Optimal Dynamic Partial Order Reduction

Parosh Abdulla Stavros Aronis Bengt Jonsson Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden

Abstract

Stateless model checking is a powerful technique for program verification, which 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). 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*, which replace the role of persistent sets in previous algorithms. First, we show how to modify an existing DPOR algorithm to work with source sets, resulting in an efficient and simple to implement algorithm. Second, we extend this algorithm with a novel mechanism, called *wakeup trees*, that allows to achieve optimality. We have implemented both algorithms in a stateless model checking tool for Erlang programs. Experiments show that source sets significantly increase the performance and that wakeup trees incur only a small overhead in both time and space.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Verification, Reliability

Keywords dynamic partial oder reduction; software model checking; systematic testing; concurrency; source sets; wakeup trees

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* 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 to capture and store a large number of global states. *Stateless model checking* [7] 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 [8] and CHESS [20].

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535845

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 and context bounding [19]. Among them, *partial order reduction* (POR) [3, 6, 21, 26] 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* [17]), 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 [3, 6, 26].

Existing POR 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, that explores only a provably sufficient subset of the enabled processes. This set is called a *persistent set* [6] (variations are *stubborn sets* [26] and *ample sets* [3]).
- The *sleep set* technique [6], that 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) [4] 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 [4, 8, 20].

Challenge Since DPOR is excellently suited as a reduction technique, several variants, improvements, and adaptations for different computation models have appeared [4, 15, 22, 24, 25]. The obtained reduction can, however, vary significantly depending on several factors, e.g. the order in which processes are explored at each point of scheduling. For a particular implementation of DPOR (with sleep sets) [11], up to an order of magnitude of difference in the number of explored interleavings has been observed, when different strategies are used. For specific communication models, specialized algorithms can achieve better reduction [25]. Heuristics for choosing which next process to explore have also been investigated without conclusive results [14].

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 [10]. At first sight, this seems to imply that sleep sets can give optimal reduction. What it actually implies, however, is that when the algorithm tries an interleaving which 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 will show in this paper, persistent sets are not powerful enough to completely prevent sleep-set blocked exploration.

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 communication via shared variables and message passing.

Contributions In this paper, we present a new DPOR algorithm, called *optimal-DPOR*, which is provably optimal in that it always explores exactly one interleaving per Mazurkiewicz trace, and never initiates any sleep set-blocked exploration. Our optimal algorithm 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 are often smaller than persistent sets and 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.

We will use a two-step approach to describe our optimal algorithm. In the first step, we develop a simpler DPOR algorithm, called *source-DPOR*, which is based on source sets. It is derived by modifying the classical DPOR algorithm by Flanagan and Godefroid [4] so that persistent sets are replaced by source sets. The power of source sets can already be observed in the algorithm *source-DPOR*: it 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 9.

Although *source-DPOR* often achieves optimal reduction, it may sometimes encounter sleep-set blocked explorations. Therefore, in the second step, we combine source sets with a novel mechanism, called *wakeup trees*, thus 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 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 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 the DPOR algorithms and the space cost of maintaining wakeup trees is very small in our experience.

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 to make finer distinctions, leading to better reduction, than many other approaches that define a happens-before relation which is based on program statements, possibly taking into account the local state in which they are executed [3, 4, 6, 8, 14, 25, 26]. 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.

We have implemented both *source-DPOR* and *optimal-DPOR* as extensions for Concuerror [2], 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 which 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 significant size, we obtain optimal reduction in the number of interleavings even with *source-DPOR*, therefore significantly outperforming the original DPOR algorithm not only in number of interleavings but in total execution time as well.

Organization In the next section, we illustrate the basic new ideas of our technique. We introduce our computational model and formulation of the partial-order framework in Section 3. In Section 4 we introduce source sets. 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. Implementation of the algorithms is described in Section 8, and experimental evaluation in Section 9. The paper 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.

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

Example 1: Writer-readers code excerpt.

Source Sets In Example 1, the three processes p, q, and r perform dependent accesses to the shared variable `x`. Two accesses are dependent if they access the same variable and at least one is a write. 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 sequence of scheduled process steps). A DPOR

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

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

Example 2: Program with control flow.

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 which contains p in the initial state is $\{p, q, r\}$. To see this, suppose that, e.g. r is not in the persistent set P, i.e. $P = \{p, q\}$. Then the 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. 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 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, the number of sleep-set blocked explorations increases significantly (see Table 2 in Section 8).

Sleep sets were introduced [6] to prevent redundant exploration. They are manipulated as follows: (i) after exploring interleavings that begin with some process p, the process p is added to the sleep set, and (ii) when a process step that is dependent with p is executed, p is removed from the sleep set. The effect is that the algorithm need never explore a step of a process in the sleep set. In Example 1, for instance, after having explored executions starting with p, the process p is added to the sleep set. When exploring executions that start with q, the process p is still in the sleep set after the first step of q, and should not be explored next, since executions that start with $q.p$ are equivalent to executions that start with $p.q$.

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

We illustrate this by Example 2, a program with four processes, p, q, r, s. Two events are dependent if they access the same shared variable, i.e. `x,y` or `z`. Variables `m,n,l` are local. Each statement accessing a global variable has a unique label; e.g., 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 on `l`, and the condition check on `n`. If the value of `n` is 1, the condition check on `l` is also part of `(6)`, which ends just before the assignment on `x` that has the label `(7)`. Similar assumptions are made for the other local statements.

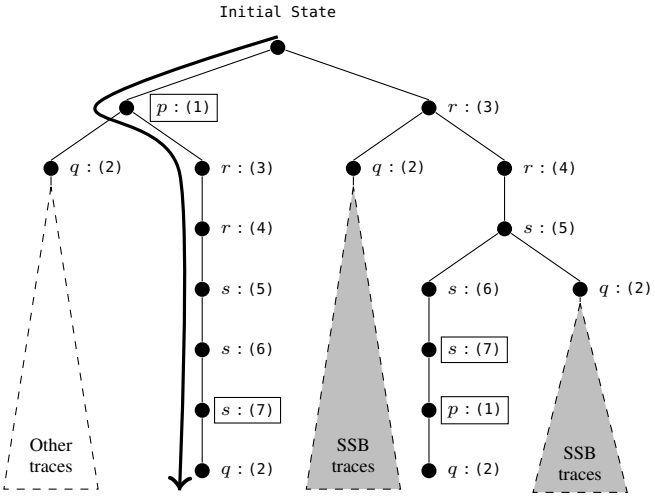

Figure 1. Explored interleavings for Example 2.

Consider a DPOR algorithm that starts the exploration with p, explores the interleaving $p.r.r.s.s.s$ (marked in Figure 1 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, i.e., 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 1 between the two chunks labeled "SSB 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 1.

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, 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.

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 (global) *state* of the system, which is made up of the local states of each process and the shared state of the system. We assume that the state space does not contain cycles, and that executions have bounded length. We do not restrict to a specific mode of process interaction, allowing instead the use of shared variables, messages, etc.

Let Σ be the set of (global) 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 $\textit{execute}_p : \Sigma \mapsto \Sigma$ which moves the system from one state to a subsequent state. Each such application of the function $\textit{execute}_p$ represents an atomic *execution step* of process p, which may depend on and affect the global 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 such as VeriSoft [7].

The execution of a process is said to *block* in some state s if the process cannot continue (i.e. $\textit{execute}_p(s)$ is undefined): for example, trying to receive a message in a state where the message queue is empty. To simplify the presentation, we assume in Sections 3–7 that a process does not disable another process, i.e. if p is enabled and another process q performs a step, then p is still enabled. This assumption is valid for Erlang programs. Note that a process can disable itself, e.g. after a step such that the next statement is a `receive` statement.

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 (global) state of the system after E, which is denoted $s_{[E]}$. For a state s, let $enabled(s)$ denote the set of processes p that are enabled in s (i.e., for which $\textit{execute}_p(s)$ is defined). We use . to denote concatenation of sequences of processes. Thus, if p is not blocked after E, then $E.p$ is an execution sequence.

An *event* of E is a particular occurrence of a process in E. We use $\langle p, i\rangle$ to denote the ith event of process p in the execution sequence E. In other words, the event $\langle p, i\rangle$ is the ith execution step of process p in the execution sequence E. We use $\textit{dom}(E)$ to denote the set of events $\langle p, i\rangle$ which are in E, i.e. $\langle p, i\rangle \in \textit{dom}(E)$ iff E contains at least i steps of p. We will use $e, e', \ldots$ to range over events. We use $\textit{proc}(e)$ to denote the process p of an event $e = \langle p, i\rangle$. If $E.w$ is an execution sequence, obtained by concatenating E and w, then $\textit{dom}_{[E]}(w)$ denotes $\textit{dom}(E.w) \setminus \textit{dom}(E)$, i.e. the events in $E.w$ which are in w. As a special case, we use $\textit{next}_{[E]}(p)$ to denote $\textit{dom}_{[E]}(p)$.

We use $<_E$ to denote the total order between events in E, i.e. $e <_E e'$ denotes that e occurs before e' in E. We use $E' \leq E$ to denote that the sequence E' is a prefix of the sequence E.

3.2 Event Dependencies

A central concept in DPOR algorithms is that of a happens-before relation between events in an execution sequence (also called a *causal relation* [24]). 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 $\textit{dom}(E)$, $e \rightarrow_E e'$ means that e "happens before", or "causally precedes" e'. For instance, e can be the transmission of a message that is received by e', or e can be a write operation to a shared variable that is accessed by e'.

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 3.1. 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 a reception of a different one.

In practice, the happens-before assignment function is implemented as expected by relating accesses to the same variables, transmissions and receptions of the same messages, etc., typically using *vector clocks* [16]. In Section 8 we describe such an assignment suitable for Erlang programs.

DEFINITION 3.1. 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 a partial order on $\textit{dom}(E)$, which is included in $<_E$.
2. The execution steps of each process are totally ordered, i.e. $\langle p, i\rangle \rightarrow_E \langle p, i{+}1\rangle$ whenever $\langle p, i{+}1\rangle \in \textit{dom}(E)$,
3. If E' is a prefix of E, then $\rightarrow_E$ and $\rightarrow_{E'}$ are the same on $\textit{dom}(E')$.
4. Any linearization E' of $\rightarrow_E$ on $\textit{dom}(E)$ is an execution sequence which 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 with the same "happens-before" relation. We use $E \simeq E'$ to denote 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 $E \simeq E'$ if and only if $E.w \simeq E'.w$.
7. If p, q, and r are different processes, then if $\textit{next}_{[E]}(p) \rightarrow_{E.p.r} \textit{next}_{[E.p]}(r)$ and $\textit{next}_{[E]}(p) \not\rightarrow_{E.p.q} \textit{next}_{[E.p]}(q)$, then $\textit{next}_{[E]}(p) \rightarrow_{E.p.q.r} \textit{next}_{[E.p.q]}(r)$. □

The first six properties should be obvious for any reasonable happens-before relation. The only non-obvious one would be the last. Intuitively, if the next step of p happens before the next step of r after the sequence E, 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 in any reasonable computation model that we could think of. As examples, one situation is when p and q read a shared variable that is written by r. Another situation is that p sends a message that is received by r. If an intervening process q is independent with p, it cannot affect this message, and so r still receives the same message.

Properties 4 and 5 together imply, as a special case, that if e and e' are two consecutive events in E with $e \not\rightarrow_E e'$, then they can be swapped and the (global) 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 $E \models p \diamondsuit w$ denotes that $E.p.w$ is an execution sequence such that $\textit{next}_{[E]}(p) \not\rightarrow_{E.p.w} e$ for any $e \in \textit{dom}_{[E.p]}(w)$. In other words, $E \models p \diamondsuit w$ states that the next event of p would not "happen before" any event in w

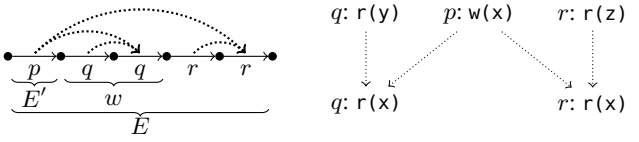

Figure 2. A sample run of the program in Example 1 is shown to the left. This run is annotated by a happens-before relation (the dotted arrows). To the right, the happens-before relation is shown as a partial order. Notice that $E' \models q\diamondsuit r$ since q and r are not happens-before related in $E'.r.q$. We also observe that $I_{[E']}(w) = \{q\}$, as q is the only process occuring 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.

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 $E \models p\diamondsuit q$ denotes that the next steps of p and q are independent after E. We use $E \not\models p\diamondsuit w$ to denote that $E \models p\diamondsuit w$ does not hold.

For a sequence w and $p \in w$, let $w \setminus p$ denote the sequence w with its first occurrence of p removed, and let $w \restriction p$ denote the prefix of w up to but not including the first occurrence of p. For an execution sequence E and an event $e \in \mathit{dom}(E)$, let $pre(E, e)$ denote the prefix of E up to, but not including, the event e. For an execution sequence E and an event $e \in E$, let $\mathit{notdep}(e, E)$ be 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 $e \not\to_E e'$).

A central concept in most DPOR algorithms is that of a race. Intuitively, two events, e and e' in an execution sequence E, where e occurs before e' in E, are in a race if

- e happens-before e' in E, and
- e and e' are "concurrent", i.e. there is an equivalent execution sequence $E' \simeq E$ in which e and e' are adjacent.

Formally, let $e \lessdot_E e'$ denote that $\mathit{proc}(e) \neq \mathit{proc}(e')$, that $e \to_E e'$, and that there is no event $e'' \in \mathit{dom}(E)$, different from e' and e, such that $e \to_E e'' \to_E e'$.

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, this may lead to a different global state: therefore the algorithm must try to explore a corresponding execution sequence. Let $e \precsim_E e'$ denote that $e \lessdot_E e'$, and that the race can be reversed. Formally, if $E' \simeq E$ and e occurs immediately before e' in E', then $\mathit{proc}(e')$ was not blocked before the occurrence of e.

In Figure 2, 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 contain the processes that can perform "first steps" in the possible future execution sequences. Let us first define two related notions of possible "first steps" in a sequence.

- For an execution sequence $E.w$, let $I_{[E]}(w)$ denote the set of processes that perform events e in $\mathit{dom}_{[E]}(w)$ that have no "happens-before" predecessors in $\mathit{dom}_{[E]}(w)$. More formally, $p \in I_{[E]}(w)$ if $p \in w$ and there is no other event $e \in \mathit{dom}_{[E]}(w)$ with $e \to_{E.w} \mathit{next}_{[E]}(p)$.

Algorithm 1: Source-DPOR algorithm

```
1  Initially Explore(⟨⟩, ∅);
2  Explore(E, Sleep) ;
3    if ∃p ∈ (enabled(s_[E]) \ Sleep) then
4        backtrack(E) := {p} ;
5        while ∃p ∈ (backtrack(E) \ Sleep) do
6            foreach e ∈ dom(E) such that(e ≾_{E.p} next_[E](p)) do
7                let E′ = pre(E, e);
8                let v = notdep(e, E).p ;
9                if I_[E′](v) ∩ backtrack(E′) = ∅ then
10                   add some q′ ∈ I_[E′](v) to backtrack(E′);
11           let Sleep′ := {q ∈ Sleep | E⊨p◇q} ;
12           Explore(E.p, Sleep′);
13           add p to Sleep ;
```

- For an execution sequence $E.w$, define $WI_{[E]}(w)$ as the union of $I_{[E]}(w)$ and the set of processes p such that $p \in enabled(s_{[E]})$ and $E \models p\diamondsuit w$.

The point of these concepts is that for an execution sequence $E.w$:

- $p \in I_{[E]}(w)$ if and only if there is a sequence w' such that $E.w \simeq E.p.w'$, and
- $p \in WI_{[E]}(w)$ if and only if there are sequences w' and v such that $E.w.v \simeq E.p.w'$.

DEFINITION 4.1 (Source Sets). Let E be an execution sequence, and let W be a set of sequences, such that $E.w$ is an execution sequence 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.

5. Source-DPOR

In this section, we present the *source-DPOR* algorithm. It is shown in Algorithm 1. As mentioned in the introduction, it is the first step towards our optimal algorithm, and is derived from the classical DPOR algorithm [4] by replacing persistent sets with source sets. *Source-DPOR* performs a depth-first search, using the recursive procedure $Explore(E, \mathit{Sleep})$, where E is the stack, i.e. the past execution sequence explored so far, and *Sleep* is a sleep set, i.e. a set of processes that need not be explored from $s_{[E]}$. The algorithm maintains, for each prefix E' of E, a set $backtrack(E')$ of processes that will eventually be explored from E'.

$Explore(E, \mathit{Sleep})$ initializes $backtrack(E)$ to consist of an arbitrary enabled process which is not in *Sleep*. Thereafter, for each process p in $backtrack(E)$ which is not in *Sleep*, the algorithm performs two phases: race detection (lines 6–10) and state exploration (lines 11–13).

In the race detection phase, the algorithm first finds the events e in E that are in a race with the next step of p, and where the race can be reversed (line 6). For each such event $e \in \mathit{dom}(E)$, the algorithm must explore an execution sequence in which the race is reversed. Using the notation of the algorithm, such an execution sequence is equivalent to a sequence of form $E'.v.\mathit{proc}(e).z$, where v is obtained by appending p after the sequence $\mathit{notdep}(e, E)$ of events

that occur after e in $E.p$, but do not "happen after" e, and z is any continuation of the execution. Note that we insert all of $notdep(e, E)$ before $next_{[E]}(p)$, since $notdep(e, E)$ includes all events that follow e in E that "happen before" $next_{[E]}(p)$, which must be performed before $next_{[E]}(p)$ when the race is reversed. The events in E that "happen after" e, should still occur after e, in the sequence z. A sequence equivalent to $E'.v.proc(e).z$ can be performed by taking a step of a process in $I_{[E']}(v)$ immediately after E'. The algorithm therefore (at line 9) checks 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 $E'.v.proc(e).z$ has been or will be explored by the algorithm.

In the exploration phase, exploration is started recursively from $E.p$, using an appropriately initialized sleep set. Sleep sets are manipulated as follows: (i) after finishing exploration of $E.p$, the process p is added to sleep set at E, and (ii) when the exploration of $E.p$ is started recursively from E, the sleep set $Sleep'$ of $E.p$ is initialized to be the set of processes currently in the sleep set of E that are independent with p after E (i.e., $Sleep' = \{q \in Sleep \mid E \models q \diamondsuit p\}$). The effect is that the algorithm need never explore $E.p$ for any process $p \in Sleep$, since that would always lead to a sequence which is equivalent to one that has been explored from a prefix of E. For processes p that were added according to Case (i) above, this is obvious. To see why a process q in the initial sleep set $Sleep'$ of $E.p$ need not be explored, note that any execution sequence of form $E.p.q.v$ is equivalent to the execution sequence $E.q.p.v$ (by $E \models q \diamondsuit p$). Since the algorithm guarantees to have explored some sequence in $[E.q.p.v]_\simeq$ whenever q is in the sleep set of E, it need not explore $E.p.q$.

Correctness Algorithm 1 is correct in the sense that for all maximal execution sequences E, the algorithm explores some execution sequence in $[E]_\simeq$. For lack of space, the proof is omitted. The main part of the proof establishes that when $Explore(E, Sleep)$ returns, the set $Sleep$ 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.

On Source Sets and Persistent Sets The mechanism by which source-DPOR produces source sets rather than persistent sets is the test at line 9. In persistent set-based DPOR algorithms [4, 14, 22, 23, 25], this test must be stronger, and at least guarantee that $backtrack(E')$ contains a process q such that q performs some event in v which "happens-before" $next_{[E]}(p)$ in $E.p$. Such a test guarantees that the first event in v which 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 9 does not require the added process to perform an event which "happens-before" $next_{[E]}(p)$ in $E'.v$. 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 $proc(e)$ 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'.

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 7–11 of Algorithm 1. At these lines, it is found that some execution sequence starting with $E'.v$ should be performed in order to reverse the detected race. However, at lines 10–11, only a single process from the sequence v is entered into $backtrack(E')$, thus "forgetting" information about how to reverse the race. Since the new exploration after $E'.q$ does not "remember" this sequence v, it may explore a completely different sequence, which could 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.

To define wakeup trees, we first generalize the relations $p \in I_{[E]}(w)$ and $p \in WI_{[E]}(w)$ to the case when p is a sequence. 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 $E.w.w'$.

As special cases, 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 2, 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 following recursive definition.

LEMMA 6.1. The relation $v \sim_{[E]} w$ holds if either $v = \langle\rangle$, or v is of form $p.v'$, and either

- $p \in I_{[E]}(w)$ and $v' \sim_{[E.p]} (w \setminus p)$, or
- $E \models p \diamondsuit w$ and $v' \sim_{[E.p]} w$. □

The following lemma states some useful properties.

LEMMA 6.2. 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 $v \sqsubseteq_{[E]} w'$, and (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. $p \in WI_{[E]}(w)$ and $w' \sqsubseteq_{[E]} w$ imply $p \in WI_{[E]}(w')$;
4. $p \in WI_{[E]}(w)$ and $E \models p \diamondsuit q$ and $E \models q \diamondsuit w$ imply $p \in WI_{[E]}(q.w)$. □

The above properties follow from the definitions.

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 the ordering $\prec$. 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.

DEFINITION 6.3 (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) $q \notin I_{[E]}(w)$ for any sequence w that is explored after E. If, in addition, $E \models q \diamondsuit w$, then q is still in the sleep set at $E.w$. To prevent this, we therefore require $q \notin WI_{[E]}(w)$, which is the same as Property 1, i.e., $WI_{[E]}(w) \cap P = \emptyset$. Property 2 implies that if 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 6.4. If $\langle B, \prec\rangle$ is a wakeup tree after $\langle E, P\rangle$ and $w, w' \in B$ and w is a leaf which 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 which is consistent with w' after E.

Proof: We prove the lemma by contradiction. Assume $w' \prec w$. Then 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 $p \notin WI_{[E.u]}(v)$ by Property 2 of Definition 6.3. Hence $p \not\sim_{[E.u]} v$, which implies $u.p \not\sim_{[E]} u.v$, which implies $u.p.v' \not\sim_{[E]} u.v$, i.e. $w' \not\sim_{[E]} w$. □

For a wakeup tree $\langle B, \prec\rangle$ and a process $p \in B$, define *subtree*$(\langle B, \prec\rangle, p)$ to denote the subtree of $\langle B, \prec\rangle$ rooted at p, i.e. *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'.

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 $WI_{[E]}(w) \cap P = \emptyset$, we define the operation $insert_{[E]}(w, \langle B, \prec\rangle)$, with the properties:

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

A simple construction of $insert_{[E]}(w, \langle B, \prec\rangle)$ is the following: Let v be the smallest (w.r.t. to $\prec$) sequence v' in B such that $v' \sim_{[E]} w$. If v is a leaf, $insert_{[E]}(w, \langle B, \prec\rangle)$ can be taken as $\langle B, \prec\rangle$ and we are done. 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 Example 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 $insert_{[\langle\rangle]}(q.q, \langle B, \prec\rangle)$ adds $q.q$ as a new leaf with $p \prec q.q$. If we thereafter perform $insert_{[\langle\rangle]}(r.r, \langle B, \prec\rangle)$, 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 the optimal algorithm, shown in Algorithm 2. The algorithm performs a depth-first search, using the recursive procedure $Explore(E, \mathit{Sleep}, \mathit{WuT})$, 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 $Explore(E, \mathit{Sleep}, \mathit{WuT})$. If *WuT* is empty, then $Explore(E, \mathit{Sleep}, \mathit{WuT})$ is free to explore any extension of E.

Like Algorithm 1, the algorithm runs in two modes: race detection (lines 3–8) and state exploration (lines 9–22), 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 7 is precise only when the used sequence v, which is defined at line 6, includes all events in the entire execution that do not "happen after" e, also those that occur after e'. Therefore v can be defined only when E is a maximal execution sequence.

Algorithm 2: Optimal-DPOR algorithm.

```
1  Initially Explore(⟨⟩, ∅, ⟨{⟨⟩}, ∅⟩);
2  Explore(E, Sleep, WuT) ;
3    if enabled(s_[E]) = ∅ then
4      foreach e, e' ∈ dom(E) such that (e ≾_E e') do
5        let E' = pre(E, e);
6        let v = (notdep(e, E).proc(e')) ;
7        if sleep(E') ∩ WI_[E'](v) = ∅ then
8          insert_[E'](v, wut(E'))
9    else
10     if WuT ≠ ⟨{⟨⟩}, ∅⟩ then
11       wut(E) := WuT ;
12     else
13       choose p ∈ enabled(s_[E]);
14       wut(E) := ⟨{p}, ∅⟩ ;
15     sleep(E) := Sleep ;
16     while ∃p ∈ wut(E) do
17       let p = min_≺{p ∈ wut(E)};
18       let Sleep' = {q ∈ sleep(E) | E ⊨ p◇q} ;
19       let WuT' = subtree(wut(E), p) ;
20       Explore(E.p, Sleep', WuT');
21       add p to sleep(E) ;
22       remove all sequences of form p.w from wut(E) ;
```

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 $sleep(E')$, which is the current sleep set at E'. In a similar way, for each prefix E' of E, the algorithm maintains $wut(E')$, which is the current wakeup tree at E'.

Let us now explain the race detection mode, which is entered whenever the exploration reaches the end of a complete sequence (i.e. $enabled(s_{[E]}) = \emptyset$). In this mode, the algorithm investigates all races that can be reversed in the just explored sequence E. Such a race consists of two events e and e' in E, such that $e \precsim_E e'$. Let $E' = pre(E, e)$ and let $v = notdep(e, E).proc(e')$, i.e. the subsequence of E consisting of the events that occur after e but do not "happen after" e, followed by $proc(e')$ (this notation is introduced at lines 5–6). 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, i.e. 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, 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 mode, which is entered if exploration has not reached the end of an execution sequence, first the wakeup tree $wut(E)$ is initialized to *WuT*. If *WuT* is empty, then (as we will state in Lemma 7.2) 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 $Explore(E.p, \textit{Sleep}', \textit{WuT}')$ has returned, the sleep set *sleep*(E) is extended with p, and all sequences beginning with p are removed from *wut*(E).

7.1 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 $Explore(E, \textit{Sleep}, \textit{WuT})$ 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 of form $Explore(E, \cdot, \cdot)$. 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.

We begin by establishing some useful invariants.

LEMMA 7.1. If $E.p \propto E.w$ then $p \notin I_{[E]}(w)$.

Proof: After the call to $Explore(E.p, \cdot, \cdot)$ has returned, we have that $p \in \textit{sleep}(E)$. It follows from the rules for manipulating sleep sets, that if $E.w \in \mathcal{E}$ and $E.p \propto E.w$ then $p \notin I_{[E]}(w)$. □

LEMMA 7.2. Whenever Algorithm 2 is inside a call of form $Explore(E, \textit{Sleep}, \textit{WuT})$, then

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

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 $w \in \textit{wut}(E)$ for some w, then $E.w \in \mathcal{E}$.
2. If $w \prec w'$ for $w, w' \in \textit{wut}(E)$ 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 $16-22$. □

We can now give the proof of correctness for the algorithm:

THEOREM 7.4. Whenever a call to $Explore(E, \textit{Sleep}, \textit{WuT})$ returns during Algorithm 2, then for all maximal execution sequences of form $E.w$, the algorithm has explored some execution sequence E' which is in $[E.w]_{\simeq}$.

Since initially the algorithm is called with $Explore(\langle\rangle, \emptyset, \langle\{\langle\rangle\}, \emptyset\rangle)$, Theorem 7.4 implies that for all maximal execution sequences of form E the algorithm explores some execution sequence E' which is in $[E]_{\simeq}$.

Proof of Theorem 7.4: 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 is 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: Consider an arbitrary execution sequence $E \in \mathcal{E}$ that is explored by the algorithm. Let *Final_sleep*(E) denote the value of *sleep*(E) when $Explore(E, \textit{Sleep}, \textit{WuT})$ returns. Let $done(E)$ denote *Final_sleep*$(E) \setminus$ *Sleep*, i.e. the set of processes that are explored from $s_{[E]}$. As inductive hypothesis, we assume that the theorem holds for all execution sequences E' with $E' \propto E$.

The inductive step is proven by contradiction. Assume that $E.w$ is a maximal execution sequence such that the algorithm has not explored any execution sequence E' in $[E.w]_{\simeq}$. We first prove that this implies *WI*$_{[E]}(w) \cap$ *Final_sleep*$(E) = \emptyset$. The proof is by contradiction. Assume that there is a $p \in$ *WI*$_{[E]}(w)$ with $p \in$ *Final_sleep*(E), i.e. $p \in$ *sleep*(E) at some point during the exploration. Let E' be the longest prefix of E such that $E'.p \in \mathcal{E}$, and define w' by $E'.w' = E$. By the handling of sleep sets, we have $E' \models p \diamondsuit w'$. It follows by the definition of *WI*$_{[E]}(w)$ that there is a w'' such that $E.w \simeq E.p.w'' \simeq E'.w'.p.w'' \simeq E'.p.w'.w''$. By the inductive hypothesis applied to $E'.p$, the algorithm has explored some execution sequence in $[E'.p.w'.w'']_{\simeq} = [E.w]_{\simeq}$, which gives a contradiction.

For each process $p \in$ *Final_sleep*(E), let E'_p be the prefix of E such that $E'_p.p$ is the last (w.r.t. $\propto$) execution sequence of form $E''_p.p$, with E''_p being a prefix of E, that precedes E (w.r.t. $\propto$). (Note that if $p \in done(E)$ then $E'_p = E$, but if $p \in$ *Sleep* then E'_p is a strict prefix of E.) Let w'_p be defined by $E = E'_p.w'_p$ (note that $E'_p \models p \diamondsuit w'_p$). Let w_p be the longest prefix of w such that $E \models p \diamondsuit w_p$, and let e_p be the first event in *dom*$_{[E]}(w)$ which is not in w_p. (Such an event e_p must exist, since otherwise $w_p = w$ which implies $E \models p \diamondsuit w$, which implies $p \in$ *WI*$_{[E]}(w)$, which contradicts *WI*$_{[E]}(w) \cap$ *Final_sleep*$(E) = \emptyset$.) Let $q \in$ *Final_sleep*(E) be such that w_q is a longest prefix among the prefixes w_p for $p \in$ *Final_sleep*(E). If there are several processes $p \in$ *Final_sleep*(E) such that w_p is the same longest prefix, then let q be the process among these such that $E'_q.q$ is minimal (w.r.t. $\propto$). Let w_R be $w_q.\textit{proc}(e_q)$.

Consider the exploration of $E'_q.q$, which happens in the call $Explore(E'_q.q, \textit{Sleep}', \cdot)$ with *Sleep′* as the sleep set argument. Since $E'_q.q$ is an execution sequence, and $E'_q \models q \diamondsuit (w'_q.w_q)$, it follows that $E'_q.w'_q.w_q.q$ is an execution sequence. Since $E'_q.w'_q.w_q.e_q$ is an execution sequence, and q does not disable e_q (since q and *proc*(e_q) are different), we conclude that $E'_q.w'_q.w_q.q.e_q$ and hence $E'_q.q.w'_q.w_q.e_q = E'_q.q.w'_q.w_R$ is an execution sequence.

We next establish that *WI*$_{[E'_q.q]}(w'_q.w_R) \cap$ *Sleep′* $= \emptyset$, using a proof by contradiction, as follows. Assume that some process p is in *WI*$_{[E'_q.q]}(w'_q.w_R) \cap$ *Sleep′*. By the construction of *Sleep′* at line 18, the process p must be in *sleep*(E'_q) just before the call to $Explore(E'_q.q, \textit{Sleep}', \cdot)$, and satisfy $E'_q \models p \diamondsuit q$. From $p \in$ *WI*$_{[E'_q.q]}(w'_q.w_R)$ and Property 3 of Lemma 6.2, we infer $p \in$ *WI*$_{[E'_q.q]}(w'_q.w_q)$. From this, $E'_q \models p \diamondsuit q$ and Property 4 of Lemma 6.2, it follows that $p \in$ *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 $E'_q \models q \diamondsuit (w'_q.w_q)$), implies $p \in$ *WI*$_{[E'_q]}(w'_q.w_q.q)$, which by Property 3 of Lemma 6.2 imply $p \in$ *WI*$_{[E'_q]}(w'_q.w_q)$. From this, and the property that $p \in$ *sleep*(E'_q) just before the call to $Explore(E'_q.q, \textit{Sleep}', \cdot)$, we have by the handling of sleep sets that $p \notin I_{[E'_q]}(w'_q.w_q)$, which together with $p \in$ *WI*$_{[E'_q]}(w'_q.w_q)$ implies $E'_q \models p \diamondsuit w'_q.w_q$. This, and the fact that $p \in$ *sleep*(E'_q) just before the call to $Explore(E'_q.q, \textit{Sleep}', \cdot)$, implies that $p \in$ *Final_sleep*$(E'_q.w'_q)$, i.e. $p \in$ *Final_sleep*(E). Since $E'_q \models p \diamondsuit w'_q.w_q$, we have by construction that $e_p = e_q$. But since among the processes p with $e_p = e_q$ we chose q to be the first one for which a call of form $Explore(E'_q.p, \cdot, \cdot)$ was performed, we have that $p \notin$ *sleep*(E'_q) just before the call to $Explore(E'_q.q, \textit{Sleep}', \cdot)$, whence $p \notin$ *Sleep′*. Thus we have a contradiction.

Let z' be any sequence that makes $E'_q.q.w'_q.w_R.z'$ maximal (such a z' can always be found, since $E'_q.q.w'_q.w_R$ is an execution

sequence). From $WI_{[E'_q.q]}(w'_q.w_R) \cap \mathit{Sleep}' = \emptyset$ (proven in the preceding paragraph) follows $WI_{[E'_q.q]}(w'_q.w_R.z') \cap \mathit{Sleep}' = \emptyset$. Hence, no execution sequence in $[E'_q.q.w'_q.w_R.z']_\simeq$ was explored before the call to $Explore(E'_q.q, \mathit{Sleep}', \cdot)$ (otherwise, there would be a call $Explore(E''.p, \cdot, \cdot)$ with E'' a prefix of E'_q and $p \in \mathit{Sleep}'$, and defining w'' by $E''.w'' = E'_q$, we would have $E'' \models p \Diamond w''$ and $p \in WI_{[E'_q.q]}(w'_q.w_R.z')$, thus contradicting $WI_{[E'_q.q]}(w'_q.w_R.z') \cap \mathit{Sleep}' = \emptyset$). By the inductive hypothesis for $E'_q.q$ applied to $w'_q.w_R.z'$, the algorithm then explores some sequence of form $E'_q.q.z$ 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 $E'_q.q.z$, i.e. 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.

Let $v = (\mathit{notdep}(\mathit{next}_{[E'_q]}(q), E'_q.q.z).\mathit{proc}(e_q))$ be the sequence v at line 6 in the algorithm. Let x be $\mathit{notdep}(\mathit{next}_{[E'_q]}(q), E'_q.q.z)$ and let x' be $\mathit{notdep}(\mathit{next}_{[E'_q]}(q), E'_q.q.w'_q.w_R.z')$. We note that $w'_q.w_q$ is a prefix of x'. From $E'_q.q.z \simeq E'_q.q.w'_q.w_R.z'$ and the definitions of x and x', it follows that $E'_q.x \simeq E'_q.x'$, and hence that $E'_q.x.\mathit{proc}(e_q) \simeq E'_q.x'.\mathit{proc}(e_q)$. Let x'' be obtained from x' by adding $\mathit{proc}(e_q)$ 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 $E'_q.x'' \simeq E'_q.x'.\mathit{proc}(e_q)$. Since $w'_q.w_R$ is a prefix of x'' we have $w'_q.w_R \sqsubseteq_{[E'_q]} x''$, which by $E'_q.x'' \simeq E'_q.x'.\mathit{proc}(e_q)$ implies $w'_q.w_R \sqsubseteq_{[E'_q]} x'.\mathit{proc}(e_q)$, which by $E'_q.x.\mathit{proc}(e_q) \simeq E'_q.x'.\mathit{proc}(e_q)$ implies $w'_q.w_R \sqsubseteq_{[E'_q]} v$. By properties of sleep sets and the construction of w_R, it follows that $\mathit{sleep}(E'_q) \cap WI_{[E'_q]}(w'_q.w_R) = \emptyset$. This implies, by $w'_q.w_R \sqsubseteq_{[E'_q]} v$, that $\mathit{sleep}(E'_q) \cap WI_{[E'_q]}(v) = \emptyset$. Thus, the test at line 7 will succeed, and after performing line 8, 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 $q \notin WI_{[E'_q]}(v)$ we have, by definition of *insert*, that and $E'_q \not\models q \Diamond y$. By Property 1 of Lemma 7.3 we then have $E'_q.y \in \mathcal{E}$.

From $w'_q.w_R \sqsubseteq_{[E'_q]} v$ and $y \sim_{[E'_q]} v$, it follows by Property 2 of Lemma 6.2 that $y \sim_{[E'_q]} w'_q.w_R$. Furthermore, from $E'_q \models q \Diamond w'_q$ (which follows from $E'_q \models q \Diamond w'_q.w_q$) and $E'_q \not\models q \Diamond y$, 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 2 of Lemma 6.2, we infer $y \sim_{[E'_q]} u.p'$. If $u.p' \in \mathit{wut}(E'_q)$ when y is inserted, we infer by Lemma 6.4 and Property 2 of Lemma 7.3 that $E'_q.y \propto E'_q.u.p'$. If $u.p' \notin \mathit{wut}(E'_q)$ when y is inserted, 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 $E'_q.u \models p \Diamond (p'.v'')$. This implies that $p \in \mathit{sleep}(E'_q.w'_q)$, i.e., $p \in \mathit{sleep}(E)$. Hence by the construction of w_R, we have $p \notin WI_{[E'_q.u.p'.v'']}(w_R)$, which together with $E'_q.u \models p \Diamond (p'.v'')$ 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 the theorem is proven. □

7.2 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.* [10] establishes that sleep sets alone are sufficient to prevent exploration of two equivalent *maximal* execution sequences:

THEOREM 7.5. *Optimal-DPOR never explores two maximal execution sequences which are equivalent.*

Proof: Assume that E_1 and E_2 are two equivalent maximal execution sequences, which 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 7.1 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 7.6 (Sleep Set Blocking). During the execution of Algorithm 2, a call to $Explore(E, \mathit{Sleep}, \mathit{WuT})$ is *sleep set blocked* if $enabled(s_{[E]}) \neq \emptyset$ and $enabled(s_{[E]}) \subseteq \mathit{Sleep}$. □

Now let us state and prove the corresponding optimality theorem.

THEOREM 7.7. *During any execution of Algorithm 2, no call to $Explore(E, \mathit{Sleep}, \mathit{WuT})$ is ever sleep set blocked.*

Proof: Consider a call $Explore(E, \mathit{Sleep}, \mathit{WuT})$ during the exploration. Then any sequence in *WuT* is enabled after E. By Lemma 7.2, *WuT* is a wakeup tree after $\langle E, \mathit{Sleep} \rangle$. Thus, if $\mathit{Sleep} \neq \emptyset$, then *WuT* contains a sequence w such that $\mathit{Sleep} \cap WI_{[E]}(w) = \emptyset$. Letting p be the first process in w, this implies $p \notin \mathit{Sleep}$, implying that p is enabled and thus $enabled(s_{[E]}) \not\subseteq \mathit{Sleep}$. □

8. Implementation

In this section we describe our implementation in the context of Concuerror [2], a stateless model checking tool for Erlang.

Erlang Erlang is an industrially relevant programming language based on the actor model of concurrency [1]. In Erlang, actors are realized by language-level processes implemented by the runtime system instead of being directly mapped to OS 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 `!` (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

```
1  -module(readers).
2  -export([readers/1]).
3
4  readers(N) ->
5    ets:new(tab, [public, named_table]),
6    Writer = fun() -> ets:insert(tab, {x, 42}) end,
7    Reader = fun(I) -> ets:lookup(tab, I), ets:lookup(tab, x) end,
8    spawn(Writer),
9    [spawn(fun() -> Reader(I) end) || I <- lists:seq(1, N)],
10   receive after infinity -> deadlock end.
```

Figure 3. Writer-readers program in Erlang.

when a process mailbox does not contain any matching message. Optionally, a `receive` may contain an `after` clause which 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. Each ETS table is owned by the process that created it and its memory is reclaimed by the runtime system when this process exits.

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 3 shows Example 1 written in Erlang, generalized to `N` instead of just two readers. A public ETS table shared between `N+1` processes: `N` readers and one writer. The writer inserts a key-value pair, using `x` as a key. Each of the `N` readers 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 `x`. 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.

Concuerror Concuerror [2] is a systematic testing tool for finding concurrency errors in Erlang programs or verifying their absence. 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 interleaving. To achieve this, 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. The tool employs two techniques to reduce the number of explored traces: (i) an optimization which avoids exploring traces that involve processes blocking on `receive` expressions [2] and (ii) *context-bounding* [19], a technique that restricts the number of explored traces with respect to a user-supplied parameter, which Concuerror calls *preemption bound*. In this respect, Concuerror is similar to the CHESS tool [20].

Our implementation We extended Concuerror with three DPOR algorithms: (i) the algorithm presented by Flanagan and Godefroid with the sleep set extension [5], (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) 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 the following:

- Two `send`s 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 `send`s 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` which executes its `after` clause "happens before" a subsequent `send` which sends a message that it can consume.

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

Concuerror uses a *vector clock* [16] 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 paper [4]. 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.

9. Experiments

We report experimental results that compare the performance of the three DPOR algorithms, which we will refer to as 'classic' (for the algorithm of [5]), 'source' and 'optimal'. We run all benchmarks on a desktop with an i7-3770 CPU (3.40 GHz), 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`, which instructs the tool to use an infinite preemption bound, i.e., verify these programs.

Performance on two "standard" benchmarks First, we report performance on the two benchmarks from the DPOR paper [4]: `filesystem` and `indexer`. These are benchmarks that have been used to evaluate another DPOR variant (DPOR-CR [22]) and a technique based on unfoldings [11]. As both programs use locks, we had to emulate a locking mechanism using Erlang. To make this translation we used particular language features:

`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 tried to be interleaved in a different way than they occur. Thus, `acquire`s 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. Upon 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. Upon 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 `receive`s.

Benchmark	Traces Explored			Time		
	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)	1024	32	32	2m11s	8.52s	8.86s
filesystem(19)	4096	64	64	8m33s	18.62s	19.57s
indexer(12)	78	8	8	0.74s	0.11s	0.10s
indexer(15)	341832	4096	4096	56m20s	50.24s	52.35s

Table 1. Performance of DPOR algorithms on two benchmarks.

indexer: This benchmark uses a CAS primitive instruction to check whether a specific entry in a matrix is 0 and set 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 the operation returns `false`; otherwise the operation returns `true` and the table now contains the new entry.

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 1 shows the number of traces that the 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, Kähllönen *et al.* [11] 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 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 of Figure 3. The results, for 2, 8 and 13 readers are shown in Table 2. For 'classic' DPOR the number of explored traces is $O(3^{\mathtt{N}})$ here, while source- and optimal-DPOR only explore $2^{\mathtt{N}}$ traces. Both numbers are exponential in `N` but, as can be seen in the table, for e.g. $\mathtt{N} = 13$ source- and optimal-DPOR finish in about one and a half minute, while the DPOR algorithm with the sleep set extension [5] explores two orders of magnitude more (mostly sleep-set blocked) traces and needs almost one and a half hours to complete.

Benchmark	Traces Explored			Time		
	classic	source	optimal	classic	source	optimal
readers(2)	5	4	4	0.02s	0.02s	0.02s
readers(8)	3281	256	256	13.98s	1.31s	1.29s
readers(13)	797162	8192	8192	86m 7s	1m26s	1m26s
lastzero(5)	241	79	64	1.08s	0.38s	0.32s
lastzero(10)	53198	7204	3328	4m47s	45.21s	27.61s
lastzero(15)	9378091	302587	147456	1539m11s	55m 4s	30m13s

Table 2. Performance of DPOR algorithms on more benchmarks.

Variables: `int array[0..N] := {0,0,...,0}, i;`

Thread 0**:** `for (i := N; array[i] != 0; i--);`

Thread j **(**$j \in 1..$`N`**):** `array[`j`] := array[`j`-1] + 1;`

Figure 4. The pseudocode of the `lastzero(N)` benchmark.

Benchmark	Traces Explored			Time		
	classic	source	optimal	classic	source	optimal
dialyzer	12436	3600	3600	14m46s	5m17s	5m46s
gproc	14080	8328	8104	3m 3s	1m45s	1m57s
poolboy	6018	3120	2680	3m 2s	1m28s	1m20s
rushhour	793375	536118	528984	145m19s	101m55s	105m41s

Table 3. Performance of DPOR algorithms on four real programs.

	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

Table 4. Memory consumption (in MB) for selected benchmarks.

The second benchmark is the lastzero(`N`) program whose pseudocode is shown in Figure 4. Its `N`+1 threads operate on an array of `N`+1 elements which are all initially zero. In this program, thread 0 searches the array for the zero element with the highest index, while the other `N` threads read one of the array elements and update the next one. The final state of the program is uniquely defined by the values of `i` and `array[1..N]`. What happens here is that thread 0 has control flow that depends on data that is exposed to races and represents a case when *source-DPOR* may encounter sleep-set blocking, that the *optimal-DPOR* algorithm avoids. As can be seen in Table 2, *source-DPOR* explores about twice as many traces than *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: (i) dialyzer: a parallel static code analyzer included in the Erlang distribution; (ii) gproc: an extended process dictionary(iii) poolboy: a worker pool factory[1]; and (iv) 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 instruments. The total number of lines of instrumented code for testing the first three programs is 44596, 9446 and 79732, respectively.

Table 3 shows the results. Here, the performance differences are not as profound as in synthetic benchmarks. Still, some general conclusions can be drawn: (1) Both source- and optimal-DPOR explore less traces than 'classic' (from 50% up to 3.5 times fewer) and require less time to do so (from 42% up to 2.65 times faster). (2) Even in real programs, the number of sleep-set blocked explorations is significant. (3) 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). (4) *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, due to space limitations, 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 4 shows numbers for gproc, the real program which requires most memory, and for all benchmarks where the difference between source and optimal is more than one MB. 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.

[1] `https://github.com/uwiger/gproc` and `https://github.com/devinus/poolboy`

10. Related Work

In early approaches to stateless model checking [8], it was observed that reduction was needed to combat the explosion in number of explored interleavings. Several reduction methods have been proposed including partial order reduction and context bounding [19]. Since early persistent set techniques [3, 6, 26] relied on static analysis, sleep set techniques were used in VeriSoft [7]. It was observed that sleep sets are sufficient to prevent the complete exploration of different but equivalent interleavings [10], but additional techniques were needed to reduce sleep-set blocked exploration.

Dynamic partial order reduction [4] showed how to construct persistent sets on-the-fly "by need", leading to better reduction. Similar techniques have been applied in testing and symbolic execution, e.g., to *concolic testing*, where new test runs are initiated in response to detected races [24]. Several variants, improvements, and adaptations of DPOR for stateless model checking [14, 25] and concolic testing [22, 23] have appeared, all based on persistent sets. Our 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 present a technique for exploring all Mazurkiewicz traces in a setting with a restricted set of primitives (message passing and monitors) for process interaction [15]. 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 software model checking. The technique of Lei and Carver guarantees to avoid re-exploration of different but equivalent maximal executions (corresponding to Theorem 7.5), but reports blocked executions.

Kahlon *et al.* [12] 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 can *not* be used by stateless model checkers that enumerate the execution sequences by state-space exploration. Kähkönen *et al.* [11] use unfoldings [18], which can also obtain optimal reduction in number of interleavings. However, this technique has significantly larger overhead than DPOR-like techniques, and 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* [6, 9, 13].

11. Conclusion

We have presented *optimal-DPOR*, a new DPOR algorithm which is the first to be provably optimal in that it is guaranteed both to explore the minimal number of executions and to avoid sleep set blockings. It is based on a novel class of sets, called *source sets*. Source sets make existing DPOR algorithms significantly more efficient, and can be extended with wakeup trees to achieve optimality. In the derivation of the optimal algorithm, we have first presented a simpler algorithm, *source-DPOR*, which maintains less information than the optimal algorithm. On the other hand, the extra overhead of maintaining wakeup trees is very moderate in practice (never more than 10% in our experiments), which is a good trade-off for having an optimality guarantee and the possibility to run arbitrarily faster. We intend to further explore the ideas behind source sets and wakeup trees, not only for verification but also for new ways of testing programs.

Acknowledgments

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.

References

[1] J. Armstrong. Erlang. *Comm. of the ACM*, 53(9):68–75, 2010.

[2] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in Erlang programs. In *ICST*, 2013.

[3] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *STTT*, 2:279–287, 1999.

[4] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.

[5] C. Flanagan and P. Godefroid. Addendum to *Dynamic partial-order reduction for model checking software*, 2005. Available at `http://research.microsoft.com/en-us/um/people/pg/`.

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

[7] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186. ACM Press, 1997.

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

[9] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *CAV*, volume 697 of *LNCS*, 1993.

[10] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.

[11] K. Kähkönen, O. Saarikivi, and K. Heljanko. Using unfoldings in automated testing of multithreaded programs. In *ASE*, pages 150–159. ACM, 2012.

[12] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, volume 5643 of *LNCS*, pages 398–413. Springer, 2009.

[13] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[14] S. Lauterburg, R. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *FASE*, volume 6013 of *LNCS*, pages 308–322. Springer, 2010.

[15] Y. Lei and R. Carver. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32(6):382–403, 2006.

[16] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, Ch. de Bonas, France, 1989. Elsevier.

[17] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, 1986.

[18] K. McMillan. A technique of a state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[19] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.

[21] D. Peled. All from one, one for all, on model-checking using representatives. In *CAV*, volume 697 of *LNCS*, pages 409–423, 1993.

[22] O. Saarikivi, K. Kähkönen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In *ACSD*. IEEE, 2012.

[23] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FASE*, volume 3922 of *LNCS*, pages 339–356, 2006.

[24] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, volume 4383 of *LNCS*, pages 166–182. Springer, 2007.

[25] S. Tasharofi et al. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *FMOODS/FORTE*, volume 7273 of *LNCS*, pages 219–234. Springer, 2012.

[26] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 491–515, 1990.