Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers

Elvira Albert
Complutense University of Madrid
elvira@fdi.ucm.es

Maria Garcia de la Banda
Faculty of IT, Monash University, Australia
maria.garciadelabanda@monash.edu

Miguel Gómez-Zamalloa
Complutense University of Madrid
mzamalloa@fdi.ucm.es

Miguel Isabel
Complutense University of Madrid
miguelis@ucm.es

Peter J. Stuckey
Faculty of IT, Monash University, Australia
peter.stuckey@monash.edu

ABSTRACT

Dynamic Partial Order Reduction (DPOR) algorithms are used in stateless model checking to avoid the exploration of equivalent execution sequences. DPOR relies on the notion of *independence* between execution steps to detect equivalence. Recent progress in the area has introduced more accurate ways to detect independence: Context-Sensitive DPOR considers two steps p and t independent in the current state if the states obtained by executing $p \cdot t$ and $t \cdot p$ are the same; Optimal DPOR with Observers makes their dependency conditional to the existence of future events that observe their operations. We introduce a new algorithm, Optimal Context-Sensitive DPOR with Observers, that combines these two notions of conditional independence, and goes beyond them by exploiting their synergies. Experimental evaluation shows that our gains increase exponentially with the size of the considered inputs.

ACM Reference Format:
Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. 2019. Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACKNOWLEDGMENTS

This work was funded partially by the Spanish MECD FPU Grant FPU15/04313, the Spanish MINECO project TIN2015-69175-C4-2-R, the CM project P2018/TCS-4314 and the Australian ARC project DP180100151.

1 INTRODUCTION

Partial Order Reduction (POR) considers two execution sequences equivalent if one can be obtained from the other by swapping adjacent, *independent* execution steps. Each such equivalence class is called a Mazurkiewicz [15] trace, and POR guarantees that exploring one sequence per equivalence class is sufficient to cover all. Early POR algorithms [8, 10, 18] relied on static approximations of independence. The Dynamic-POR (DPOR) algorithm [9] was a breakthrough because it uses the information witnessed during the actual execution of the sequence to decide dynamically what to explore. Thus, it often explores less sequences than approaches based on static approximations. As a result, DPOR is considered one of the most scalable techniques for software verification.

The cornerstone of DPOR is the notion of *(in)dependence*, which is used to decide if two concurrent execution steps p and t (do not) interfere with each other and, thus, both $p.t$ and $t.p$ sequences must (not) be explored. To guarantee soundness, DPOR approximates independence and, thus, can lose precision if it treats execution steps as interfering when they are not. Optimal DPOR (ODPOR) [2] ensures optimality (never explores equivalent execution sequences), but only w.r.t. *unconditional* independence, which requires execution steps to be independent in any possible state. In practice, syntactic approximations are used to detect unconditional independence: typically, two execution steps are considered dependent if both access the same variable and at least one modifies it.

Any DPOR algorithm can thus improve its efficiency by using a more accurate independence notion [11]. Two recent approaches – DPOR$_{cs}$ (Context-Sensitive DPOR) [3] and ODPOR^{ob} (Optimal-DPOR with Observers) [6] – have achieved this by integrating orthogonal notions of *conditional independence* into DPOR:

- *DPOR$_{cs}$*: introduced the notion of *context-sensitive* independence, which only requires execution steps p and t be independent in the state S where they appear. This is determined by executing sequences $p.t$ and $t.p$ in S, and checking if the two states reached are equal. Consider, for example, the three concurrent processes p, q and r below, and the execution tree in Fig. 2(a), which will be explained throughout the paper.

```
int x = 0;
p: x = 1;
q: x = 2;
r: assert (x < 3);
```

Assume p is scheduled first and we reach state 1, where x==1. Executing either $q \cdot r$ or $r \cdot q$ in state 1 yields the same final state: x==2 and the assertion holds. Therefore, DPOR$_{cs}$ considers r and q independent in the context of state 1.

ISSTA 2019, 15â19 July, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- *ODPOR^{ob}*: introduced the notion of *observability*, where dependencies between execution steps p and t are conditional to the existence of future steps, called observers, which read the values modified by p and t. Consider again the three concurrent processes p, q and r above, and the execution tree in Fig. 2(a). Assume r is scheduled first reaching state 9, where x==0 and the assertion holds. ODPOR^{ob} considers q and p independent since, while their interleaved execution leads to different final states, variable x is not observed later.

DPOR$_{cs}$ and ODPOR^{ob} modified the DPOR algorithm to exploit their notions of independence. We present a further modification of DPOR, called ODPOR$^{ob}_{cs}$ (Optimal Context-Sensitive DPOR with Observers), that not only combines and exploits these two powerful notions, but also takes advantage of their synergy to gain further pruning. Let us consider the leftmost branch of the execution tree in Fig. 2(a). DPOR$_{cs}$ does not consider p and q independent, as they give different values to variable x. ODPOR^{ob} does not consider them independent either, as r observes the different values they give to x. However, DPOR$^{ob}_{cs}$ does consider them as independent, as the assertion of observer r evaluates to *true* after executing both $p \cdot q$ and $q \cdot p$. Two major contributions are needed for this:

1. DPOR$_{cs}$ was formulated over Source-DPOR [1]. Thus, it did not include the extension of *wakeup trees* used by ODPOR to ensure optimality, and later used to handle observers. Our first contribution is the formulation of DPOR$_{cs}$ over ODPOR, which we name *Optimal Context-Sensitive DPOR* (ODPOR$_{cs}$).
2. Our second contribution is to integrate observability into ODPOR$_{cs}$. For this, we modify context-sensitive independence to be *modulo observability*, which only requires equivalence for variables affected by future observers.

We have implemented our ODPOR$^{ob}_{cs}$ algorithm and experimentally evaluated with benchmarks from [3], [5] and [6]. Our results show our algorihtm can explore exponentially less sequences than either DPOR$_{cs}$ or ODPOR^{ob}.

2 PRELIMINARIES

2.1 Basics of DPOR and ODPOR

An *event* (p, i) of execution sequence E represents the i-th occurrence of process p in E. We use $e <_E e'$ to denote that event e occurs before event e' in sequence E, s.t. $<_E$ establishes a total order between events in E.

The core concept in ODPOR is that of the *happens-before* partial order among the events in execution sequence E, denoted by $\rightarrow_E$. This relation is used to define a subset of the $<_E$ total order, such that any two sequences with the same happens-before order are equivalent. Let $dom(E)$ denote the set of events in E. Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence with the same happens-before relation $\rightarrow_{E'}$ as $\rightarrow_E$. Thus, $\rightarrow_E$ 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 equivalent.

The happens-before relation is also used for defining the notion of *race*. Event e is said to be in race with event e' in execution E, written $e \lessdot_E e'$, if the events belong to different processes, e happens-before e' in E ($e \rightarrow_E e'$), and the two events are "concurrent" ($\exists E'$ s.t. $E' \simeq E$ and the two events are adjacent in E'). We write $e \precsim_E e'$ to denote that e is in a reversible race with e', i.e., e is in a race with e' and the two can be reversed ($\forall E'$ s.t. $E' \simeq E$ and e appears immediately before e', e' is not blocked).

Optimality in ODPOR is achieved through the use of *wakeup trees*. A wakeup tree is an *ordered tree* $\langle B, \prec \rangle$, where the set of nodes B is a finite prefix-closed set of sequences of processes, with the empty sequence ϵ at the root. The children of node w, of form $w.p$ for some set of processes p, are ordered by $\prec$. Intuitively, a wakeup tree of sequence E, written $wut(E)$, is composed of partial execution sequences that must be explored from E, as they (a) reverse the order of detected races, and (b) are provably not equivalent. As a result, ODPOR does not even initiate equivalent explorations, achieving exponential reductions over earlier DPOR algorithms.

To ensure an execution sequence v does not lead to equivalent explorations if inserted in $wut(E)$, the *sleep* and *weak initials* sets [1] are used. The sleep set of execution E, $Sleep(E)$, contains the processes that should not be explored from E, as they lead to equivalent executions. The weak initials set of sequence w from execution E, $WI_{[E]}(w)$, contains any process with no "happens-before" predecessors in $dom_{[E]}(w)$, where $dom_{[E]}(w)$ denotes the subset of events in execution sequence $E.w$ that are in w, i.e., $dom(E.w) \backslash dom(E)$. Then, v is known to lead to equivalent explorations from E if $Sleep(E) \cap WI_{[E]}(v) \neq \emptyset$.

Other notation we use includes: $\hat{e}$, denoting the process of event e; $s_{[E]}$, the state after executing sequence E; *enabled(s)*, the set of processes that can perform an execution step from state s; $pre^{+}(E, e)$ and $pre(E, e)$, the prefix of sequence E up to, including and not including e, respectively; $insert_{[E]}(v, wut(E))$, the extension of $wut(E)$ with new sequence v; and $subtree(\langle B, \prec \rangle, p)$, the subtree of the wakeup tree $\langle B, \prec \rangle$ rooted at process $p \in B$, that is, the tree $\langle B', \prec' \rangle$, where $B' = \{w | p.w \in B\}$ and $\prec'$ is the restriction of $\prec$ to B'.

2.2 ODPOR with observers

The notion of dependency we use in this paper extends the traditional one based on the concept of *observer* introduced in [6].

Definition 2.1 (observers(e, e', E)[6]). Given an execution sequence E, for all events $e, e' \in dom(E)$ where $e \lessdot_E e'$, there exists a set $O = observers(e, e', E) \subseteq dom(E)$ such that:

1. For all $o \in O$, it holds that $e \rightarrow_E o$, $o \neq e'$, and $o \not\rightarrow_E e'$.
2. For all $o, o' \in O$, it holds that $o \not\rightarrow_E o'$.
3. If $E' \simeq E$, then $observers(e, e', E') = O$.
4. For every prefix E' of E such that $e, e' \in dom(E')$:
    - If O is empty, then $e \rightarrow_{E'} e'$.
    - If O is nonempty, then $e \rightarrow_{E'} e'$ iff $dom(E') \cap O \neq \emptyset$.
5. If $e \precsim_E e'$, for all sequences w s.t. $E.w$ is a sequence, and all events $e'' \in dom(E)$:
    - If $e \not\rightarrow_E e''$, then $e \not\lessdot_{E.w} e''$.
    - If $e'' \not\rightarrow_E e'$, then $e'' \not\lessdot_{E.w} e'$.
6. For all $e'' \in dom(E)$ such that $e' \rightarrow_E e''$, it holds that $O \cap observers(e', e'', E) = \emptyset$.
7. If $O = \{o\}$ and $E = E'.\hat{o}$ for some o and E', then for any $E'' \simeq E'$, either $e \rightarrow_{E''.\hat{o}} e'$ or $e' \rightarrow_{E''.\hat{o}} e$.

Intuitively, $observers(e, e', E)$ is the set of other events in E, independent of each other (by Property 2), that read the value written

```
1:  procedure RaceDetection($E$)
2:    for all $e, e' \in dom(E)$ such that $e \precsim_E e'$ do
3:      let $E' = pre(E, e)$;
4:      if $observers(e, e', E) \neq \emptyset$ then
5:        let $o = max_E(observers(e, e', E))$;
6:        let $v = notdep^*(e, e', E).\hat{e'}.\hat{e}.(notobs^*(e, e', E) \backslash \hat{e'}).\hat{o}$;
7:      else
8:        let $v = notdep^*(e, e', E).\hat{e'}$;
9:    if $v \notin redundant(E', done)$ then
10:     $\langle wut(E') \rangle := insert_{[E']}(v, wut(E'))$;
```

Figure 1: RaceDetection of ODPOR^{ob} [6]

Algorithm 1 ODPOR$_{cs}$ algorithm

```
11: procedure explore($E$,$WuT$,$\underline{DnD}$)
12:   $\underline{dnd(E) := DnD}$;
13:   done$(E) := \emptyset$;
14:   if $enabled(s_{[E]}) = \emptyset$ then $RaceDetection(E)$;
15:   else if $WuT \neq \langle \{\epsilon\}, \emptyset \rangle$ then
16:     $wut(E) := WuT$;
17:   $\underline{\text{else if } enabled(s_{[E]}) \backslash dnd(E) = \emptyset}$ then
18:     $\underline{\text{for each } p \in dnd(E) \text{ such that } |p| = 1 :}$
19:       $\underline{RaceDetection(E.p)}$;
20:   else
21:     choose $p \in enabled(s_{[E]})\underline{\backslash dnd(E)}$;
22:     $wut(E) := \langle \{\epsilon, p\}, \{(p, \epsilon)\} \rangle$;
23:   while $\exists p \in wut(E)$ do
24:     let $p = min_{\prec}\{p \in wut(E)\}$;
25:     $\underline{\text{if } p \in dnd(E) \text{ then}}$
26:       $\underline{RaceDetection(E.p)}$;
27:     else
28:       let $WuT' = subtree(wut(E), p)$;
29:       $\underline{\text{let } DnD' = \{v \mid v \in dnd(E), p \notin v, E \models p \diamond v\}}$
30:                  $\underline{\cup \{v \mid (p.v) \in dnd(E)\}}$;
31:       $Explore(E.p, WuT', \underline{DnD'})$;
32:       add $p$ to $done(E)$;
33:     remove all sequences of form $p.w$ from $wut(E)$;
34: procedure RaceDetection($E$)
35:   for all $e, e' \in dom(E)$ such that $e \precsim_E e'$ do
36:     let $E' = pre(E, e)$; $\underline{\text{let } dont = \epsilon}$;
37:     let $v = notdep^*(e, e', E).\hat{e'}$; $\underline{v := v.I_{\text{fut}}(E', v, E)}$;
38:     $\underline{\text{if } s_{[pre^+(E,e')]} = s_{[E'.(v.suc(e,E))_{\leq_E e'}]}}$ then
39:       $\underline{dont := v.\hat{e}}$;
40:     if $v \notin redundant(E', done)$ then
41:       $\langle wut(E') \rangle := insert_{[E']}(v, wut(E'))$;
42:       $\underline{\text{add } dont \text{ to } dnd(E')}$;
```

by e (e') for any variable also written by e' (e) (by Property 4). By an abuse of notation, we will sometimes treat this set as a sequence.

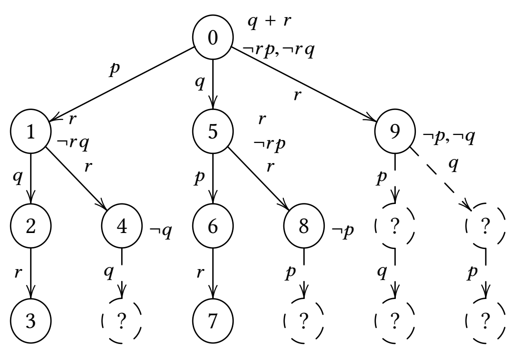
Thus, the happens-before relation used in this section is the one defined in [6], based on the notion of observability discussed previously (except from Sec. 3, where we use the relation in [2]). Intuitively, two processes p and q are *dependent modulo observability* in execution E if either *enable*s the other (i.e., executing $E.p$ introduces q, or vice versa), or (ii) $s_{[E'.p.q]} \neq s_{[E'.q.p]}$, where $E' < E$, and there exists in E at least an observer reading the variable written by both of them. The code in black of procedure *Explore* of Algorithm 1 (lines 11-33, excluding lines 13, 25-27 and 29-30) and the code of procedure *RaceDetection* of Fig. 1 corresponds to the ODPOR^{ob} algorithm [6], which extends the original ODPOR [2] with the notion of observers, and is our starting point. ODPOR^{ob} carries out a depth-first exploration of the execution tree from execution sequence E (initially empty) using DPOR. Essentially, it dynamically finds reversible races and is able to backtrack at the appropriate scheduling points to reverse them. For this purpose, it keeps two sets at every prefix E' of E: the usual wakeup tree $wut(E')$, with the execution sequences that must be explored from E', and the set $done(E')$ of processes that have already been explored from E'.

ODPOR^{ob} starts by selecting (line 24) the leftmost process p in the wakeup tree, according to its order $\prec$, that is enabled by state $s_{[E]}$ (due to line 14). If there is such a process, it sets WuT' as the subtree of $wut(E)$ with root p (line 28), and recursively explores every sequence in WuT' from $E.p$ (line 31). Note that $wut(E)$ might grow as this recursion progresses, due to later executions of line 10. After the recursion finishes, it adds p to $done(E)$, removes from $wut(E)$ all sequences that start from p, and iterates selecting a new p. Once an entire sequence E has been explored ($enabled(s_{[E]}) = \emptyset$), the algorithm performs the race detection phase (line 14). This starts by finding all pairs of events e and e' in $dom(E)$ such that $e \precsim_E e'$. For each such pair, it sets E' to $pre(E, e)$ and checks if the race between e and e' is *observed* (line 4).

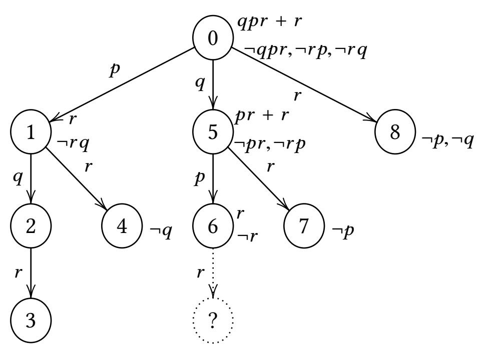
If the race is not observed, v is set to $notdep^*(e, e', E).\hat{e'}$ (line 8), where $notdep^*(e, e', E)$[1] is the subsequence of processes $\hat{e''}$ of E such that events e'' hold $e <_E e''$ and $e \not\rightarrow_E e''$. If it is observed, the race must be reversed and observed by the same observers. Thus, the last (max_E) observer o executed in E is selected (line 5) and used to compute v (line 6), where $notobs^*(e, e', E)$[1] denotes the subsequence of E containing any process $\hat{e''}$ such that $e \rightarrow_E e''$, but e'' does not observe the race $e \precsim_E e'$, and $o' \not\rightarrow_E e''$ for any observer o' of the race. There is a small change in line 5 with respect to [6]: we select o as the last (rather than an arbitrary) observer from $observers(e, e', E)$. The reason for this will be clear in Sec. 4.1. Finally, if v is not redundant for E' (line 9), it is inserted into $wut(E')$ (line 10). To detect if v is redundant from E', ODPOR^{ob} cannot use sleep sets because they are not sufficiently precise, in the presence of observers, for avoiding redundant explorations without missing non-redundant ones [6]. Instead, ODPOR^{ob} uses the set *done*: $v \in redundant(E', done)$ iff $E'.v$ is an execution sequence and there is a partitioning $E' = w.w'$ such that $done(w) \cap WI_{[w]}(w'.v) \neq \emptyset$.

Example 2.2. Consider again the processes p, q and r in Fig. 2(a). Since they all have a single event, by abuse of notation, we will refer to events by their process name. The algorithm starts at state 0 in Fig. 2(a), with both E and WuT empty. The execution first chooses p, and explores sequence p with an empty *done* set to state 1. The execution then chooses q and explores sequence $p.q$ with an empty *done* set to state 2. Since now only r can be chosen, the execution explores sequence $p.q.r$ to state 3. Now, the race detection phase detects an observed race for p and q, as they both write variable x

[1]The mark * in functions $notdep^*$ and $notobs^*$ indicates they will be redefined later. Function $notdep^*(e, e', E)$ does not use parameter e', it will be used once redefined.



(a) Full tree computed by ODPOR; dashed fragment not computed by DPOR$_{cs}$ (nor by ODPOR$_{cs}$); Arrow labels: scheduled process. Node labels up: wakeup trees ($v + w$ is a tree with two traces). Node labels down: don't-do (dnd).



(b) Full tree computed by ODPOR$_{cs}$; dotted fragment not computed by ODPOR$^{ob}_{cs}$. Labels are as in 2(a).

Figure 2: Execution trees computed by DPOR algorithms for our running example starting from x==0

and are observed by r (line 4). It then creates sequence $q.p.r$ in line 6, which will later lead to sequence $q.p$ and will thus make r observe the value written by p. The created sequence is added to $wut(\epsilon)$ of state 0 in line 10. A race between q and r is also detected and r is added to $wut(p)$. Execution then backtracks to state 1, adding q to $done(p)$ on the way. Next, it chooses r and continues, exploring the first five executions in Fig. 2(a). Once the fifth one is completed, it checks if p is in an observed race with q. Since it is not, as there is no observer after them, the sixth execution is not even started. Thus, ODPOR^{ob} explores one sequence less than ODPOR.

3 OPTIMAL CONTEXT-SENSITIVE DPOR

The happens-before relation used in this section is the one in [2], which does not consider observability. Essentially, DPOR$_{cs}$ works as follows: when a reversible race $e \precsim_E e'$ is detected, it not only updates the appropriate structures to ensure the race is reversed on backtracking, but also checks whether events e and e' are independent in the current context E, that is, whether $s_{[E.\hat{e}.\hat{e'}]} = s_{[E.\hat{e'}.\hat{e}]}$. If they are, it stores a sequence in a new *don't-do* set (in the original DPOR$_{cs}$ it was stored in the sleep set) at every prefix E' of E, indicating that this sequence must not be explored in full when backtracking to E'. Consider the working example of Fig. 2(a). When DPOR$_{cs}$ reaches state 3 (execution sequence $p.q.r$), it realizes q and r can be regarded as independent in context p, as $s_{[p.q.r]} = s_{[p.r.q]}$ even though they are dependent according to the happens-before relation in [2] and the usual syntactic approximation, since q writes global variable x and r reads it. Hence, it adds $r.q$ to the don't-do set of state 3. Once r is explored, q is not executed because it is in the don't-do set of state 4, which prevents the whole exploration of $p.r.q$.

As mentioned before, our first contribution is the reformulation of DPOR$_{cs}$ as an extension of ODPOR, rather than of Source-DPOR. This yields an optimal DPOR$_{cs}$ algorithm (see below), which we refer to as ODPOR$_{cs}$. Later in Sec. 4, this extension also allows us to integrate easily the notion of observers, i.e., integrate DPOR$_{cs}$ with ODPOR^{ob}. Reformulating DPOR$_{cs}$ in terms of ODPOR is challenging due to two main problems:

- Problem I: While Source-DPOR performs race detection at every state, ODPOR must delay race detections until each sequence is explored.
- Problem II: As shown in [3], the effectiveness of DPOR$_{cs}$ is highly dependent on exploring don't-do sequences as soon as possible. Indeed, DPOR$_{cs}$ uses these sequences to guide the selection of the next process to be explored. However, the wakeup trees of ODPOR fix part of these decisions, which can affect guidance.

The ODPOR$_{cs}$ algorithm corresponds to the code of Algorithm 1. It is explained in detail in Secs. 3.1 and 3.2, which detail how problems I and II, respectively, have been overcome. Finally, Sec. 3.3 discusses its correctness and optimality.

3.1 Overcoming Problem I

Delaying race detections until the entire sequence is explored, complicates the implementation of the context-sensitive checks, as they need access to intermediate states. One could recover these states by, for example, re-executing the sequence of events to reach them, or storing them, either in full or by means of incremental state updates, to be undone on backtracking. One could also perform (part of) the checks on the fly during the exploration, instead of at the end, thus reducing the number of intermediate states needed. The preferred strategy will depend on the available memory and the concrete language features. In any case, the following assumes access to all states of the current sequence.

The new context-sensitive check corresponds to the underlined blue code in line 38 of Algorithm 1 (for now, we use the black code for v in line 37; it will be redefined in Sec. 3.2). Black code of Algorithm 1 is common both to ODPOR and ODPOR^{ob} and it was explained in Sec. 2.2. Intuitively, given a reversible race $e \precsim_E e'$ for events e and e', the check succeeds if the state right after the race, $s_{[pre^+(E,e')]}$, is the same as that obtained when the race is reversed, $s_{[E'.(v.suc(e,E))_{\leq_E^{e'}}]}$, where $suc(e,E)$ is the subsequence w of E that starts with $\hat{e}$ and contains all $\hat{e''}$ s.t. $e \rightarrow_E e''$, and $w_{\leq_E^{e'}}$ is the subsequence of w in E of processes that execute events up to, and including, e' (i.e., keeps $\hat{e''}$ only if $e'' \leq_E e'$). As a result, the sequence $E'.(v.suc(e,E))_{\leq_E^{e'}}$ executes the same events as $pre^+(E,e')$ but with the race reversed. Assuming we have access to $s_{[pre^+(E,e)]}$ and $s_{[E']}$, we only need to compute the state after the sequence $(v.suc(e,E))_{\leq_E^{e'}}$ from $s_{[E']}$. If the check succeeds, sequence $v.\hat{e}$ is added to the don't-do set $dnd(E')$ (line 42). Note that, unlike in the original DPOR$_{cs}$, v contains the processes of events executed after e' in E, that are independent of e and, thus, also independent of e'. This issue is further discussed in Sec. 3.2.

As in the original DPOR$_{cs}$, if a sequence w is added to the don't-do set of state s, w can be inherited down once we backtrack to s, possibly being reduced until it eventually becomes a unitary sequence and the exploration stops. In that case, race detection must be forced explicitly. This is the task of the new *if* statement in lines 25 and 26. Similarly, if every process enabled in $s_{[E]}$ is also in $dnd(E)$ for sequence E, then the exploration of E stops and race detection is forced explicitly, in this case for every unitary sequence in $dnd(E)$ (lines 17, 18 and 19). The support to inherit down don't-do sequences is the same as in the original DPOR$_{cs}$, corresponding to lines 29 and 30. Essentially, $E.p$ inherits each sequence v where $p.v \in dnd(E)$ (line 30), and where every process in v (line 29) is independent of p in E (denoted as $E \models p \diamond v$), i.e, where the event in $dom_{[E]}(p)$ does not happen-before any event in $dom_{[E.p]}(v)$.

Example 3.1. Let us explain the exploration performed by ODPOR$_{cs}$ on our running example in Fig. 2(a). The algorithm first explores sequence $p.q.r$ and then performs race detection. For the reversible race between q and r, the check (line 38) $s_{[p.q.r]} = s_{[p.r.q]}$ succeeds and, hence, $r.q$ is added to $dnd(p)$. The algorithm also finds a reversible race between p and q, but this time $s_{[p.q]} \neq s_{[q.p]}$ and, thus, nothing is added to $dnd(\epsilon)$. After backtracking to state 1 with r, sequence $r.q$ is inherited down to state 4 as q (line 30). Hence, this exploration is stopped at state 4 and race-detection is explicitly invoked (lines 25 and 26). For the reversible race between p and r, the check $s_{[p.r]} = s_{[r.p]}$ also succeeds adding $r.p$ to $dnd(\epsilon)$. At this point $wut(\epsilon)$ contains q and r. After backtracking to state 0 with q, p and r can be executed. Let us suppose that $q.p.r$ is completely explored. This exploration is analogous to that of $p.q.r$. Therefore, $r.p$ will be added to $dnd(q)$, stopping the exploration at state 7. Due to the reversible race between q and r, the algorithm checks $s_{[q.r]} = s_{[r.q]}$, which succeeds adding $r.q$ to $dnd(\epsilon)$. Finally, after backtracking to state 0 with r, sequences $r.p$ and $r.q$ are inherited down to state 8, as p and q, respectively (line 30). Hence, exploration stops at state 8.

3.2 Overcoming Problem II

Consider the processes p and q from our running example, and the initial exploration $E_1 = t.t'.p.q$, where t is a process defined as $t : y = 1$; and t' is another instance of the same process t. Let us assume, for now, that ODPOR$_{cs}$ uses the original definition of sequence v (line 37), that is, $v = notdep^*(e,e',E).\hat{e'}$. For the reversible race between p and q, ODPOR$_{cs}$ adds q to $wut(t.t')$. Hence, upon backtracking to $t.t'$, it will explore $E_2 = t.t'.q.p$. For the reversible race between t and t', ODPOR$_{cs}$ sets v to $p.q.t'$ and adds it to $wut(\epsilon)$. Also, since $s_{[t.t']} = s_{[t'.t]}$, it adds $p.q.t'.t$ to $dnd(\epsilon)$. Later, when backtracking to ϵ and exploring $p.q.t'$, sequence $t'.t$ is inherited down to $dnd(p.q)$. Hence, t is inherited down to $dnd(p.q.t')$, causing the exploration to stop and the race-detection phase to start (line 26) for $p.q.t'.t$. This detects a race between p and q, causing the exploration of $t'.t.q.p$, which is redundant to E_2 (as $s_{[t.t']} = s_{[t'.t]}$).

Such a redundant trace would not have been explored by DPOR$_{cs}$. This is because DPOR$_{cs}$ (as well as Source-DPOR and the original DPOR) does not record the sequence to be explored upon backtracking but, rather, an initial event to explore plus the sequences that should not be selected (by means of the so called *backtrack-set* and *sleep set*). This allows using don't-do sequences to guide DPOR$_{cs}$ decisions regarding what to explore, achieving earlier and more effective context-sensitive prunings. However, wakeup trees are essential for ODPOR to achieve optimality. Therefore, the challenge is to determine whether it is possible to keep optimality, while at the same time being able to exploit don't-do sequences at least as effectively as DPOR$_{cs}$.

In order to reverse race $e \precsim_E e'$, it suffices to have all ancestors of e' before it. Let us then re-define $notdep^*(e,e',E)$ as $ance(e,e',E)$, the subsequence of E containing the processes whose events occurs after e and happen-before $\hat{e'}$ (and thus, independent with e). This solves the problem in the above example: for the race between t and t' in E_1, the sequences added to $wut(\epsilon)$ and $dnd(\epsilon)$ would be t' and $t'.t$, respectively. This is however not enough since, in order to achieve optimality, v needs to include part of the processes of E whose corresponding events are independent with the ones in v, thus being detected as redundant in line 38. Let us define the set of *future initials*, written $I_{\text{fut}}(E',v,E)$, that contains any process with no "happens-before" predecessors in $dom_{[E'.v]}(w)$ (i.e., $WI_{[E']}(w) \setminus v$), where $E = E'.w$. Intuitively, every event executed in w is dependent with one in $v.I_{\text{fut}}(E',v,E)$ (i.e., $\forall \hat{e} \in w, \exists \hat{e'} \in v.I_{\text{fut}}(E',v,E)$ such that $e' \rightarrow_E e$). Indeed, the future initials are also required in sequence v, so that when an exploration is stopped by a don't-do sequence (line 26), the corresponding race detection phase has enough information to build the appropriate sequences for each detected new race. As a result, we redefine v as $notdep^*(e',e',E).\hat{e'}.I_{\text{fut}}(E',v,E)$ (line 37) with $notdep^*(e,e',E) = ance(e,e',E)$. In the example above, for the race between t and t' in E_1, the new sequences added to $wut(\epsilon)$ and $dnd(\epsilon)$ are $t.p$ and $t.p.t'$, respectively.

3.3 Correctness, Optimality and Final remarks

The correctness of the ODPOR$_{cs}$ algorithm follows from the correctness of ODPOR, and the fact that context-sensitive checks only remove equivalent Mazurkiewicz traces. The optimality of ODPOR$_{cs}$ with respect to the Mazurkiewicz traces based on any conditional independence is not guaranteed, since it only detects certain cases of context-sensitive independence. However, it has similar optimality results as [2] (i.e., for the Mazurkiewicz traces based on unconditional independence): if ODPOR$_{cs}$ explores a sleep set blocked execution E, then ODPOR explores completely an execution with the same happens-before relation than E. We do not compute sleep sets but they can be obtained from the *dnd* and *done* sets. Furthermore, DPOR$_{cs}$ never explores more traces than ODPOR.

Definition 3.2 (Sleep set and Sleep set blocked execution [2]). Given an execution sequence E and $dnd(E)$ set and a $done(E')$ set for each prefix $E' \leq E$, we define $Sleep(E)$ as the set of processes $\{p \mid p \in dnd(E)$ such that $\exists E' \leq E, p \in done(E')\}$. A call to $Explore(E, WuT, DnD)$ is sleep set blocked during the execution of Algorithm 1 if $enabled(s_{[E]}) \subseteq Sleep(E)$.

This section focuses on proving the correctness and optimality of ODPOR$_{cs}$ and, thus, the original check [2] is used ($sleep(E') \cap WI_{[E']}(v) \neq \emptyset$). However, a similar reasoning can be done for the check in [6]: $v \in redundant(E, done)$.

Lemma 3.3. *If Algorithm 1 discovers that $s_{[pre^+(E',e')]} = s_{[E_0.(v.suc(e,E))_{\leq_E^{e'}}]}$, for any complete sequence E of the form $E = E_0.v.\hat{e}.u'.w'$ that contains a race $e' \precsim_E e$, there is a complete sequence $E' = pre^+(E', e').w$ that defines a different Mazurkiewicz trace $T' = \!\!\longrightarrow_{E'}$ and leads to an identical final state.*

Theorem 3.4 (Soundness of ODPOR$_{cs}$). *For each Mazurkiewicz trace T defined by the happens-before relation, $Explore(\epsilon, \langle\{\epsilon\}, \emptyset\rangle, \emptyset)$ of Algorithm 1 explores a complete execution sequence that either implements T, or reaches an identical state as one that implements T.*

Let us prove now the optimality of Algorithm 1.

Lemma 3.5. *Let $E'.v.u$ be a complete execution such that $v \in wut(E)$ and $v.u \in dnd(E)$ and v' the sequence created to reverse a race found in $E'' < E'.v.u$. For all w, such that $E'.v.u.w$, let v_w be the corresponding sequence to reverse the same race in $E'' < E'.v.u.w$, then:*

$$Sleep(E'') \cap WI_{[E'']}(v') \neq \emptyset \Leftrightarrow Sleep(E'') \cap WI_{[E'']}(v_w) \neq \emptyset$$

Theorem 3.6 (Optimality of ODPOR$_{cs}$). *Algorithm 1 never explores two maximal execution sequences which are equivalent.*

Finally, let us conclude this section by noting that both DPOR$_{cs}$ and ODPOR$_{cs}$ are likely to be highly beneficial for programs with large atomic code sections (e.g., monitors, concurrent objects, and message-passing systems), where the usual approximation of dependence can be rather imprecise. It is also likely to be beneficial for programs with assertions, as these only result in two possibly (local) states: either the assertion holds or it does not. Hence, the context-sensitive independence check is more likely to succeed.

4 OPTIMAL CONTEXT-SENSITIVE DPOR WITH OBSERVERS

The ODPOR^{ob} and ODPOR$_{cs}$ algorithms of Secs. 2.2 and 3 can be combined simply by joining their codes together. This also requires using the happens-before relation based on the notion of observability of [6] throughout the algorithm. The exploration performed by such a "union" algorithm would be the intersection of the explorations of ODPOR^{ob} and ODPOR$_{cs}$, and its prunings the union of the ODPOR^{ob} and ODPOR$_{cs}$ prunings.

This section goes beyond the union of the algorithms and proposes in Secs. 4.1 and 4.2 two enhancements that exploit the combination and the synergy between the notions of context-sensitive independence and observability. The resulting algorithm ODPOR$^{ob}_{cs}$ is presented in Algorithm 2. Finally, Sec. 4.3 studies the soundness of ODPOR$^{ob}_{cs}$.

Algorithm 2 ODPOR$^{ob}_{cs}$ algorithm

```
43: procedure EXPLORE($E$,$WuT$,$DnD$)
44:   $dnd(E) := DnD$;
45:   done$(E) := \emptyset$;
46:   if $enabled(s_{[E]}) = \emptyset$ then $RaceDetection(E)$;
47:   else if $WuT \neq \langle\{\epsilon\}, \emptyset\rangle$ then
48:     $wut(E) := WuT$;
49:   else if $enabled(s_{[E]}) \backslash dnd(E) = \emptyset$ then
50:     for each $p \in dnd(E)$ such that $|p| = 1$ :
51:       $RaceDetection(E.p)$;
52:   else
53:     choose $p \in enabled(s_{[E]}) \backslash dnd(E)$;
54:     $wut(E) := \langle\{\epsilon, p\}, \{(p, \epsilon)\}\rangle$;
55:   while $\exists p \in wut(E)$ do
56:     let $p = min_{\prec}\{p \in wut(E)\}$;
57:     if $p \in dnd(E)$ then
58:       $RaceDetection(E.p)$;
59:     else
60:       let $WuT' = subtree(wut(E), p)$;
61:       let $DnD' = \{v \mid v \in dnd(E), p \notin v, E \models p \diamond v\}$
62:           $\cup \{(u.v) \mid (u.p.v) \in dnd(E), E \models_{u.p.v} p \diamond u\}$;
63:       $Explore(E.p, WuT', DnD')$;
64:       add $p$ to $done(E)$;
65:     remove all sequences of form $p.w$ from $wut(E)$;
66: procedure RACEDETECTION($E$)
67:   for all $e, e' \in dom(E)$ such that $e \precsim_E e'$ do
68:     let $E' = pre(E, e)$; let $dont = \epsilon$;
69:     if $observers(e, e', E) \neq \emptyset$ then
70:       let $o = max_E(observers(e, e', E))$;
71:       let $v = notdep^*(e, e', E).\hat{e'}.\hat{e}.(notobs^*(e, e', E) \backslash \hat{e'}).\hat{o}$;
72:       let $o_s = observers(e, e', E)$; $v := v.I_{\text{fut}}(E', v, E))$;
73:       if $\bigwedge_{o' \in o_s} s_{[pre^+(E,o')]} =_{o'}^{e,e'} s_{[E'.v_{\leq_E^{o}}.(\hat{o}_s \backslash \hat{o})]}$ then
74:         $dont := v.(\hat{o}_s \backslash \hat{o})$;
75:     else
76:       let $v = notdep^*(e, e', E).\hat{e'}$; $v := v.I_{\text{fut}}(E', v, E)$;
77:       if $s_{[pre^+(E,e')]} = s_{[E'.(v.suc(e,E))_{\leq_E^{e'}}]}$ then
78:         $dont := v.\hat{e}$;
79:     if $v \notin redundant(E', done)$ then
80:       $\langle wut(E')\rangle := insert_{[E']}(v, wut(E'))$;
81:       add $dont$ to $dnd(E')$;
```

4.1 Refining the context-sensitive check for write-write races

Consider again the race detection phase on our running example of Fig. 2(b), after exploring the sequence $p.q.r$. The "union" algorithm still finds a reversible race $p \precsim_E q$ observed by r. After setting v to $q.p.r$ in line 71, the check $s_{[p.q]} = s_{[q.p]}$ in line 77 fails (recall this line is temporarily assumed to be out of the **else**). Hence, nothing is added to $dnd(\epsilon)$ and $q.p.r$ is added to $wut(\epsilon)$. Interestingly, sequence $q.p.r$ is equivalent to the already explored $p.q.r$, from the point of view of the observer, i.e., while the value of variable x is different, the assert in r holds in both cases.

It thus seems natural to perform a further and different context-sensitive check that compares the states *modulo observability*, e.g., compares $s_{[p.q.r]}$ and $s_{[q.p.r]}$ only considering the observation performed by r. Since the observed value in both cases makes the assert hold, $q.p.r$ could be added to $dnd(\epsilon)$, thus stopping the exploration of the third sequence at state 6. More precisely, given a race $e \precsim_E e'$ observed by o, we say that two states s and s' are *equivalent modulo observability*, written $s =_o^{e,e'} s'$, if the effect produced by observing e or e' is the same. If o is an assertion, the effect is the same if after both events o evaluates to the same Boolean value. Similarly, if o is an assignment to a variable y, where there is at least a variable which is read on its right-hand side and modified by both e and e', then the effect is the same if the value of y in s and s' is the same.

The implementation of the refined check corresponds to the underlined red code in lines 72 and 73 of Algorithm 2. First, it sets o_s to the subsequence of observer processes $observers(e, e', E)$. Then, after extending v with the required processes (see explanation below), it checks that for every observer process o' in o_s (this time treated as a set), the state after executing o' is equivalent modulo observability to the state obtained by the alternative sequence $E'.v_{\leq_E^o}.(\hat{o}_s \setminus \hat{o})$, which contains the reversed race, followed by observer o and the remaining observers.

As with the original context sensitive check (see Sec. 3.2), in order to be effective, it is important not to include in v unnecessary processes before the reversed race, while at the same time including at the end those that are necessary to keep optimality. The solution is analogous to that of Sec. 3.2. First, unnecessary processes are taken away from sequence v (line 71). In particular, $notdep^*$ inherits the redefinition of Sec. 3.2, and $notobs^*(e, e', E)$ is redefined as the subsequence of processes of E, excluding the occurrence e, whose events happen-before those in $observers(e, e', E)$. Finally, to ensure optimality, v is extended with $I_{\text{fut}}(E', v, E)$ (line 71), to ensure it has enough information to detect redundancies.

Note that all the predecessors of other observers are in $v_{\leq_E^o}$, thanks to the choice of o as $max_E(observers(e, e', E))$. Thus, we can execute $\hat{o}_s \setminus \hat{o}$ ($\hat{o}$ is already in v) without problem after $E'.v_{\leq_E^o}$. Note also that we cannot use $s_{[pre^+(E,o)]}$ to perform all the checks because, after every $o' \in o_s$ has been executed, there may be another event $e'' < o \in E$ such that $o' \rightarrow_E e''$, which would invalidate the check by modifying the value of the variables used in the check. That is why we use $s_{[pre^+(E,o')]}$ for each $o' \in o_s$ to perform each check in line 73. Another possibility, which could be more efficient in certain contexts (and does not require accessing these intermediate states), would be to perform all the checks with the state $s_{[notdep^*(e,e',E).\hat{e}.\hat{e'}.(notobs^*(e,e',E)\setminus\{\hat{e}\}).\hat{o}_s]}$, (where the race between e and e' has not been reversed).

The following provides the intuition behind the need to consider every observer $o' \in observers(e, e', E)$ for the new check, rather than just the selected one o. Consider our running example extended with one more observer $r' : assert(x < 2)$; and an initial exploration of the sequence $E = p.q.r.r'$. For the race between p and q we have that $observers(p, q, E) = \{r, r'\}$. Let us assume the algorithm selects $o := r$. If the new check only considers o (instead of every $o' \in o_s$), the check succeeds (the assert holds in both cases) and, hence, $q.p.r$ is added to $dnd(\epsilon)$. This prevents the exploration of sequence $q.p.r.r'$(where the assert of r' does not hold) which is not equivalent to any previously explored sequence. In this concrete example, this does not cause losing any different final result (the assert of r' also fails in other combinations). However, this would not be the case in an example where the only possibility for the assert of r' to fail would be to execute it after $q.p$.

Note that this new check is only applied in the case of write-write races followed by an observer (i.e. when the algorithm enters the **if** of line 69) and that it can only be finer than the original check. That is why in the final algorithm, the blue code of lines 76, 77 and 78 goes within the **else** scope, hence replacing the original check for the case of write-write races. For those races that are not observed, the original check is still applied in line 77.

Example 4.1. Let us extend our running example with a process r_2 := `assert(x < 2);` which is enabled only after executing r and let us suppose that $E = p.q.r.r_2$ is the first exploration explored by Algorithm 2. We detect a race between p and q because $observers(p, q, E) = \{r\}$, so the race is observed by r. Now, the check in line 69 is true, $p.q.r$ is equivalent modulo observer r to $q.p.r$, so $q.p.r$ is added to $dnd(\epsilon)$. However, $q.p.r.r_2$ and $p.q.r.r_2$ have a different effect in r_2 (let us notice that r_2 is not an observer for these executions). We also detect a race between q and r in E, so r and $r.q$ are added to $wut(p)$ and $dnd(p)$, respectively. Now, $p.r.r_2.q$ is also explored. Let us notice that the effect of r is always true for any possible execution and the effect of r_2 is true (in $p.r.r_2.q$) or false (in $p.q.r.r_2$) depending on the execution.

4.2 Refining the inheritance of don't-do sequences

One could think that whenever a sequence w is added to a $dnd(E')$ set of sequence E' due to the new refined check, then a prefix of w is also added to $wut(E')$. Indeed, if the refined check of line 73 succeeds, the sequence $v.(\hat{o}_s \setminus \hat{o})$ is added to $dnd(E')$, and the sequence v is inserted to $wut(E')$. However, it is possible for the sequence not to be added to $wut(E')$ if it already contains an equivalent sequence (which had been added before). In such cases, the dnd sequence might not be propagated successfully during the exploration of the corresponding sequence in $wut(E')$, resulting in unnecessary exploration.

Example 4.2. Let us consider our running example but replacing process r by $r : o = x$;, and first exploring sequence $E_1 = p.p'.q.r$, where p' is another instance of the same process p. For the race between p and q, the refined check builds the alternative sequence $p'.q.p.r$ (note that p' happens-before q in E_1). The obtained observation is $o == 1$, whereas in the original E_1 it was $o == 2$, hence $p'.q.p.r$ is added to $wut(\epsilon)$ but not to $dnd(\epsilon)$. The algorithm explores four more sequences before backtracking to the root, including sequence $E_2 = p.q.p'.r$. In this case, for the race between p and p', the refined check builds the alternative sequence $q.p'.p.r$ (note that q happens-before p' in E_2). The obtained observation both in E_2 and $q.p'.p.r$ is $o == 1$. Hence, $q.p'.p.r$ is added to $dnd(\epsilon)$ but not to $wut(\epsilon)$, since it is equivalent to $p'.q.p.r$, which was added before. The propagation of dnd sequences in Algorithm 1 (underlined blue code of lines 29 and 30) is not able to propagate down $q.p'.p.r$ when exploring $p'.q.p.r$, even though they are equivalent sequences.

The refined propagation allows to generalize the previous propagation of *dnd* sequences, which can be seen in the underlined red code of line 62 of Algorithm 2. Essentially, a sequence $u.p.v$ will now be propagated as $u.v$, if p is independent of all processes in u. In addition, the new case can take advantage of observability using the information of the trace $E'.u.p.v$. We define $E \models_{u.\hat{q}.\hat{p}.v} \hat{q} \diamond \hat{p}$ if $E.u \models \hat{q} \diamond \hat{o}$, (i.e., they are unconditional independent), or $\exists \hat{w} \in v$, such that the set of variables written both by p and q is overwritten by w and $\forall \hat{r} \in v$ that observes any of these variables, $w <_{E.u.\hat{q}.\hat{p}.v} r$. Intuitively, this refined propagation allows transitively propagating equivalences between the *dnd* set and the *WuT* of a state.

In the case of Example 4.2, when backtracking to the root to explore $p'.q.p.r$, the sequence $q.p'.p.r$ in $dnd(\epsilon)$ is propagated down to $dnd(p')$ as $q.p.r$. This allows detecting $p'.q.p.r$ as redundant. Indeed, $\epsilon \models_{qp'pr} q \diamond p'$ in $q.p'.p.r$ since r is not observing their effect (it observes the subsequent write p), whereas they would be dependent with the traditional notion of dependency.

Let us finally point out that this refinement is also applicable to the ODPOR$_{cs}$ algorithm of Sec. 3, and also to the original DPOR$_{cs}$ algorithm of [3], although in these contexts it would be much less likely to be applied.

4.3 Correctness and Optimality

The theorem for ODPOR$^{ob}_{cs}$ is analogous to the one in Sec. 3.3, but using the definition of *equivalence modulo observability*, introduced in Sec. 4.1. As in [6], the optimality used in this theorem (based on not exploring redundant maximal execution sequences) is weaker than the one in Sec. 3.3 (based on not exploring sleep set blocked executions). This is because, as we have mentioned before, *sleeps sets* cannot be used with observers to achieve the stronger optimality.

Lemma 4.3. *If Algorithm 2 discovers that* $s_{[pre^+(E',o')]} =^{e,e'}_{o'} s_{[E_0.v_{\leq^o_E}.(\hat{o}_s \backslash \hat{o})]}$ $\forall o' \in o_s$, *for any complete sequence* E *of the form* $E = E_0.v.(\hat{o}_s \backslash \hat{o}).w'$ *that contains a race* $e' \precsim_E e$ *observed by* $o_s = observers(e, e', E)$ *and* $o = max_E(o_s)$, *there is a complete sequence* $E' = pre^+(E', o).w$ *that defines a different Mazurkiewicz trace* $T' = \longrightarrow_{E'}$ *and leads to an identical final state modulo observability.*

Lemma 4.4 (soundness of new inheritance). *Let* E' *be an execution such that* $p.q.u \in wut(E')$, $q.p.u.v \in dnd(E')$, *and* $E' \models_{q.p.u.v} p \diamond q$. *If* $E = E'.p.q.u.v$ *and* $E'' = E'.q.p.u.v$, *then* $s_{[E]} = s_{[E'']}$ *modulo observability.*

Theorem 4.5 (soundness and optimality of ODPOR$^{ob}_{cs}$). *For each Mazurkiewicz trace* T *defined by the happens-before relation, Explore*$(\epsilon, \langle\{\epsilon\}, \emptyset\rangle, \emptyset)$ *of Algorithm 2 explores a complete execution sequence that either implements* T, *or reaches an equivalent state modulo observability as one that implements* T. *Furthermore, Algorithm 2 never explores two maximal execution sequences which are equivalent.*

5 EXPERIMENTS

This section reports on an experimental comparison of the performance of DPOR$_{cs}$ [3], ODPOR^{ob} [6] and our proposed ODPOR$^{ob}_{cs}$ (the implementation will be released as open-source after the double-blind reviewing). We have used three sets of benchmarks: The first one is a subset of the synthetic programs used in [6] to compare ODPOR and ODPOR^{ob}. Benchmarks FR, FR-a, LW, and abs are similar to our running example, while Lam is a mutual exclusion protocol. We have not included apr_1, an Apache library written in C that is too complex to be translated to our language. Similarly, we have excluded the second set of benchmarks used in [6], as they are written in Erlang and exploit the notion of observability inherent to a `receive` synchronization primitive, not supported in our language [12]. Our second set of benchmarks is a subset of the classical concurrent programs used in [3] to compare Source-DPOR and DPOR$_{cs}$. They feature typical distributed and concurrent algorithmic patterns in which computations are split into smaller atomic subcomputations that concurrently interleave their executions, and work on shared data. Our set includes two concurrent sorting algorithms, QS and MS, concurrent Fibonacci, Fib, a database protocol, DBP, and a consumer producer interaction, BB. We excluded Pi, PSort and Reg, as they were already optimal in DPOR$_{cs}$ and behave as Fib and MS. Our third set of benchmarks include two larger programs: MapRed, an implementation of a map-reduce model developed by a company (440 lines of code); and SDN [5], a model of a software-defined network featuring a safety policy violation (490 lines of code).

We have executed each benchmark with 4 size increasing input parameters and a timeout of 120 seconds. When reached, we write $>X$ to indicate that, for the corresponding measure, we encountered X units at timeout (i.e., it is at least X). Table 1 shows the results of the executions. An exception for this is Lam, for which we only show one input (corresponding to two processes trying to access the critical section), since it is not tractable from three processes on in our implementation (in the implementation of [6] it becomes intractable from four processes on). Column E shows the number of execution sequences, S the number of states explored, and T the time in seconds needed to compute them. Times are obtained on an Intel(R) Core(TM) i7 CPU at 2.5Ghz with 8GB of RAM (Linux Kernel 5.4.0). Columns G^{cs}_T and G^{ob}_T show the time speedup of ODPOR$^{ob}_{cs}$ over DPOR$_{cs}$ and ODPOR^{ob}, respectively, computed by dividing their respective times by that of ODPOR$^{ob}_{cs}$. To measure memory requirements, we compute for each explored trace, the sum of the cardinality of all its *dnd* sets, and show in $\mathbf{M_D}$ the maximum of these sums. In addition, $\mathbf{M_S}$ shows the maximum number of states stored, which corresponds to the number of states of the longest explored trace.

The results from the first set of benchmarks show that ODPOR$^{ob}_{cs}$ can explore exponentially less sequences than DPOR$_{cs}$ and ODPOR^{ob}. In most cases we obtain speedups with respect to both methods, although when the reduction in sequences is small, the overhead of the more complex context-sensitive checks of ODPOR$^{ob}_{cs}$ does not pay off. For FR and FR-a, ODPOR$^{ob}_{cs}$ obtains gains over both algorithms, scaling by several orders of magnitude. For LW(n), ODPOR^{ob} behaves very well, only exploring n sequences. Thus, ODPOR$^{ob}_{cs}$ obtains similar results and the overhead is small. The same happens for abs. When compared with DPOR$_{cs}$, we achieve reductions of up to 4 orders of magnitude. Since most examples reach the timeout, the gains can be bigger than the ones shown.

In the second set of benchmarks DPOR$_{cs}$ is already optimal for Fib and MS. Hence, the addition of observers has no benefit and the

	DPOR$_{cs}$			ODPOR^{ob}			ODPOR$^{ob}_{cs}$					Speed-up	
Bench.	**E**	**S**	**T**	**E**	**S**	**T**	**E**	**S**	**T**	**M$_D$**	**M$_S$**	**G$^{cs}_T$**	**G$^{ob}_T$**
FR(3)	17	36	0.02	13	29	0.03	8	27	0.04	8	6	0.5x	0.7x
FR(5)	416	865	0.35	81	247	0.17	29	141	0.14	43	8	2.7x	1.3x
FR(7)	21k	42k	34.21	449	2k	2.28	68	456	0.82	128	10	42.1x	2.9x
FR(9)	>51k	>104k	120.00	3k	12k	29.92	129	2k	5.18	328	12	>23.2x	5.8x
FR-a(4)	24	107	0.05	33	86	0.05	1	40	0.04	17	7	1.3x	1.6x
FR-a(6)	720	4k	2.17	193	680	0.88	1	119	0.25	51	9	9.0x	3.7x
FR-a(8)	>20k	>88k	120.00	2k	5k	10.79	1	270	1.18	116	11	>101.9x	9.2x
FR-a(10)	>18k	>79k	120.00	>5k	>26k	120.00	1	517	5.09	224	13	>23.6x	>23.6x
LW(3)	6	17	0.01	3	10	0.01	1	10	0.01	3	6	0.9x	0.7x
LW(5)	120	327	0.16	5	21	0.02	1	21	0.03	8	8	7.8x	0.9x
LW(7)	6k	14k	8.96	7	36	0.06	1	36	0.07	15	10	137.8x	0.9x
LW(10)	>32k	>85k	120.00	10	66	0.32	1	66	0.31	29	13	>396.0x	1.1x
abs(2)	4	28	0.02	2	15	0.01	1	12	0.01	6	9	2.0x	1.2x
abs(3)	54	577	0.30	2	26	0.02	1	23	0.02	15	12	19.8x	1.3x
abs(4)	2k	33k	29.34	2	44	0.04	1	40	0.05	25	15	598.6x	0.8x
abs(5)	>4k	>109k	120.00	2	58	0.09	1	54	0.10	37	18	>1250.0x	0.9x
Lam(2)	37	605	0.30	30	470	0.59	26	456	0.46	12	29	0.7x	1.3x
Fib(3)	1	18	0.01	6	22	0.01	1	18	0.01	2	10	0.8x	1.0x
Fib(4)	1	43	0.02	90	250	0.18	1	43	0.03	5	18	0.7x	8.2x
Fib(5)	1	99	0.04	4k	11k	22.36	1	99	0.09	10	30	0.5x	266.2x
Fib(6)	1	228	0.13	>2k	>6k	120.00	1	228	0.63	20	50	0.2x	>192.6x
QS(8)	1	763	0.31	4k	12k	23.67	1	309	0.19	7	30	1.7x	130.8x
QS(10)	1	4k	1.47	>6k	>31k	120.00	1	607	0.50	9	38	3.0x	>243.4x
QS(13)	1	25k	14.83	>2k	>15k	120.00	1	2k	1.68	12	50	8.9x	>71.6x
QS(15)	1	99k	72.95	>826	>10k	120.00	1	3k	3.56	14	58	20.6x	>33.8x
MS(7)	1	70	0.03	2k	4k	5.12	1	68	0.06	6	26	0.5x	91.4x
MS(9)	1	121	0.06	14k	37k	116.75	1	107	0.14	8	34	0.4x	877.8x
MS(11)	1	172	0.09	>4k	>13k	120.00	1	166	0.33	15	42	0.3x	>372.7x
MS(14)	1	254	0.14	>2k	>5k	120.00	1	224	1.08	14	54	0.2x	>111.5x
DBP(5)	361	5k	2.89	32	210	0.24	4	65	0.09	5	32	35.2x	2.9x
DBP(6)	2k	21k	66.59	64	451	0.73	4	73	0.14	6	38	479.1x	5.3x
DBP(7)	>3k	>26k	120.00	128	964	2.23	5	109	0.28	7	44	>431.7x	8.0x
DBP(8)	>3k	>27k	120.00	256	3k	6.78	5	117	0.44	8	50	>275.9x	15.6x
BB(3)	11	38	0.02	20	49	0.03	5	23	0.02	5	7	1.0x	2.0x
BB(5)	80	326	0.13	252	671	0.56	17	103	0.09	9	11	1.5x	6.7x
BB(7)	580	3k	1.18	4k	10k	18.02	65	459	0.82	13	15	1.5x	22.2x
BB(8)	5k	21k	12.49	>10k	>28k	120.00	257	3k	15.87	17	19	0.8x	>7.6x
MapRed	9	162	114	118	856	2961	9	162	185	24	26	0.6x	16.0x
SDN	22	242	83	58	287	229	16	83	52	20	14	1.6x	4.4x

Table 1: Experimental evaluation results

slower context-sensitive checks introduce a slowdown. For DBP, observers achieve important gains and the combination with context sensitivity gives further benefits. For QS, we obtain significant gains over both algorithms, although those over ODPOR^{ob} do not scale. In most benchmarks, we have been able to identify which of the extensions proposed in the paper are leading to the gains. In particular, the gains in QS are achieved due to the extension of Sec. 3. The refined context-sensitive check is fundamental for the gains achieved in FR and FR-a. Finally, the new way of inheriting the dnd sets leads to the gains of abs and DBP.

The results from the third set of benchmarks give evidence of the potential of our algorithm when applied over larger programs. For MapRed, both ODPOR$^{ob}_{cs}$ and DPOR$_{cs}$ explore 9 executions in 185 ms. and 114 ms. respectively, while ODPOR^{ob} explores 118 executions and takes almost 3 seconds, since there is no gain in using observers in this case. For SDN, ODPOR$^{ob}_{cs}$ explores 16 executions in 52 ms., whereas DPOR$_{cs}$ explores 22 executions in 83 ms. and ODPOR^{ob} 58 in 229 ms.

Regarding the memory requirements of ODPOR$^{ob}_{cs}$, the results show that **M$_S$** (i.e., the maximal length of the explored traces) remains low in all examples and grows linearly with the input size. The same holds for **M$_D$** except for FR and FR-a, where an exponential growth can be observed. As already mentioned in Sec. 3 (and also discussed in [17] for a different algorithm), there are a number of strategies to reduce the amount of information to be stored.

In summary, we argue that our experimental results show the exponential reduction that can be achieved by ODPOR$^{ob}_{cs}$, as our gains increase exponentially at least w.r.t. one of the algorithms in all examples we have considered.

6 CONCLUSIONS AND RELATED WORK

DPOR is one of the most scalable techniques used in the verification of concurrent systems. Recent work has introduced orthogonal notions of conditional independence into DPOR: DPOR$_{cs}$ [3] proposes a context-sensitive check in the current state to detect more accurately independence among processes, ODPOR^{ob} [6] proposes a finer notion of independence which is conditional to the existence of observers that read the values written by the processes. We propose a seamless integration of DPOR$_{cs}$ and ODPOR^{ob}, via two major technical extensions to DPOR$_{cs}$, namely: (1) incorporating (and using effectively) the notion of *wakeup tree* used by ODPOR^{ob}, and (2) refining the context-sensitive check (and the sequences computed with it) to take observers into account. As shown in our experimental evaluation, the resulting algorithm achieves prunings that go beyond the combination of the individual algorithms.

Other recent approaches have considered alternative ways of refining the detection of independence. Data-Centric DPOR [7] focuses on the read-write of variables. It defines two traces to be observationally equivalent if every read event observes the same write event in both traces. In contrast, we use the notion of observability introduced by [6], which is based on observing interference of operations, not just individual writes. The equivalence relation used by Data-Centric is proven in [7] to see more traces as equivalent than the one based on Mazurkiewicz traces, which is the one used in our work and in all other variants of the DPOR algorithm of [9]. The drawback of Data-Centric is that is optimal only for programs with acyclic communication graphs. Instead, our work is an extension of an optimal algorithm [6].

Another approach is to generate *independence constraints* (ICs), which ensure the independence of each pair of processes in the program. The work in [13, 19] generated for the first time ICs for processes with a single instruction following some predefined patterns. Recently, Constrained DPOR [4] proposed to generate ICs in a pre-phase, using an SMT solver. It later used the generated ICs within DPOR in a similar way to how our context-sensitive checks are used. In addition, it can perform another type of pruning using the notion of *transitive uniform* conditional independence –which ensures the ICs hold along the whole execution trace (and ensures uniformity as defined in [11, 14]). The extension of Constrained DPOR with observers, to the best of our knowledge, has not been studied yet. We believe the integration of wakeup trees could be done similarly to our proposal in Sec. 3, and the enhancements in Sec. 4 would be applicable also in the Constrained DPOR framework. Still, the combination of transitive uniformity and observability remains to be investigated.

An orthogonal approach to increase scalability, introduced in Quasi-Optimal POR [16], is to approximate the optimal exploration using a provided constant k. In essence, by using approximation, alternatives are computed in polynomial time, rather than making an NP-complete exploration, as in ODPOR. Another orthogonal improvement is to inspect dependencies over event chains [17].

REFERENCES

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

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal Dynamic Partial Order Reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014.

[3] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter Stuckey. Context Sensitive Dynamic Partial Order Reduction. In Victor Kuncak and Rupak Majumdar, editors, *29th International Conference on Computer Aided Verification (CAV 2017)*, volume 10426 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2017.

[4] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained Dynamic Partial Order Reduction. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 392–410. Springer, 2018.

[5] Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, and Alexandra Silva. Sdn-actors: Modeling and verification of SDN programs. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, pages 550–567, 2018.

[6] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 229–248, 2018.

[7] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *PACMPL*, 2(POPL):31:1–31:30, 2018.

[8] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.

[9] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.

[10] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[11] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *CAV*, pages 438–449, 1993.

[12] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

[13] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, pages 398–413, 2009.

[14] Shmuel Katz and Doron A. Peled. Defining conditional independence using collapses. *TCS*, 101(2):337–359, 1992.

[15] Antoni W. Mazurkiewicz. Trace theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, pages 279–324, 1986.

[16] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 2018.

[17] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 456–469, 2015.

[18] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[19] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of*

Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008.