $w' \in W'$ with $E.w' \simeq E.w.v$ for some v. Then the set of first processes of sequences in W' is a source set for W after E.*

This theorem implies that a necessary condition for the correctness of *any* DPOR algorithm is that the set of explored process steps is a source set. It it also a sufficient condition: The proofs of correctness for our algorithms source-DPOR (Theorem 5.2) and optimal-DPOR (Theorem 7.4) contain as key lemmas the property that the set of explored processes is a source set (Claim 5.4 and Claim 7.5, respectively).

Before continuing, we first generalize the relations $p \in I_{[E]}(w)$ and $p \in WI_{[E]}(w)$ to the case where p is a sequence. These generalizations will be used in the proof of soundness of the source-DPOR algorithm, and in the definition of wakeup trees.

Definition 4.5. Let E be an execution sequence and let v and w be sequences of processes.

- Let $v \sqsubseteq_{[E]} w$ denote that there is a sequence v' such that $E.v.v'$ and $E.w$ are execution sequences with $E.v.v' \simeq E.w$. Intuitively, $v \sqsubseteq_{[E]} w$ if, after E, the sequence v is a possible way to start an execution that is equivalent to w.
- Let $v \sim_{[E]} w$ denote that there are sequences v' and w' such that $E.v.v'$ and $E.w.w'$ are execution sequences with $E.v.v' \simeq E.w.w'$. Intuitively, $v \sim_{[E]} w$ if, after E, the sequence v is a possible way to start an execution that is equivalent to an execution sequence of form $E.w.w'$.

It can be seen that Definition 4.5 generalizes Definition 4.1, since for a process p we have $p \in I_{[E]}(w)$ iff $p \sqsubseteq_{[E]} w$ and $p \in WI_{[E]}(w)$ iff $p \sim_{[E]} w$.

As examples, in Figure 4, we have $q.r \sqsubseteq_{[E']} q.q.r.r$ but $q.q \not\sqsubseteq_{[E']} r.r$. We also have $q.q \sim_{[E']} r.r$, since $E'.q.q.r.r \simeq E'.r.r.q.q$. Note that $\sim_{[E]}$ is not transitive. The relation $v \sim_{[E]} w$ can be checked using the properties specified in the following lemma.

Lemma 4.6. *The relation $v \sim_{[E]} w$ holds if either*

(1) $v = \langle\rangle$, or
(2) v is of form $p.v'$, and either
(a) $p \in I_{[E]}(w)$ ("p is an initial of w after E") and $v' \sim_{[E.p]} (w \setminus p)$, or
(b) $E \vdash p \diamond w$ ("p is independent of w after E") and $v' \sim_{[E.p]} w$.

Proof. We examine each case separately.

(1) If $v = \langle\rangle$, then let $v' = w$ and $w' = \langle\rangle$. Then $E.v.v' \simeq E.v' \simeq E.w \simeq E.w.w'$, and hence $v \sim_{[E]} w$.

(2) If $v = p.v'$, then

  (a) If $p \in I_{[E]}(w)$ and $v' \sim_{[E.p]} (w \setminus p)$, then by $p \in I_{[E]}(w)$ and Definition 4.1(1) we get that there exists w' such that $E.w \simeq E.p.w'$. In the proof of that Lemma (forward direction), it is easy to see that such a w' is exactly $w \setminus p$, therefore $E.w \simeq E.p.(w \setminus p)$. By $v' \sim_{[E.p]} (w \setminus p)$, we get that there exist sequences v'' and w'' such that $E.v.v'' \simeq E.p.v'.v'' \simeq E.p.(w \setminus p).w'' \simeq E.w.w''$, and hence $v \sim_{[E]} w$.

  (b) If $E \vdash p \diamond w$ and $v' \sim_{[E.p]} w$, then from $E \vdash p \diamond w$ we have that $E.p.w \simeq E.w.p$. From $v' \sim_{[E.p]} w$, we also have that there exist v'' and w' such that $E.v.v'' \simeq E.p.v'.v'' \simeq E.p.w.w' \simeq E.w.p.w'$, and hence $v \sim_{[E]} w$. □

The following lemma states some useful properties.

Lemma 4.7. *Let E be an execution sequence, and let v, w, and w' be sequences. Then*

(1) $E.w' \simeq E.w$ implies that (i) $v \sqsubseteq_{[E]} w$ iff $v \sqsubseteq_{[E]} w'$, (ii) $w \sqsubseteq_{[E]} v$ iff $w' \sqsubseteq_{[E]} v$, and (iii) $v \sim_{[E]} w$ iff $v \sim_{[E]} w'$;
(2) $v \sqsubseteq_{[E]} w$ and $w \sim_{[E]} w'$ imply $v \sim_{[E]} w'$;
(3) $v \sqsubseteq_{[E]} w'$ and $w \sqsubseteq_{[E]} w'$ imply $v \sim_{[E]} w$;
(4) $p \in WI_{[E]}(w)$ and $w' \sqsubseteq_{[E]} w$ imply $p \in WI_{[E]}(w')$;
(5) $p \in WI_{[E]}(w)$ and $E \vdash p \diamond q$ and $E \vdash q \diamond w$ imply $p \in WI_{[E]}(q.w)$.

All of the above properties follow easily from the definitions, so we omit their proofs.

5 SOURCE-DPOR

Having established the concept of source sets, Algorithm 1 presents the *source-DPOR* algorithm. As mentioned in the introduction, this algorithm is derived from the classical persistent-set-based DPOR algorithm of Flanagan and Godefroid (2005b) by replacing persistent sets by source sets. The modification only consists in a small change to a single test in the algorithm. Recall that, as mentioned at the end of Section 3.1, in this section we assume that processes cannot disable each other; in Section 8, we show how the algorithm is extended to the case where process can disable each other.

5.1 Algorithm

Source-DPOR uses the recursive procedure *Explore*(E, *Sleep*) to perform a depth-first search, where E can be interpreted as the stack of the search, that is, the past execution sequence explored so far and *Sleep* is a sleep set, that is, a set of processes that (provably) need not be explored from $s_{[E]}$. For each prefix $E' \leq E$, the algorithm maintains a set *backtrack*(E') of processes that will eventually be explored from E', allowing additions to any backtrack set of a state in the stack to be made by recursive calls to *Explore*.

Explore(E, *Sleep*) initializes *backtrack*(E) to consist of an arbitrary enabled process that is not in *Sleep* (line 3). Thereafter, for each process p in *backtrack*(E) that is not in *Sleep*, the algorithm performs two phases: race detection (lines 5–9) and state exploration (lines 10–12).

In the race detection phase, the algorithm first finds the events $e \in dom(E)$ that are in a reversible race with the next step of p (line 5). For each such event $e \in dom(E)$, the algorithm must explore execution sequences in which the race is reversed. Such sequences are obtained by leaving unchanged that part of E, which follows E' and depends neither on e nor on $next_{[E]}(p)$, and thereafter performing the event $next_{[E]}(p)$ instead of e. The unchanged part is obtained as $notdep(e, E)$. Note that some events in $notdep(e, E)$ may happen-before $next_{[E]}(p)$, and it is then necessary to

ALGORITHM 1: Source-DPOR algorithm.

```
// In all algorithms, we use ":=" notation for data shared among recursive calls and "let" for local variables.
Initial call: $Explore(\langle\rangle, \emptyset)$
1  $Explore(E, Sleep)$
2      if $\exists r \in (enabled(s_{[E]}) \setminus Sleep)$ then          // If not at a maximal or "sleep set blocked" execution...
3          $backtrack(E) := \{r\}$;                                    // ... initialize backtrackwith arbitrary $r$
4          while $\exists p \in (backtrack(E) \setminus Sleep)$ do              // Pick an unexplored $p$
5              foreach $e \in dom(E)$ such that $(e \precsim_{E.p} next_{[E]}(p))$ do   // $e$ and $next_{[E]}(p)$ race
6                  let $E' = pre(E, e)$;
7                  let $v = notdep(e, E).p$;                            // Find events independent with $e$
8                  if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ then         // Ensure an initial exists in backtrack or...
9                      add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;        // ... add an initial
10             let $Sleep' = \{q \in Sleep \mid E \vdash p \diamond q\}$;        // Compute next sleep set
11             $Explore(E.p, Sleep')$;                                 // Recursively call Explore
12             add $p$ to $Sleep$;                                     // Mark $p$ as explored
```

perform them before $next_{[E]}(p)$ can be executed. To this, we add p to ensure that $next_{[E]}(p)$ is executed instead of e. Let v be $notdep(e, E).p$. By construction, any execution sequence that starts with $E'.v$ is guaranteed to be inequivalent to any execution sequence that starts with E. To ensure that an execution sequence of form $E'.v$ will be explored, the algorithm checks (at line 8) whether some process in $I_{[E']}(v)$ is already in $backtrack(E')$. If not, then a process in $I_{[E']}(v)$ is added to $backtrack(E')$. This ensures that a sequence equivalent to $E'.v$ has been or will be explored.

The exploration (phase) is started recursively from $E.p$, using an appropriately initialized sleep set. According to rule (ii) in the explanation of sleep sets in Section 2, the sleep set for the exploration of $E.p$ should be initialized to be the set of processes currently in the sleep set of E that are independent with p after E (i.e., $Sleep' = \{q \in Sleep \mid E \vdash q \diamond p\}$).

On Source Sets and Persistent Sets. The mechanism by which source-DPOR produces source sets rather than persistent sets is the test at line 8. In DPOR algorithms based on persistent sets, such as those of Flanagan and Godefroid (2005b), Lauterburg et al. (2010), Tasharofi et al. (2012), Sen and Agha (2006), and Saarikivi et al. (2012), this test must be stronger and at least guarantee that $backtrack(E')$ contains a process q such that q performs some event in v that "happens-before" $next_{[E]}(p)$ in $E.p$. Thus, if the test at line 8 and the addition at line 9 are strengthened into

$$\begin{aligned}&\textbf{if } \nexists e' \in dom_{[E']}(v).\left[e' \rightarrow_{E.p} next_{[E]}(p) \wedge \widehat{e'} \in (I_{[E']}(v) \cap backtrack(E'))\right] \textbf{ then}\\ &\qquad \text{add } \widehat{e'} \text{ for some } e' \in dom_{[E']}(v) \text{ with } \left[e' \rightarrow_{E.p} next_{[E]}(p) \wedge \widehat{e'} \in I_{[E']}(v)\right] \text{ to } backtrack(E'),\end{aligned}$$

then the algorithm ensures that $backtrack(E')$ will be a persistent set when $Explore(E, Sleep)$ returns. These lines guarantee that the first event in v that is dependent with some process in $backtrack(E')$ is performed by some process in $backtrack(E')$, thus making $backtrack(E')$ a persistent set. In contrast, our test at line 8 does not require the added process to perform an event that "happens-before" $next_{[E]}(p)$. Consider, for instance, that v is just the sequence $q.p$, where q is independent with p after E'. Then, since the event of q does not "happen-before" the event of p, there is an execution sequence $E'.p.q$ in which p is dependent with the process $\widehat{e}$ in $backtrack(E')$ but need not be in $backtrack(E')$. On the other hand, since $q \in I_{[E']}(p.q)$, the set $backtrack(E')$ (together with the initial sleep set at E') is still a source set for the possible continuations after E'.

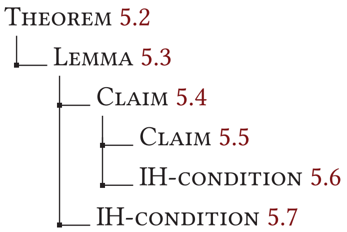

Fig. 5. Structure of correctness proof for Algorithm 1.

5.2 Correctness

In this section, we prove that Algorithm 1 is correct in the sense that for each execution sequence E, it explores an execution sequence in $[E.v]_{\simeq}$ for some v. In particular, if E is maximal, then Algorithm 1 explores an execution sequence in $[E]_{\simeq}$. We formalize this statement in Theorem 5.2. In Figure 5, we give an overview of the structure of its proof.

We first prove some auxiliary properties of initials and weak initials (Lemma 5.1). We then state Lemma 5.3 (a statement about each call to *Explore*, from which Theorem 5.2 can be derived) and prove it by induction on the order in which states (i.e., execution sequences) are backtracked by the algorithm. A key idea, stated as Claim 5.4, is that when *Explore*(E, *Sleep*) returns, the set *Sleep* (the *final sleep set*) will be a source set for W after E, where W is the set of suffixes w such that $E.w$ is an execution sequence. To prove Claim 5.4 by contradiction, we assume that there is some $w \in W$ that does not have a weak initial in the final sleep set. Then there is a sequence of form $E.q.w_R$, which (as proven in Claim 5.5) does not have any weak initial in the final sleep set. Now, by the induction hypothesis (which is applicable by IH-condition 5.6), the algorithm will actually explore a sequence equivalent to $E.q.w_R$, in which it will detect a particular race. However, the algorithm guarantees (lines 8 and 9) that the backtrack set at E contains a process that contradicts the initial assumption. Having shown that the final sleep set is a source set for W after E, we can prove Lemma 5.3 by induction (using IH-condition 5.7), thereby also establishing the converse of Theorem 4.4.

We begin by establishing some properties that will be used in the proof:

LEMMA 5.1. *The following properties hold:*

(1) *If* $E \vdash w$, $E \vdash w'$ *and* $E.w \simeq E.w'$, *then*
  (a) $I_{[E]}(w) = I_{[E]}(w')$
  (b) $WI_{[E]}(w) = WI_{[E]}(w')$
(2) *If* $E \vdash w.p$, *then*
  (a) $I_{[E]}(w) \subseteq I_{[E]}(w.p)$
  (b) $WI_{[E]}(w.p) \subseteq WI_{[E]}(w)$
(3) $E \vdash q \diamond w \rightarrow I_{[E.q]}(w) = I_{[E]}(w)$

PROOF. We prove each of the cases, in order:

1(a). If $p \in I_{[E]}(w)$, then by Definition 4.1(1) there exists w'' such that $E.p.w'' \simeq E.w$. Since $E.w \simeq E.w'$, we then have $E.p.w'' \simeq E.w'$, which by the same definition entails $p \in I_{[E]}(w')$.

1(b). If $p \in WI_{[E]}(w)$, then by Definition 4.1(2) there exists w'' and v such that $E.p.w'' \simeq E.w.v$. Since $E.w \simeq E.w'$, by Property (6) of Definition 3.2, we also have $E.w.v \simeq E.w'.v$ and transitively $E.p.w'' \simeq E.w'.v$, which by Definition 4.1(2) entails $p \in WI_{[E]}(w')$.

2(a). Trivial, from Property (3) of Definition 3.2, since $E.w$ is a prefix of $E.w.p$, and therefore any event not having others happening before it in w maintains this status.

2(b). If $p' \in w$, then $p' \in WI_{[E]}(w.p)$ implies $p' \in I_{[E]}(w)$, which implies $p' \in WI_{[E]}(w)$. If $p' \notin w$, then if $p' = p$, $p \in WI_{[E]}(w.p)$ implies $E \vdash p \diamond w$ and hence $p \in WI_{[E]}(w)$; otherwise, $p' \in WI_{[E]}(w.p)$ implies $E \vdash p' \diamond w.p$, which also implies $E \vdash p' \diamond w$ and therefore $p \in WI_{[E]}(w)$.

3. From Property (4) of Definition 3.2 and the definition of $E \vdash q \diamond w$, it follows immediately that

$$E \vdash q \diamond w \rightarrow E.q.w \simeq E.w.q. \tag{1}$$

Also, from Definition 4.1(1),

$$p \in I_{[E.q]}(w) \Leftrightarrow E.q.w \simeq E.q.p.w' \text{ (for some } w'\text{)}, \tag{2}$$

$$p \in I_{[E]}(w) \Leftrightarrow E.w \simeq E.p.w'' \text{ (for some } w''\text{)}. \tag{3}$$

We will assume that p is in one set and show that p is also in the other:

(i) (Forward direction) Assume $p \in I_{[E.q]}(w)$. From Equation (2), the events in $dom_{[E.q]}(p.w')$ are the same as in $dom_{[E.q]}(w)$ and they have the same (non-) dependency with $next_{[E]}(q)$, because $E \vdash q \diamond w$.

Therefore,

$$E \vdash q \diamond (p.w') \rightarrow E.q.p.w' \simeq E.p.w'.q \qquad \text{similar to (1).} \tag{4}$$

Then

$$\begin{aligned} p \in I_{[E.q]}(w) &\rightarrow E.q.w \simeq E.q.p.w' && \text{by (2)} \\ &\rightarrow E.w.q \simeq E.p.w'.q && \text{by (1) and (4)} \\ &\rightarrow E.w \simeq E.p.w' && \text{by Property (6) of Def. 3.2} \\ &\rightarrow p \in I_{[E]}(w) && \text{by (3).} \end{aligned}$$

(ii) (Reverse direction) Assume $p \in I_{[E]}(w)$. From $E \vdash q \diamond w$, it follows that $E.w.q$ is an execution sequence, therefore:

$$\begin{aligned} p \in I_{[E]}(w) &\rightarrow E.w \simeq E.p.w'' && \text{by (3)} \\ &\rightarrow E.w.q \simeq E.p.w''.q && \text{by Property (6) of Def. 3.2.} \end{aligned} \tag{5}$$

The events in $dom_{[E.q]}(p.w'')$ are the same as $dom_{[E.q]}(w)$, and they have the same (non-)dependency with $next_{[E]}(q)$, because $E \vdash q \diamond w$. Therefore

$$E \vdash q \diamond (p.w'') \rightarrow E.q.p.w'' \simeq E.p.w''.q \qquad \text{similar to (1).} \tag{6}$$

Continuing from (5),

$$\begin{aligned} (5) \quad &= E.w.q \simeq E.p.w''.q \\ &\rightarrow E.q.w \simeq E.q.p.w'' && \text{by (1) and (5)} \\ &\rightarrow p \in I_{[E.q]}(w) && \text{by (2).} \end{aligned}$$

This concludes the proof of all cases of the lemma. □

Theorem 5.2 (Correctness of Source-DPOR). *For all execution sequences E, Algorithm 1 explores some execution sequence E', which is in $[E.v]_{\simeq}$ for some v. In particular, for all maximal execution sequences E, Algorithm 1 explores some execution sequence E', which is in $[E]_{\simeq}$.*

The proof is given at the end of this section, after establishing a series of intermediate lemmas. We begin by introducing some notation. Let:

- $\mathit{init\_sleep}(E)$ be the value of Sleep when calling $\mathit{Explore}(E, \mathit{Sleep})$,
- $\mathit{final\_sleep}(E)$ be the value of Sleep when $\mathit{Explore}(E, \mathit{Sleep})$ returns,
- $\mathit{done}(E)$ be $\mathit{final\_sleep}(E) \setminus \mathit{init\_sleep}(E)$, that is, the set of processes that are explored from $s_{[E]}$ (added in Sleep by line 12),
- $\mathit{to\_be\_explored}(E)$ be the set $\{w \mid E \vdash w \wedge (I_{[E]}(w) \cap \mathit{init\_sleep}(E) = \emptyset)\}$ of possible continuations w of E such that $I_{[E]}(w) \cap \mathit{init\_sleep}(E) = \emptyset$; intuitively, this is the set of sequences that should be explored by the call $\mathit{Explore}(E, \mathit{Sleep})$,
- $e \precsim_E p$ denote that p is not blocked after E and that $e \precsim_{E.p} \mathit{next}_{[E]}(p)$.

We are now ready to state the lemma that will eventually be used to prove correctness.

Lemma 5.3. *Whenever a call to* $\mathit{Explore}(E, \mathit{Sleep})$ *returns during Algorithm 1, then for all sequences* $w \in \mathit{to\_be\_explored}(E)$*, the algorithm has explored some execution sequence* $E.w'$ *which is in* $[E.w.v]_{\simeq}$ *for some* v*.*

Proof. The proof is by induction on the order in which states (i.e., execution sequences) are backtracked by the algorithm. The first time $\mathit{Explore}$ returns, it has just explored a maximal sequence, so the base case holds trivially. Consider a sequence E. As inductive hypothesis, we assume that the statement in the lemma holds for all execution sequences $E.p$ with $p \in \mathit{done}(E)$. Whenever we invoke the induction hypothesis, we need to establish a condition of form $w' \in \mathit{to\_be\_explored}(E.p)$ for some process p and sequence w'. We will refer to this as the IH-condition and try to separate its establishment from the rest of the flow in the proof. We structure the proof of the lemma as a sequence of *claims*.

Claim 5.4. $\mathit{done}(E)$ *is a source set for* $\mathit{to\_be\_explored}(E)$ *after* E*.*

By the definition of source sets (Definition 4.3), this claim states that for each $w \in \mathit{to\_be\_explored}(E)$, there is some $p \in \mathit{done}(E)$ such that $p \in \mathit{WI}_{[E]}(w)$. Since, by the definition of $\mathit{to\_be\_explored}(E)$, the set of sequences w with $E \vdash w$ is the union of $\mathit{to\_be\_explored}(E)$ and the set $\{w \mid E \vdash w \wedge \mathit{init\_sleep}(E) \neq \emptyset\}$, it implies that the set $\mathit{final\_sleep}(E)$ is a source set for $\{w \mid E \vdash w\}$.

Proof of Claim 5.4. By contradiction. Assume a $w \in \mathit{to\_be\_explored}(E)$, such that for all $p \in \mathit{done}(E)$ we have $p \notin \mathit{WI}_{[E]}(w)$. We will prove that such a sequence cannot exist.

If $\mathit{done}(E)$ is empty, then no process was explored from that state: This can only happen if the check on line 2 has failed (otherwise process p on that line will later be added by line 12), which contradicts that $\mathit{to\_be\_explored}(E)$ is non-empty. Therefore, $\mathit{done}(E)$ is not empty.

For each process $p \in \mathit{done}(E)$, let w_p be the longest prefix of w such that $E \vdash p \diamond w_p$, and let e_p be the first event in $\mathit{dom}_{[E]}(w)$ which is not in w_p. We have

$$p \neq \widehat{e_p} \qquad \text{(otherwise from } E \vdash p \diamond w_p \text{ we derive } p \in \mathit{WI}_{[E]}(w)\text{).} \tag{7}$$

Let q be such that w_q is a longest prefix among the prefixes w_p for $p \in \mathit{done}(E)$. If there are several processes p such that w_p is the same longest prefix, then let q be the process among these which is explored first from $s_{[E]}$. Finally, let w_R be $w_q.\widehat{e_q}$ (Figure 6).

Given these definitions, all of the following are valid execution sequences:

$$\begin{aligned} E \vdash w &\rightarrow E \vdash w_q.\widehat{e_q} \\ &\rightarrow E.w_q \vdash \widehat{e_q} \end{aligned} \tag{8}$$

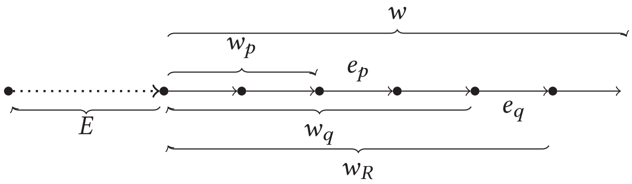

Fig. 6. Illustrating E, w, w_p, e_p, w_q, e_q, and w_R.

as well as the following:

$$\begin{aligned} E \vdash q \diamond w_q &\rightarrow E.w_q \vdash q && \\ &\rightarrow E.w_q \vdash q.\widehat{e_q} && \text{by (7), (8) and Assumption 3.1} \\ &\rightarrow E \vdash q.w_q.\widehat{e_q} && \text{by } E \vdash q \diamond w_q \\ &\rightarrow E \vdash q.w_R && \text{by the definition of } w_R. \end{aligned} \tag{9}$$

We also note that

$$p \notin WI_{[E]}(w_R) \text{ for all } p \in done(E) \qquad \text{by the construction of } w_R. \tag{10}$$

As the next step in the proof of Claim 5.4, we establish the following claim:

Claim 5.5. *For all $p \in done(E)$ with $p \neq q$ we have $p \notin WI_{[E]}(q.w_R)$.*

Proof of Claim 5.5. We distinguish among the following cases:

1. If $e_p \neq e_q$, then w_p is a strict prefix of w_q. Therefore:

$$\begin{aligned} e_p \in w_q &\rightarrow p \notin WI_{[E]}(w_q) && \text{by } E \nvdash p \diamond w_q \\ &\rightarrow p \notin WI_{[E]}(w_q.q) && \text{by 5.1.2b} \\ &\rightarrow p \notin WI_{[E]}(q.w_q) && \text{by } E.q.w_q \simeq E.w_q.q\text{, 5.1.1b} \\ &\rightarrow p \notin WI_{[E]}(q.w_R) && \text{by 5.1.2b.} \end{aligned}$$

2a. If $e_p = e_q$ and $E \nvdash p \diamond q$, then $p \notin WI_{[E]}(q.w_R)$ follows directly.
2b. If $e_p = e_q$, $E \vdash p \diamond q$, and $E \nvdash p \diamond q.w_q$, then $p \notin WI_{[E]}(q.w_R)$ follows directly.
2c. If $e_p = e_q$, $E \vdash p \diamond q$, and $E \vdash p \diamond q.w_q$, then:

$$\begin{aligned} E \vdash p \diamond q.w_q &\rightarrow E \vdash p \diamond w_q.q && \text{by } E \vdash q \diamond w_q \\ &\rightarrow E.w_q \vdash p \diamond q && \qquad (11) \\ e_p = e_q &\rightarrow E.w_q \nvdash p \diamond \widehat{e_q} && \\ &\rightarrow E.w_q \nvdash p \diamond q.\widehat{e_q} && \text{by (11) and Property (7) of Def. 3.2} \\ &\rightarrow E \nvdash p \diamond w_q.q.\widehat{e_q} && \\ &\rightarrow E \nvdash p \diamond q.w_q.\widehat{e_q} && \text{by } E \vdash q \diamond w_q \\ &\rightarrow p \notin WI_{[E]}(q.w_R). && \end{aligned}$$

This concludes the proof of Claim 5.5. □

We will next invoke the inductive hypothesis for $E.q$ and w_R. Let us first establish the IH-condition:

IH-condition 5.6. $w_R \in to\_be\_explored(E.q)$.

Proof of IH-condition 5.6. That $E.q \vdash w_R$ was established in (9). We must prove that $I_{[E.q]}(w_R) \cap \mathit{init\_sleep}(E.q) = \emptyset$. By the handling of sleep sets, $\mathit{init\_sleep}(E.q)$ may extend $\mathit{init\_sleep}(E)$ by some of the processes in $\mathit{done}(E)$. From the assumptions for w we have that for all $p \in \mathit{done}(E)$:

$$\begin{aligned} p \notin WI_{[E]}(w) &\rightarrow p \notin I_{[E]}(w) && \text{by Lemma 4.2(1)} \\ &\rightarrow p \notin I_{[E]}(w_q) && \text{by 5.1.2a} \\ &\rightarrow p \notin I_{[E.q]}(w_q) && \text{by } E \vdash q \diamond w_q\text{, 5.1.3.} \end{aligned} \tag{12}$$

Any other process in $\mathit{init\_sleep}(E.q)$ was also in $\mathit{init\_sleep}(E)$ (passed along by line 10). Again, from the assumptions for w, we have that for all $p \in \mathit{init\_sleep}(E)$,

$$\begin{aligned} I_{[E]}(w) \cap \mathit{init\_sleep}(E) = \emptyset &\rightarrow p \notin I_{[E]}(w) \\ &\rightarrow p \notin I_{[E.q]}(w_q) && \text{(same reasoning).} \end{aligned} \tag{13}$$

Since in (12) and (13) we considered $I_{[E.q]}(w_q)$, the only process that can be in $I_{[E.q]}(w_R) \cap \mathit{init\_sleep}(E.q)$ is $\widehat{e_q}$. We have two cases.

- If there is no event e in w_q such that $e \rightarrow_{E.w_R} e_q$, then by the construction of e_q, we have $E \nvdash q \diamond \widehat{e_q}$, implying $\widehat{e_q} \notin \mathit{init\_sleep}(E.q)$ (as it is filtered by line 10).
- Otherwise, let e be the last event in w_q such that $e \rightarrow_{E.w_R} e_q$. Let w'_q be the prefix of w_q that precedes e, and let w''_q be the suffix of w_q that follows e, that is, $w_q = w'_q.\widehat{e}.w''_q$. By construction, we have $E.w'_q \nvdash \widehat{e} \diamond \widehat{e_q}$ and $E.w'_q \vdash \widehat{e} \diamond q$. By Property (7) of Definition 3.2, we infer $E.w'_q \nvdash \widehat{e} \diamond (q.\widehat{e_q})$, which implies $e \rightarrow_{E.w_q.q.\widehat{e_q}} e_q$, which implies $e \rightarrow_{E.q.w_q.\widehat{e_q}} e_q$. Hence $e \rightarrow_{E.q.w_R} e_q$, and so $\widehat{e_q} \notin I_{[E.q]}(w_R)$.

This concludes the establishment of IH-condition 5.6. □

Using the inductive hypothesis for $E.q$, the algorithm explores some sequence of the form $E.q.w'$ such that

$$E.q.w' \simeq E.q.w_R.v' \text{ for some } v'. \tag{14}$$

Note that in w', the event e_q exists, but need not be the last occurring of the events e_p for $p \in \mathit{done}(E)$, since such independent events may have been reordered from $w_R.v'$ to w'.

Let w'_q be the prefix of w' up to, but not including, e_q. By the construction of w_q, we have $\mathit{next}_{[E]}(q) \precsim_{E.q.w_q} e_q$. From (14), it also follows that $\mathit{next}_{[E]}(q) \precsim_{E.q.w'_q} e_q$ (this is the same race as the race $\mathit{next}_{[E]}(q) \precsim_{E.q.w_q} e_q$, that is, between $\mathit{next}_{[E]}(q)$ and e_q over the sequence w_q). Since the sequence $E.q.w'$ is one actually explored by the algorithm, line 5 will detect the race $\mathit{next}_{[E]}(q) \precsim_{E.q.w'_q} e_q$. At that point,

- p in the algorithm will correspond to $\widehat{e_q}$,
- E' in the algorithm will correspond to E, and
- e in the algorithm will correspond to $\mathit{next}_{[E]}(q)$.

The algorithm will extract (lines 6 and 7) the sequence $v = \mathit{notdep}(\mathit{next}_{[E]}(q), E.q.w'_q).\widehat{e_q}$ from $E.q.w'_q.\widehat{e_q}$ by removing events that have $\mathit{next}_{[E]}(q)$ happening-before them and identify its initials

$I_{[E]}(v)$. Below let us use $\overset{*}{\to}$ to denote multiple steps. For any such $q' \in I_{[E]}(v)$ it follows that

$$\begin{aligned} q' \in I_{[E]}(v) &\to q' \in I_{[E]}(q.w'_q.\widehat{e_q}) && \text{by Lemma 4.2(1) and definition of } \textit{notdep} \\ &\overset{*}{\to} q' \in I_{[E]}(q.w') && \text{by Lemma 5.1.2a} \\ &\to q' \in I_{[E]}(q.w_R.v') && \text{by (14), Lemma 5.1.1a} \\ &\to q' \in WI_{[E]}(q.w_R) && \text{by Lemma 4.2(2).} \qquad (15) \end{aligned}$$

The algorithm then guarantees (in lines 8 and 9) that there is one such $q' \in I_{[E]}(v)$ that is added to *backtrack*(E). Additionally, by (15), we have that no other event happens-before q' in $E.q.w'$. Therefore $q' \notin$ *init_sleep*(E); otherwise, it would have stayed in the sleep set (transferred by line 10) and could not have been chosen for execution.

Together, $q' \in$ *backtrack*(E) and $q' \notin$ *init_sleep*(E) imply that $q' \in$ *done*(E), but this contradicts Claim 5.5. Such a sequence w can therefore not exist, and this concludes the proof of Claim 5.4. □

We now move on to the proof of Lemma 5.3. By Claim 5.4, for any $E.w$, we have $p \in WI_{[E]}(w)$ for some $p \in$ *done*(E). Let q be the process among these that is explored first from $s_{[E]}$. We intend to apply the inductive hypothesis for $E.q$ and $w \setminus q$:

IH-condition 5.7. $(w \setminus q) \in$ *to_be_explored*$(E.q)$.

Proof of IH-condition 5.7 We must establish two properties: (i) that $E.q \vdash w \setminus q$ and (ii) that $I_{[E.q]}(w \setminus q) \cap$ *init_sleep*$(E.q) = \emptyset$. We infer $E.q \vdash w \setminus q$ from $E \vdash w$ and $q \in WI_{[E]}(w)$. For the second property, $I_{[E.q]}(w \setminus q)$ may extend $I_{[E]}(w)$ only by processes $q' \in w$ such that $next_{[E]}(q) \to_{E.w} \langle q', j \rangle$, where $\langle q', j \rangle$ is the first event of q' in w. However, any such process q' cannot be in *init_sleep*$(E.q)$, since in this case $E \nvdash q \diamond q'$ so q' is removed from the sleep set when exploring $E.q$. Given also that *init_sleep*$(E.q) \setminus$ *init_sleep*(E) does not contain any process in $I_{[E]}(w)$ (recall that q was the first process in $I_{[E]}(w)$ to be explored), we infer IH-condition 5.7. □

The conclusion from the inductive hypothesis is that the algorithm explores some sequence $E.q.\hat{w}$ that is in $[E.q.(w \setminus q).v]_{\simeq}$ for some v. We have two cases:

- $q \in w$. From $q \in WI_{[E]}(w)$, we infer $E.q.(w \setminus q).v \simeq E.w.v$, which establishes the lemma.
- $q \notin w$, that is, $(w \setminus q) = w$. From $E \vdash q \diamond w$, we infer $E.q.\hat{w} \simeq E.q.w.v \simeq E.w.q.v$.

This concludes the proof of Lemma 5.3. □

Using Lemma 5.3, we can finally conclude the proof of the main theorem.

Proof of Theorem 5.2. Since Algorithm 1 is initially called with *Explore*$(\langle\rangle, \emptyset)$, Lemma 5.3 implies correctness in the sense that for all execution sequences E the algorithm explores some execution sequence E' that is in $[E.v]_{\simeq}$ for some v. In particular, if E is maximal, then $E' \simeq E$. □

6 WAKEUP TREES

As we described earlier, *source-DPOR* may still lead to sleep-set-blocked explorations. We therefore present an algorithm, called *optimal-DPOR*, which is provably optimal in that it always explores exactly one interleaving per Mazurkiewicz trace and never encounters sleep set blocking. Optimal-DPOR is obtained by combining source sets with a novel mechanism, called *wakeup trees*, which control the initial steps of future explorations.

Wakeup trees can be motivated by looking at lines 6–9 of Algorithm 1. At these lines, it is found that some execution sequence starting with $E'.v$ should be performed to reverse a detected race. However, at line 9, only a single process from the sequence v is entered into *backtrack*(E'), thus "forgetting" information about how to reverse this race. Since the new exploration after $E'.q$ does not "remember" this sequence v, it may explore a completely different sequence, which could

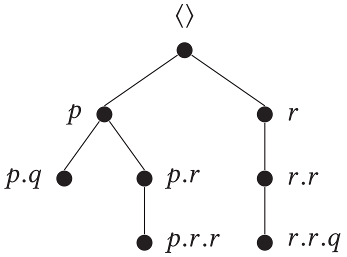

Fig. 7. A wakeup tree. The order of exploration $\prec$ is $p.q$, followed by $p.r.r$, $p.r$, and so on, until we finish at $\langle\rangle$.

potentially lead to sleep set blocking. To prevent such a situation, we replace the backtrack set by a so-called *wakeup tree*. The wakeup tree contains initial fragments of the sequences that are to be explored after E'. Each fragment guarantees that no sleep set blocking will be encountered during the exploration.

6.1 Formal Definition

Let us define an *ordered tree* as a pair $\langle B, \prec\rangle$, where B (the set of *nodes*) is a finite prefix-closed set of sequences of processes, with the empty sequence $\langle\rangle$ being the root. The children of a node w, of form $w.p$ for some set of processes p, are ordered by $\prec$. In $\langle B, \prec\rangle$, such an ordering between children has been extended to the total order $\prec$ on B by letting $\prec$ be the induced post-order relation between the nodes in B. This means that if the children $w.p_1$ and $w.p_2$ are ordered as $w.p_1 \prec w.p_2$, then $w.p_1 \prec w.p_2 \prec w$ in the induced post-order. An example of the ordering of sequences is given in Figure 7.

Definition 6.1 (Wakeup Tree). Let E be an execution sequence, and P be a set of processes. A *wakeup tree after* $\langle E, P\rangle$ is an ordered tree $\langle B, \prec\rangle$, such that the following properties hold:

(1) $WI_{[E]}(w) \cap P = \emptyset$ whenever w is a leaf of B;
(2) whenever $u.p$ and $u.w$ are nodes in B with $u.p \prec u.w$, and $u.w$ is a leaf, then $p \notin WI_{[E.u]}(w)$.

Intuitively, a wakeup tree after $\langle E, P\rangle$ is intended to consist of initial fragments of sequences that should be explored after E to avoid sleep set blocking, when P is the current sleep set at E. To see this, note that if $q \in P$, then (by the way sleep sets are handled) q cannot be in $I_{[E]}(w)$ for any sequence w that is explored after E. If, however, $E \vdash q \diamond w$, then q is still in the sleep set at $E.w$ and may never be removed. To prevent this, we therefore require $q \notin WI_{[E]}(w)$, which is the same as Property 1, that is, $WI_{[E]}(w) \cap P = \emptyset$. Property 2 implies that when a process p is added to the sleep set at $E.u$, after exploring $E.u.p$, then by the same reasoning as above, it will have been removed from the sleep set when we reach $E.u.w$.

The *empty* wakeup tree is the tree $\langle\{\langle\rangle\}, \emptyset\rangle$, which consists only of the root $\langle\rangle$. We state a useful property of wakeup trees.

Lemma 6.2. *If $\langle B, \prec\rangle$ is a wakeup tree after $\langle E, P\rangle$ and $w, w' \in B$ and w is a leaf that satisfies $w' \sim_{[E]} w$, then $w \preceq w'$.*

The lemma states that any leaf w is the smallest (w.r.t. $\prec$) node in the tree that is consistent with w after E.

Proof. We prove the lemma by contradiction. Assume $w' \prec w$. Then by the definition of ordered trees there are u, p, v, v' such that $w' = u.p.v'$ and $w = u.v$ such that $u.p \prec u.v$. Since $u.v$

is a leaf, we have $p \notin \mathit{WI}_{[E.u]}(v)$ by Property 2 of Definition 6.1. Hence $p \not\sim_{[E.u]} v$, which implies $u.p \not\sim_{[E]} u.v$, which implies $u.p.v' \not\sim_{[E]} u.v$, that is, $w' \not\sim_{[E]} w$. □

For a wakeup tree $\langle B, \prec \rangle$ and a process $p \in B$, define $\mathit{subtree}(\langle B, \prec \rangle, p)$ to denote the subtree of $\langle B, \prec \rangle$ rooted at p, that is, $\mathit{subtree}(\langle B, \prec \rangle, p) = \langle B', \prec' \rangle$, where $B' = \{w \mid p.w \in B\}$ and $\prec'$ is the restriction of $\prec$ to B'.

6.2 Inserting Sequences in a Wakeup Tree

Let $\langle B, \prec \rangle$ be a wakeup tree after $\langle E, P \rangle$. For any sequence w such that $E.w$ is an execution sequence with $\mathit{WI}_{[E]}(w) \cap P = \emptyset$, we need an operation $\mathit{insert}_{[E]}(w, \langle B, \prec \rangle)$ that satisfies the following properties:

(1) $\mathit{insert}_{[E]}(w, \langle B, \prec \rangle)$ is also a wakeup tree after $\langle E, P \rangle$,
(2) any leaf of $\langle B, \prec \rangle$ remains a leaf of $\mathit{insert}_{[E]}(w, \langle B, \prec \rangle)$, and
(3) $\mathit{insert}_{[E]}(w, \langle B, \prec \rangle)$ contains a leaf u with $u \sim_{[E]} w$.

A simple implementation of $\mathit{insert}_{[E]}(w, \langle B, \prec \rangle)$ is the following: Let v be the smallest (w.r.t. to $\prec$) sequence in B such that $v \sim_{[E]} w$. If v is a leaf, then the result of $\mathit{insert}_{[E]}(w, \langle B, \prec \rangle)$ can be taken as $\langle B, \prec \rangle$ (leaving the tree unmodified). Otherwise, let w' be a shortest sequence such that $w \sqsubseteq_{[E]} v.w'$, and add $v.w'$ as a new leaf, which is ordered after all already existing nodes in B of form $v.w''$.

As an illustration, using the program of Figure 1, assume that a wakeup tree $\langle B, \prec \rangle$ after $\langle \langle \rangle, \emptyset \rangle$ contains p as the only leaf. Then the operation $\mathit{insert}_{[\langle \rangle]}(q.q, \langle B, \prec \rangle)$ adds $q.q$ as a new leaf with $p \prec q.q$. If we thereafter perform $\mathit{insert}_{[\langle \rangle]}(r.r, \langle B, \prec \rangle)$, then the wakeup tree remains the same, since $q.q \sim_{[\langle \rangle]} r.r$, and $q.q$ is already a leaf.

7 OPTIMAL-DPOR

In this section, we present a DPOR algorithm that achieves optimal reduction.

7.1 Algorithm

The *optimal-DPOR* algorithm, shown as Algorithm 2, performs a depth-first search using the recursive procedure *Explore*(E, *Sleep*, *WuT*), where E and *Sleep* are as in Algorithm 1, and *WuT* is a wakeup tree after $\langle E, \mathit{Sleep} \rangle$, containing extensions of E that are guaranteed to be explored (in order) by *Explore*(E, *Sleep*, *WuT*). If *WuT* is empty, then *Explore*(E, *Sleep*, *WuT*) is free to explore any extension of E.

Like Algorithm 1, the algorithm runs in two phases: race detection (lines 2–7) and state exploration (lines 8–21), but it is slightly differently organized. Instead of analyzing races at every invocation of *Explore*, races are analyzed in the entire execution sequence only when a maximal execution sequence has been generated. The reason for this is that the test at line 6 is precise only when the used sequence v, which is defined at line 5, includes all events in the entire execution that do not "happen after" e, as well as those that occur after e'. Therefore v can be defined only when E is a maximal execution sequence.

In the race detection phase, Algorithm 2 must be able to access the current sleep set for each prefix E' of the currently explored execution sequence E. For each such prefix E', the algorithm therefore maintains a set of processes $\mathit{sleep}(E')$, which is the current sleep set at E'. In a similar way, for each prefix E' of E, the algorithm maintains $\mathit{wut}(E')$, which is the current wakeup tree at E'.

Let us now explain the race detection phase, which is entered whenever the exploration reaches the end of a complete sequence (i.e., $\mathit{enabled}(s_{[E]}) = \emptyset$). In this phase, the algorithm investigates all races that can be reversed in the just explored sequence E. Such a race consists of two events e and

ALGORITHM 2: Optimal-DPOR algorithm.

```
    Initial call: $Explore(\langle\rangle, \emptyset, \langle\{\langle\rangle\}, \emptyset\rangle)$
1   $Explore(E, Sleep, WuT)$
2       if $enabled(s_{[E]}) = \emptyset$ then                     // Race detection only at maximal execution sequences
3           foreach $e, e' \in dom(E)$ such that $(e \precsim_E e')$ do          // For each racing pair e, e'
4               let $E' = pre(E, e)$;                                             // Goto state before e
5               let $v = notdep(e, E).\widehat{e'}$;                       // Find events independent with e
6               if $sleep(E') \cap WI_{[E']}(v) = \emptyset$ then       // Has an equivalent already been explored?
7                   $wut(E') := insert_{[E']}(v, wut(E'))$;             // If not, insert into the wakeup tree
8       else                                               // If not at a maximal execution sequence, explore...
9           if $WuT \neq \langle\{\langle\rangle\}, \emptyset\rangle$ then
10              $wut(E) := WuT$;                                     // ... either using an existing wakeup tree
11          else
12              choose $p \in enabled(s_{[E]})$;                     // ... or by selecting an arbitrary p...
13              $wut(E) := \langle\{\langle\rangle, p\}, \{(p, \langle\rangle)\}\rangle$;        // ... and making a wakeup tree from it
14          $sleep(E) := Sleep$;
15          while $\exists p \in wut(E)$ do                           // While the wakeup tree is not empty...
16              let $p = \min_{\prec}\{p \in wut(E)\}$;                           // ... pick next branch, ...
17              let $Sleep' = \{q \in sleep(E) \mid E \vdash p \diamond q\}$;            // ... compute next sleep set...
18              let $WuT' = subtree(wut(E), p)$;         // ... and wakeup tree (a subtree of the current),...
19              $Explore(E.p, Sleep', WuT')$;                     // ... and do a recursive call to Explore
20              remove all sequences of form $p.w$ from $wut(E)$;                 // When done, cleanup...
21              add $p$ to $sleep(E)$;                                     // ... and mark p as explored
```

e' in E, such that $e \precsim_E e'$. Let $E' = pre(E, e)$ and let $v = notdep(e, E).\widehat{e'}$, that is, the sub-sequence of E consisting of the events that occur after e but do not "happen after" e, followed by $\widehat{e'}$ (this notation is introduced at lines 4 and 5). The reversible race $e \precsim_E e'$ indicates that there is another execution sequence, which performs v after E' and in which the race is reversed, that is, the event e' happens before the event e. Since $E'.v$ is incompatible with the currently explored computation, the algorithm must now make sure that it will be explored if it was not explored previously. If some $p \in sleep(E')$ is in $WI_{[E']}(v)$, then some execution equivalent to one starting with $E'.v$ will have been explored previously. If not, then we perform the operation $insert_{[E']}(v, wut(E'))$ to make sure that some execution equivalent to one starting with $E'.v$ will be explored in the future.

In the exploration phase, which is entered if exploration has not reached the end of a maximal execution sequence, first the wakeup tree $wut(E)$ is initialized to WuT. If WuT is empty, then (as we will state in Lemma 7.1) the sleep set is empty, and an arbitrary enabled process is entered into $wut(E)$. The sleep set $sleep(E)$ is initialized to the sleep set that is passed as argument in this call to *Explore*. Thereafter, each sequence in $wut(E)$ is subject to exploration. We find the first (i.e., minimal) single-process branch p in $wut(E)$ and call *Explore* recursively for the sequence $E.p$. In this call, the associated sleep set $Sleep'$ is obtained from $sleep(E)$ in the same way as in Algorithm 1. The associated wakeup tree WuT' is obtained as the corresponding subtree of $wut(E)$. Thereafter, *Explore* is called recursively for the sequence $E.p$ with the modified sleep set $Sleep'$ and wakeup tree WuT'. After $Explore(E.p, Sleep', WuT')$ has returned, the sleep set $sleep(E)$ is extended with p, and all sequences beginning with p are removed from $wut(E)$.

7.2 Correctness

Let us now prove the correctness of the optimal-DPOR algorithm. Throughout, we assume a particular completed execution of optimal-DPOR. This execution consists of a number of terminated calls to $Explore(E, Sleep, WuT)$ for some values of the parameters E, $Sleep$, and WuT. Let $\mathcal{E}$ denote the set of execution sequences E that have been explored in some call $Explore(E, \cdot, \cdot)$. Define the ordering $\propto$ on $\mathcal{E}$ by letting $E \propto E'$ if $Explore(E, \cdot, \cdot)$ returned before $Explore(E', \cdot, \cdot)$. Intuitively, if one were to draw an ordered tree that shows how the exploration has proceeded, then $\mathcal{E}$ would be the set of nodes in the tree, and $\propto$ would be the post-order between nodes in that tree.

For an arbitrary execution sequence $E \in \mathcal{E}$ let

- $final\_sleep(E)$ denote the value of $sleep(E)$ at the point when $Explore(E, Sleep, WuT)$ returns;
- $done(E)$ denote $final\_sleep(E) \setminus Sleep$, that is, the set of processes that are explored from $s_{[E]}$.

We begin by establishing some useful invariants.

Lemma 7.1. *Whenever Algorithm 2 is inside a call $Explore(E, Sleep, WuT)$, then the following two invariants hold:*

(1) $wut(E)$ is a wakeup tree after $\langle E, sleep(E)\rangle$,
(2) if WuT is empty, then $Sleep$ is empty.

Proof. We establish these two invariants jointly by induction over the steps of the algorithm. The properties hold at the beginning of the initial call, since $sleep(E)$ and $wut(E)$ are both empty. We verify that each property is preserved by the steps of the algorithm.

(1) We need to consider the following cases.
- Whenever $wut(E)$ is updated at line 7, it follows by Property 1 of the operation $insert_{[E]}(v, wut(E))$ that $wut(E)$ remains a wakeup tree after $\langle E, sleep(E)\rangle$.
- Consider a new call to $Explore(E.p, Sleep', WuT')$ at line 19, which occurs inside a call $Explore(E, Sleep, WuT)$, and where $Sleep'$ and WuT' are obtained from the current values of $sleep(E)$ and $wut(E)$ by $Sleep' = \{q \in sleep(E) \mid E \vdash p \diamond q\}$ and $WuT' = subtree(wut(E), p)$, at lines 17 and 18. Since the parameters $Sleep'$ and WuT' will be used to initialize $sleep(E.p)$ (at line 14) and $wut(E.p)$ (at line 10), we must check that WuT' is a wakeup tree after $\langle E.p, Sleep'\rangle$. By the inductive hypothesis, $wut(E)$ is a wakeup tree after $\langle E, sleep(E)\rangle$. We establish that WuT' is a wakeup tree after $\langle E.p, Sleep'\rangle$ by checking the two properties of Definition 6.1. Let $WuT' = \langle B', \prec'\rangle$ and let $wut(E) = \langle B, \prec\rangle$.
  (a) Let w be a leaf of B'; then $p.w$ is a leaf of B. Let $q \in Sleep'$. By the update at line 17, we have $q \in sleep(E)$ and $E \vdash p \diamond q$. By the inductive hypothesis $q \notin WI_{[E]}(p.w)$, which using $E \vdash p \diamond q$ implies $q \notin WI_{[E.p]}(w)$. Hence $WI_{[E.p]}(w) \cap Sleep' = \emptyset$ whenever w is a leaf of B'.
  (b) Let $u.q$ and $u.w$ be nodes in B' with $u.q \prec' u.w$, and $u.w$ is a leaf. Then $p.u.q$ and $p.u.w$ are nodes in B with $p.u.q \prec p.u.w$, and $p.u.w$ a leaf. By the inductive hypothesis, it follows that $q \notin WI_{[E.p.u]}(w)$, which is precisely what we must establish for the inductive step.

  If $wut(E.p)$ is initialized at line 13 instead of at line 10, then this initialization is performed in a call of form $Explore(E.p, Sleep', WuT')$, where WuT' is the empty wakeup tree. By Invariant 2 of the inductive hypothesis, $Sleep'$ is empty. Since $\langle \{\langle\rangle, p\}, \{(p, \langle\rangle)\}\rangle$ is trivially a wakeup tree after $\langle E.p, \emptyset\rangle$, the property follows.
- Consider the update to $sleep(E)$ and $wut(E)$ at lines 20 and 21. We must prove that they preserve the two properties of Definition 6.1.

(a) Since the update removes nodes from $wut(E)$, and p is added to $sleep(E)$, it suffices to check that $p \notin WI_{[E]}(w)$ whenever w is a leaf of $wut(E)$ that remains after the operation, that is, w is not of form $p.w'$. Before the operation, it was the case that p and w were nodes of $wut(E)$ with $p \prec w$ and w a leaf. From the inductive hypothesis, Property 2 of Definition 6.1 implies that $p \notin WI_{[E]}(w)$, which completes the proof for this case.

(b) Since $wut(E)$ is modified by removing a branch from the root, this property is preserved.

(2) Consider a new call to $Explore(E.p, Sleep', WuT')$ at line 19. If WuT' is empty, then p must have been a leaf in $wut(E)$ just before the call. By Property 1 of the inductive hypothesis, $wut(E)$ was then a wakeup tree after $\langle E, sleep(E)\rangle$; therefore (by Property 1 of Definition 6.1), we have $E \nvdash p \diamond q$ for all $q \in sleep(E)$, which by construction of $Sleep'$ at line 17 implies that $Sleep'$ is empty. □

Lemma 7.2. *If $E.p \propto E.w$, then $p \notin I_{[E]}(w)$.*

Proof. We use a proof by contradiction. Assume that $E.p \propto E.w$ but $p \in I_{[E]}(w)$. The property $p \in I_{[E]}(w)$ implies that w is of form $w'.p.w''$ with $E \vdash p \diamond w'$. The property $E.p \propto E.w$ implies that the sequence $E.w$ is explored after the call to $Explore(E.p, \cdot, \cdot)$ has returned. After the call to $Explore(E.p, \cdot, \cdot)$ has returned, we have that $p \in sleep(E)$ by the insertion at line 21. By the way sleep sets are updated at line 17, and the assumption $E \vdash p \diamond w'$, we conclude that p remains in the sleep set at the calls $Explore(E.w''', \cdot, \cdot)$ whenever w''' is a prefix of w'. Thus $p \in sleep(E.w')$ just before the call $Explore(E.w'.p, \cdot, \cdot)$ is performed. By Invariant 1 of Lemma 7.1, we then have that $wut(E.w')$ is a wakeup tree after $\langle E.w', sleep(E.w')\rangle$. To perform the call $Explore(E.w'.p, \cdot, \cdot)$, it must be the case that p is a node in $wut(E.w')$. But this contradicts Property (1) of Definition 6.1, which says that $p \notin WI_{[E.w']}(p.v)$ where v is chosen so $p.v$ is a leaf of $wut(E.w')$. Thus, the lemma is proven. □

The following lemma captures the relationship between wakeup trees and $\langle \mathcal{E}, \propto\rangle$.

Lemma 7.3. *Let $\langle \mathcal{E}, \propto\rangle$ be the tree of explored execution sequences. Consider some point in the execution, and the wakeup tree $wut(E)$ at that point, for some $E \in \mathcal{E}$.*

(1) If $w \in wut(E)$ for some w, then $E.w \in \mathcal{E}$.
(2) If $w \prec w'$ for $w, w' \in wut(E)$, then $E.w \propto E.w'$.

Proof. The properties follow by noting how the exploration from any $E \in \mathcal{E}$ is controlled by the wakeup tree $wut(E)$ at lines 15–21. □

We can now prove that Algorithm 2 is correct in the sense that for each maximal execution sequence E, it explores an execution sequence in $[E]_{\simeq}$. This is formalized in Theorem 7.4. Similarly to Lemma 5.3, Theorem 7.4 is proven by induction on the order in which states (i.e., execution sequences) are backtracked by the algorithm. The inductive step is proven by contradition, assuming that some maximal sequence $E.w$ is unexplored after the call for E returns. The proof of this inductive step is structured as a sequence of claims, structured in the way shown in Figure 8. First it is shown that the assumption implies Claim 7.5, which is analogous to Claim 5.4 in the proof of Lemma 5.3 and essentially states that w is a counterexample to the statement that *final_sleep*(E) is a source set for the set of sequences w with $E \vdash w$. Thereafter, the sequence of Claims 7.6–7.9 establish that the algorithm must have explored some sequence that exposes a race, which by Claim 7.10 causes the algorithm to include a leaf with properties that contradict the initial assumption in the inductive step, thereby concluding the proof of the theorem.

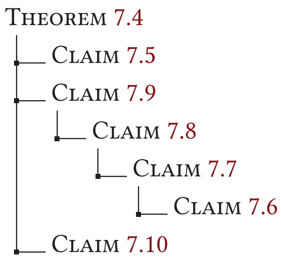

Fig. 8. Structure of correctness proof for Algorithm 2.

Theorem 7.4 (Correctness of Optimal-DPOR). *Whenever a call to* $Explore(E, Sleep, WuT)$ *returns during Algorithm 2, then for all maximal execution sequences* $E.w$*, the algorithm has explored some execution sequence* E' *that is in* $[E.w]_{\simeq}$*.*

Since the initial call to the algorithm uses the arguments $Explore(\langle\rangle, \emptyset, \langle\{\langle\rangle\}, \emptyset\rangle)$, Theorem 7.4 implies that for all maximal execution sequences E the algorithm explores some execution sequence E' that is in $[E]_{\simeq}$. Since any execution sequence can be extended to a maximal one, the theorem also implies the analog of Theorem 5.2, namely that for all execution sequences E, Algorithm 2 explores some execution sequence E' that is in $[E.v]_{\simeq}$ for some v.

Proof. By induction on the set of execution sequences E that are explored during the considered execution, using the ordering $\propto$ (i.e., the order in which the corresponding calls to *Explore* returned).

Base Case: This case corresponds to the first sequence E for which the call $Explore(E, \cdot, \cdot)$ returns. By the algorithm, E is already maximal, so the theorem trivially holds.

Inductive Step: Let us assume that there exist values E, *Sleep* and *WuT*, such that when the call to $Explore(E, Sleep, WuT)$ returns, there is a maximal sequence $E.w$ such that the algorithm has not explored any execution sequence E' in $[E.w]_{\simeq}$. We will show that this leads to a contradiction.

For such w to exist, E cannot be maximal, so $final\_sleep(E)$ contains at least one process. For $p \in final\_sleep(E)$, define

- E'_p, such that $E'_p \leq E$, $E'_p.p \in \mathcal{E}$, and $E'_p.p$ is the last execution sequence of this form that precedes E (w.r.t. $\propto$). If $p \in done(E)$, then $E'_p = E$, but if $p \in Sleep$, then E'_p is a strict prefix of E.
- w'_p, as $E = E'_p.w'_p$. It follows that $E'_p \vdash p \diamond w'_p$.

Inductive Hypothesis: The theorem holds for all execution sequences E' with $E' \propto E$.

Claim 7.5. $WI_{[E]}(w) \cap final\_sleep(E) = \emptyset$.

Using the terminology of source sets, this claim states that w is a counterexample to the statement that $final\_sleep(E)$ is a source set for the set of sequences w with $E \vdash w$.

Proof. By contradiction. Assume that there is a $p \in WI_{[E]}(w)$ with $p \in final\_sleep(E)$. Since w is maximal, $p \in WI_{[E]}(w)$ implies $p \in I_{[E]}(w)$, therefore from Definition 4.1(1) there is a w'' such that $E.w \simeq E.p.w'' \simeq E'_p.w'_p.p.w'' \simeq E'_p.p.w'_p.w''$. By the inductive hypothesis applied to $E'_p.p$, the algorithm has explored some execution sequence in $[E'_p.p.w'_p.w'']_{\simeq} = [E.w]_{\simeq}$, which contradicts the main assumption about w. □

For $p \in \mathit{final\_sleep}(E)$, also define

- w_p, as the longest prefix of w such that $E \vdash p \diamond w_p$,
- e_p, as the first event in $\mathit{dom}_{[E]}(w)$ that is not in w_p. Such an event e_p must exist, otherwise $w_p = w$, which implies $E \vdash p \diamond w$, which implies $p \in \mathit{WI}_{[E]}(w)$, which contradicts Claim 7.5.

Finally, define

- $q \in \mathit{final\_sleep}(E)$, such that w_q is a longest prefix among w_p. If there are several processes $p \in \mathit{final\_sleep}(E)$ such that w_p is the same longest prefix, then pick q such that $E'_q.q$ is minimal (w.r.t. $\propto$).
- w_R, as $w_q.\widehat{e_q}$.
- Sleep' as the initial sleep set argument in the exploration of $E'_q.q$, that is, it started by calling $\mathit{Explore}(E'_q.q, \mathit{Sleep}', \cdot)$.

Claim 7.6. $E'_q \vdash q.w'_q.w_R$.

Proof. Since $E'_q \vdash q$ (actually explored) and $E'_q \vdash q \diamond (w'_q.w_q)$, it follows that $E'_q \vdash w'_q.w_q.q$. Given also that $E'_q \vdash w'_q.w_q.e_q$ and q does not disable e_q (since q and $\widehat{e_q}$ are different), we conclude that $E'_q \vdash w'_q.w_q.q.e_q$ and hence $E'_q \vdash q.w'_q.w_q.e_q$ which can be written as $E'_q \vdash q.w'_q.w_R$. □

Claim 7.7. $\mathit{WI}_{[E'_q.q]}(w'_q.w_R) \cap \mathit{Sleep}' = \emptyset$.

Proof. We will establish the stronger property $\mathit{WI}_{[E'_q.q]}(w'_q.w_q) \cap \mathit{Sleep}' = \emptyset$. Claim 7.6 has shown that the execution sequence is valid. The proof is then by contradiction: Assume that some process p is in $\mathit{WI}_{[E'_q.q]}(w'_q.w_q) \cap \mathit{Sleep}'$.

By the construction of Sleep' at line 17, the process p must be in $\mathit{sleep}(E'_q)$ just before the call to $\mathit{Explore}(E'_q.q, \mathit{Sleep}', \cdot)$ and satisfy $E'_q \vdash p \diamond q$. From $p \in \mathit{WI}_{[E'_q.q]}(w'_q.w_q)$, $E'_q \vdash p \diamond q$, and Property 4.7(5), it follows that $p \in \mathit{WI}_{[E'_q]}(q.w'_q.w_q)$, which, using $E'_q.q.w'_q.w_q \simeq E'_q.w'_q.w_q.q$ (which follows from $E'_q \vdash q \diamond (w'_q.w_q)$), implies $p \in \mathit{WI}_{[E'_q]}(w'_q.w_q.q)$, which by Property 4.7(4) implies $p \in \mathit{WI}_{[E'_q]}(w'_q.w_q)$.

Since, again by Property 4.7(4), $p \in \mathit{WI}_{[E'_q]}(w'_q.w_q)$ also implies $p \in \mathit{WI}_{[E'_q]}(w'_q)$, during exploration of $E'_q.w'_q$ no event in w'_q removes p from the sleep set. Thus, $p \notin w'_q$ and p will end up in $\mathit{sleep}(E'_q.w'_q)$ and from there in $\mathit{final\_sleep}(E'_q.w'_q)$, which means $p \in \mathit{final\_sleep}(E)$. It hence cannot hold that $p \in I_{[E]}(w_q)$, since this by Property 4.2(1) implies $p \in I_{[E]}(w)$, and hence $p \in \mathit{WI}_{[E]}(w)$, violating Claim 7.5. Since there can be no event in w_q happening-before p (otherwise the same event would prohibit $p \in \mathit{WI}_{[E'_q]}(w'_q.w_q)$) the only way for $p \notin I_{[E]}(w_q)$ is by $p \notin w_q$.

Therefore $p \notin w'_q.w_q$ which by $p \in \mathit{WI}_{[E'_q]}(w'_q.w_q)$ entails $E'_q \vdash p \diamond w'_q.w_q$. By choice of q, we then have necessarily that $e_p = e_q$ (otherwise w_p would be longer than w_q). But since among the processes p with $e_p = e_q$ we chose q to be the first one for which a call of the form $\mathit{Explore}(E'_q.p, \cdot, \cdot)$ was performed, we have that $p \notin \mathit{sleep}(E'_q)$ just before the call to $\mathit{Explore}(E'_q.q, \mathit{Sleep}', \cdot)$, whence $p \notin \mathit{Sleep}'$. Thus we have a contradiction. □

Claim 7.8. *Let z' be any sequence such that $E'_q.q.w'_q.w_R.z'$ is maximal (such a z' can always be found, since $E'_q.q.w'_q.w_R$ is an execution sequence). Then, the algorithm explores some sequence $E'_q.q.z$ in $[E'_q.q.w'_q.w_R.z']_{\simeq}$.*

Proof. From Claim 7.7, it follows $\mathit{WI}_{[E'_q.q]}(w'_q.w_R.z') \cap \mathit{Sleep}' = \emptyset$. Therefore, no execution sequence in $[E'_q.q.w'_q.w_R.z']_{\simeq}$ was explored before the call to $\mathit{Explore}(E'_q.q, \mathit{Sleep}', \cdot)$; otherwise, there would be a call $\mathit{Explore}(E''.p, \cdot, \cdot)$ with E'' a prefix of E'_q and $p \in \mathit{Sleep}'$, and, defining w'' by $E''.w'' = E'_q$, we would have $E'' \vdash p \diamond w''$ and $p \in \mathit{WI}_{[E'_q.q]}(w'_q.w_R.z')$, thus contradicting

$\mathit{WI}_{[E'_q.q]}(w'_q.w_R.z') \cap \mathit{Sleep}' = \emptyset$. By the inductive hypothesis for $E'_q.q$ applied to $w'_q.w_R.z'$, the algorithm then explores some sequence $E'_q.q.z$ in $[E'_q.q.w'_q.w_R.z']_{\simeq}$. □

By the construction of w_R, we have $\mathit{next}_{[E'_q]}(q) \precsim_{E'_q.q.w'_q.w_R.z'} e_q$. From $E'_q.q.z \simeq E'_q.q.w'_q.w_R.z'$, it follows that the same race between $\mathit{next}_{[E]}(q)$ and e_q will also occur in $E'_q.q.z$, that is, we have $\mathit{next}_{[E'_q]}(q) \precsim_{E'_q.q.z} e_q$. Since the sequence $E'_q.q.z$ is actually explored by the algorithm, it will encounter the race $\mathit{next}_{[E'_q]}(q) \precsim_{E'_q.q.z} e_q$. When handling it,

- E in the algorithm will correspond to $E'_q.q.z$ in this proof,
- e in the algorithm will correspond to $\mathit{next}_{[E'_q]}(q)$ in this proof, and
- e' in the algorithm will correspond to e_q in this proof,
- $v = (\mathit{notdep}(\mathit{next}_{[E'_q]}(q), E'_q.q.z).\widehat{e_q})$ will be the sequence v at line 5 in Algorithm 2.

Claim 7.9. *$w'_q.w_R \sqsubseteq_{[E'_q]} v$.*

Proof. Let

- x be $\mathit{notdep}(\mathit{next}_{[E'_q]}(q), E'_q.q.z)$
- x' be $\mathit{notdep}(\mathit{next}_{[E'_q]}(q), E'_q.q.w'_q.w_R.z')$. Note that $w'_q.w_q$ is a prefix of x'.

From $E'_q.q.z \simeq E'_q.q.w'_q.w_R.z'$ (Claim 7.8) and the definitions of x and x', it follows that $E'_q.x \simeq E'_q.x'$ and hence that $E'_q.x.\widehat{e_q} \simeq E'_q.x'.\widehat{e_q}$. Let x'' be obtained from x' by adding $\widehat{e_q}$ just after the prefix $w'_q.w_q$ (i.e., in the same place that it has in $w'_q.w_R.z'$). Since e_q "happens after" $\mathit{next}_{[E'_q]}(q)$ in $E'_q.q.w'_q.w_R.z'$, it follows that no events in x' "happen after" e_q in $E'_q.q.w'_q.w_R.z'$. Hence $E'_q.x'' \simeq E'_q.x'.\widehat{e_q}$. Since $w'_q.w_R$ is a prefix of x'', we have $w'_q.w_R \sqsubseteq_{[E'_q]} x''$, which by $E'_q.x'' \simeq E'_q.x'.\widehat{e_q}$ implies $w'_q.w_R \sqsubseteq_{[E'_q]} x'.\widehat{e_q}$, which by $E'_q.x.\widehat{e_q} \simeq E'_q.x'.\widehat{e_q}$ implies $w'_q.w_R \sqsubseteq_{[E'_q]} v$. □

Claim 7.10. *$\mathit{sleep}(E'_q) \cap \mathit{WI}_{[E'_q]}(w'_q.w_R) = \emptyset$.*

Proof. Assume that some process p is in $\mathit{WI}_{[E'_q]}(w'_q.w_R)$. Then

(1) If $p \in w'_q$, then it has no event happening before it, which implies that it cannot have been in $\mathit{sleep}(E'_q)$, since then it could not have been taken out of the sleep set. Thus $p \notin \mathit{sleep}(E'_q)$.
(2) Therefore $p \notin w'_q$. By $p \in \mathit{WI}_{[E'_q]}(w'_q.w_R)$, we have that $E'_q \vdash p \diamond w'_q$, which assuming $p \in \mathit{sleep}(E'_q)$, means that p will still be in the sleep set after w'_q and therefore $p \in \mathit{final\_sleep}(E)$. Then:
  (a) If $p \in w_R$, then $p \in I_{[E]}(w_R)$, from which we get $p \in I_{[E]}(w)$ therefore $p \in \mathit{WI}_{[E]}(w)$. Since $p \in \mathit{final\_sleep}(E)$ this contradicts Claim 7.5.
  (b) Therefore $p \notin w_R$ must hold. From $p \in \mathit{WI}_{[E'_q]}(w'_q.w_R)$ and $p \notin w'_q.w_R$, we have that $E'_q \vdash p \diamond w'_q.w_R$, which implies $E'_q.w'_q \vdash p \diamond w_R$, which is equivalent to $E \vdash p \diamond w_R$. But then $w_R = w_q.\widehat{e_q}$ is a prefix of w_p, implying that w_p is strictly longer than w_q. This contradicts the fact that q was chosen as the process in $\mathit{final\_sleep}(E)$ with the longest prefix w_q satisfying $E \vdash q \diamond w_q$.

Therefore, there can be no such $p \in \mathit{WI}_{[E'_q]}(w'_q.w_R)$ and Claim 7.10 is proven. □

From Claim 7.9, Claim 7.10, and Property 4.7(4), we get $\mathit{sleep}(E'_q) \cap \mathit{WI}_{[E'_q]}(v) = \emptyset$. Thus, the test at line 6 will succeed, and after performing line 7, the wakeup tree $\mathit{wut}(E'_q)$ will (by the specification of *insert*) contain a leaf y such that $y \sim_{[E'_q]} v$. Since at this point $q \in \mathit{wut}(E'_q)$ and $q \notin \mathit{WI}_{[E'_q]}(v)$ we have, by definition of *insert*, that $E'_q \nvdash q \diamond y$. By Property 7.3(1), we then have $E'_q.y \in \mathcal{E}$.

From Claim 7.9 and $y \sim_{[E'_q]} v$, it follows by Property 4.7(2) that $y \sim_{[E'_q]} w'_q.w_R$. Furthermore, from $E'_q \vdash q \diamond w'_q$ (which follows from $E'_q \vdash q \diamond w'_q.w_q$) and $E'_q \nvdash q \diamond y$, it follows that y is not a prefix of w'_q.

Let u be the longest common prefix of y and $w'_q.w_R$. We claim that w'_q is a strict prefix of u. Otherwise, there are different processes p, p' and a sequence v'' such that $u.p'.v'' = w'_q$ and $u.p$ is a prefix of y. From $y \sim_{[E'_q]} w'_q.w_R$ and Property 4.7(2), we infer $y \sim_{[E'_q]} u.p'$. If $u.p' \in wut(E'_q)$ when y is inserted, then we infer by Lemma 6.2 and Property 7.3(2) that $E'_q.y \propto E'_q.u.p'$. If $u.p' \notin wut(E'_q)$ when y is inserted, then we also infer $E'_q.y \propto E'_q.u.p'$, since then y will be explored before $u.p'$. Thus $E'_q.y \propto E'_q.u.p'$, which implies $E'_q.u.p \propto E'_q.u.p'.v''$, which by handling of sleep sets implies $p \notin I_{[E'_q.u]}(p'.v'')$. By $y \sim_{[E'_q]} w'_q$, implying $u.p \sim_{[E'_q]} u.p'.v''$, we have $p \sim_{[E'_q.u]} p'.v''$, which is the same as $p \in WI_{[E'_q.u]}(p'.v'')$. By $p \notin I_{[E'_q.u]}(p'.v'')$, this implies $E'_q.u \vdash p \diamond (p'.v'')$. This implies that $p \in sleep(E'_q.w'_q)$, that is, $p \in sleep(E)$. Hence by the construction of w_R, we have $p \notin WI_{[E'_q.u.p'.v'']}(w_R)$, which together with $E'_q.u \vdash p \diamond (p'.v'')$ implies $p \notin WI_{[E'_q.u]}(p'.v''.w_R)$, which implies $u.p \not\sim_{[E'_q]} u.p'.v''.w_R$, which implies $y \not\sim_{[E'_q]} w'_q.w_R$, which contradicts the construction of y.

Thus, w'_q is a strict prefix of u. Since $E'_q.y$, and hence $E'_q.u$, is explored by the algorithm, we have $E'_q.u \propto E'_q.w'_q$. Moreover, since u is a prefix of $w'_q.w_R$, we infer that $E'_q.u$ is a prefix of $E'_q.w'_q.w$. This means that there is a sequence w'' such that $E'_q.u.w'' \simeq E.w$. It follows by the inductive hypothesis applied to $E'_q.u$ that the algorithm has explored some maximal sequence in $[E'_q.u.w'']_{\simeq}$ and hence in $[E.w]_{\simeq}$. This contradicts the assumption at the beginning of the inductive step. This concludes the proof of the inductive step, and Theorem 7.4 is proven. □

7.3 Optimality

In this section, we prove that optimal-DPOR is optimal in the sense that it never explores two different but equivalent execution sequences and never encounters sleep set blocking. The following theorem, which is essentially the same as Theorem 3.2 of Godefroid et al. (1995), establishes that sleep sets alone are sufficient to prevent exploration of two equivalent *maximal* execution sequences.

Theorem 7.11. *Optimal-DPOR never explores two maximal execution sequences that are equivalent.*

Proof. Assume that E_1 and E_2 are two equivalent maximal execution sequences that are explored by the algorithm. Then they are both in $\mathcal{E}$. Assume w.l.o.g. that $E_1 \propto E_2$. Let E be their longest common prefix, and let $E_1 = E.p.v_1$ and $E_2 = E.v_2$. By Lemma 7.2, we have $p \notin I_{[E]}(v_2)$, which contradicts $E_1 \simeq E_2$ and the maximality of E_1 and E_2. □

We will now prove that Algorithm 2 is optimal in the sense that it never encounters sleep set blocking. Let us first define this precisely.

Definition 7.12 (Sleep Set Blocking). A call to *Explore*$(E, Sleep, WuT)$ is *sleep set blocked* during the execution of Algorithm 2 if $enabled(s_{[E]}) \neq \emptyset$ and $enabled(s_{[E]}) \subseteq Sleep$.

Now let us state and prove the corresponding optimality theorem.

Theorem 7.13 (Optimality of Optimal-DPOR). *During any execution of Algorithm 2, no call to Explore*$(E, Sleep, WuT)$ *is ever sleep set blocked.*

Proof. Consider a call *Explore*$(E, Sleep, WuT)$ during the exploration. Then any sequence in WuT is enabled after E. By Lemma 7.1, WuT is a wakeup tree after $\langle E, Sleep \rangle$. Thus, if $Sleep \neq \emptyset$, then WuT contains a sequence w such that $Sleep \cap WI_{[E]}(w) = \emptyset$. Letting p be the first process in w, this implies $p \notin Sleep$, implying that p is enabled and thus $enabled(s_{[E]}) \not\subseteq Sleep$. □

8 EXTENDING SOURCE-DPOR AND OPTIMAL-DPOR TO SUPPORT BLOCKING

In this section, we consider how to extend the techniques of the previous sections for the general situation in which processes can disable each other. In other words, we will no longer make use

ALGORITHM 3: Source-DPOR algorithm considering locks.

```
Initial call: $Explore(\langle\rangle, \emptyset)$

$Explore(E, Sleep)$
    foreach lock $l$ do
        foreach $p$ whose next operation after $E$ is lock($l$) do
            let $e$ be the last event in $E$ with $op_{[E]}(e) = \texttt{lock}(l)$;
            let $E' = pre(E, e)$;
            let $v = notdep(e, E).p$;
            if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ then
                add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;
    if $\exists r \in (enabled(s_{[E]}) \setminus Sleep)$ then
        $backtrack(E) := \{r\}$;
        while $\exists p \in (backtrack(E) \setminus Sleep)$ do
            foreach $e \in dom(E)$ such that$(e \precsim_{E.p} next_{[E]}(p))$ do
                let $E' = pre(E, e)$;
                let $v = notdep(e, E).p$;
                if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ then
                    add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;
            let $Sleep' = \{q \in Sleep \mid E \vdash p \diamond q\}$;
            $Explore(E.p, Sleep')$;
            add $p$ to $Sleep$;
```

of Assumption 3.1. Processes can disable each other via shared synchronization objects, such as locks, condition variables, and so on. It is also possible for processes to disable each other via less-visible mechanisms; one such method is to use `await` statements, by means of which any update to a shared variable can disable an `await` statement that depends on this shared variable. When extending our techniques to the case in which processes can disable each other, it turns out that the general case of `await` statements requires a more sophisticated modification than the case of locks. We will therefore first consider how to extend Algorithm 1 and 2 to handle locks and thereafter consider the general case.

8.1 Handling Locks

Let us extend Algorithm 1 and 2 to the case that disabling between processes happens only via locks. We assume a set of locks that are disjoint from ordinary memory locations and only used in lock operations. Each lock l can take values 0 or 1 (initial value) and supports the operations lock(l) and unlock(l), where lock(l) decrements l from 1 to 0 if l is 1 and blocks if l is 0 and where unlock(l) assigns 1 to l (and never blocks). The only operation that can disable another process is lock(l), which disables any other operation of form lock(l). We assume that the happens-before relation relates a lock(l) operation with any other operation of form lock(l) or unlock(l) on the same lock, that the lock(l) operation is independent of other operations, and that two unlock(l) operations by different threads are independent. Note that we do not assume that programs use locks in a "well-structured" way or that the program is data-race-free.

8.1.1 Extending Source-DPOR to Handle Locks.. To handle locks, the basic modification to Algorithm 1 is to extend the concept of races. An important part of Algorithm 1 (at line 5) is to detect a race between two events e and $next_{[E]}(p)$. Let us denote $next_{[E]}(p)$ by e'. In the absence of locks, whenever e is in a race with e', and e is performed before e', then e' must appear sometime later in the execution, since it cannot be disabled by e. This implies that the race detection mechanism of line 5 in Algorithm 1 is sufficient for detecting any race. However, in the presence of locks, we must detect races also in the case where e' has been disabled by e. The needed extension is to consider the case where e' performs a lock() operation, regardless of whether $\widehat{e'}$ is blocked or not, and to investigate alternative executions in which e' is performed before the most recently performed lock() operation. Such an extension should work when $\widehat{e'}$ is blocked, since it is necessary to explore the case where e' occurs before the most recently performed lock() operation, when it is not blocked. Additionally, it should *also* work when $\widehat{e'}$ is not blocked. The reason is that Algorithm 1 will only find the "race" between the enabled lock event e' and the previous unlock() operation: Obviously, these two events cannot be reversed. Instead, it is necessary to consider the race between e' and the most recently performed lock() operation; these events can be performed in reverse order.

Our extension of Algorithm 1 to programs with locks is shown in Algorithm 3. The extension consists of adding an extra case to the race detection phase, found at lines 2–8. This extra case considers all processes p that are about to perform a `lock()` operation on some lock l, regardless of whether they are blocked. For each such process, we find the preceding `lock(`l`)` operation in E, since that operation is in a race with the next step of process p, according to our extended definition of "race." Note that the next step of process p, which is a `lock(`l`)` operation, can be either enabled or disabled. Thereafter, the race detection proceeds analogously as in the non-blocking case: At line 6, the sequence v is constructed as a sequence of events that can follow E' in an execution where p takes the lock before $\widehat{e}$. At line 7, it is tested whether the backtrack set at E' already contains a process that makes such an extension of E' possible; otherwise, such a process is added at line 8.

8.1.2 Correctness of Algorithm 3.. Let us next consider how to extend the proof of correctness for Algorithm 1 to become a proof of correctness for Algorithm 3. Here we present the needed modifications to the proof of Theorem 5.2. We thus consider how to extend the proof of Lemma 5.3.

The proof remains the same as in Section 5.2 up to Property (9), stated in the proof of Claim 5.4. This property, that is, $E \vdash q.w_R$, is no longer true if there is a lock l such that $next_{[E]}(q)$ and e_q perform the operation `lock(`l`)`, since $\widehat{e_q}$ is now disabled after $E.q.w_q$. We therefore here explain how the proof should be continued for this case; all other cases can be handled as in Section 5.2.

We first infer, from $E \vdash q \diamond w_q$, that w_q does not contain any operation on the lock l. We then replace Claim 5.5 by the following stronger one:

For all $p \in done(E)$ with $p \neq q$ we have $p \notin WI_{[E]}(q.w_q)$.

This property is actually shown in Cases 1 and 2a in the proof for Claim 5.5. The other cases, that is, Cases 2b and 2c, are not applicable when $next_{[E]}(q)$ and e_q both perform the operation `lock(`l`)`, since these cases assume $e_p = e_q$ and $E \vdash p \diamond q$, which is not possible when e_q performs the operation `lock(`l`)`. This follows by noting that, since the next events of p and q conflict with e_q, they must both be lock or unlock operations on l. Since $next_{[E]}(q)$ performs the operation `lock(`l`)`, we infer $E \nvdash p \diamond q$.

We next establish a slight modification of IH-condition 5.6, namely that $w_q \in to\_be\_explored(E.q)$. The proof of this is contained in the proof of IH-condition 5.6: We first note in formula (12) that $p \notin I_{[E.q]}(w_q)$ for any $p \in done(E)$ and then in formula (13) that $p \notin I_{[E.q]}(w_q)$ for any $p \in init\_sleep(E)$, concluding that $I_{[E.q]}(w_q) \cap init\_sleep(E.q) = \emptyset$.

We next claim that the algorithm will actually explore an execution sequence of form $E.q.w'$ that is in $[E.q.w_q.v'']_{\simeq}$ for some v'' (i.e., such that $w_q \sqsubseteq_{[E.q]} w'$), such that if w' contain some `lock(`l`)`

operation(s), then e_q is the first of these. To prove this claim, note that after $E.q.w_q$, process $\widehat{e_q}$ is prepared to perform a `lock(`l`)` operation and will continue to be prepared to do so all the time until it actually performs such a `lock(`l`)` operation (note that immediately after $E.q.w_q$, the lock l is still taken). We split the proof into two cases:

(1) If *to_be_explored*$(E.q)$ contains a sequence of form $w_q.z$ in which z contains an `unlock(`l`)` operation, then *to_be_explored*$(E.q)$ also contains such a sequence of form $w_q.z$ in which event e_q immediately follows the first of these `unlock(`l`)` operations, since process $\widehat{e_q}$ is continuously prepared to perform `lock(`l`)` operation after $E.q.w_q$. The inductive hypothesis for $E.q$ guarantees that the algorithm explores $E.q.w'$ for some w' with $z \sqsubseteq_{[E.q]} w'$. Since the operations `lock(`l`)` are dependent with each other, e_q is the first event in w' to perform a `lock(`l`)` operation.

(2) If *to_be_explored*$(E.q)$ contains no sequence of form $w_q.z$ in which z contains an `unlock(`l`)` operation, then the inductive hypothesis for $E.q$ guarantees that the algorithm explores $E.q.w'$ for some w' with $w_q \sqsubseteq_{[E.q]} w'$. Since w' must be in *to_be_explored*$(E.q)$, we conclude that w' contains no `unlock(`l`)`, and hence no `lock(`l`)` operation.

Having proven this claim, let u be the shortest prefix of w' such that $w_q \sqsubseteq_{[E.q]} u$. Note that after w_q, the event e_q (with operation `lock(`l`)`) is the next to be performed by $\widehat{e_q}$, although the lock l may still be taken immediately after u. Since, according to the just-proven claim, u contains no `lock(`l`)` operation, then $\mathit{next}_{[E]}(q)$ is the last event in $E.q.u$ that performs a `lock(`l`)` operation, and so the race detection at lines 2–8 of Algorithm 3 will make sure that *done*(E) contains some q' with $q' \in I_{[E]}(v)$, where $v = v'.\widehat{e_q}$ for $v' = \mathit{notdep}(\mathit{next}_{[E]}(q), E.q.u)$. In this case, we derive a contradiction in a similar way as in the proof of Claim 5.4. By the construction of v', we have $v' \sqsubseteq_{[E.q]} w'$. From $w_q \sqsubseteq_{[E.q]} w'$ and $v' \sqsubseteq_{[E.q]} w'$, and Property 4.7(3), we infer $w_q \sim_{[E.q]} v'$. This implies, using $E \vdash q \diamond w_q$ and $E \vdash q \diamond v'$, that $w_q \sim_{[E]} v'$ (this can be established in a similar way as Lemma 5.1.3). Now $q' \in I_{[E]}(v)$ means that either (i) $q' \in I_{[E]}(v')$ or that (ii) $q' = \widehat{e_q}$ and $E \vdash q' \diamond v'$. Case (i) implies (by the construction of v') that $\mathit{next}_{[E]}(q')$ does not perform any operation on the lock l and (using $w_q \sim_{[E]} v'$) that $q' \in \mathit{WI}_{[E]}(w_q)$. Since e_q performs the operation `lock(`l`)`, this implies $q' \in \mathit{WI}_{[E]}(w_R)$, which contradicts Property (10). In Case (ii), from $w_q \sqsubseteq_{[E.q]} u$ and $E \vdash q \diamond w_q$, the construction of v' implies $w_q \sqsubseteq_{[E.q]} v'$. Also considering $E \vdash q \diamond v'$, we infer $w_q \sqsubseteq_{[E]} v'$. Since $q' \in \mathit{done}(E)$, the construction of w_R implies that $E \nvdash q' \diamond w_q$, which together with $w_q \sqsubseteq_{[E]} v'$ implies $E \nvdash q' \diamond v'$, contradicting the assumption $E \vdash q' \diamond v'$ for case (ii).

This concludes the proof of Claim 5.4. The rest of the proof proceed as in the proof of Theorem 5.2. □

8.1.3 Extending Optimal-DPOR to Handle Locks.. Let us next present our extension of Algorithm 2 to programs with locks, resulting in Algorithm 4. Similarly as in Algorithm 3, the algorithm extends the lock detection phase to consider races between `lock()` operations, one of which is potentially disabled. The extension for locks is handled by lines 8–14. These lines handle the case when event e' in a race $e \precsim_E e'$ is actually blocked for the reason that both e and e' are `lock(`l`)` operations on the same lock l. In this case, e' may not be present in the execution sequence E at a position where it is blocked. For this case, we construct the sequence v as in line 6 of Algorithm 3. First, at line 10, the events that happen-after e are removed from E, resulting in the suffix w. We then construct v by adding a process p that is prepared to perform a `lock(`l`)` operation. Since by construction w does not contain any operations on the lock l, and since the lock could be taken after E by $\widehat{e}$, it is ensured that the lock can be taken after $E'.w$ by any process that is ready to do so. Thereafter, we perform the same test as in Algorithm 2 to decide whether the wakeup tree at E' must be extended to explore some sequence that is equivalent to $E'.v$.

ALGORITHM 4: Optimal-DPOR algorithm considering locks.

```
Initial call: $Explore(\langle\rangle, \emptyset, \langle\{\langle\rangle\}, \emptyset\rangle)$

$Explore(E, Sleep, WuT)$
    if $enabled(s_{[E]}) = \emptyset$ then
        foreach $e, e' \in dom(E)$ such that $(e \precsim_E e')$ do
            let $E' = pre(E, e)$;
            let $v = (notdep(e, E).\widehat{e'})$;
            if $sleep(E') \cap WI_{[E']}(v) = \emptyset$ then
                $wut(E') := insert_{[E']}(v, wut(E'))$;
        foreach $e \in dom(E)$ such that $op_{[E]}(e) = \texttt{lock}(l)$ do
            let $E' = pre(E, e)$;
            let $w = notdep(e, E)$;
            foreach $p$ such that $op_{[E'.w.p]}(next_{[E'.w]}(p)) = \texttt{lock}(l)$ do
                let $v = w.p$;
                if $sleep(E') \cap WI_{[E']}(v) = \emptyset$ then
                    $wut(E') := insert_{[E']}(v, wut(E'))$;
    else
        if $WuT \neq \langle\{\langle\rangle\}, \emptyset\rangle$ then
            $wut(E) := WuT$;
        else
            choose $p \in enabled(s_{[E]})$;
            $wut(E) := \langle\{p\}, \emptyset\rangle$;
        $sleep(E) := Sleep$;
        while $\exists p \in wut(E)$ do
            let $p = \min_{\prec}\{p \in wut(E)\}$;
            let $Sleep' = \{q \in sleep(E) \mid E \vdash p \diamond q\}$;
            let $WuT' = subtree(wut(E), p)$;
            $Explore(E.p, Sleep', WuT')$;
            add $p$ to $sleep(E)$;
            remove all sequences of form $p.w$ from $wut(E)$;
```

Let us next consider how to extend the soundness proof of Algorithm 2 to cover Algorithm 4. The overall structure of the proof remains the same; below we indicate where and how the proof needs to be extended.

The first extension is needed when we get to Claim 7.6, and the event performed by q is of form $\texttt{lock}(l)$. In this case, e_q must also perform $\texttt{lock}(l)$ (by another process). But it is obvious that now $E'_q \vdash q.w'_q.w_R$ does not hold, since both $next_{[E'_q]}(q)$ and e_q take the lock. We therefore replace Claim 7.6 with $E'_q \vdash q.w'_q.w_q$. For similar reasons, Claim 7.7 must be replaced by the stronger $WI_{[E'_q.q]}(w'_q.w_q) \cap Sleep' = \emptyset$, which is actually also proven in the text of the proof of Claim 7.7. Continuing along these lines, the text of Claim 7.8 should be changed to the following:

> *Let z' be any sequence such that $E'_q.q.w'_q.w_q.z'$ is maximal. (Such a z' can always be found, since $E'_q.q.w'_q.w_q$ is an execution sequence.) Then, the algorithm explores some sequence $E'_q.q.z$ in $[E'_q.q.w'_q.w_q.z']_{\simeq}$.*

```
                          Initially: x = y = 0
  p :              q :              r :              s :
  x +:= 1;         x +:= 1;         x +:= 3;         await (x = 3)
                                                         then y := 1;
```

Fig. 9. Example motivating line 11 of Algorithm 6.

After the proof of Claim 7.8, the algorithm will handle the new kind of race (lines 8–14) between `lock(l)` operations, with $v = w.p$. At the end, we end up with the same Claim 7.9 as in the proof, that is, $w'_q.w_R \sqsubseteq_{[E'_q]} v$, where the last process in w_R is actually $\widehat{e_q}$. Here, it is important to observe that e_q, being of form `lock(l)`, is actually independent with all events in w except possibly the ones already in $w'_q.w_q$. Thereafter, the proof goes on just as in the non-blocking case.

8.2 Allowing Arbitrary Blocking in the Execution Model

Let us next consider how to extend algorithms Source-DPOR and Optimal-DPOR to a general execution model, where disabling between processes can happen via any operations, in a computational model that need only satisfy the requirements of Definition 3.2. If disabling between processes happens only via clearly identified synchronization mechanisms, such as locks, then Algorithm 3 and 4 with obvious modifications are sufficient. However, if disabling can occur in less obvious ways, the modification is less straightforward. The reason is that, in the general case, races may be more difficult to detect. We illustrate this by a small example.

Consider the program in Figure 9. All four statements are atomic. In this example, we define the following happens-before relation on the events in an execution sequence: Let the statement of s be related with any other statement, and let the statements of processes p, q, and r all be unrelated if the statement of s does not occur somewhere between them. The statements of processes p, q, and r are all related if the statement of s occurs between them. Such a happens-before relation is somewhat non-standard; another natural choice of happens-before relation would be to let it relate all pairs of statements, since they all touch the variable x. However, our choice is also consistent with the requirements on a happens-before relation in Definition 3.2, since the order in which the atomic statements of p, q, and r are performed does not matter when they occur consecutively. With this happens-before assignment, the program has only two Mazurkiewicz traces: one in which only p, q, and r occur (whereafter s is blocked) and one consisting of $r.s.p.q$ and $r.s.q.p$.

To illustrate the difficulty of detecting races, suppose that we first explore the sequence $p.q.r$, after which s is blocked. In some way, we must now define a "race." There is a race between p and s, but such a race is visible only in explorations that start with $r.p$. More generally, this example shows that in the performed execution sequence (in this case $p.q.r$), one may have to extract a subsequence (in this case r) to make the race visible. The subsequence r should clearly be consistent with $p.q.r$ in the sense that $r \sqsubseteq_{[\langle\rangle]} p.q.r$.

Let us from this example extract a general pattern for how to extend the race detection phase of our algorithms. In this race detection, we should try to find a sequence that plays the role of v, as defined in line 6 in Algorithm 3 and in line 12 of Algorithm 4. In Algorithms 3 and 4, we could take v to be the sequence of all statements that do not happen-after p, that is, $q.r.s$, since any happens-before closed subset of v will enable s. However, in the present case, it may happen that only a specific subsequence of v can enable s. In this case, the subsequence is r, implying that we should take v as $r.s$. Continuing with the example in Figure 9, we see that there are several possible choices for the sequence v, but of all these only r will lead to a state where s can be disabled by

ALGORITHM 5: Source-DPOR algorithm for general blocking.

```
   Initial call: $Explore(\langle\rangle, \emptyset)$
1  $Explore(E, Sleep)$
2      foreach $p$ that is blocked in $s_{[E]}$ do
3          foreach $e \in dom(E)$ do
4              let $E' = pre(E, e)$;
5              let $w = notdep(e, E)$;
6              foreach subsequence $u$ of $w$ such that $u \sqsubseteq_{[E']} w$ do
7                  if $\widehat{e}$ disables $p$ after $E.u$ then
8                      let $v = u.p$;
9                      if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ then
10                         add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;
11     if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ then
12         $backtrack(E) := \{p\}$;
13         while $\exists p \in (backtrack(E) \setminus Sleep)$ do
14             foreach $e \in dom(E)$ such that$(e \precsim_{E.p} next_{[E]}(p))$ do
15                 let $E' = pre(E, e)$;
16                 let $v = notdep(e, E).p$;
17                 if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ then
18                     add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;
19             let $Sleep' = \{q \in Sleep \mid E \vdash p \diamond q\}$;
20             $Explore(E.p, Sleep')$;
21             add $p$ to $Sleep$;
```

another process. It is, of course, possible to develop optimization schemes that would disregard many of these choices, but we here refrain from doing this to keep the structure of the algorithm simple.

Let us now incorporate these insights into general extensions of Algorithms 3 and 4. The generalization of Algorithm 3 is shown in Algorithm 5. The extended race detection phase is shown at lines 2–10. Whenever a process p is blocked, we try to detect whether some other process $\widehat{e}$ disables p. As described above, we may have to move $\widehat{e}$ past some subsequence u to make the disabling visible. As in Algorithm 3, the algorithm first extracts the sequence w that does not happen-after e. Thereafter, it considers all possible subsequences u of w that are consistent w.r.t. the happens-before relation. Whenever $\widehat{e}$ disables some process p after u, then the sequence $u.p$ is used as v in the race detection phase. The rest of the race detection proceeds as before.

The proof of soundness for Algorithm 5 can be obtained by a small modification of the proof for Algorithm 1. As for Algorithm 3, we must replace w_R by w_q in Claim 5.4, Claim 5.5, and in related properties, just as in the proof for Algorithm 3. We define w'_q precisely as in the proof for Algorithm 3 and note that $w'_q \sqsubseteq_{[E.q]} w'$. Let w''_q be the reordering of the events in w'_q as they occur in w'. Then we can take u as w''_q at line 6 of the algorithm. By construction q disables $\widehat{e_q}$ after $E.w''_q$, and the rest of the proof proceeds as in the proof for Algorithm 3.

The extension of Algorithm 4 is shown in Algorithm 6. The corresponding extension of the race detection phase is now at lines 8–15.

ALGORITHM 6: Optimal-DPOR algorithm for general blocking.

```
    Initial call: $Explore(\langle\rangle, \emptyset, \langle\{\langle\rangle\}, \emptyset\rangle)$
1   $Explore(E, Sleep, WuT)$
2       if $enabled(s_{[E]}) = \emptyset$ then
3           foreach $e, e' \in dom(E)$ such that $(e \precsim_E e')$ do
4               let $E' = pre(E, e)$;
5               let $v = (notdep(e, E).\widehat{e'})$;
6               if $sleep(E') \cap WI_{[E']}(v) = \emptyset$ then
7                   $wut(E') := insert_{[E']}(v, wut(E'))$;
8           foreach $e \in dom(E)$ do
9               let $E' = pre(E, e)$;
10              let $w = notdep(e, E)$;
11              foreach subsequence $u$ of $w$ such that $u \sqsubseteq_{[E']} w$ do
12                  foreach $p$ such that $\widehat{e}$ disables $p$ after $E'.u$ do
13                      let $v = u.p$;
14                      if $sleep(E') \cap WI_{[E']}(v) = \emptyset$ then
15                          $wut(E') := insert_{[E']}(v, wut(E'))$;
16      else
17          if $WuT \neq \langle\{\langle\rangle\}, \emptyset\rangle$ then
18              $wut(E) := WuT$;
19          else
20              choose $p \in enabled(s_{[E]})$;
21              $wut(E) := \langle\{p\}, \emptyset\rangle$;
22          $sleep(E) := Sleep$;
23          while $\exists p \in wut(E)$ do
24              let $p = \min_{\prec}\{p \in wut(E)\}$;
25              let $Sleep' = \{q \in sleep(E) \mid E \vdash p \diamond q\}$;
26              let $WuT' = subtree(wut(E), p)$;
27              $Explore(E.p, Sleep', WuT')$;
28              add $p$ to $sleep(E)$;
29              remove all sequences of form $p.w$ from $wut(E)$;
```

Let us consider how to extend the soundness proof of Algorithm 2 to cover Algorithm 6. As for the locking case, the structure of the proof remains the same. The main modifications are as follows:

- Claim 7.6 is replaced by $E'_q \vdash q.w'_q.w_q$;
- Claim 7.7 is replaced by $WI_{[E'_q.q]}(w'_q.w_q) \cap Sleep' = \emptyset$;
- in Claim 7.8, w_R is replaced by w_q.

In the text between Claim 7.8 and Claim 7.9, the text about race detection should be replaced by the observation that by choosing u as the subsequence of w with $w'_q.w_q \simeq E'_q u$, then the test at line 12 will succeed. The rest of the proof proceeds as for Algorithm 2.

Initially q_1, r_1, s_1 and all p_i are spawned. Also, $\mathtt{x}_i = \mathtt{y}_i = \mathtt{z}_i = 0$, for all i.

```
$p_i$ :
if $x_i$ = 1 && i < n then
  spawn $q_{i+1}$;
  spawn $r_{i+1}$;
  spawn $s_{i+1}$;
```

```
$q_i$ :
$y_i$ := 1;
```

```
$r_i$ :
m := $y_i$;
if m = 0 then
  $z_i$ := 1;
```

```
$s_i$ :
n := $z_i$;
l := $y_i$;
if n = 1 then
  if l = 0 then
    $x_i$ := 1;
```

Fig. 10. The program of Figure 2 extended with more processes and global variables.

9 COMPARISON AND TRADEOFFS BETWEEN SOURCE-DPOR AND OPTIMAL-DPOR

In this section, we compare source-DPOR and optimal-DPOR on synthetic programs that highlight interesting performance differences, based on the number of explored interleavings, as well as in time and memory consumption. More specifically, we present:

(1) A program that shows that the use of wakeup trees allows optimal-DPOR to avoid exploration of an exponential number of sleep set blocked traces. In contrast, non-optimal DPOR algorithms in general, and source-DPOR in particular, are forced to explore these traces and therefore can require arbitrarily more time than optimal-DPOR. As we describe, this does not happen only under the worst scheduling scenario but also even when the scheduling algorithm that is employed is a "reasonable" one.
(2) A simpler program, which we also use as a benchmark in the next section, that also shows that a wakeup sequence is necessary for the optimal algorithm to avoid sleep set blocking.
(3) A program that shows that wakeup trees may make the optimal algorithm require exponentially more space than source-DPOR.

The first program, shown in Figure 10, is an extended version of the example presented earlier in Figure 2. It has been extended in the following way: n determines how many groups of four processes (which correspond to p, q, r, and s from Figure 2) are used. Each of the q_i, r_i, and s_i processes operate on the respective $\mathtt{x}_i$, $\mathtt{y}_i$, and $\mathtt{z}_i$ variables. The code of the p_i processes reads $\mathtt{x}_i$ and only spawns q_{i+1}, r_{i+1} and s_{i+1} if the write on $\mathtt{x}_i$ by s_i has been completed. This pair of read-write operations is in a race. The other processes also have racing read-write pairs on variables $\mathtt{y}_i$ and $\mathtt{z}_i$.

When starting an interleaving to reverse the race of the top s_n process with the top p_n process, the only way for p_n to be removed from the sleep set is for each "lower" group to be scheduled exactly as shown in Figure 3, as this is the only way for s_n to execute its write to $\mathtt{x}_n$. If a "wrong step" is taken at any point, then s_n will not do the write and the respective trace will be sleep set blocked.

For $n = 1$, there exist four interleavings where the read on $\mathtt{x}_1$ by p_1 happens before the respective write by s_1 and one more where it appears afterwards, just as shown in Figure 3. However, with just the first step (which must be taken by r_1) specified, it can be the case for source-DPOR to schedule q_1 before s_1 has read $\mathtt{y}_1$, thus altering s_1's behaviour and leading to sleep set blocking. As there are four ways to schedule q_n, r_n, and s_n processes, only one of which reveals the write on $\mathtt{x}_n$, p_n will remain in the sleep set and the other three schedules will encounter sleep set blocking.

For higher values of n, when the race of $\mathtt{x}_i$ in the ith group is reversed successfully, four new interleavings are possible by the $i + 1$-th group. The number of Mazurkiewicz traces for this program is $4n + 1$: four traces for the "top" group, plus a fifth that splits in four when it spawns the

Table 1. Traces Explored by Source- and Optimal-DPOR for the Program in Figure 10

	Traces Explored			Time	
n	Source	Optimal	Sleep Set Blocked	Source	Optimal
1	8	5	3	0.03s	0.02s
2	20	9	11	0.08s	0.04s
3	41	13	28	0.20s	0.07s
4	78	17	61	0.41s	0.10s
5	145	21	124	0.81s	0.17s
6	268	25	243	1.56s	0.20s
7	373	29	344	2.38s	0.26s
8	674	33	641	4.70s	0.34s
9	1,222	37	1,185	8.79s	0.44s

```
   Shared variables:  int array[0..N] := {0,0,...,0};

          Process 0:  for (int i := N; array[i] != 0; i--);
Process j (j ∈ 1..N):  array[j] := array[j-1] + 1;
```

Fig. 11. Pseudocode of the lastzero(N) example; measurements for it can be found in Section 11.

next group of processes and so on until the final group, which does not split and has just a single extra interleaving.

Wrong schedulings within the ith group are still sleep set blocked. Additionally, to reverse the pair operating on x_{i+1}, all the lower groups must be scheduled correctly, but this is only possible in exactly one interleaving, whereas all deviations lead to sleep-set-blocked interleavings. As a result, the tree of the "previous" level (where p_{i+1} was not in a sleep set) is duplicated as sleep-set-blocked explorations, leading to the exponential growth of traces that encounter sleep set blocking shown in Table 1. This program shows therefore an exponential difference between non-optimal DPOR algorithms (and source-DPOR) and optimal-DPOR. As seen in Table 1, this difference does not just affect the number of traces that the two algorithms explore but also the time performance of the two algorithms. Thus, optimal-DPOR can perform exponentially better in time than non-optimal DPOR algorithms.

The second program is lastzero(N), whose pseudocode is shown in Figure 11. Its N+1 processes operate on an array of N+1 elements that are all initially zero. In this program, process 0 searches the array for the highest index that has a zero value, while the other N processes read one of the array values and write that value increased by one in the next index. The final state of the program is uniquely defined by the values of i and array[1..N].

Here, again, process 0 has control flow that depends on the values in the array, which are all exposed to racing operations. This is generally a typical case where source-DPOR may encounter sleep set blocking, which optimal-DPOR can avoid. If some of the processes with higher indices have not completed their execution, then processes with lower indices that are in a sleep set, because of a race with process 0's access, will never be removed, as process 0 will "stop" at a high index and not access the lower indices. Again, an accurate wakeup sequence is necessary to avoid sleep set blocking in this program. We present measurements for the lastzero(N) program in Section 11.

```
     Shared variables:  int barrier := N + 1, counter1, counter2;

            Process 0:  inc counter1;

 Process j (j ∈ 1..N):  inc counter2;
                        wait(barrier);

        Process N + 1:  wait(barrier);
                        inc counter1;
```

Fig. 12. Pseudocode of the wakeup_stress(N) example.

Table 2. Nodes in Largest Backtrack Set/Wakeup Tree, Memory and Time Usage of the Source- and Optimal-DPOR Algorithms (As Implemented in Concuerror) for the Program in Figure 12

		Source			Optimal		
n	Traces Explored	Nodes	Memory	Time	Nodes	Memory	Time
1	2	2	22MB	0.01s	7	22MB	0.01s
2	4	3	22MB	0.02s	21	22MB	0.02s
3	12	4	22MB	0.05s	79	22MB	0.06s
4	48	5	22MB	0.24s	374	23MB	0.24s
5	240	6	22MB	1.21s	2,159	27MB	1.23s
6	1,440	7	22MB	8.74s	14,698	42MB	8.96s
7	10,080	8	22MB	1m02s	115,091	114MB	1m07s
8	80,640	9	22MB	9m30s	1,018,094	801MB	9m48s
9	725,760	10	22MB	89m14s	9,916,765	8,364MB	99m2s

The third and final program is wakeup_stress(N), whose pseudocode is shown in Figure 12. Here each of N processes increments counter2 atomically, and two more processes, 0 and N + 1, increment counter1, with process N + 1 waiting for all N processes to finish before doing its operation. Under the assumption that the inc operations are dependent (and wait operations are not), this program has $2 * (N!)$ Mazurkiewicz traces, differing in the order in which the N processes increment counter2 and processes 0 and N + 1 increment counter1.

Whenever an attempt is made to reverse a race, any chosen process will eventually execute an appropriate inc operation, removing any other process from the sleep set, so source-DPOR will never encounter sleep set blocking.

On the other hand, all such inc operations are also racing with each other, therefore introducing new branches in a wakeup tree. Particularly for the race between processes 0 and N + 1, each of the N! schedules of the N processes (that must all finish to enable process N + 1) differs in the happens-before order of the inc counter2 operations. As a result, the wakeup tree at the point where the first inc counter1 operation is originally executed will eventually have N! branches.

In Table 2, we show the total memory usage of Concuerror (the stateless model checking tool which, as we describe in the next section, is our implementation vehicle) when exploring the interleavings of wakeup_stress(N). It is evident that the base memory usage of the tool is approximately 22MB, but as the number of interleavings increases the memory used by wakeup trees, which are present only in optimal-DPOR, starts to become noticeable and follows an exponential trend. The timing measurements, however, show that the overhead of maintaining the wakeup trees is negligible, even in this case, and the optimal algorithm has a time performance that is very similar to source-DPOR.

In general, the purpose of wakeup trees is to "guide the first steps" of future explorations with enough information to ensure that the only traces explored are Mazurkiewicz traces which are distinct from both all previous explored traces and from each other. As a result, an exponential blowup in the size of wakeup trees can only be the result of still needing to explore exponentially many Mazurkiewicz traces, which in turn means that the exponential worst case kicks in only when the DPOR algorithm needs to do exponential work anyway.

10 IMPLEMENTATION

In this section, we describe our implementation in the context of Concuerror, a stateless model checking tool for Erlang. Erlang is an industrially relevant programming language based on the actor model of concurrency (Armstrong 2010). In Erlang, actors are realized by language-level processes implemented by the runtime system instead of being directly mapped to operating system threads. Each Erlang process has its own private memory area (stack, heap, and mailbox) and communicates with other processes via message passing. A call to the `spawn` function creates a new process p and returns a *process identifier* (PID) that can be used to send messages to p. Messages are sent *asynchronously* using the `!` operation (or `send` function). Messages get placed in the mailbox of the receiving process in the order they arrive. A process can then consume messages using *selective* pattern matching in `receive` expressions, which are *blocking* operations when a process mailbox does not contain any matching message. Optionally, a `receive` may contain an `after` clause that specifies a *timeout* value (either an integer or the special value `infinity`) and a value to be returned if the timeout time (in ms) is exhausted.

Erlang processes do not share any memory by default. Still, the Erlang implementation comes with a key-value store mechanism, called *Erlang Term Storage (ETS)*, that allows processes to create memory areas where terms shared between processes can be inserted, looked up, and updated. Such areas are the ETS tables that are explicitly declared `public`. The runtime system automatically serializes accesses to these tables when this is necessary and also comes with mechanisms that guarantee atomicity of some operations (e.g., a bulk `insert`). Each ETS table is owned by the process that created it and its memory is reclaimed by the runtime system when this process exits if no other process has inherited this table.

Erlang has all the ingredients needed for concurrency via message passing and most of the ingredients (e.g., reads and writes to shared data, etc.) needed for concurrent programming using shared memory. Unsurprisingly, Erlang programs are prone to "the usual" errors associated with concurrent execution, although the majority of them revolves around message passing and misuse of built-in primitives implemented in C.

Figure 13 shows the program of Figure 1 written in Erlang, generalized to `N` instead of just two readers. On line 5, a public ETS table named `tab` is created and is shared between `N+1` processes: `N` readers and one writer. The writer inserts a key-value pair, using `k` as a key. Each of the `N` readers spawned on line 9 tries to read two entries from this table: some entry with a different key in each process (an integer in the range `1..N`) and the entry keyed by `k`. The `receive` expression on line 10 forces the process executing the `readers` code to get stuck at this point, ensuring that the process owning the table stays alive, which in turn preserves the ETS table.

As mentioned, our implementation vehicle is Concuerror (Christakis et al. 2013), a stateless model checking tool for finding concurrency errors in Erlang programs or verifying their absence. The tool is publicly available at http://parapluu.github.io/Concuerror/.

Given a program and a test to run, Concuerror uses a stateless search algorithm to systematically explore the execution of the test under conceptually all process interleavings. To achieve this,

```
-module(readers).
-export([readers/1]).

readers(N) ->
  ets:new(tab, [public, named_table]),
  Writer = fun() -> ets:insert(tab, {k, 42}) end,
  Reader = fun(I) -> ets:lookup(tab, I), ets:lookup(tab, k) end,
  spawn(Writer),
  [spawn(fun() -> Reader(I) end) || I <- lists:seq(1, N)],
  receive after infinity -> deadlock end.
```

Fig. 13. Writer-readers program in Erlang.

the tool employs a source-to-source transformation that inserts instrumentation at *preemption points* (i.e., points where a context switch is allowed to occur) in the code under execution. This instrumentation allows Concuerror to take control of the scheduler when the program is run, without having to modify the Erlang VM in any way. In the current VM, a context switch may occur at any function call. Concuerror inserts preemption points only at process actions that interact with (i.e., inspect or update) shared state. Concuerror supports the complete Erlang language and can instrument programs of any size, including any libraries they use.

For benchmarking purposes, we extended a previous version of Concuerror[1] with three DPOR algorithms: (i) the "classic" DPOR algorithm with the sleep set extension as presented by Flanagan and Godefroid (2005a), (ii) source-DPOR, and (iii) optimal-DPOR. To implement these, we had to encode rules for dependencies between operations that constitute preemption points. These rules are shared between all DPOR variants. For lookups and inserts to ETS tables (i.e., reads and writes to shared data), the rules are standard (two operations conflict if they act on the same key and at least one is an insert). For sending and receiving operations, the "happens before" relation ($\rightarrow_E$) is as follows:

- Two sends are ordered by $\rightarrow_E$ if they send to the same process, even if the messages are the same. (Note that if we would not order two sends that send the same message, then when we reorder them, the corresponding `receive` will not "happen after" the same send statement.)
- A send "happens before" the `receive` statement that receives the message it sent. A race exists between these statements only if the `receive` has an `after` clause.
- A `receive` that executes its `after` clause "happens before" a subsequent send that sends a message that it can consume.

There are also other race-prone primitives in Erlang, but it is beyond the scope of this article to describe how they interact.

Concuerror uses a *vector clock* (Mattern 1989) for each process at each state to calculate the happens-before relation for any two events. The calculation of the vector clocks uses the ideas presented in the original DPOR article (Flanagan and Godefroid 2005b). The only special case is for the association of a send with a `receive`, where we instrument the message itself with the vector clock of the sending process.

[1] The current (2016–2017) version of Concuerror supports the optimal-DPOR algorithm by default and the source-DPOR algorithm as an option. The "classic" algorithm is no longer available as an option.

11 EXPERIMENTS

We report experimental results that compare the performance of the three DPOR algorithms that, for convenience, we will refer to as "Classic" (for the algorithm of Flanagan and Godefroid (2005a)), "Source," and "Optimal." We ran all benchmarks on a desktop with an i7-3770 CPU (3.40GHz) with 16GB of RAM running Debian Linux 3.2.0-4-amd64. The machine has four physical cores, but presently Concuerror uses only one of them. In all benchmarks, Concuerror was started with the option `-p inf` (use "infinity" as preemption bound), which instructs the tool to explore traces without limiting the number of preemptions in each explored trace, that is, to verify these programs.[2]

Performance on two "standard" benchmarks. First, we report performance on the two benchmarks from the DPOR article (Flanagan and Godefroid 2005b): filesystem and indexer. These are benchmarks that have been used in the literature to evaluate another DPOR variant (DPOR-CR of Saarikivi et al. (2012)) and a technique based on unfoldings by Kähkönen et al. (2012). As both programs use locks, we had to emulate a locking mechanism using Erlang. To make this translation, we used particular language features for these two benchmarks.

filesystem. This benchmark uses two lock-handling primitives, called `acquire` and `release`. The assumptions made for these primitives are that an `acquire` and a `release` operation on the same lock are never co-enabled and should therefore not be interleaved in a way that differs from the one in which they occur. Thus, `acquires` are the only operations that can be swapped, if possible, to get a different interleaving.

We implemented the lock objects in Erlang as separate processes. To acquire the lock, a process sends a message with its identifier to the "lock process" and waits for a reply. On receiving the message, the lock process uses the identifier to reply and then waits for a `release` message. Other `acquire` messages are left in the lock's mailbox. On receiving the `release` message, the lock process loops back to the start, retrieving the next `acquire` message and notifying the next process. This behavior can be implemented in Erlang using two selective `receives`.

indexer. This benchmark uses a CAS primitive instruction to check whether a specific entry in a matrix is zero and sets it to a new value. The "Erlang way" to do this is to try to execute an `insert_new` operation on an ETS table: If another entry with the same key exists, then the operation returns `false`; otherwise, the operation returns `true` and the table now contains the new entry. Two `insert_new` operations on the same key are always dependent.

Both benchmarks are parametric on the number of threads they use. For filesystem, we used 14, 16, 18, and 19 threads. For indexer, we used 12 and 15 threads.

Table 3 shows the number of traces that the three algorithms explore as well as the time it takes to explore them. It is clear that our algorithms, which in these benchmarks explore the same (optimal) number of interleavings, beat "classic" DPOR with sleep sets by a margin that becomes wider as the number of threads increases. As a sanity check, Kähkönen et al. (2012) report that their unfolding-based method is also able to explore only 8 paths for indexer(12), while their prototype implementation of DPOR extended with sleep sets and support for commutativity of reads and

[2] The version of Concuerror that we used to obtain the experimental results of this section is available at https://github.com/aronisstav/Concuerror/releases/tag/JACM_submission

Table 3. Performance of DPOR Algorithms on Two Benchmarks

	Traces Explored			Time		
Benchmark	Classic	Source	Optimal	Classic	Source	Optimal
filesystem(14)	4	2	2	0.54s	0.36s	0.35s
filesystem(16)	64	8	8	8.13s	1.82s	1.78s
filesystem(18)	1,024	32	32	2m11s	8.52s	8.86s
filesystem(19)	4,096	64	64	8m33s	18.62s	19.57s
indexer(12)	78	8	8	0.74s	0.11s	0.10s
indexer(15)	341,832	4,096	4,096	56m20s	50.24s	52.35s

Table 4. Performance of DPOR Algorithms on More Benchmarks

	Traces Explored			Time		
Benchmark	Classic	Source	Optimal	Classic	Source	Optimal
readers(2)	5	4	4	0.02s	0.02s	0.02s
readers(8)	3,281	256	256	13.98s	1.31s	1.29s
readers(13)	797,162	8,192	8,192	86m 7s	1m26s	1m26s
lastzero(5)	241	79	64	1.08s	0.38s	0.32s
lastzero(10)	53,198	7,204	3,328	4m47s	45.21s	27.61s
lastzero(15)	9,378,091	3,025,87	147,456	1,539m11s	55m 4s	30m13s

writes explores between 51 and 138 paths (with 85 as median value). The numbers we report (78 for “classic” DPOR and 8 for our algorithms) are very similar.

Performance on two synthetic benchmarks. Next, we compare the algorithms on two synthetic benchmarks that expose differences between them. The first is the readers program in Figure 13. The results, for 2, 8, and 13 readers, are shown in Table 4. For “classic” DPOR, the number of explored traces is $O(3^{\mathsf{N}})$ here, while source- and optimal-DPOR only explore 2^{N} traces. Both numbers are exponential in N but, as can be seen in the table, for, for example, $\mathsf{N} = 13$ both source-DPOR and optimal-DPOR finish in about one and a half minutes, while the DPOR algorithm with the sleep set extension (Flanagan and Godefroid 2005a) explores two orders of magnitude more (mostly sleep-set-blocked) traces and needs almost one and a half hours to complete.

The second benchmark is the lastzero(N) program presented earlier in Figure 11. As can be seen in Table 4, source-DPOR explores about twice as many traces as optimal-DPOR and, naturally, even if it uses a cheaper test, takes almost twice as much time to complete.

Performance on real programs. Finally, we evaluate the algorithms on four Erlang applications. The programs are as follows:

dialyzer. A parallel static code analyzer (Aronis and Sagonas 2012) included in the Erlang/OTP distribution.
gproc. An extended process dictionary (https://github.com/uwiger/gproc).
poolboy. A worker pool factory (https://github.com/devinus/poolboy).
rushhour. A program that uses processes and ETS tables to solve the Rush Hour puzzle in parallel.

The last program, rushhour, is complex but self-contained (917 lines of code). The first three programs, besides their code, call many modules from the Erlang libraries, which Concuerror also

Table 5. Performance of DPOR Algorithms on Four Real Programs

	Traces Explored			Time		
Benchmark	Classic	Source	Optimal	Classic	Source	Optimal
dialyzer	12,436	3,600	3,600	14m46s	5m17s	5m46s
gproc	14,080	8,328	8,104	3m 3s	1m45s	1m57s
poolboy	6,018	3,120	2,680	3m 2s	1m28s	1m20s
rushhour	793,375	536,118	528,984	145m19s	101m55s	105m41s

Table 6. Memory Requirements (in MB) for Selected Benchmarks

	filesystem(19)	indexer(15)	gproc	rushhour
Classic	92.98	245.32	557.31	24.01
Source	66.07	165.23	480.96	24.01
Optimal	76.17	174.60	481.07	31.07

instruments. The total number of lines of instrumented code for testing the first three programs is 44, 596, 9, 446, and 79, 732, respectively.

Table 5 shows the results. Here, the performance differences are not as profound as in synthetic benchmarks. Still, some general conclusions can be drawn:

- Both source-DPOR and optimal-DPOR explore less traces than "classic" (from 50% up to 3.5 times fewer) and require less time to do so (they run from 42% up to 2.65 times faster).
- Even in real programs, the number of sleep-set-blocked explorations is significant.
- Regarding the number of traces explored, source-DPOR is quite close to optimal but manages to completely avoid sleep-set-blocked executions in only one program (in dialyzer).
- *Source-DPOR* is faster overall but only slightly so compared to *optimal-DPOR* even though it uses a cheaper test. In fact, its maximal performance difference percentage-wise from *optimal-DPOR* is a bit less than 10% (in dialyzer again).

Although we do not include a full set of memory consumption measurements, we mention that all algorithms have very similar, and quite low, memory needs. Table 6 shows numbers for gproc, the real program that requires most memory, and for all benchmarks where the difference between source and optimal is more than 1MB. From these numbers, it can also be deduced that the size of the wakeup tree is small. In fact, the average size of the wakeup trees for these programs is less than three nodes (it ranges from 2.14 to 2.38 nodes). In our experience, this holds more generally: In all practical applications we have tried, the space requirements of the optimal algorithm are very moderate and the exponential worst-case memory requirements do not seem to manifest themselves in practice. Further evidence for this provides the fact that, so far at least, we have not received any issue from a user reporting extensive memory consumption when running Concuerror.

12 RELATED WORK

In early approaches to stateless model checking, it was observed that reduction was needed to combat the explosion in number of explored interleavings. Consequently, since then, several reduction methods have been proposed, including partial order reduction and bounding techniques such as bounding the depth (Godefroid 1997), the number of context switches, and the number of preemptive context switches (Musuvathi and Qadeer 2007) or the number of delays that an otherwise deterministic scheduler is allowed (Emmi et al. 2011). Since early persistent set techniques

(Clarke et al. 1999; Godefroid 1996; Valmari 1991) relied on static analysis, sleep set techniques were also used to dynamically prevent explorations from processes that would be provably redundant. However, it was observed that, although sleep sets are sufficient to prevent the complete exploration of different but equivalent interleavings (Godefroid et al. 1995), additional techniques were needed to reduce sleep-set-blocked exploration.

The dynamic partial order reduction algorithm of Flanagan and Godefroid (2005b) showed how to construct persistent sets on-the-fly "by need," leading to better reduction. Similar techniques have been combined with dynamic symbolic execution, which is also known as *concolic testing*, where new test runs are initiated in response to detected races by flipping these races using postponed sets (Sen and Agha 2007). Since then, several variants, improvements, and adaptations of DPOR for stateless model checking (Lauterburg et al. 2010; Tasharofi et al. 2012) and concolic testing (Saarikivi et al. 2012; Sen and Agha 2006) have appeared, all based on persistent sets. Our source-DPOR and optimal-DPOR algorithms can be applied to all these contexts to provide increased or optimal reduction in the number of explored interleavings.

A related area is *reachability testing*, in which test executions of concurrent programs are steered by the test harness. Lei and Carver (2006) present a technique for exploring all Mazurkiewicz traces in a setting with a restricted set of primitives (message passing using FIFO channels and monitors) for process interaction. The scheduling of new test executions explicitly pairs message transmissions with receptions and could potentially require significant memory, compared to the more lightweight approach of stateless model checking. The technique of Lei and Carver guarantees to avoid re-exploration of different but equivalent maximal executions (corresponding to Theorem 7.11) but reports blocked executions. Moreover, their technique requires a non-trivial amount of memory for storing interleavings that are yet to be explored.

Kahlon et al. (2009) present a normal form for executions of concurrent programs and prove that two different normal-form executions are not in the same Mazurkiewicz trace. This normal form can be exploited by SAT- or SMT-based bounded model checkers, but it *cannot* be used by stateless model checkers that enumerate the execution sequences by state-space exploration. Kähkönen et al. (2012) and, very recently, Rodríguez et al. (2015) use unfoldings (McMillan 1995), which can also obtain optimal reduction in number of interleavings. However, unfolding-based techniques have significantly larger cost per test execution than DPOR-like techniques, and the technique of Kähkönen et al. (2012) also needs an additional post-processing step for checking non-local properties such as races and deadlocks. A technique for using transition-based partial order reduction for message-passing programs, without moving to an event-based formulation is to refine the concept of dependency between transitions to that of *conditional dependency* (Godefroid 1996; Godefroid and Pirottin 1993; Katz and Peled 1992).

Another line of work that can be used to test multi-threaded programs that take inputs from unbounded domains are approaches based on *generalized symbolic execution*, starting with the work of Khurshid et al. (2003), which took advantage of the partial order and symmetry reduction capabilities of the model checker component of the Java PathFinder tool. Broadly speaking, most early approaches based on symbolic execution do not achieve the reduction in explored traces that DPOR techniques can offer and do not define any notion of optimality.

Recently, Huang (2015) characterized a generalization (i.e., weaker form) of Mazurkiewicz traces based on a new criterion: the *maximal causal model* for a concurrent computation from a given execution trace, a notion defined by Serbanuta et al. (2012). Maximal causality characterizes the largest possible set of equivalent interleavings in each (weaker form of) Mazurkiewicz trace by taking semantic information into consideration, namely the values of reads and writes. When applied in stateless model checking, maximal causality enables exploration of the entire state space of a concurrent program with respect to a given input with a provably minimal number of executions. The corresponding algorithm, called Maximal Causality Reduction (MCR), relies on an offline constraint analyzer to formulate constraints that are then solved using an SMT solver. Its implementation, which is not publicly available, was compared against an implementation of the original DPOR algorithm with iterative context bounding (Musuvathi and Qadeer 2007) and was found to outperform it in most cases. In view of this result, it would be interesting to compare MCR, both theoretically and experimentally, with a better DPOR algorithm, such as optimal-DPOR, at some point.

13 CONCLUDING REMARKS

We have presented a new theoretical foundation for partial order reduction, based on *source sets*, and two new algorithms for dynamic partial order reduction, called *source-DPOR* and *optimal-DPOR*. The latter algorithm, which combines source sets with a novel mechanism called *wakeup trees* to achieve optimality, is the first DPOR algorithm to be provably optimal in the sense that it is guaranteed both to completely explore the minimal number of executions and avoid even initiating executions that lead to sleep set blocking. As shown in the experimental evaluation, the extra overhead of maintaining wakeup trees is very moderate in practice (never more than 10% in our experiments), which is a good tradeoff for having an optimality guarantee and the possibility to run arbitrarily (exponentially) faster than other DPOR algorithms. However, as also shown in this article, wakeup trees can be exponential in size in situations where an exponential number of executions needs to be explored by the optimal algorithm. In such situations, the source-DPOR algorithm, which maintains less information than the optimal algorithm, could be used as a fallback in a tool that implements both algorithms. Another reason to prefer the source-DPOR algorithm is its implementation simplicity; indeed, one only needs to substitute persistent sets with source sets in an implementation of the original DPOR algorithm of Flanagan and Godefroid (2005b) to obtain an implementation of source-DPOR.

Due to its implementation simplicity, we have chosen the source-DPOR algorithm to be the core algorithm used in Nidhugg, a stateless model checking tool for C/pthreads programs under sequential consistency and also under relaxed memory models, and achieved performance that outperforms state-of-the-art tools of similar capabilities (Abdulla et al. 2015). Recently, we have applied Nidhugg to systematically test various flavors of the Read-Copy-Update (RCU) synchronization mechanism of the Linux kernel. We have been able to reproduce, within seconds, various safety and liveness bugs that have been reported for Tree RCU's implementation and have verified the Grace-Period guarantee, the basic guarantee that RCU offers, on non-preemptible builds of several Linux kernel versions. The source-DPOR algorithm has been instrumental in verifying this property in reasonable time. For more information, refer to Kokologiannakis and Sagonas (2017).

We intend to further explore the ideas behind source sets and wakeup trees, not only for verification but also for new ways of testing concurrent programs.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for comments and suggestions that have improved the presentation and simplified some of the proofs, and the editor, Rajeev Alur, for his swift handling of our article. Thanks also to Magnus Lång for his careful reading of this article before its final submission to the publisher.

REFERENCES

Parosh Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless model checking for TSO and PSO. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'15),* Christel Baier and Cesare Tinelli (Eds.), LCNS, Vol. 9035, Springer, Berlin, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

Joe Armstrong. 2010. Erlang. *Commun. ACM* 539 (Sept. 2010). 68–75. https://doi.org/10.1145/1810891.1810910

Stavros Aronis and Konstantinos Sagonas. 2012. On using Erlang for parallelization—experience from parallelizing Dialyzer. In *Proceedings of the 13th International Symposium on Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.), LCNS, Vol. 7829, Springer, Berlin, 295–310. https://doi.org/10.1007/978-3-642-40447-4_19

Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. 2013. Systematic testing for detecting concurrency errors in Erlang programs. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST'13)*. IEEE Computer Society, Los Alamitos, CA, 154–163. https://doi.org/10.1109/ICST.2013.50

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1983. Automatic verification of finite-state concurrent systems using temporal logics specification: A practical approach. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 117–126. https://doi.org/10.1145/567067.567080

Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. 1999. State space reduction using partial order techniques. *Int. J. Softw. Tools Technol. Transfer* 2, 3 (Nov. 1999), 279–287. https://doi.org/10.1007/s100090050035

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, New York, NY, 20–36. https://doi.org/10.1145/2814270.2814297

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 411–422. https://doi.org/10.1145/1926385.1926432

Cormac Flanagan and Patrice Godefroid. 2005a. Addendum to *Dynamic Partial-order Reduction for Model Checking Software*. Retrieved from http://research.microsoft.com/en-us/um/people/pg/

Cormac Flanagan and Patrice Godefroid. 2005b. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. ACM, New York, NY, 110–121. https://doi.org/10.1145/1040305.1040315

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Ph.D. Dissertation. University of Liège. https://doi.org/10.1007/3-540-60761-7

Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. ACM, New York, NY, 174–186. https://doi.org/10.1145/263699.263717

Patrice Godefroid. 2005. Software model checking: The VeriSoft approach. *Formal Methods Syst. Des.* 26, 2 (March 2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. 1995. State-space caching revisited. *Formal Methods Syst. Des.* 7, 3 (Nov. 1995), 227–241. https://doi.org/10.1007/BF01384077

Patrice Godefroid and Didier Pirottin. 1993. Refining dependencies improves partial-order verification methods. In *Computer Aided Verification*, Costas Courcoubetis (Ed.), LCNS, Vol. 697, Springer, London, UK, 438–449. https://doi.org/10.1007/3-540-56922-7_36

Patrice Godefroid and Pierre Wolper. 1991. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Computer Aided Verification*, Kim G. Larsen and Arne Skou (Eds.), LCNS, Vol. 575, Springer-Verlag, London, UK, 332–342. https://doi.org/10.1007/3-540-55179-4_32

Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, 165–174. https://doi.org/10.1145/2737924.2737975

Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. 2012. Using unfoldings in automated testing of multithreaded programs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, 150–159. https://doi.org/10.1145/2351676.2351698

Vineet Kahlon, Chao Wang, and Aarti Gupta. 2009. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.), LCNS, Vol. 5643, Springer, Berlin, 398–413. https://doi.org/10.1007/978-3-642-02658-4_31

Harmen Kastenberg and Arend Rensink. 2008. Dynamic partial order reduction using probe sets. In *Concurrency Theory*, Franck van Breugel and Marsha Chechnik (Eds.), LNCS, Vol. 5201, Springer, 233–247. https://doi.org/10.1007/978-3-540-85361-9_21

Shmuel Katz and Doron Peled. 1992. Defining conditional independence using collapses. *Theoret. Comput. Sci.* 101, 2 (July 1992), 337–359. https://doi.org/10.1016/0304-3975(92)90054-J

Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Hubert Garavel and John Hatcliff (Eds.), LNCS, Vol. 2619, Springer, Berlin, Heidelberg, 553–568. https://doi.org/10.1007/3-540-36577-X_40

Michalis Kokologiannakis and Konstantinos Sagonas. 2017. Stateless model checking of the Linux Kernel's hierarchical read-copy-update (tree RCU). In *Proceedings of the 24th International SPIN Symposium on Model Checking of Software (SPIN'17)*. ACM, New York, NY, 172–181. DOI:https://doi.org/doi.org/10.1145/3092282.3092287

Leslie Lamport. 1978. Time, clocks and the ordering of events in a distributed system. *Comm. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. 2010. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *Fundamental Approaches to Software Engineering, 13th International Conference (FASE'10)*, David S. Rosenblum and Gabriele Taentzer (Eds.), LNCS, Vol. 6013, Springer, Berlin Heidelberg, 308–322. https://doi.org/10.1007/978-3-642-12029-9_22

Yu Lei and Richard H. Carver. 2006. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.* 32, 6 (June 2006), 382–403. https://doi.org/10.1109/TSE.2006.56

Friedemann Mattern. 1989. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, M. Cosnard et al. (Ed.). North-Holland/Elsevier, 215–226.

Antoni Mazurkiewicz. 1987. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.), LNCS, Vol. 255, Springer, Berlin, 279–324. https://doi.org/10.1007/3-540-17906-2_30

Kenneth L. McMillan. 1995. A technique of a state space search based on unfolding. *Formal Methods Syst. Des.* 6, 1 (Jan. 1995), 45–65. https://doi.org/10.1007/BF01384314

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 446–455. https://doi.org/10.1145/1250734.1250785

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerald Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 267–280. *http://dl.acm.org/citation.cfm?id=1855741.1855760*

Doron Peled. 1993. All from one, one for all, on model-checking using representatives. In *Computer Aided Verification*, Costas Courcoubetis (Ed.), LNCS, Vol. 697, Springer-Verlag, London, UK, 409–423. https://doi.org/10.1007/3-540-56922-7_34

Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *Proceedings of the International Symposium on Programming*, Mariangiola Dezani-Ciancaglini and Ugo Montanari (Eds.), LNCS, Vol. 137, Springer Verlag, Berlin, 337–351. https://doi.org/10.1007/3-540-11494-7_22

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based partial order reduction. In *Proceedings of the 26th International Conference on Concurrency Theory (CONCUR'15)*, LIPIcs, Vol. 42, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 456–469. https://doi.org/10.4230/LIPIcs.CONCUR.2015.456

Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Pearson Education Limited, Essex, UK.

Olli Saarikivi, Kari Kähkönen, and Keijo Heljanko. 2012. Improving dynamic partial order reductions for concolic testing. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design (ACSD'12)*. IEEE, Los Alamitos, CA, 132–141. https://doi.org/10.1109/ACSD.2012.18

Koushik Sen and Gul Agha. 2006. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering, 9th International Conference (FASE'06)*, Luciano Baresi and Reiko Heckel (Eds.), LCNS, Vol. 3922, Springer, Berlin, 339–356. https://doi.org/10.1007/11693017_25

Koushik Sen and Gul Agha. 2007. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, Eyal Bin, Avi Ziv, and Shmuel Ur (Eds.), LNCS, Vol. 4383, Springer, Berlin, 166–182. https://doi.org/10.1007/978-3-540-70889-6_13

Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. 2012. Maximal causal models for sequentially consistent systems. In *Runtime Verification, Third International Conference, RV 2012, Revised Selected Papers.* Shaz Qadeer and Serdar Tasiran (Eds.), LCNS, Vol. 7687, Springer, Berlin, 136–150. https://doi.org/10.1007/978-3-642-35632-2_16

Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems.* Holger Giese and Grigore Rosu (Eds.), LNCS, Vol. 7273, Springer, Berlin Heidelberg, 219–234. https://doi.org/10.1007/978-3-642-30793-5_14

Antti Valmari. 1991. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, Grzegorz Rozenberg (Ed.), LCNS, Vol. 483, Springer-Verlag, London, UK, 491–515. https://doi.org/10.1007/3-540-53863-1_36

Received April 2016; revised March 2017; accepted March 2017