Optimal Dynamic Partial Order Reduction with Context-Sensitive Independence and Observers

Elvira Albert[a], Maria Garcia de la Banda[b], Miguel Gómez-Zamalloa[a], Miguel Isabel[a,*], Peter Stuckey[b]

[a] *Universidad Complutense de Madrid, Spain*
[b] *Faculty of IT, Monash University, Australia*

Abstract

Dynamic Partial Order Reduction (DPOR) algorithms are used in stateless model checking of concurrent programs to avoid the exploration of equivalent execution sequences. In order to detect equivalence, DPOR relies on the notion of *independence* between execution steps. As this notion must be approximated, it can lose precision and thus treat execution steps as interfering when they are not. Our work is inspired by recent progress in the area that has introduced more accurate ways to exploit conditional notions of 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. This article introduces 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. The implementation of our algorithm has been undertaken within the Nidhugg model checking tool. Our experimental evaluation, using benchmarks from the previous works, shows that our algorithm is able to effectively combine the benefits of both context-sensitive and observers-based independence and that it can produce exponential reductions over both of them.

Keywords: Software verification, Concurrent programs, stateless model checking, partial order reduction

1. Introduction

A fundamental challenge in the verification and testing of concurrent programs arises from the combinatorial explosion that results from exploring the different ways in which processes/threads can interleave. There are several proposals to reduce the number of explored interleavings, such as depth- and context-bounding [25]. Among all proposals, Dynamic Partial Order reduction (DPOR) stands out due to its proven scalability, which results from being a *stateless* approach, i.e., one where global states do not need to be explicitly stored. DPOR is the dynamic realization of Partial-Order Reduction (POR) [14, 15, 12]. POR is a general theory that provides full coverage of all possible executions of concurrent programs by identifying equivalence classes of redundant executions, and only exploring one representative of each class. 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 [24] trace, and POR guarantees that exploring one sequence per equivalence class is sufficient to cover all. Early POR algorithms [15, 12, 31] relied on static approximations of independence. The Dynamic-POR (DPOR) algorithm [13] 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 sequences p followed by t (written $p.t$) and the reversed $t.p$ 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) [1] ensures optimality, that is, only one sequence per equivalence class is explored and the exploration of any equivalent execution sequence is not even initiated. However, this is done w.r.t. a restricted notion of independence, which usually requires execution steps to be independent in any possible state during the current execution. In practice, syntactic approximations are used to detect 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 [16]. Two recent approaches – DPOR_{cs} (Context-Sensitive DPOR) [4] and ODPOR^{ob} (Optimal-DPOR with Observers) [9] – have achieved this by integrating orthogonal notions of *conditional independence* into DPOR:

*Corresponding author: Miguel Isabel, Department of Sistemas Informáticos y Computación , C/ Profesor José García Santesmases, s/n Complutense University of Madrid, E-28040 - Madrid (Spain). Phone/Fax +34 91 3947641 / +34 91 3947529.

Email addresses: elvira@sip.ucm.es (Elvira Albert), maria.garciadelabanda@monash.edu (Maria Garcia de la Banda), mzamalloa@ucm.es (Miguel Gómez-Zamalloa), miguelis@ucm.es (Miguel Isabel), peter.stuckey@monash.edu (Peter Stuckey)

- *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. Let us consider the following simple example, borrowed from [9], with three concurrent processes p, q and r (each containing a single instruction) where the global variable x is initialized to 0:

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

  Assume p is scheduled first and we reach a state s where x==1. Executing either $q.r$ or $r.q$ from s yields the same final state: x==2 and the assertion holds. Thus, DPOR$_{cs}$ considers r and q independent in s (e.g., in x==1).

- *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. Assume r is scheduled first reaching a state s where x==0 and the assertion holds. ODPOR^{ob} considers q and p independent in s since, while their interleaved execution leads to different final states, variable x is not observed later.

1.1. Summary of Contributions

DPOR$_{cs}$ and ODPOR^{ob} modified the DPOR algorithm to exploit their notions of independence. As our overall contribution, we present a further modification of DPOR, called Optimal Context-Sensitive DPOR with Observers (ODPOR$^{ob}_{cs}$), that not only combines and exploits these two powerful notions, but also takes advantage of their synergy to gain further pruning. Let us consider again the same three processes and the trace $p.q.r$. DPOR$_{cs}$ does not consider p and q independent in this trace, 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, ODPOR$^{ob}_{cs}$ does consider them as independent in this trace, as the assertion of observer r evaluates to *true* after executing either $p.q$ or $q.p$. The following major contributions are needed for this:

1. *Optimal Context-Sensitive DPOR.* DPOR$_{cs}$ was originally formulated in [4] 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. *Extending 1 with observers.* Our second contribution is to integrate observability into ODPOR$_{cs}$, obtaining ODPOR$^{ob}_{cs}$. For this, we modify context-sensitive independence to be *modulo observability*, which only requires equivalence for variables affected by future observers.
3. *Implementation and experiments.* We have implemented ODPOR$^{ob}_{cs}$, DPOR$_{cs}$ and ODPOR$_{cs}$ within the Nidhugg model checking tool [23] and performed an experimental evaluation with benchmarks from [4] and [9]. Our experimental results show that ODPOR$^{ob}_{cs}$ can produce exponential reductions over both DPOR$_{cs}$ and ODPOR^{ob}, and that, in the worst cases, it scales similarly as the best of the other systems for the considered benchmark.

This article revises and extends a previous conference paper published in the proceedings of ISSTA'19 [5]. In addition to fixing some errors, extending the explanations and intuitions, discussing the most recent related work and adding new and more detailed examples, we have important improvements both on the practical and theoretical sides of the work. On the practical one, rather than using our implementations within the SYCO tool [6], we have reimplemented the ODPOR$^{ob}_{cs}$, DPOR$_{cs}$ and ODPOR$_{cs}$ algorithms within their Nidhugg model checking tool [23]. This very significant effort was necessary to facilitate the comparison with algorithms other than ours, particularly ODPOR [1] and ODPOR^{ob} [9]. We then performed a new and more detailed experimental evaluation using some of the original benchmarks of [9] and new encodings of those of [4]. This also required implementing the necessary support to handle atomic blocks, which are needed to encode the actors based concurrency of the benchmarks in [4]. On the theoretical side, we have made three major improvements. First, we have added correctness proofs of both the ODPOR$_{cs}$ and ODPOR$^{ob}_{cs}$ algorithms. Second, we have proposed a refined and more efficient context-sensitive check that can be exploited not only in ODPOR$_{cs}$ and ODPOR$^{ob}_{cs}$, but also in the original DPOR$_{cs}$ of [4]. And third, we have developed a simpler and more accurate definition of the notion of equivalence modulo observability which is key in ODPOR$^{ob}_{cs}$.

1.2. Structure of the Article

The rest of the article is organized as follows. Section 2 introduces the technical notation and recalls background knowledge on the DPOR and ODPOR formalisms. Section 3 first describes the original DPOR$_{cs}$ approach and then introduces the modifications and extensions needed to define ODPOR$_{cs}$, as well as the associated proofs of soundness and optimality. Section 4 presents ODPOR$^{ob}_{cs}$: the integration of observability into ODPOR$_{cs}$. The results of our implementation and experimental evaluation are reported in Section 5. Finally, Section 6 discusses related work and our final conclusions.

2. Background

2.1. Basics of DPOR

As previous work on DPOR, we assume the state space does not contain cycles, executions have finite unbounded length and processes are deterministic (i.e., at a given time there is at most one event a process can execute). An *execution sequence E* is a finite sequence of *execution steps* of the system processes that is performed from the initial state. We denote by $s_{[E]}$ the,

uniquely defined, state after executing sequence E. We use ϵ to denote the empty sequence and . to denote concatenation of sequences of process steps. E.g., $p.p.q$ denotes the execution sequence where first p performs two steps, followed by a step of q. An *event* p_i of execution sequence E represents the i-th occurrence of process p in E, e.g., event p_2 denotes the second execution step of process p in the previous sequence.

Running Example. Let us consider an extension of the example in Section 1. The program contains two global variables x and y which are initialized to 0, and three processes, p, q and r. Process p contains two assignments over variables y and x respectively. Process q only includes an assignment over variable x. Finally, process r includes two assertions that check if variable x is less than 3 and 2, respectively.

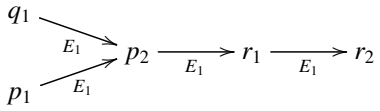
```
Int x = 0;   Int y = 0;

process p :  | process q :  | process r :
y = 1;       |    x = 2;    | assert x < 3;
x = 1;       |              | assert x < 2;
```

In this example the first assert always holds, while the second can either hold or not depending on whether x has been assigned to 2 when r_2 executes. For instance, given execution sequence $E_1 = q.p.p.r.r$, both asserts hold in $s_{[E_1]}$. This is also the case for sequence $E_2 = p.p.r.r.q$, even though we have $x = 2$ in $s_{[E_2]}$ and $x = 1$ in $s_{[E_1]}$. In contrast, for sequence $E_3 = p.p.q.r.r$, the second assert does not hold since process q writes 2 over variable x before process r reads it.

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 DPOR 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. DPOR algorithms use this relation to reduce the number of equivalent execution sequences explored, with ODPOR ensuring that only one execution sequence in each equivalence class is explored.

Example 1. *Let us consider again sequence $E_1 = q.p.p.r.r$ of our running example. Its total order of events is: $q_1 <_{E_1} p_1 <_{E_1} p_2 <_{E_1} r_1 <_{E_1} r_2$. Its happens-before relation is as follows:*



Another possible linearization of this happens-before relation is hence $E'_1 = p.q.p.r.r$. Since both E_1 and E'_1 have the same happens-before relation we have $E_1 \simeq E'_1$.

As done in [1] to support any computation model, our approach assumes the existence of a *happens-before assignment* function that assigns a happens-before relation to any execution sequence. The precision of such function can vary as long as it satisfies the set of properties specified in [1]. For the examples throughout the paper we assume a happens-before assignment based on the traditional approximation of dependency for programs with shared variables. Namely, two events p and q are considered *dependent* if either one *enable*s the other (i.e., executing $E.p$ introduces q, or vice versa), or if both access the same variable and at least one modifies it.

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'). This implies that there is no intermediate event e'' in E such that $e \rightarrow_E e'' \rightarrow_E e'$. We also 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). These are the only races the algorithms need to dynamically check for independence.

Example 2. *In both sequences E_1 and E'_1 of our running example, we have $q_1 \precsim_{E_1} p_2$, i.e., event q_1 is in a reversible race with event p_2. This is because the events belong to different processes, they are concurrent, i.e., $\exists E'$ s.t. $E' \simeq E_1$ where the two events are adjacent (namely, $E' = p.q.p.r.r$), and none of them can block the other one.*

Other notation we use includes: $\hat{e}$, denoting the process of event e (e.g., $\hat{q_1}$ is q in our example); *enabled(s)*, the set of processes that can perform an execution step from state s (e.g., a locking operation can be enabled if the lock is not taken, or disabled when it is taken); and $pre^+(E, e)$, respectively $pre(E, e)$, the prefix of sequence E up to e, including and not including e (e.g., $pre^+(E', p_2)$ is $p.q.p$ and $pre(E', p_2)$ is $p.q$ in the previous example).

2.2. Optimal DPOR

The code in black of Algorithm 1 (excluding underlined blue parts) corresponds to the ODPOR algorithm [1] adapted so that it does not make use of sleep-sets, as proposed in [9]. ODPOR carries out a depth-first exploration of the execution tree from an (initially empty) execution sequence E. 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 for every sequence E explored: the *wakeup tree* of E containing the execution sequences that must be explored from E, and the set of processes that are *done* from E, that is, have already been explored from it.

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 are ordered by $\prec$ and have the form $w.p$ for some set of processes p. We write $wut(E)$ to denote the wakeup tree of sequence E. Intuitively, $wut(E)$ is composed

of partial execution sequences that must be explored from E, as they (a) reverse the order of detected races, and (b) are not provably equivalent.

The algorithm makes use of the following functions over wakeup trees:

- $insert_{[E]}(v, wut(E))$ returns the extension of $wut(E)$ with a new sequence v.
- $subtree(\langle B, \prec \rangle, p)$ returns the subtree of wakeup tree $\langle B, \prec \rangle$ rooted at process $p \in B$, i.e., the tree $\langle B', \prec' \rangle$, where $B' = \{w | p.w \in B\}$ and $\prec'$ is the restriction of $\prec$ to B'.

Algorithm 1 ODPOR$_{cs}$ algorithm

```
1:  procedure Explore(E, WuT, DnD)
2:    dnd(E) := DnD;
3:    done(E) := ∅;
4:    if enabled(s_[E]) = ∅ then RaceDetection(E);
5:    else if WuT ≠ ⟨{ϵ}, ∅⟩ then
6:      wut(E) := WuT;
7:    else if enabled(s_[E])\dnd(E) = ∅ then
8:      for each p ∈ dnd(E) such that |p| = 1 :
9:        RaceDetection(E.p);
10:   else
11:     choose p ∈ enabled(s_[E])\dnd(E);
12:     wut(E) := ⟨{ϵ, p}, {(p, ϵ)}⟩;
13:   while ∃p ∈ wut(E) do
14:     let p = min_≺{p ∈ wut(E)};
15:     if p ∈ dnd(E) then
16:       RaceDetection(E.p);
17:     else
18:       let WuT' = subtree(wut(E), p);
19:       let DnD' = {v | v ∈ dnd(E), p ∉ v, E ⊨ p ⋄ v}
20:                  ∪ {v | (p.v) ∈ dnd(E)};
21:       Explore(E.p, WuT', DnD');
22:       add p to done(E);
23:     remove all sequences of form p.w from wut(E);
24: procedure RaceDetection(E)
25:   for all e, e' ∈ dom(E) such that e ≿_E e' do
26:     let E' = pre(E, e); let dont = ϵ;
27:     let v = notdep*(e, e', E).ê'; v := v.I_fut(E', v, E);
28:     if s_[pre+(E,e')] = s_[E'.(v.suc(e,E))_≤E'] then
29:       dont := v.ê;
30:     if v ∉ redundant(E', done) then
31:       wut(E') := insert_[E'](v, wut(E'));
32:       add dont to dnd(E');
```

ODPOR starts by selecting (line 14) the leftmost process p in the wakeup tree, according to its order $\prec$, that is enabled by state $s_{[E]}$ (due to line 4). If there is such a process, it sets WuT' as the subtree of $wut(E)$ with root p (line 18), and recursively explores every sequence in WuT' from $E.p$ (line 21). Note that $wut(E)$ might grow as this recursion progresses, due to later executions of line 31. After the recursion finishes, it adds p to $done(E)$, removes from $wut(E)$ all sequences that start with p, and iterates selecting a new p. Once a *complete* sequence E has been explored (E is said to be complete if $enabled(s_{[E]}) = \emptyset$), the algorithm performs the race detection phase (line 4). This starts by finding all pairs of events e and e' in $dom(E)$ such that $e \precsim_E e'$, that is, e is in a reversible race with e'. For each such pair, it sets E' to $pre(E, e)$ and v to $notdep^*(e, e', E).\hat{e'}$ (line 27), where $notdep^*(e, e', E)$[1] is the subsequence of processes $\hat{e'}$ of E such that $e \prec_E e''$ and $e \not\rightarrow_E e''$, that is, e occurs before e'' in E but does not happen-before it. Let us explain the intuition of this construction. In order to reverse the race, we need to explore a sequence in which e' is executed before e without changing any other happens-before relation. Such sequence must start by $E'.v.e'$ where v includes all events that are dependent with e'. In order to ensure optimality, v might also need to include other events. The sequence computed by $notdep^*$ includes all events that are independent with e and, therefore, all events that are dependent with e' (otherwise there would be no race) and also those that might be needed to ensure optimality. That is, it includes those events that are necessary and possibly more. Later, $notdep^*$ will be refined to include the shortest sequence that is needed.

Finally, if the exploration of v in E' is not redundant w.r.t already explored sequences (line 30), i.e., it does not lead to equivalent explorations, it is inserted into $wut(E')$ (line 31). Redundancy is detected by means of the so called *weak initials* sets [1]. The weak initials set of sequence w from execution E, denoted $WI_{[E]}(w)$, contains any process (in w or not) 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., $dom(E.w)\backslash dom(E)$.

Example 3. *In our running example, the set $dom_{[q.p]}(p.r.r)$ contains the events executed by sequence $p.r.r$ after executing $q.p$, that is, $\{p_2, r_1, r_2\}$. The weak initials set $WI_{[\epsilon]}(q.p.p.r.r)$ contains q since it does not have happens-before predecessors in $\{p_2, r_1, r_2\}$; and p since p_1 does not have happens-before predecessors in $\{p_2, r_1, r_2\}$ either (as can be seen in Example 1).*

The redundancy check of line 30 is then defined as follows: $v \in redundant(E', done)$ iff $E'.v$ is an execution sequence and there is a partitioning $E' = w.w'$ such that $done(w) \cap WI_{[w]}(w'.v) \neq \emptyset$. Intuitively, this check determines if the algorithm has already explored a subsequence that is equivalent to $E'.v$, by succeeding if a process without happens-before predecessors in the subset of events of $E'.v$ that are in $w'.v$, has already been explored (done) from sequence w. This is shown in action later in Example 4 (in the RaceDetection at state 15).

Note that in the original ODPOR algorithm [1] this check is implemented using sleep-sets. However, [9] shows that sleep-sets are not suitable as a mechanism to avoid redundant exploration in the presence of observers, and proposes this alternative definition that, as we have seen, relies on the done and weak-initials sets. We have therefore adopted this solution to ensure the ODPOR algorithm is ready for the integration of observers.

Example 4. *The execution tree that ODPOR explores for the running example is shown in Figure 1. Each node in the tree*

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

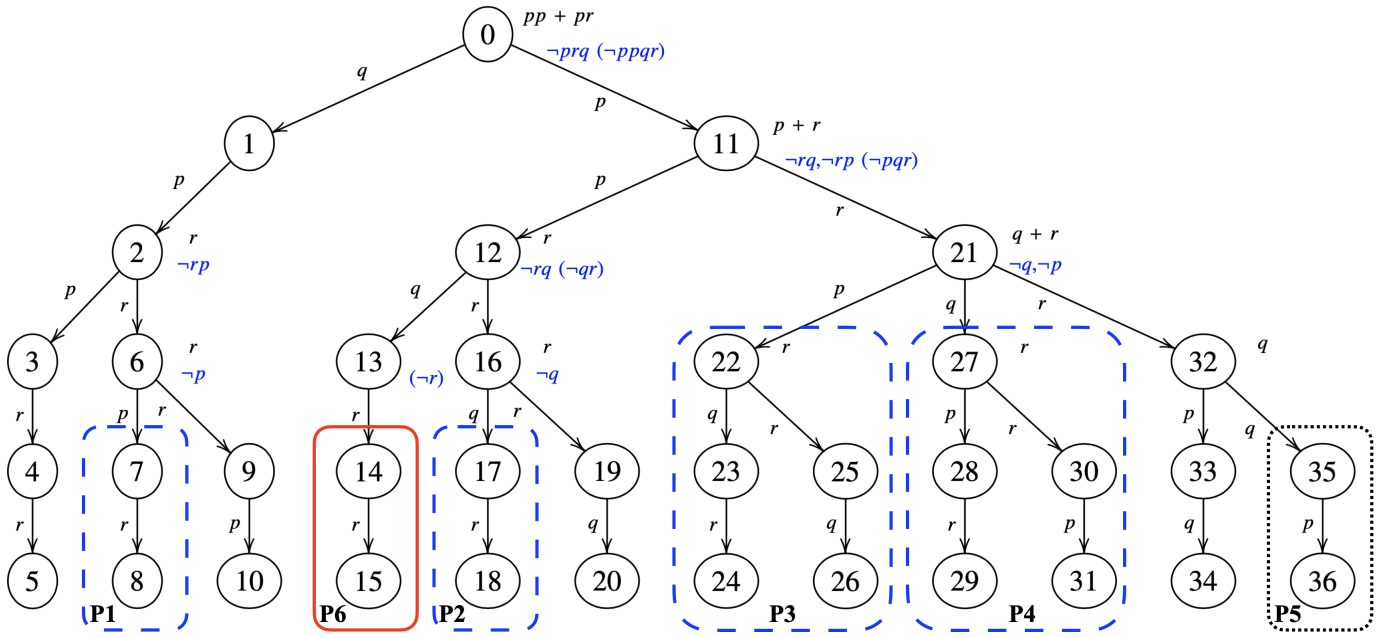


Figure 1: Execution tree computed by the ODPOR algorithms for our running example. Subtrees P1 to P4 (dashed boxes in blue) are pruned by ODPOR$_{cs}$ and DPOR$_{cs}$; P5 (dotted box) is pruned by ODPOR^{ob} and P1-P6 are all pruned by ODPOR$^{ob}_{cs}$. Arrow labels indicate the scheduled process; upper node labels the wakeup trees ($v + w$ is a tree with two traces); and lower node labels (in blue) the don't-do sets (dnd).

State	Cause	Effects
5	$p_2 \precsim_E r_1$ (access to x)	r added to $wut(q.p)$
5	$q_1 \precsim_E p_2$ (write to x)	$p.p$ added to $wut(\epsilon)$
8	$p_2 \precsim_E r_2$ (access to x)	r added to $wut(q.p.r)$
8	$q_1 \precsim_E r_1$ (access to x)	$p.r$ added to $wut(\epsilon)$
15	$q_1 \precsim_E r_1$ (access to x)	r added to $wut(p.p)$
15	$p_2 \precsim_E q_1$ (write to x)	q not added to $wut(p)$
18	$q_1 \precsim_E r_2$ (access to x)	r added to $wut(p.p.r)$
18	$p_2 \precsim_E r_1$ (access to x)	r added to $wut(p)$
24	$q_1 \precsim_E r_2$ (access to x)	r added to $wut(p.r.p)$
24	$p_2 \precsim_E q_1$ (write to x)	q added to $wut(p.r)$
26	$p_2 \precsim_E r_2$ (access to x)	r added to $wut(p.r)$
29	$p_2 \precsim_E r_2$ (access to x)	r added to $wut(p.r.q)$
34	$p_2 \precsim_E q_1$ (write to x)	q added to $wut(p.r.r)$

Figure 2: Main actions performed by ODPOR on Example 4

$$wut(\epsilon) = \langle\{\epsilon,\ q,\ p,\ p.p,\ p.r\}, \{(q, p.p), (p.p, p.r)\}\rangle$$

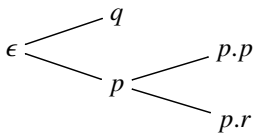


Figure 3: $wut(\epsilon)$ at state 10 in Example 4

corresponds to a different call to Explore*. The meaning of the different labels is explained in the figure caption. The lower node labels (in blue) and the boxes labeled P1-P6 should be ignored for now. They correspond, respectively, to don't-do sets (see Section 3.1), and to prunings that the other algorithms perform. Figure 2 lists the main actions that are carried out by the algorithm showing, for each of them, the state, the cause (i.e. the reversible race) and the effects of the action (i.e. the updates on wakeup trees).*

ODPOR starts with both E and WuT empty (state 0). Let us assume the algorithm first explores sequence q.p.p.r.r (by picking respectively these processes in each recursive call in line 11), thus reaching state 5 *and invoking* RaceDetection *(line 4). Since the happens-before relation of this explored execution is the one shown in Example 1, a reversible race between p_2 and r_1 is detected (p_2 happens-before r_1, appears immediately before r_1 and does not block it) setting E' to $q.p$ and v to r. This causes the addition of r to $wut(q.p)$ (see upper label of state 2), indicating the algorithm will eventually have to backtrack to state* 2 *to explore process r. A reversible race between q_1 and p_2 is also detected, setting E' to ϵ and v to $p.p$ (line 27), thus adding $p.p$ to $wut(\epsilon)$ (upper label of state 0). The algorithm then backtracks to state* 2 *(the first one with non-empty wut), along the way adding r to the done sets of states* 4 *and* 3*, and p to the done set of state* 2 *(line 22). Then it explores q.p.r (as determined by its wut, which has r) reaching state 6. Let's assume it then explores q.p.r.p.r thus reaching state 8 and invoking* RaceDetection*. Two reversible races are detected: one between r_2 and p_2, which causes the addition of r to $wut(q.p.r)$ (upper label of state 6), and one between r_1 and q_1, which causes the addition of p.r to $wut(\epsilon)$ (upper label of state 0). The algorithm then backtracks to state* 6 *whose non-empty wut has r (adding the appropriate processes to the done sets), explores q.p.r.r.p and invokes* RaceDetection *at state* 10*, which detects no new races. Figure 3 shows the $wut(\epsilon)$ at this point, both in textual and graphical forms. Node q will*

be deleted from it when backtracking to state 0 *and hence sequences p.p and p.r remain to be explored.*

The algorithm now backtracks to state 0 *(removing r from the wut of states* 6 *and* 2 *on the way due to line 23, and appropriately updating the done sets as before) whose wut has both p.p and p.r. Let us assume p.p is explored first, reaching p.p.q.r.r (at state* 15*) and invoking* RaceDetection*. It detects two reversible races: one between r_1 and q_1, which causes the addition of r to wut(p.p) (state* 12*), and one between q_1 and p_2. In this latter case q is not added to wut(p) (state* 11*) because we have $q \in redundant(p, done)$ (line 30), which means the exploration that would be performed from p.q is redundant to that already performed from q.p. Applying the above definition of redundant, $E'.v$, in this case let $E' = p$ and $v = q$, is an execution sequence and there is a partitioning $E' = w.w'$, in this case let $w = \epsilon$ and $w' = p$, such that we have a non-empty intersection between $done(\epsilon) = \{q\}$ and $WI_{[\epsilon]}(p.q) = \{p, q\}$.*

The algorithm then backtracks to state 12 *(updating the wut and done sets of intermediate states as usual) whose wut has r, and explores p.p.r.q.r reaching state* 18*. The invocation to* RaceDetection *detects two reversible races: one between r_2 and q_1, which causes the addition of r to wut(p.p.r) (and the subsequent exploration of sequence p.p.r.r.q), and one between r_1 and p_2, which causes the addition of r to wut(p) (though this has no effect since r is already inherited down to wut(p) from wut(ϵ)). The algorithm then backtracks to state* 11 *whose wut now only has r and explores the subtree rooted at state* 21*. Let us just list the reversible races that are responsible for updating the wakeup trees of states* 21, 22 *and* 27*: At state* 24 *two reversible races are detected: one between r_2 and q_1, which causes the addition of r to wut(p.r.p), and one between q_1 and p_2, which causes the addition of q to wut(p.r). At state* 26 *the reversible race between r_2 and p_2 causes the addition of r to wut(p.r). At state* 29 *the reversible race between r_2 and p_2 causes the addition of r to wut(p.r.q). Finally, at state* 34 *the reversible race between q_1 and p_2 causes the addition of q to wut(p.r.r).*

Overall, ODPOR has explored 12 *executions that matches the* 12 *possible happens-before relations corresponding to all ordering combinations of events q, p_2, r_1 and r_2, with the restriction that r_1 cannot go after r_2. A full systematic exploration would have to explore 30 executions. Note also that ODPOR has not even had to initiate any redundant exploration. Instead, the original DPOR algorithm very often needs to explore partial executions before noticing they lead to redundant explorations (the so called sleep-set-blocked explorations).*

3. Optimal Context-Sensitive DPOR

3.1. Context-Sensitive DPOR

The first algorithm that has used notions of conditional independence within the state of the art DPOR algorithm is Context-Sensitive DPOR [4] ($DPOR_{cs}$). This algorithm exploits *context-sensitive independence*, which requires checking the commutativity in the current state (the context) of every two events that are in race. If they commute, the algorithm prevents the exploration of the reversed race by adding information to the sleep set. $DPOR_{cs}$ was built in [4] on top of Source-DPOR [1] and includes the following two main features:

- Sleep-set extension. Sequences of processes are more expressive than single processes, thus sleep sets are generalized to contain sequences of processes. We denote this new kind of set as the *don't-do* set. Sequences in the don't-do sets are ignored, as they are proven to lead to redundant exploration thanks to context-sensitive independence.
- State equivalence check. When a reversible race $e \precsim_E e'$ is detected, in addition to recording the required information to ensure the race is reversed on backtracking, the algorithm 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 so, the corresponding don't-do set is extended with the sequence of the reversed race to avoid its exploration.

Example 5. *Let us consider the exploration of sequence q.p.p.r.r in the example of Figure 1. When $DPOR_{cs}$ reaches state 4, it realizes p_2 and r_1 can be regarded as independent in context q.p, as $s_{[q.p.p.r]} = s_{[q.p.r.p]}$, even though they are dependent according to the happens-before relation in [1] with the usual syntactic approximation, since p_2 writes global variable x and r_1 reads it. Hence, it adds r.p to the don't-do set of state* 2 *(see its lower label, which is shown in negative form, i.e., ¬r.p). Once r is explored, p is not executed because it is in the don't-do set of state* 6 *(don't-do sets are propagated downwards in the tree by eliminating the explored process – in this case r – from the sequence – in this case from r.p), which prevents the full exploration of q.p.r.p.r.*

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), referred to as $ODPOR_{cs}$, which enables the integration of the notion of observers (as done in Section 4). Reformulating $DPOR_{cs}$ in terms of ODPOR is challenging due to two main problems:

- Challenge I: While Source-DPOR performs race detection at every state, ODPOR must delay race detections until the sequence being explored is complete.
- Challenge II: As shown in [4], 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 full code of Algorithm 1 (both black and underlined in blue). The algorithm is discussed in detail in Sections 3.2 and 3.3, which explain how challenges I and II, respectively, have been faced. In addition, Section 3.4 proposes a refinement to the context-sensitive check of both $ODPOR_{cs}$ and $DPOR_{cs}$ that makes it potentially much more efficient in certain cases. The proofs of correctness and optimality for $ODPOR_{cs}$ are provided in AppendixA.

3.2. Challenge 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 28 of Algorithm 1 (for now, we use the black code for v in line 27; it will be redefined in Section 3.3). Recall that the black code of Algorithm 1 is common to both ODPOR and ODPOR^{ob}, and was described in Section 2.2. Intuitively, given a reversible race $e \precsim_E e'$ for events e and e', the new 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 and v is not redundant for E', sequence $v.\hat{e}$ is added to the don't-do set $dnd(E')$ (line 32). Note that, unlike in the original DPOR$_{cs}$, v contains the processes of events executed after e' in E, that do not happen-after e and, thus, neither happen-after e'. This issue is further discussed in Section 3.3.

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 (referred to as a *don't-do set blocked exploration*). In that case, race detection must be forced explicitly. This is the task of the new *if* statement in lines 15 and 16. 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 7, 8 and 9). The support to inherit down don't-do sequences is the same as in the original DPOR$_{cs}$, corresponding to lines 19 and 20. Essentially, $E.p$ inherits each sequence v where $p.v \in dnd(E)$ (line 20), and where every process in v (line 19) 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 6. *Let us explain the exploration performed by ODPOR$_{cs}$ on our running example, which is shown in Figure 1. The lower node labels (in blue) that start with ¬ indicate sequences in the corresponding don't-do sets, and the nodes within dashed boxes (labeled P1, P2, P3 and P4) correspond to nodes that are not explored and hence are pruned by ODPOR$_{cs}$. Sequences starting with ¬ in parenthesis should be ignored by now. In the following we explain the most relevant additional actions that ODPOR$_{cs}$ performs w.r.t ODPOR, some of which lead to the above-mentioned prunings. Figure 4 lists such actions, showing, for each of them, the state, the cause (i.e. the result of the context-sensitive check or the inherited sequence in the don't-do set, where ⇐ should be read as "inherited from") and the effects of the action (i.e. the updates on don't-do sets or the pruning carried out).*

St.	Cause	Effects
5	$p_2 \precsim_E r_1$ and $s_{[q.p.p.r]} = s_{[q.p.r.p]}$	$r.p$ added to $dnd(q.p)$
5	$q_1 \precsim_E p_2$ and $s_{[q.p.p]} \neq s_{[p.p.q]}$	-
6	$p \in dnd(q.p.r) \Leftarrow r.p \in dnd(q.p)$	Pruning P1
10	$q_1 \precsim_E r_1$ and $s_{[q.p.r]} = s_{[p.r.q]}$	$p.r.q$ added to $dnd(\epsilon)$
15	$q_1 \precsim_E r_1$ and $s_{[p.p.q.r]} = s_{[p.p.r.q]}$	$r.q$ added to $dnd(p.p)$
16	$q \in dnd(p.p.r) \Leftarrow r.q \in dnd(p.p)$	Pruning P2
20	$p_2 \precsim_E r_1$ and $s_{[p.p.r]} = s_{[p.r.p]}$	$r.p$ added to $dnd(p)$
21	$p \in dnd(p.r) \Leftarrow r.p \in dnd(p)$	Pruning P3
21	$q \in dnd(p.r) \Leftarrow r.q \in dnd(p)$	Pruning P4

Figure 4: Main actions performed by ODPOR$_{cs}$ on Example 6

In the invocation to RaceDetection *at state* 5*, i.e., after exploring sequence q.p.p.r.r, for the race between p_2 and r_1, the new context-sensitive check at line 28 checks if $s_{[q.p.p.r]} = s_{[q.p.r.p]}$. Since the check succeeds r.p is added to dnd(q.p) (lines 29 and 32). Also at state* 5*, for the race between p_2 and q_1, ODPOR$_{cs}$ checks whether $s_{[q.p.p]} = s_{[p.p.q]}$, but this time the check fails and, hence, nothing is added to $dnd(\epsilon)$.*

When backtracking to state 2 *with r, sequence r.p in $dnd(q.p)$ is inherited down as p in $dnd(q.p.r)$ (line 20). This causes the pruning of box P1. At state* 6*, r is chosen to be explored, because p is in $dnd(q.p.r)$. p is then removed from $dnd(q.p.r.r)$ in line 19 since p_2 and r_1 are dependent (i.e., we do not have $q.p.r \models p \diamond r$). The algorithm fully explores sequence q.p.r.r.p and then invokes again* RaceDetection *at state* 10*. For the reversible race between q_1 and r_1, the check $s_{[q.p.r]} = s_{[p.r.q]}$ succeeds and, hence, p.r.q is added to $dnd(\epsilon)$.*

The race between r_2 and p_2 causes the addition of p to $wut(q.p.r)$. Hence, when the algorithm backtracks to state 6*, and since p is both in $wut(q.p.r)$ and $dnd(q.p.r)$,* RaceDetection *is explicitly invoked (lines 15 and 16), though this time nothing new is added to any wakeup-tree nor any don't-do set.*

ODPOR$_{cs}$ then fully explores sequence p.p.q.r.r and invokes RaceDetection *at state* 15*. For the race between r_1 and q_1 the algorithm checks whether $s_{[p.p.q.r]} = s_{[p.p.r.q]}$, and since the check succeeds r.q is added to $dnd(p.p)$. This causes the pruning of box P2. The algorithm then backtracks to state* 12 *with r, explores sequence p.p.r.r.q and invokes* RaceDetection *at state* 20*. For the race between r_1 and p_2 the algorithm now checks whether $s_{[p.p.r]} = s_{[p.r.p]}$, and since the check succeeds r.p is added to $dnd(p)$ (which already had r.q in it).*

The algorithm then backtracks to state 11 *with r. It inherits down r.p and r.q to state* 21 *as p and q respectively, hence forcing it to select process r at state* 21*, causing the prunings of P3*

and P4. Finally, sequences p.r.r.p.q and p.r.r.q.p are explored as in ODPOR.

Overall, ODPOR$_{cs}$ has explored 6 *executions (instead of the* 12 *explored by ODPOR), plus the additional explorations that need to be computed by the new context-sensitive checks.*

3.3. Challenge II

To illustrate this challenge, let us consider the processes p and q from our running example, and the initial exploration $E_1 = t.t'.p.p.q$, where t is a process defined as $t : z = 1$; 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 27), that is, $v = notdep^*(e, e', E).\hat{e'}$. For the reversible race between p_2 and q_1, ODPOR$_{cs}$ adds q to $wut(t.t'.p)$. Hence, upon backtracking to $t.t'.p$, it will explore $E_2 = t.t'.p.q.p$. For the reversible race between t_1 and t'_1, ODPOR$_{cs}$ sets v to $p.p.q.t'$ (since all p_1, p_2 and q are independent of t_1, and also of t'_1) and adds it to $wut(\epsilon)$. Also, since $s_{[t.t']} = s_{[t'.t]}$, it adds $p.p.q.t'.t$ to $dnd(\epsilon)$. Later, when backtracking to ϵ and exploring $p.p.q.t'$, sequence $t'.t$ is inherited down to $dnd(p.p.q)$, which in turn causes t to be inherited down to $dnd(p.p.q.t')$, causing the exploration to stop and the race-detection phase to start (line 16) for $p.p.q.t'.t$. This detects a race between p_2 and q_1, causing the exploration of $t'.t.p.q.p$, 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 occur after e and happen-before $\hat{e'}$ (and thus, are independent with e, since otherwise e and e' would not be in race). 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. However, it is 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 28. 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 16), 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 27) 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.

3.4. A More Efficient Context-Sensitive Check

As explained above, for any given race, the state equivalence check of ODPOR$_{cs}$ (and DPOR$_{cs}$) compares the state of the current execution sequence right after the race ($pre^+(E, e')$) against that of an alternative sequence that executes the same events but with the race reversed. To ensure this, such an alternative sequence includes, after the reversed race, events that are not related to the race, namely $suc(e, E)_{\precsim_E^{e'}}$. The computation of these alternative sequences may produce an overhead on the ODPOR$_{cs}$ (and DPOR$_{cs}$) algorithm. Therefore, we would like to make them as efficient as possible without compromising the effectiveness of the prunings performed due to the context-sensitive checks.

To illustrate this, let us consider the process p from our running example, a new process t defined as $t : y = 1$ and an initial exploration $E_1 = p.p.t$. For the race between p_1 and t_1 the state equivalence check compares the states for E_1 and $E_2 = t.p.p$ and succeeds. We can observe that in E_2 the second step of p (that is $suc(p_1, E_1)_{\precsim_{E_1}^{t_1}}$) is completely unrelated to the race between p_1 and t_1 but it has been added to ensure both E_1 and E_2 have the same events. Indeed, if we take it out from E_2 the check would fail, i.e. $s_{[p.p.t]} \neq s_{[t.p]}$. However, if the check would focus only on the variables (memory addresses) which are involved in the race, in this case variable x, then the check will succeed even with the shortened sequence $t.p$.

To illustrate the potential impact this improvement may have in the algorithm, let us consider a generalization of the above pattern with a process $p' : for(i = 0; i < n; i{+}{+})\ a[i] = 0$; that initializes the first n elements of an array a, a process $t' : a[0] = 0$; that simply writes on the first position of the array, and the initial exploration $E'_1 = p'.p'.\ldots.p'.t'$ (where all steps of process p' are executed before process t'). For the race between p'_1 and t'_1 the state equivalence check would compare the states of E'_1 and $E'_2 = t'.p'.p'.\ldots.p'$ and succeed. Here we can observe the potential overhead that we may have in the computation of E'_2. The proposal is to take out from E'_2 the events that are not related to the race, hence considering instead $E'_2 = t'.p'$, and make the state equivalence check focus only on variable $a[0]$, otherwise the check would fail.

To accomplish this, we rewrite line 27 of Algorithm 1 as:

$$s_{[pre^+(E,e')]} \overset{\mathcal{V}_E^{e,e'}}{=\!=} s_{[E'.(v.\hat{e})_{\precsim_E^{e'}}]}$$

where $suc(e, E)_{\precsim_E^{e'}}$ has been taken out from the sequence of the right-hand side, and $\mathcal{V}_E^{e,e'}$ added on top of the equality, indicating the equality is checked on the subset of variables $\mathcal{V}_E^{e,e'}$, which is defined as follows. Let $W_E(e)$ and $W_E(\{e_1, \ldots e_n\})$ denote the set of variables (memory addresses) written in execu-

tion E by event e and by the set of events $\{e_1, \dots e_n\}$, respectively. Let $\mathcal{I}_E^e(x)$ be the set of variables written by event e in execution E whose written value has been affected by variable x. We use $\bar{E}$ to denote the alternative sequence including event e, that is, $E'.(v)_{\precsim_{\bar{E}}^{e'}}.\hat{e}$, where E' and v are as defined in lines 26 and 27 of Algorithm 1. We define the set of variables involved in a race between e and e' in execution E, written $\mathcal{V}_E^{e,e'}$, as:

$$\begin{aligned}
\mathcal{V}_E^{e,e'} = \; & (W_E(e) \cap W_E(e')) \cup && (1)\\
& (\textstyle\bigcup_{x \in W_E(e)} \mathcal{I}_{\bar{E}}^{e'}(x)) \cup && (2)\\
& (\textstyle\bigcup_{x \in W_{\bar{E}}(e')} \mathcal{I}_{\bar{E}}^{e}(x)) \cup && (3)\\
& (W_E(\{e, e'\}) \setminus W_{\bar{E}}(\{e, e'\})) \cup && (4)\\
& (W_{\bar{E}}(\{e, e'\}) \setminus W_E(\{e, e'\})) && (5)
\end{aligned}$$

The first term, labeled (1), includes the variables that are written in both events. The second and third terms include, for the original and the alternative executions E and $\bar{E}$ respectively, the variables written by the second event in the race whose written value has been affected by a variable written in the first event of the race in the corresponding execution. Finally, the fourth and fifth terms include the variables that are written by either e or e' in one of the executions that are not written by e nor e' in the other execution.

Example 7. *Let us illustrate the need for each of the terms above with the following examples of events e, e' and execution sequence $E = e.e'$:*

- *Consider $e : x = 1$; and $e' : x = 2$;. Since variable x is involved in a race between e and e', it should be included in $\mathcal{V}_E^{e,e'}$. This is captured by term (1).*
- *Consider $e : x = 1$; and $e' : y = x$;. In this case term (2) produces $\{y\}$ since the value written to y depends on whether e is executed before or not. The other terms produce the empty set. We hence get $\mathcal{V}_E^{e,e'} = \{y\}$.*
- *Consider $e : i = 1$; and $e' : v[i] = 1$; with an initial state where $i = 0$. Term (4) produces $\{v[1]\}$ whereas term (5) produces $\{v[0]\}$. The other terms produce the empty set. We hence get $\mathcal{V}_E^{e,e'} = \{v[0], v[1]\}$. If we instead consider an initial state with $i = 1$, we would get $\mathcal{V}_E^{e,e'} = \{\}$ since in this case both executions write $v[1]$ with a value that does not depend on event e. If we consider $e' : v[i] = i$; (with initial state $i = 1$) then term (2) would produce $\{v[1]\}$ since the written value now depends on event e.*

Note that in the case of assertion statements, the result of an assertion is assumed to be written to a memory address and thus it is included in the W and $\mathcal{I}$ sets. For instance, consider events $e : x = 1$; and $e' : assert\ x > 0$; with execution sequence $E = e.e'$. The result of the assertion is obviously involved in the race between e and e' and should be included in $\mathcal{V}_E^{e,e'}$. This is captured by term (2) since the memory address where the result of the assertion is written to is assumed to belong to $\mathcal{I}_E^{e'}(x)$. The rest of the terms produce the empty set in this case.

3.5. Correctness and Optimality

The following theorem ensures the soundness of our ODPOR$_{cs}$ extension.

Theorem 1 (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 to one that implements T.*

The optimality of ODPOR$_{cs}$ with respect to the Mazurkiewicz traces based on the context-sensitive notion of independence is in general not guaranteed, since it only detects certain cases of context-sensitive independence. However, it has analogous optimality results as the ODPOR algorithm (i.e., for the Mazurkiewicz traces based on the notion of independence of [1]): if ODPOR$_{cs}$ explores a don't-do set blocked execution E, then ODPOR explores completely an execution with the same happens-before relation than E.

Theorem 2 (Optimality of ODPOR$_{cs}$)**.** *Algorithm 1 never explores two complete execution sequences that are equivalent, and never initiates redundant executions.*

The proofs of the above theorems can be found in AppendixA.

4. Optimal Context-Sensitive DPOR with Observers

This section presents Optimal Context-Sensitive DPOR with Observers (ODPOR$_{cs}^{ob}$), our new DPOR algorithm that not only combines and exploits the notions of context-sensitive independence and observability, but also takes advantage of their synergy to gain further pruning. First, Section 4.1 recalls the formal notion of observability and the Optimal DPOR with Observers (ODPOR^{ob}) algorithm of [9]. Then, Section 4.2 presents the basic ODPOR$_{cs}^{ob}$ algorithm that simply joins the ODPOR^{ob} and ODPOR$_{cs}$ algorithms. Sections 4.3 and 4.4 present two enhancements that exploit the combination and the synergy between the notions of context-sensitive independence and observability. Finally, Section 4.5 studies the soundness of the enhanced ODPOR$_{cs}^{ob}$ algorithm.

4.1. Optimal DPOR with Observers

The notion of observability [9] allows dependencies between execution events to be conditional to the existence of later events called observers. The typical example in the context of programs with shared variables occurs when there are two events e and e' that write over the same variable, but the written value is not later read or observed. In such a case the idea is not to consider the two events as interfering and, hence, e does not happen-before e'. The intuition behind this approach is that only operations that observe a value (e.g. assertions or receives) can influence the control flow and lead to erroneous or unexpected behaviors, whereas other operations (e.g., writes or sends) cannot affect program behaviour if no future operation observes their effects.

Again, as in [1], in order not to restrict to any specific computation model, Aronis et. al. [9] assume the existence of a *happens-before assignment* function that assigns a happens-before relation to any execution sequence. The precision of

such function can vary as long as it satisfies a set of properties, which relax those of [1] in order to consider the notion of observability. Let us recall the new properties related to the notion of observability.

Definition 1 (*observers*(e, e', E)[9])**.** *Given an execution sequence E, and any two 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 $o \nrightarrow_E e'$.*
2. *For all $o, o' \in O$, it holds that $o \nrightarrow_E o'$.*
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 $e \nrightarrow_E e''$, then $e \not\lessdot_{E.w} e''$.*
   - *If $e'' \nrightarrow_E e'$, 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, in the usual particular case of variable reads and writes, $observers(e, e', E)$ is the set of other events in E, independent of each other (by Property 2), that read the value written 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.

From this point on, the presented algorithms will rely on such a generic happens-before assignment, i.e., one that considers observability and satisfies the above properties. In the examples we will continue to assume a concrete happens-before assignment based on the traditional approximation of dependency for programs with shared variables, extended to consider observability as follows. Two events p and q are considered *dependent modulo observability* if: (i) one *enable*s the other, or (ii) one writes over a variable that the other reads, or (iii) both write over a variable that is later read by an observer.

Example 8. *Let us consider sequence $E_1 = q.p.p.r.r$. We have $q_1 \rightarrow_{E_1} p_2$ since event r_1 is an observer of the race. However, event r_2 is not an observer in spite of reading variable x. This is because of Condition 2 of Definition 1 (r_1 is already an observer). On the other hand, r_2 is an observer of the race between q_1 and p_2 in sequence $E_2 = p.r.q.p.r$ and hence we have $q_1 \rightarrow_{E_2} p_2$. Finally, in sequence $E_3 = p.r.r.p.q$ we have $p_2 \nrightarrow_{E_3} q_1$ since there is not an observer after them that reads the written value on variable x.*

The ODPOR^{ob} algorithm of [9] corresponds to the code in black of procedure Explore of Algorithm 1 (excluding underlined blue parts) and the new RaceDetection procedure of Figure 5 with the original definition of *notdep** of Section 2.2. Apart from the new happens-before assignment, the new support for handling observers corresponds to lines 4-6 of Figure 5.

```
1:  procedure RaceDetection(E)
2:    for all e, e' ∈ dom(E) such that e ≾_E e' do
3:      let E' = pre(E, e);
4:      if observers(e, e', E) ≠ ∅ then
5:        let o = max_E(observers(e, e', E));
6:        let v = notdep*(e, e',E).ê'.ê.(notobs*(e, e',E)\ê').ô;
7:      else
8:        let v = notdep*(e, e', E).ê';
9:    if v ∉ redundant(E', done) then
10:     wut(E') := insert_[E'](v, wut(E'));
```

Figure 5: RaceDetection of ODPOR^{ob} [9]

Specifically, if the race between e and e' is *observed* (line 4), 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*$^*(e, e', E)$[2] 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 $o' \nrightarrow_E e''$ for any observer o' of the race. There is a small change in line 5 with respect to [9]: we select o as the last (rather than an arbitrary) observer from $observers(e, e', E)$. The reason for this will be clear in Section 4.3.

Example 9. *The exploration of ODPOR^{ob} on our working example proceeds exactly the same as that of ODPOR (see Example 4) until state 34. This is because all write-write races until this point have a subsequent observer that reads the written value. As an example, see sequences E_2 and E_3 of Example 8. However, after exploring sequence p.r.r.p.q at state 34, ODPOR^{ob} does not consider a race between p_2 and q_1 (since there is not a later observer) and hence it does not backtrack to state 32 to explore p.r.r.q.p (pruning P5 of Figure 1). Note that while the final value of variable x in these sequences is different ($x = 1$ vs. $x = 2$), ODPOR^{ob} considers that this does not affect the program behavior, as the value is not read later.*

4.2. The Basic "Union" Algorithm

The ODPOR$_{cs}$ and ODPOR^{ob} algorithms of Sections 3 and 4.1 can be combined simply by joining their codes together, that is, the code of procedure Explore of Algorithm 1 (including underlined blue parts) and the new RaceDetection procedure of Figure 5, adding the underlined blue code of the RaceDetection of Algorithm 1, including the enhancements of Sections 3.2, 3.3 and 3.4 and relying on the happens-before relation of Section 4.1. The exploration performed by such a "union" algorithm would be the intersection of the explorations of ODPOR$_{cs}$ and ODPOR^{ob}, and its prunings the union of the ODPOR$_{cs}$ and ODPOR^{ob} prunings.

Example 10. *In our working example such basic "union" algorithm proceeds as the ODPOR$_{cs}$ algorithm until state 34 and then, as ODPOR$_{cs}$ (see Example 9), it does not backtrack to state 32 to explore p.r.r.q.p. It is hence able to avoid the exploration of the states in boxes P1, P2, P3, P4 and P5 of Figure 1.*

[2]The mark * in function *notobs** indicates that it will be redefined later.

This "union" algorithm can be seen in Algorithm 2, if we ignore the code underlined in red (which corresponds to the enhancements of Sections 4.3 and 4.4) and we take lines 76, 77 and 78 outside the scope of the **else** (as explained in Section 4.3). These modifications to the "union" algorithm are performed to exploit the synergy between the notions of context-sensitive independence and observability. The resulting algorithm is ODPOR$^{ob}_{cs}$.

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

```
43: procedure EXPLORE(E,WuT,DnD)
44:   dnd(E) := DnD;
45:   done(E) := ∅;
46:   if enabled(s_[E]) = ∅ then RaceDetection(E);
47:   else if WuT ≠ ⟨{ϵ}, ∅⟩ then
48:     wut(E) := WuT;
49:   else if enabled(s_[E])\dnd(E) = ∅ then
50:     for each p ∈ dnd(E) such that |p| = 1 :
51:       RaceDetection(E.p);
52:   else
53:     choose p ∈ enabled(s_[E])\dnd(E);
54:     wut(E) := ⟨{ϵ, p}, {(p, ϵ)}⟩;
55:   while ∃p ∈ wut(E) do
56:     let p = min_≺{p ∈ wut(E)};
57:     if p ∈ dnd(E) then
58:       RaceDetection(E.p);
59:     else
60:       let WuT' = subtree(wut(E), p);
61:       let DnD' = {v | v ∈ dnd(E), p ∉ v, E ⊨ p ⋄ v}
62:           ∪ {(u.v) | (u.p.v) ∈ dnd(E), E ⊨_{u.p.v} p ⋄ 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' ∈ dom(E) such that e ≾_E e' do
68:     let E' = pre(E, e); let dont = ϵ;
69:     if observers(e, e', E) ≠ ∅ then
70:       let o = max_E(observers(e, e', E));
71:       let v = notdep*(e, e', E).ê'.ê.(notobs*(e, e', E)\ê').ô;
72:       let o_s = observers(e, e', E); v := v.I_fut(E', v, E));
73:       if ⋀_{o'∈o_s} s_[pre+(E,o')] =^{e,e'}_{o'} s_[E'.v_{≤^o_E}.(ô_s\ô)] then
74:         dont := v.(ô_s\ô);
75:     else
76:       let v = notdep*(e, e', E).ê'; v := v.I_fut(E', v, E);
77:       if s_[pre+(E,e')] =^{v^{e,e'}_E} s_[E'.(v)_{≤^{e'}_E}] then
78:         dont := v.ê;
79:     if v ∉ redundant(E', done) then
80:       wut(E') := insert_[E'](v, wut(E'));
81:       add dont to dnd(E');
```

4.3. Refining the Context-Sensitive Check for Write-Write Races

Consider again the race detection phase on our running example of Figure 1 after exploring sequence $q.p.p.r.r$. The "union" algorithm finds a reversible race $q_1 \precsim_E p_2$ observed by r_1. After setting v to $p.p.q.r$ in line 71, the check $s_{[q.p.p]} = s_{[p.p.q]}$ in line 77 fails (recall this line is temporarily assumed to be outside the **else** scope). Hence, nothing is added at this time to $dnd(\epsilon)$ and $p.p.q.r$ is added to $wut(\epsilon)$. Interestingly, sequence $p.p.q.r.r$ is equivalent to the already explored $q.p.p.r.r$ from the point of view of the observer r_1. I.e., although the value of variable x is different, the assert executed by r_1 holds in both cases. Note that this is not the case for the assert of r_2 (it holds in $q.p.p.r.r$ but it does not in $p.p.q.r.r$), but since it is not an observer (due to Condition 2 of Definition 1) it does not need to be considered in this sequence (see Example 8).

It thus seems natural in this case to perform an enhanced context-sensitive check that compares the states *modulo observability*, e.g., compares $s_{[p.p.q.r]}$ and $s_{[q.p.p.r]}$ only considering the effect of the observation performed by r_1. Since in both cases it has the same effect (the assert holds), $p.p.q.r$ could be added to $dnd(\epsilon)$, thus stopping the exploration of the fourth derivation at state 13. 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 $s \overset{W_E(o)}{=\!=} s'$ (following the notation of Section 3.4), if the variables that are written by o have the same values in s and s'. Let us recall that in the case of observers with assertion statements, the result of the assertion is assumed to be written to a memory address and thus included in the $W_E(o)$ set. In the example above with $E = q.p.p.r.r$, the equality between $s_{[p.p.q.r]}$ and $s_{[q.p.p.r]}$ is performed only looking at the result of the assertion in r_1.

The implementation of the refined check corresponds to the underlined red code in lines 72, 73 and 74 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 $E'.v_{\precsim^o_E}.(\hat{o}_s \backslash \hat{o})$, which contains the reversed race, followed by observer o and the remaining observers.

As with the original context-sensitive check (see Section 3.3), 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 Section 3.3. First, unnecessary processes are taken away from sequence v (line 71). In particular, $notdep^*$ inherits the redefinition of Section 3.3, 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 $v_{\precsim^o_E}$, 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 $E'.v_{\precsim^o_E}$. 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 with a simple modification: the instruction $assert(x < 2)$; is executed by a different process r', enabled from the initial state. Let the sequence $E = q.p.p.r.r'$ be the initial exploration of the algorithm. For the race between q_1 and p_2 we have that $observers(q_1, p_2, E) = \{r_1, r'_1\}$. Let us assume the algorithm selects $o := r$. If the new check only considers event o_1 (instead of every $o' \in o_s$), the check succeeds (the assert holds in both cases) and, hence, $p.p.q.r$ is added to $dnd(\epsilon)$. This would prevent the exploration of sequence $p.p.q.r.r'$ (where the assert of r'_1 does not hold) which is not equivalent to any previously explored sequence. In this concrete example, this does not cause us to lose any different final result (the assert of r'_1 also fails in other combinations). However, this would not be the case in an example where the only possibility for the assert of r'_1 to fail would be to execute it after $q.p.p$.

Note that the new enhanced 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 more precise than the original check. That is why in the final algorithm, the 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 races that are not observed, the original check is still applied in lines 77 and 78.

Example 11. *Let us consider the exploration performed by ODPOR$^{ob}_{cs}$ on our running example of Figure 1 (recall that fragments in all boxes are pruned by ODPOR$^{ob}_{cs}$). In the invocation to* RaceDetection *at state* 5*, i.e., after exploring sequence q.p.p.r.r, for the write-write race between q_1 and p_2, the new enhanced context-sensitive check of line 73 only checks if the result of the assertion of r_1 is the same in both $s_{[q.p.p.r]}$ and $s_{[p.p.q.r]}$ (since we have observers$(q_1, p_2, q.p.p.r.r) = \{r_1\}$), ignoring the value of variable x which is different. Since the check succeeds p.p.q.r is added to dnd(ϵ). This causes the pruning of box P*6*. The rest of the exploration proceeds as the "union" algorithm (see Example 10). Note that the enhanced check is also executed in the call to* RaceDetection *at state* 24*, i.e., after exploring sequence p.r.p.q.r, for the write-write race between p_2 and q_1, but this time it fails since the assert of observer r_2 fails after p.r.p.q.r (where $x = 2$) but instead it succeeds in p.r.q.p.r (where $x = 1$).*

4.4. Refining the Inheritance of Don't-Do Sequences

One could expect that whenever a sequence w is added to a $dnd(E')$ set of sequence E' due to the new refined check, a prefix of w be also added to $wut(E')$. Indeed, if the refined check of line 73 succeeds, sequence $v.(\hat{o}_s \setminus \hat{o})$ is added to $dnd(E')$, and sequence v is inserted to $wut(E')$. However, it is possible for the later 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 12. *Let us consider our running example but replacing process r by $r : o = x$; and exploring first sequence $E_1 = p.q.q'.p.r$, where q' is another instance of the same process q. For the race between q_1 and p_2, the refined check builds the alternative sequence $p.q'.p.q.r$ (note that q' happens-before q in E_1). The obtained observation is $o = 2$, whereas in the original E_1 it was $o = 1$, hence $q'.p.q.r$ is added to $wut(p)$ but not to $dnd(p)$. The algorithm explores four more sequences before backtracking to the root, including sequence $E_2 = p.q.p.q'.r$. In this case, for the race between q and q', the refined check builds the alternative sequence $p.p.q'.q.r$ (note that p happens-before q' in E_2). The obtained observation both in E_2 and $p.p.q'.q.r$ is $o = 2$. Hence, $p.q'.q.r$ is added to $dnd(p)$ but not to $wut(p)$, since it is equivalent to $q'.p.q.r$, which was added before. The propagation of dnd sequences in Algorithm 1 (underlined blue code of lines 19 and 20) is not able to propagate down $p.q'.q.r$ when exploring $q'.p.q.r$, even though they are equivalent sequences.*

The refined propagation allows us 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 $E \models_{u.q.p.v} q \diamond p$ if $E.u \models q \diamond p$, (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, $w <_{E.u.q.p.v} r$. Intuitively, this refined propagation allows transitively propagating equivalences between the *dnd* set and the WuT of a state.

In the case of Example 12, when backtracking to p to explore $q'.p.q.r$, the sequence $p.q'.q.r$ in $dnd(p)$ is propagated down to $dnd(p.q')$ as $p.q.r$. This allows detecting $p.q'.p.q.r$ as redundant. Indeed, $p \models_{p.q'.q.r} p_2 \diamond q'_1$ in $p.q'.q.r$ since r_1 is not observing their effect (it observes the subsequent write q_1), 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 Section 3.1, and also to the original DPOR$_{cs}$ algorithm of [4], although in these contexts it would be much less likely to be applied.

4.5. Correctness and Optimality

The theorem for ODPOR$^{ob}_{cs}$ is analogous to the one in Section AppendixA, but using the definition of *equivalence modulo observability*, introduced in Section 4.3.

Theorem 3 (Soundness of ODPOR$^{ob}_{cs}$)**.** *For each Mazurkiewicz trace T defined by the happens-before relation,*

Explore(ϵ, $\langle\{\epsilon\},\emptyset\rangle$,$\emptyset$) of Algorithm 2 explores a complete execution sequence that either implements T, or reaches an equivalent state modulo observability as one that implements T.

Theorem 4 (Optimality of ODPOR$^{ob}_{cs}$)**.** *Algorithm 2 never explores two complete execution sequences that are equivalent.*

The proofs of the above theorems can be found in AppendixB.

5. Experiments

This section reports on our experimental comparison of the performance of ODPOR [1], ODPOR^{ob} [9], DPOR$_{cs}$ [4] and our proposed ODPOR$_{cs}$ and ODPOR$^{ob}_{cs}$. The major part of the experiments are performed using the Nidhugg tool [23], a stateless model checker for shared-memory pthreads programs written in C/C++ that operates by interpreting the LLVM intermediate representation. It is important to note that Nidhugg was developed and it is maintained by some of the authors of [1] and [9], and it is indeed used in the experimental evaluation of [9]. It therefore includes implementations of the ODPOR and ODPOR^{ob} algorithms. Our experimental evaluation has hence required the implementation and integration within the Nidhugg framework of the DPOR$_{cs}$, ODPOR$_{cs}$ and ODPOR$^{ob}_{cs}$, as well as the implementation of the support to handle atomic blocks.

We also use the SYCO tool [6], a systematic testing tool for message-passing concurrent programs written in the ABS modeling language [17], for the two largest benchmarks of [5] which were unsuitable to be translated into C/C++.

5.1. Goals of the Experimental Evaluation

The goals of our experimental evaluation are the following:

- G1 Study the overall effectiveness in terms of scalability of ODPOR$_{cs}$ and ODPOR$^{ob}_{cs}$ w.r.t ODPOR, DPOR$_{cs}$ and ODPOR^{ob}. We expect cases where ODPOR$^{ob}_{cs}$ produces exponential reductions over ODPOR^{ob} and behaves similarly to DPOR$_{cs}$ and ODPOR$_{cs}$ (G1a), cases where it produces exponential reductions over DPOR$_{cs}$ and ODPOR$_{cs}$, and behaves similarly to ODPOR^{ob} (G1b), and, cases where it produces exponential reductions over all DPOR$_{cs}$, ODPOR$_{cs}$ and ODPOR^{ob} (G1c). We also expect cases where ODPOR$^{ob}_{cs}$ may behave worse than either DPOR$_{cs}$ or ODPOR^{ob} due to the overhead of either the observers or the context-sensitive part whenever it is not effective (G1d). Overall, we expect that ODPOR$^{ob}_{cs}$ will often outperform or, at least, scale similarly than the best of the other algorithms. As regards ODPOR$_{cs}$, we expect it to perform essentially equivalent to DPOR$_{cs}$ for most benchmarks (G1e). An explanation for this is found below in Section 5.3.
- G2 Observe the potential overheads that both the context-sensitive part (G2a) and the observers part (G2b) of the algorithm may produce.
- G3 Study the effectiveness of the context-sensitive check refinements of Sections 3.4 and 4.3.

5.2. Description of the Experiments

We have used three sets of benchmarks: The first one is a subset of the classical concurrent programs used in [4] 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 three concurrent sorting algorithms (pipesort, mergesort and quicksort), a distributed workers algorithm (pi), a concurrent fibonacci algorithm (fib), and a producer-consumer algorithm (boundedbuffer).

Our second set of benchmarks contains the synthetic programs used in [9] to compare ODPOR and ODPOR^{ob} and some variations of them. Benchmarks lastwrite, floatingread and lastzero correspond to benchmarks of the same name in [9], and abs corresponds to the running example of [9]'s figure 1 generalized for n processes. Benchmarks ending with -2phases and -dif are variations of the above for the corresponding prefix. Specifically, the -2phases variations perform 2 repetitions of the code pattern in each process so that potential exploration reductions can be better observed. The -dif variations write different values to all shared variables, hence context-sensitive checks cannot take advantage of context-sensitive dependencies. That allows us to reason about the worst-case scenario for the context-sensitive part of the algorithms. We also include in this second set arraywrite, a synthetic benchmark to illustrate the potential impact of the refined check of Section 3.4, which consists of $4*N$ processes writing an N-array and two processes reading the first two positions.

Each benchmark is executed for a series of increasing input parameters in order to observe the impact of the reductions and overheads on bigger explorations. Tables 1 and 2 show the results of the executions of ODPOR, ODPOR^{ob}, DPOR$_{cs}$ and ODPOR$^{ob}_{cs}$ for the first and second set of benchmarks respectively. The results for ODPOR$_{cs}$ are not shown in the tables since they are basically the same as those of DPOR$_{cs}$. Column LE/LT shows the length of the longest execution sequence in terms of execution steps (LE) and the length of the longest trace in terms of LLVM instructions (LT). Note that such a longest execution sequence and corresponding trace is the same on every algorithm. This gives an idea on the complexity and depth of the exploration which is carried out. Also, the difference between the LE and LT values on a given exploration provides an indication on the length of the atomic blocks of the benchmark (the bigger the difference the longer the atomic blocks). Columns labeled with E show the number of finished execution sequences whereas those with T show the time in seconds of the whole exploration. 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-ob}_{cs} and G^{cs-ob}_{ob} 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$^{ob}_{cs}$.

A timeout of 150 seconds is set. When reached, we write —, except for the speedups of the first input for which the timeout is reached, in such case we write $>X$ to indicate that the speedup would be X if the process finishes right in the timeout, and hence it is guaranteed to be greater than X.

		ODPOR		ODPOR^{ob}		DPOR$_{cs}$		ODPOR$^{ob}_{cs}$		Speed-up	
Benchmark	**LE/LT**	**E**	**T**	**E**	**T**	**E**	**T**	**E**	**T**	**G$^{cs-ob}_{ob}$**	**G$^{cs-ob}_{cs}$**
pipesort (N = 3)	13/442	6	0.01s	4	0.02s	2	0.01s	2	0.01s	1.0	1.0
pipesort (N = 7)	57/1746	—	—	64	0.30s	329	4.73s	13	0.23s	1.3	20.57
pipesort (N = 9)	91/2719	—	—	256	1.94s	—	—	34	1.06s	1.83	>141.51
pipesort (N = 11)	133/3906	—	—	1024	13.00s	—	—	89	4.63s	2.81	—
pi (N = 5)	11/251	120	0.09s	120	0.09s	16	0.05s	16	0.05s	1.8	1.0
pi (N = 6)	13/297	720	0.56s	720	0.60s	61	0.17s	61	0.19s	3.16	0.89
pi (N = 7)	15/343	5040	5.28s	5040	5.63s	272	0.83s	272	0.94s	5.99	0.88
pi (N = 8)	17/389	40320	121.74s	40320	123.36s	1385	5.30s	1385	6.04s	20.42	0.88
quicksort (N = 2)	23/3491	32	0.29s	32	0.33s	1	0.24s	2	0.15s	2.2	1.6
quicksort (N = 3)	27/4230	64	0.70s	64	0.95s	1	0.38s	6	0.41s	2.32	0.93
quicksort (N = 4)	31/4947	128	2.32s	128	2.01s	1	0.53s	1	0.42s	4.79	1.26
quicksort (N = 5)	35/5864	256	4.86s	256	5.46s	1	2.06s	34	2.92s	1.87	0.71
mergesort (N = 4)	15/4722	8	0.15s	8	0.15s	1	0.07s	1	0.07s	2.14	1.0
mergesort (N = 5)	19/7745	16	0.53s	16	0.56s	1	0.19s	2	0.20s	2.8	0.95
mergesort (N = 6)	31/13993	128	9.77s	128	10.21s	1	1.40s	1	1.21s	8.44	1.16
mergesort (N = 7)	31/18131	—	—	—	—	1	1.76s	1	1.77s	>84.75	0.99
fib (N = 2)	7/155	2	0.01s	2	0.01s	1	0.01s	1	0.01s	1.0	1.0
fib (N = 3)	11/276	4	0.01s	4	0.01s	1	0.01s	1	0.01s	1.0	1.0
fib (N = 4)	19/511	16	0.03s	16	0.03s	1	0.03s	2	0.02s	1.5	1.5
fib (N = 5)	35/868	128	0.27s	128	0.28s	1	0.19s	16	0.22s	1.27	0.86
boundedbuffer (N = 7)	15/401	3432	3.09s	3432	3.24s	550	1.16s	550	1.26s	2.57	0.92
boundedbuffer (N = 8)	17/458	12870	14.28s	12870	14.88s	1487	3.69s	1487	4.05s	3.67	0.91
boundedbuffer (N = 9)	19/515	—	—	—	—	4036	11.67s	4036	13.81s	>10.86	0.85
boundedbuffer (N = 10)	21/572	—	—	—	—	10981	38.23s	10981	42.84s	—	0.89

Table 1: Experimental evaluation results. First set of benchmarks.

Finally, our third set of benchmarks (Table 3) are run within the SYCO tool and include two larger programs implemented in the ABS modeling language using active objects and message-passing: MapRed, an implementation of a map-reduce model developed by a company (440 lines of code); and SDN [8], a model of a software-defined network featuring a safety policy violation (490 lines).

5.3. Analysis of the Results

We have confirmed that ODPOR$_{cs}$ performs basically equivalent to DPOR$_{cs}$ (goal G1e). The number of finished execution sequences is exactly the same for all benchmarks. The execution time is also basically the same with a very slight overhead (of at most 4%) due to the handling of the wakeup trees. An exception for this is the lastzero benchmark where ODPOR$_{cs}$ outperforms DPOR$_{cs}$ (e.g., for $N = 6$, ODPOR$_{cs}$ takes $0.24s$ vs. the 0.34 of DPOR$_{cs}$). It is already shown in [1] that even though optimality provides a very relevant and strong theoretical result, it is difficult to find examples where ODPOR produces significant improvements in practice over Source-DPOR. The lastzero benchmark is indeed the only case where this happens in [1]. Note also that the primary goal of the development of ODPOR$_{cs}$ is to serve as an intermediate and necessary step towards the development of our finally proposed algorithm which is ODPOR$^{ob}_{cs}$.

Looking at the results on our first set of benchmarks (see Table 1) we clearly observe that ODPOR$^{ob}_{cs}$ always outperforms or behaves similarly to the best of the other algorithms (goal G1). For instance, in pipesort we observe that ODPOR^{ob} and DPOR$_{cs}$ both feature exponential reductions over ODPOR and that ODPOR$^{ob}_{cs}$ is able to produce exponential reductions over both of them (goal G1c). This is due to the benefits of combining the orthogonal reductions of ODPOR^{ob} and DPOR$_{cs}$ and, also to the synergy of the combination (namely the refinement of Section 4.3) (goal G3). In quicksort, mergesort, fib and boundedbuffer we can observe that ODPOR^{ob} performs almost identically to ODPOR since there are no observers to take advantage of. We also observe exponential reductions of both DPOR$_{cs}$ and ODPOR$^{ob}_{cs}$ over both ODPOR and ODPOR^{ob} due to context-sensitive dependencies (goal G1a). We can hence observe the overhead of the observers part of the algorithms in isolation (goal G2b) which is very low (see e.g. the times of ODPOR$^{ob}_{cs}$ versus those of DPOR$_{cs}$).

In pi we have a different situation. We observe exponential reductions of both ODPOR^{ob} and DPOR$_{cs}$ w.r.t ODPOR indicating there are both observers-based independences and also context-sensitive independences. In this case, the combination of the reductions due to both kind of independences produced by ODPOR$^{ob}_{cs}$ features exponential reductions over ODPOR^{ob}

		ODPOR		ODPOR^{ob}		DPOR$_{cs}$		ODPOR$^{ob}_{cs}$		Speed-up	
Benchmark	**LE/LT**	**E**	**T**	**E**	**T**	**E**	**T**	**E**	**T**	**G$^{cs-ob}_{ob}$**	**G$^{cs-ob}_{cs}$**
lastwrite (N = 4)	82/175	24	0.01s	4	0.01s	5	0.01s	1	0.01s	1.0	1.0
lastwrite (N = 8)	150/319	40320	15.93s	8	0.01s	1385	3.54s	1	0.03s	0.33	118.0
lastwrite (N = 12)	218/463	—	—	12	0.02s	—	—	1	0.08s	0.25	>1875.00
lastwrite (N = 16)	286/607	—	—	16	0.05s	—	—	1	0.18s	0.28	—
lastwrite-dif (N = 4)	86/187	24	0.01s	4	0.01s	24	0.02s	4	0.01s	1.0	2.0
lastwrite-dif (N = 8)	158/343	40320	16.16s	8	0.01s	40320	105.98s	8	0.03s	0.33	3532.67
lastwrite-dif (N = 12)	230/499	—	—	12	0.02s	—	—	12	0.06s	0.33	>2500.00
lastwrite-dif (N = 16)	302/655	—	—	16	0.05s	—	—	16	0.13s	0.38	—
lastwrite-2phases (N = 4)	233/495	13824	8.32s	64	0.06s	125	0.51s	1	0.03s	2.0	17.0
lastwrite-2phases (N = 6)	335/711	—	—	216	0.31s	—	—	1	0.07s	4.43	>2142.86
lastwrite-2phases (N = 8)	437/927	—	—	512	1.10s	—	—	1	0.14s	7.86	—
lastwrite-2phases (N = 10)	539/1143	—	—	1000	3.22s	—	—	1	0.26s	12.38	—
floatingread (N = 4)	97/196	120	0.06s	33	0.02s	16	0.04s	1	0.07s	0.29	0.57
floatingread (N = 5)	116/233	720	0.36s	81	0.06s	61	0.18s	1	0.26s	0.23	0.69
floatingread (N = 6)	135/270	5040	3.28s	193	0.18s	272	0.90s	1	1.33s	0.14	0.68
floatingread (N = 7)	154/307	40320	29.19s	449	0.38s	1385	6.10s	1	6.86s	0.06	0.89
floatingread-dif (N = 4)	97/196	120	0.05s	33	0.02s	120	0.11s	33	0.06s	0.33	1.83
floatingread-dif (N = 5)	116/233	720	0.29s	81	0.05s	720	0.66s	81	0.17s	0.29	3.88
floatingread-dif (N = 6)	135/270	5040	2.30s	193	0.12s	5040	6.09s	193	0.92s	0.13	6.62
floatingread-dif (N = 7)	154/307	40320	21.36s	449	0.30s	—	—	449	4.88s	0.06	>30.74
floatingread-2phases (N = 3)	152/310	576	0.30s	169	0.11s	25	0.10s	1	0.05s	2.2	2.0
floatingread-2phases (N = 4)	190/384	14400	9.35s	1089	0.82s	256	0.97s	1	0.15s	5.47	6.47
floatingread-2phases (N = 5)	228/458	—	—	6561	6.12s	3721	19.56s	1	0.61s	10.03	32.07
floatingread-2phases (N = 6)	266/532	—	—	37249	42.60s	—	—	1	2.76s	15.43	>54.35
abs (N = 3)	13/198	36	0.03s	9	0.02s	4	0.02s	1	0.02s	1.0	1.0
abs (N = 4)	17/254	576	0.45s	16	0.03s	25	0.10s	1	0.05s	0.6	2.0
abs (N = 5)	21/310	14400	19.20s	25	0.04s	256	1.05s	1	0.12s	0.33	8.75
abs (N = 6)	25/366	—	—	36	0.09s	—	—	1	0.29s	0.31	>517.24
abs-dif (N = 3)	13/198	36	0.03s	9	0.01s	36	0.04s	9	0.02s	0.5	2.0
abs-dif (N = 4)	17/254	576	0.46s	16	0.02s	576	0.65s	16	0.04s	0.5	16.25
abs-dif (N = 5)	21/310	14400	19.30s	25	0.05s	14400	25.26s	25	0.08s	0.63	315.75
abs-dif (N = 6)	25/366	—	—	36	0.09s	—	—	36	0.15s	0.6	>1000.00
abs-2phases (N = 2)	17/276	16	0.02s	16	0.02s	1	0.01s	1	0.02s	1.0	0.5
abs-2phases (N = 3)	25/388	1296	1.41s	81	0.14s	16	0.11s	1	0.05s	2.8	2.2
abs-2phases (N = 4)	33/500	—	—	256	0.66s	—	—	1	0.14s	4.71	>1071.42
abs-2phases (N = 5)	41/612	—	—	625	2.44s	—	—	1	0.34s	7.18	—
arraywrite (N = 5)	89/1659	16	0.07s	16	0.07s	1	0.03s	1	0.03s	2.33	1.0
arraywrite (N = 10)	421/7969	1024	25.09s	1024	25.99s	1	0.31s	1	0.32s	81.21	0.97
arraywrite (N = 15)	869/16539	—	—	—	—	1	1.30s	1	1.45s	>103.45	0.90
arraywrite (N = 20)	1641/31319	—	—	—	—	1	5.91s	1	6.80s	—	0.87

Table 2: Experimental evaluation results. Second set of benchmarks.

	DPOR$_{cs}$		ODPOR^{ob}		ODPOR$^{ob}_{cs}$		Speed-up	
Bench.	**E**	**T**	**E**	**T**	**E**	**T**	**G$^{cs-ob}_{ob}$**	**G$^{cs-ob}_{cs}$**
MapRed	9	114	118	2961	9	185	0.6x	16.0x
SDN	22	83	58	229	16	52	1.6x	4.4x

Table 3: Experimental evaluation results. Third set of benchmarks.

but, interestingly, not over DPOR$_{cs}$, which behaves similarly to ODPOR$^{ob}_{cs}$ (goal G1). This indicates that the prunings performed by ODPOR^{ob} are in this case a subset of those produced by DPOR$_{cs}$, hence joining them does not produce an improvement over the best of both algorithms (here DPOR$_{cs}$).

In the second set of benchmarks (see Table 2) we can observe exponential reductions in the number of finished executions in

ODPOR$^{ob}_{cs}$ w.r.t all ODPOR, DPOR$_{cs}$ and ODPOR^{ob}. Note that ODPOR$^{ob}_{cs}$ is able to catch all redundant executions hence obtaining only 1 finished execution. Again, that is possible both due to the combination of ODPOR^{ob} and DPOR$_{cs}$ and, also to the new refined check of Section 4.3 (goal G3). However, such a reduction over ODPOR^{ob} does not have a direct correlation to a corresponding reduction in time in neither lastwrite nor floatingread nor abs. This indicates that the overhead of the context-sensitive part of the algorithm does not pay-off in these cases, mainly because the reductions are being done at the very end of the executions and hence few states are being pruned (goal G1d). This can indeed be observed looking at the results we get in the -dif variations of the benchmarks where the variable writes are all done with different values, hence making all context-sensitive checks to fail. We observe that in these cases ODPOR$^{ob}_{cs}$ performs similar, and even better, as it does with the original benchmarks in terms of exploration time, clearly indicating that the prunings being made are not saving exploration effort. This is because of the overhead due to the handling and inheritance of the don't-do sets, which in the case of the -dif variations is not produced since there are no don't-do sets to be handled (goal G2a).

Although it is not shown in the tables (in order not to overload them with more data) we are also measuring the number of executions which are cut due to don't-do sets (don't-do set blocked executions, DDSBs). Note that the number of DDSBs with ODPOR$^{ob}_{cs}$ would be always bounded (above) by the difference between the number finished execution sequences of ODPOR^{ob} and ODPOR$^{ob}_{cs}$ (and the same with ODPOR$_{cs}$ and ODPOR, respectively). But it could be much less than that because by blocking a partial execution, we can be avoiding the exploration of several redundant complete executions that will be explored by ODPOR^{ob} (or ODPOR). Indeed, the number of DDSBs would be a direct indication of the effectiveness of the context-sensitive checks. The more close to the above upper bound the less effective are the checks since that would mean that the total number of explored executions (including the DDSBs) would be close to the total number of executions explored by ODPOR^{ob}. E.g., in the case of pi for $N = 7$, the number of DDSBs is 750 which means that ODPOR$^{ob}_{cs}$ is exploring a total of 1022 executions (including the DDSBs) whereas ODPOR^{ob} is exploring 5040. However, in the case of lastwrite we get $3, 7, 11$ and 15 as the numbers of DDSBs for the four runs respectively, which means that ODPOR$^{ob}_{cs}$ is exploring the same number of executions as ODPOR^{ob} although most of them are cut at the end (since they are redundant). The results on the rest of the benchmarks confirm this claim: in the cases where $G^{cs-ob}_{ob} \leq 1$, the total number of explored executions of ODPOR$^{ob}_{cs}$ (including DDSBs) is close (or even equal, as in the case of lastwrite) to the number of explorations performed by ODPOR^{ob}, whereas in the cases where G^{cs-ob}_{ob} grows, such number is much lower.

The fact that ODPOR$^{ob}_{cs}$ is able to catch many more redundant executions than ODPOR^{ob}, as we have seen, does not necessarily indicate a reduction on the overall exploration time (due to the associated overheads). However, this indicates the potential of ODPOR$^{ob}_{cs}$ to produce exponential reductions. That will happen as soon as the pattern where the reduction takes place is followed by non-trivial explorations, in such case such upcoming explorations will be effectively pruned. This can be observed looking at the results obtained in the -2phases variations of the benchmarks where we can see such exponential reductions of ODPOR$^{ob}_{cs}$ w.r.t both ODPOR^{ob} and DPOR$_{cs}$ (goal G1).

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.

Finally, in order to illustrate the potential impact of the refined check of Section 3.4 in isolation, the following table shows the exploration times of ODPOR$^{ob}_{cs}$ with and without the refined check (columns labeled respectively **With 3.4** and **Without 3.4**) on the arraywrite benchmark for a series of increasing inputs (column **N**), and the corresponding speedups.

N	With 3.4	Without 3.4	Speed-up
5	0.03s	0.03s	1.00
10	0.32s	0.41s	1.29
15	1.45s	2.13s	1.51
20	6.80s	9.63s	1.42
25	18.15s	25.13s	1.39
30	53.77s	76.26	1.42

In conclusion, our experimental results show that our algorithm ODPOR$^{ob}_{cs}$ is able to effectively combine the benefits of both DPOR$_{cs}$ and ODPOR^{ob}. On the other hand, we can observe that it also inherits their associated overheads. Whereas in the case of ODPOR^{ob} the overhead is very low, our results show notable overheads in DPOR$_{cs}$ and hence ODPOR$^{ob}_{cs}$, which in some cases do not pay-off even though they do not seem to affect the overall scalability. In this sense, we believe that the integration of the optimizations proposed in [4] could help. Specifically, we have measured that nearly half of the exploration time in both DPOR$_{cs}$ and ODPOR$^{ob}_{cs}$ is spent in the computation of the alternative execution sequences for the context-sensitive checks (that are later recomputed during the normal exploration), which we believe could be avoided (at least in part) by integrating the support to avoid recomputations described in [4].

Finally, let us conclude this section by noting that DPOR$_{cs}$, ODPOR$_{cs}$, and, consequently, ODPOR$^{ob}_{cs}$ 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 dependency based on variable accesses, even if it is computed dynamically, can be rather imprecise. They are 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.

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} [4] proposes a context-sensitive check in the current state to detect more accurately independence among processes, ODPOR^{ob} [9] 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}: (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 [11] was the first work to focus on the *reads-from* relation. 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 [9], which is based on observing interference of operations, not just individual writes. The equivalence relation used by Data-Centric is proven in [11] 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 [13]. Note that this is more unlikely to happen in programs with bigger atomic blocks like our first and third set of benchmarks (where we have indeed seen that the reads-from relation would not be exploited). A drawback of Data-Centric DPOR is that it is not always optimal. Recently, there have been other works proposing optimal DPOR algorithms based on the reads-from relation, either for sequential consistency [2], for the release-acquire semantics [3] or for the C/C++ concurrency. Also, [22] proposes an optimal algorithm for the reads-from relation which is parametric in the choice of the memory model under a set of basic assumptions. This has been recently further extended to handle locks efficiently [21] and, for the first time, to ensure linear memory consumption per equivalence class while at the same time allowing parallelization [20]. It remains to be studied whether our context-sensitive dependency can be successfully integrated within these approaches based on the reads-from relation since their underlying algorithms are fundamentally different as the classical DPOR algorithm and its variants (including [9]).

Another approach is to generate *independence constraints* (ICs), which ensure the independence of each pair of processes in the program. The work in [32, 18] generated for the first time ICs for processes with a single instruction following some predefined patterns. Constrained DPOR [7] proposed to generate ICs in a pre-phase, using a SMT solver. The generated ICs are then used within DPOR in a similar way to our context-sensitive checks. 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 [19, 16]). 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 Section 3.1, and the enhancements in Section 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 [26], 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 [29]. This work has been recently extended to the context of symbolic execution [30], where cut-off events are used to detect program states already visited in order to handle non-terminating executions.

Finally, there is a large body of related work in the context of dynamic runtime verification that include similar ideas for race and deadlock detection [28, 10, 27]. For instance, [28] proposes a complete and sound algorithm that predicts all data races of a given trace but without considering observability, whereas [10] improves it by taking into account observability.

Acknowledgments This work was funded partially by the Spanish MICINN/FEDER, UE projects PID2021-122830OB-C41, the CM project S2018/TCS-4314 and the Australian ARC project DP180100151.

[1] Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F., 2014. Optimal Dynamic Partial Order Reduction, in: Jagannathan, S., Sewell, P. (Eds.), The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014, ACM. pp. 373–384.

[2] Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K., 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. 3, 150:1–150:29. URL: https://doi.org/10.1145/3360576, doi:10.1145/3360576.

[3] Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P., 2018. Optimal stateless model checking under the release-acquire semantics. PACMPL 2, 135:1–135:29. URL: https://doi.org/10.1145/3276505, doi:10.1145/3276505.

[4] Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P., 2017. Context Sensitive Dynamic Partial Order Reduction, in: Kuncak, V., Majumdar, R. (Eds.), 29th International Conference on Computer Aided Verification (CAV 2017), Springer. pp. 526–543. URL: https://doi.org/10.1007/978-3-319-63387-9_26, doi:10.1007/978-3-319-63387-9_26.

[5] Albert, E., de la Banda, M.G., Gómez-Zamalloa, M., Isabel, M., Stuckey, P.J., 2019. Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers, in: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2019 (ISSTA'19), pp. 352–362. doi:10.1145/3293882.3330565.

[6] Albert, E., Gómez-Zamalloa, M., Isabel, M., 2016. Syco: A Systematic Testing Tool for Concurrent Objects, in: Zaks, A., Hermenegildo, M.V. (Eds.), Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016, ACM. pp. 269–270. URL: http://dx.doi.org/10.1145/2892208.2892236, doi:10.1145/2892208.2892236.

[7] Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A., 2018a. 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, Springer. pp. 392–410. URL: https://doi.org/10.1007/978-3-319-96142-2_24, doi:10.1007/978-3-319-96142-2_24.

[8] Albert, E., Gómez-Zamalloa, M., Rubio, A., Sammartino, M., Silva, A., 2018b. 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, pp. 550–567. URL: https://doi.org/10.1007/978-3-319-95582-7_33, doi:10.1007/978-3-319-95582-7_33.

[9] Aronis, S., Jonsson, B., Lång, M., Sagonas, K., 2018. 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, pp. 229–248. URL: https://doi.org/10.1007/978-3-319-89963-3_14, doi:10.1007/978-3-319-89963-3_14.

[10] Cai, Y., Yun, H., Wang, J., Qiao, L., Palsberg, J., 2021. Sound and efficient concurrency bug prediction, in: Spinellis, D., Gousios, G., Chechik, M., Penta, M.D. (Eds.), ESEC/FSE'21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, ACM. pp. 255–267. URL: https://doi.org/10.1145/3468264.3468549, doi:10.1145/3468264.3468549.

[11] Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K., 2018. Data-centric dynamic partial order reduction. PACMPL 2, 31:1–31:30. URL: http://doi.acm.org/10.1145/3158119, doi:10.1145/3158119.

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

[13] Flanagan, C., Godefroid, P., 2005. Dynamic Partial-Order Reduction for Model Checking Software, in: Palsberg, J., Abadi, M. (Eds.), Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, ACM. pp. 110–121.

[14] Godefroid, P., 1991. Using Partial Orders to Improve Automatic Verification Methods, in: Proc. of CAV'91, Springer. pp. 176–185.

[15] Godefroid, P., 1996. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. volume 1032 of *LNCS*. Springer. URL: http://dx.doi.org/10.1007/3-540-60761-7, doi:10.1007/3-540-60761-7.

[16] Godefroid, P., Pirottin, D., 1993. Refining dependencies improves partial-order verification methods (extended abstract), in: CAV, pp. 438–449.

[17] Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M., 2012. ABS: A Core Language for Abstract Behavioral Specification, in: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (Eds.), Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers, Springer. pp. 142–164.

[18] Kahlon, V., Wang, C., Gupta, A., 2009. Monotonic partial order reduction: An optimal symbolic partial order reduction technique, in: CAV, pp. 398–413.

[19] Katz, S., Peled, D.A., 1992. Defining conditional independence using collapses. TCS 101, 337–359.

[20] Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V., 2022. Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. 6, 1–28. URL: https://doi.org/10.1145/3498711, doi:10.1145/3498711.

[21] Kokologiannakis, M., Raad, A., Vafeiadis, V., 2019a. Effective lock handling in stateless model checking. Proc. ACM Program. Lang. 3, 173:1–173:26. URL: https://doi.org/10.1145/3360599, doi:10.1145/3360599.

[22] Kokologiannakis, M., Raad, A., Vafeiadis, V., 2019b. Model checking for weakly consistent libraries, in: McKinley, K.S., Fisher, K. (Eds.), Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, ACM. pp. 96–110. URL: https://doi.org/10.1145/3314221.3314609, doi:10.1145/3314221.3314609.

[23] Lang, M., Leonardsson, C., . Nidhugg. URL: https://github.com/nidhugg/nidhugg.

[24] Mazurkiewicz, A.W., 1986. 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, pp. 279–324. URL: https://doi.org/10.1007/3-540-17906-2_30, doi:10.1007/3-540-17906-2_30.

[25] Musuvathi, M., Qadeer, S., 2007. Iterative context bounding for systematic testing of multithreaded programs, in: Ferrante, J., McKinley, K.S. (Eds.), Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, ACM. pp. 446–455. URL: https://doi.org/10.1145/1250734.1250785, doi:10.1145/1250734.1250785.

[26] Nguyen, H.T.T., Rodríguez, C., Sousa, M., Coti, C., Petrucci, L., 2018. Quasi-optimal partial order reduction, in: Chockler, H., Weissenbacher, G. (Eds.), 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, Springer. pp. 354–371. URL: https://doi.org/10.1007/978-3-319-96142-2_22, doi:10.1007/978-3-319-96142-2_22.

[27] Park, J., Choi, B., Jang, S.Y., 2020. Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments. Int. J. Parallel Program. 48, 1032–1060. URL: https://doi.org/10.1007/s10766-020-00661-3, doi:10.1007/s10766-020-00661-3.

[28] Pavlogiannis, A., 2020. Fast, sound, and effectively complete dynamic race prediction. Proc. ACM Program. Lang. 4, 17:1–17:29. URL: https://doi.org/10.1145/3371085, doi:10.1145/3371085.

[29] Rodríguez, C., Sousa, M., Sharma, S., Kroening, D., 2015. Unfolding-based partial order reduction, in: 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015, pp. 456–469. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2015.456, doi:10.4230/LIPIcs.CONCUR.2015.456.

[30] Schemmel, D., Büning, J., Rodríguez, C., Laprell, D., Wehrle, K., 2020. Symbolic partial-order execution for testing multi-threaded programs, in: Lahiri, S.K., Wang, C. (Eds.), Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I, Springer. pp. 376–400. URL: https://doi.org/10.1007/978-3-030-53288-8_18, doi:10.1007/978-3-030-53288-8_18.

[31] Valmari, A., 1989. Stubborn Sets for Reduced State Space Generation, in: Rozenberg, G. (Ed.), Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings], Springer. pp. 491–515. URL: http://dx.doi.org/10.1007/3-540-53863-1_36, doi:10.1007/3-540-53863-1_36.

[32] Wang, C., Yang, Z., Kahlon, V., Gupta, A., 2008. Peephole Partial Order Reduction, in: Ramakrishnan, C.R., Rehof, J. (Eds.), 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, Springer. pp. 382–396. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_29, doi:10.1007/978-3-540-78800-3_29.

AppendixA. Proofs of Correctness and Optimality of ODPOR$_{cs}$

Let us claim some auxiliary lemmas and definitions before proving the main theorems of soundness and optimality. First, we claim a lemma needed to prove the correctness of the new context-sensitive check.

Let the sequence $E' = E_0.\hat{e}.w.\hat{e'}.w'$ is explored by Algorithm 1 and the race $e \precsim_{E'} e'$. According to the algorithm's definitions: $E_0 = pre(E', e)$ and $v = notdep(e, e', E')$.

Lemma 1. *If the following check of Algorithm 1 succeeds*

$$s_{[pre^+(E',e')]} \overset{\mathcal{V}^{e,e'}_{E'}}{=\!=} s_{[E_0.(v.\hat{e})_{\leq^{e'}_{E'}}]}, \text{ then}$$
$$s_{[pre^+(E',e')]} = s_{[E_0.(v.suc(e,E'))_{\leq^{e'}_{E'}}]}.$$

Proof. First, we know that $E' = E_0.\hat{e}.w.\hat{e'}.w'$, then $pre^+(E', e') = E_0.\hat{e}.w.\hat{e'}$, and sequence $\hat{e}.w$, can be divided in two sets $notdep(e, e', E')$ and $suc(e, E')$. Let us introduce a simple notation $suc^-(e, E')$ to denote the sequence $suc(e, E')$ without the first process $\hat{e}$. Then, $E_0.\hat{e}.w.\hat{e'} \simeq E_0.\hat{e}.notdep(e, e', E').suc^-(e, E').\hat{e'} \simeq E_0.notdep(e, e', E).\hat{e}.\hat{e'}.suc(e, E')$ (taking into account that after $E_0.notdep(e, e', E).\hat{e}$, we have $\hat{e'} \diamond suc^-(e, E')$). Because of the hypothesis, we know that $s_{[E_0.notdep(e,e',E).\hat{e}.\hat{e'}.suc^-(e,E')]} = s_{[pre^+(E',e')]} \overset{\mathcal{V}^{e,e'}_{E'}}{=\!=} s_{[E_0.(v.\hat{e})_{\leq^{e'}_{E'}}]} = s_{[E_0.notdep(e,e',E).\hat{e'}.\hat{e}]}$.

Let us suppose there exists a variable y such that

$$s_{[pre^+(E',e')]}(y) \neq s_{[E_0.(v.suc(e,E'))_{\leq^{e'}_{E'}}]}(y).$$

Then, $s_{[E_0.notdep(e,e',E).\hat{e}.\hat{e'}.suc^-(e,E')]}(y) \neq s_{[E_0.notdep(e,e',E).\hat{e'}.suc(e,E')]}(y)$. Two cases can be distinguished:

- If $y \in \mathcal{V}^{e,e'}_{E'}$, then because of the hypothesis we know that the value of y is not modified by $suc^-(e, E')$ at state $s_{[E_0.notdep(e,e',E).\hat{e}.\hat{e'}]}$. But it is modified by $suc^-(e, E')$ at state $s_{[E_0.notdep(e,e',E).\hat{e'}.\hat{e}]}$, then the write instruction must depend on a value of another variable x in a condition. This variable must be different in both states after the execution of $\hat{e}.\hat{e'}$ and $\hat{e'}.\hat{e}$, respectively. However, this is not possible since x must be modified by e or e' and then, $x \in \mathcal{V}^{e,e'}_{E'}$, and, thus, by the initial hypothesis, x should have the same value, producing the same result in the condition of variable y. We get a contradiction.
- If $y \notin \mathcal{V}^{e,e'}_{E'}$ we get a similar contradiction. Since they write different values over variable y, and both executions share the same prefix $E_0.notdep(e, e', E)$, there must be a variable in $\mathcal{V}^{e,e'}_{E'}$ with a different value. Otherwise, the value of y must be the same in both executions.

Consequently, we have that for every program variable y

$$s_{[pre^+(E',e')]}(y) = s_{[E_0.(v.suc(e,E'))_{\leq^{e'}_{E'}}]}(y).$$

□

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.

Lemma 2. *If the following check of Algorithm 1 succeeds* $s_{[pre^+(E',e')]} = s_{[E_0.(v.suc(e,E'))_{\leq^{e'}_{E'}}]}$, *then 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* $T' =\!\!\rightarrow_{E'}$ *and leads to an identical final state.*

Proof. Let $E = E_0.v.\hat{e}.u'.w'$ be a complete execution sequence. Let us notice here that

1. $E_0.v \simeq E_0.v_{\leq^{e'}_{E'}}.v_{>^{e'}_{E'}}$,
2. $E_0.v.\hat{e}.u' \simeq E_0.v.suc(e, E')_{<^{e'}_{E'}}.suc(e, E')_{>^{e'}_{E'}}.w''$, where w'' are possibly some events added that do not depend on any event in $dom_{[E_0.v]}(suc(e, E'))$ and
3. $E_0.v_{\leq^{e'}_{E'}} \models v_{>^{e'}_{E'}} \diamond suc(e, E')_{<^{e'}_{E'}}$.

Now,

$$\begin{aligned} E &\simeq^{1,2} E_0.v_{\leq^{e'}_{E'}}.v_{>^{e'}_{E'}}.suc(e, E')_{<^{e'}_{E'}}.suc(e, E')_{>^{e'}_{E'}}.w''.w' \\ &\simeq^{3} E_0.v_{\leq^{e'}_{E'}}.suc(e, E')_{<^{e'}_{E'}}.v_{>^{e'}_{E'}}.suc(e, E')_{>^{e'}_{E'}}.w''.w' \\ &\simeq E_0.(v.suc(e, E'))_{\leq^{e'}_{E'}}.v_{>^{e'}_{E'}}.suc(e, E')_{>^{e'}_{E'}}.w''.w' \end{aligned}$$

Since $s_{[pre^+(E',e')]} = s_{[E_0.(v.suc(e,E'))_{\leq^{e'}_{E'}}]}$, we have that $s_{[E'']} = s_{[E]}$ where $E'' = pre^+(E', e').w$ and $w = v_{>^{e'}_{E'}}.suc(e, E')_{>^{e'}_{E'}}.w''.w'$. Note that in execution sequence E we have $e' \rightarrow_E e$, whereas in E', we have $e \rightarrow_{E'} e'$. □

After claiming the previous lemmas, let us prove now the soundness of ODPOR$_{cs}$.

Proof of Theorem 1 (Soundness of ODPOR$_{cs}$). Consider an execution of Explore$(\epsilon, \langle\{\epsilon\}, \emptyset\rangle, \emptyset)$ without the additions for context-sensitivity, and assuming we always choose an enabled process (if it is possible, from the wakeup tree) that would not be blocked by the *don't-do* set in the extended algorithm, wherever possible. This is exactly the ODPOR algorithm of [1] which is guaranteed to explore a complete execution sequence that implements each T [24].

Suppose that some Mazurkiewicz trace T is omitted by ODPOR$_{cs}$, C is the complete execution sequence that implements T ($T =\!\!\rightarrow_C$) and is explored by the original ODPOR algorithm. This sequence must be cut by our algorithm. Thus, it must be of the form $C = E'.v'.\hat{e}.y$, where our algorithm added $v.\hat{e}$ to $dnd(E')$ after succeeding in the check

$$s_{[pre^+(C'',e')]} \overset{\mathcal{V}^{e,e'}_{C''}}{=\!=} s_{[E'.(v.\hat{e})_{\leq^{e'}_{C''}}]}$$

exploring a sequence C'', and $v'.\hat{e}$ is $v.\hat{e}$ possibly having added some events that do not depend on any event in $dom_{[E']}(v.\hat{e})$, as otherwise the *don't-do* entry would have been removed. Hence, there exists a complete execution sequence $E'.v.\hat{e}.w.y$ with the same happens-before relation as C, obtained by moving after $v.\hat{e}$ events that are independent of $v.\hat{e}$ (those with processes in w). By Lemma 1, we can apply Lemma 2 and we get that there is a different trace T' which leads to the same state as C. Since ODPOR explores a complete execution sequence for each Mazurkiewicz trace, it must include a complete execution sequence C' that implements T'. Note that C' has the same happens-before relation as $C'' = pre^+(C'', e').w.y$.

We now show that C' must have been explored before C, i.e. it appears to the left of C in the ODPOR exploration tree. Sequence $pre^+(C'', e')$ clearly appears to the left of C in the ODPOR tree, or it could not be used to add the *don't-do* entry that blocked C. Let us suppose instead that C' appears to the right of C in the exploration tree. Let E'' be the largest common prefix of C and C'. Now $C = E''.q.w'$ for some q. Since C' appears to the right of C, then q will be in the *don't-do* sets for (the remainder of) sequence C' unless it is removed by some dependent event. Let $e' = next_{[E'']}(q)$, that is, the next event that q executes after E''.

Suppose that $E''.q \leq E'$, then the first change is above E'. The happens-before relation for C' must then have some event e'' (after $E''.q$) such that $e'' \rightarrow_{C'} e'$, but this cannot be the case since $\rightarrow_{C'} = \rightarrow_{pre^+(C'',e').w.y}$ where this does not occur.

Suppose that $E' \leq E''$ and, thus, the first change is at, or after, the point where $pre^+(C'', e'')$ and $E'.v.\hat{e}$ differ. Clearly C' must appear to the right of $E'.\hat{(e)}$ in the tree (otherwise it would be to the left of C). Hence, $\hat{e}$ is in the *don't-do* set (for the remainder) of C' after E' until removed by dependent events. Suppose event e'' removes $\hat{e}$. Then, we have that $e'' \rightarrow_{C'} e$. This is a contradiction since this does not occur in $pre^+(C'', e').w.y$.

Hence, C' must appear to the left of C in the ODPOR tree. If C' exists in the tree visited by ODPOR$_{cs}$ we are done, since we have found an equivalent complete sequence. Otherwise, we can apply the same construction to discover an equivalent complete sequence that occurs to the left in the original tree. The procedure must terminate since, eventually, we reach the left-most branch, which cannot be removed by the context-sensitive additions to the algorithm. □

Now, let us claim a lemma needed for the optimality theorem.

Lemma 3. *Let $E = E'.\hat{e}.w.\hat{e'}.w$ be an execution such that $e \precsim_E e'$. Let v' and v be $dep(e', E).\hat{e'}.I_{fut}(E', v, E)$ and $notdep^*(e, E).\hat{e'}$, respectively.*

$$v \notin redundant(E', done) \Leftrightarrow v' \notin redundant(E', done)$$

Finally, let us sketch the proof of Theorem 2.

Proof sketch of Theorem 2 (Optimality of ODPOR$_{cs}$). This theorem follows from: (1) the optimality of ODPOR, (2) the fact that the algorithm can cut executions that are not happens-before equivalent but whose result is the same as that of other complete executions already explored (Lemma 2), and (3) the fact that races found in cut executions have enough information to detect redundancies correctly (Lemma 3). By Lemma 3, we detect the same redundancies than ODPOR, whenever ODPOR$_{cs}$ explores a don't-do set blocked execution E. Then, when detecting a race in E, the wakeup tree is updated if and only if it would also be updated in case E would have been completely explored. Consequently, if both algorithms detect the same redundancies, and ODPOR does not initiates any of them, ODPOR$_{cs}$ does not either. The happens-before relation used in ODPOR requires (by the soundness of ODPOR) to explore an equivalent execution to E. Let us notice that E does not have the same happens-before relation than any execution explored by ODPOR$_{cs}$. □

AppendixB. Proofs of Correctness and Optimality of ODPOR$^{ob}_{cs}$

Let us prove two lemmas that are the key for the soundness and optimality of Algorithm 2.

Lemma 4. *If Algorithm 2 checks that $s_{[pre^+(E',o')]} =^{e,e'}_{o'} s_{[E_0.v_{\leq^o_{E'}}.(\hat{o}_s \backslash \hat{o})]}$ $\forall o' \in o_s$, then 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 $T' = \rightarrow_{E'}$ and leads to an identical final state modulo observability.*

Proof. Let $E = E_0.v.(\hat{o}_s \backslash \hat{o}).w'$ be a complete execution sequence. Here, let us denote $notobs(e, e', E)$ as $notobs$ and notice that

$$\begin{aligned} E &= E_0.v.(\hat{o}_s \backslash \hat{o}).w' \\ &\sim E_0.ance(e, e', E).\hat{e'}.\hat{e}.notobs.\hat{o}.(\hat{o}_s \backslash \hat{o}).w' \\ &\sim E_0.ance(e, e', E).\hat{e'}.\hat{e}.notobs_{<^o_E}.notobs_{>^o_E}.\hat{o}.(\hat{o}_s \backslash \hat{o}).w' \\ &\sim E_0.ance(e, e', E).\hat{e'}.\hat{e}.notobs_{<^o_E}.\hat{o}.notobs_{>^o_E}.(\hat{o}_s \backslash \hat{o}).w' \\ &\sim E_0.v_{\leq^o_E}.(\hat{o}_s \backslash \hat{o}).v_{>^o_E}.w' = E_0.v_{\leq^o_E}.(\hat{o}_s \backslash \hat{o}).w \end{aligned}$$

Since $s_{[pre^+(E',o')]} =^{e,e'}_{o'} s_{[E_0.v_{\leq^o_E}.(\hat{o}_s \backslash \hat{o})]} \forall o' \in o_s$, we have that $s_{[E]}$ is equivalent to $s_{[E']}$ modulo observability where $E' = pre^+(E', o).w$. Note that in E we have $e' \rightarrow_E e$, whereas in E' we instead have $e \rightarrow_{E'} e'$. □

Lemma 5 (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.*

Proof. Let us distinguish two cases depending on $E' \models_{q.p.u.v} p \diamond q$:

- If $E' \models_{q.p.u.v} p \diamond q$ holds because $E' \models p \diamond q$, then they are independent, hence $E'.p.q \sim E'.q.p \Rightarrow s_{[E'.p.q]} = s_{[E'.q.p]} \Rightarrow s_{[E]} = s_{[E'']}$.
- Otherwise, $E' \models_{q.p.u.v} p \diamond q$ holds because it takes advantage of observability, (i.e., $\exists \hat{w} \in u.v$ such that the set of variables written by both p and q is overwritten by w and $\forall \hat{r} \in u.v$ that observes any of these variables, $w <_{E'.q.p.u.v} r$). Then, $u.v$ is of the form $u_1.\hat{w}.u_2.\hat{r}.v_3$ for all $\hat{r} \in u.v$, and, although $s_{[E'.p.q]}$ may be different to $s_{[E'.q.p]}$, we know that $s_{[E'.p.q.u_1.\hat{w}]} = s_{[E'.q.p.u_1.\hat{w}]}$ and hence $s_{[E]} = s_{[E'']}$ modulo observability.

□

The proofs of soundness and optimality of ODPOR$^{ob}_{cs}$ are analogous to those of the soundness and optimality theorems of ODPOR$_{cs}$ relying on Lemmas 4 and 5.