Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers

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

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

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

Miguel Isabel  
Complutense University of Madrid  
Spain  
miguelis@ucm.es

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

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

CCS CONCEPTS
• Software and its engineering → Software verification and validation.

KEYWORDS
Testing, Software Verification, Model-Checking, Partial-Order Reduction.

1 INTRODUCTION
Partial Order Reduction (POR) considers two execution sequences equivalent if one can be obtained from the other by swapping adjacent, independent execution steps. Each such equivalence class is called a Mazurkiewicz [16] trace, and POR guarantees that exploring one sequence per equivalence class is sufficient to cover all. Early POR algorithms [9, 11, 19] relied on static approximations of independence. The Dynamic-POR (DPOR) algorithm [10] was a breakthrough because it uses the information witnessed during the actual execution of the sequence to decide dynamically what to explore. Thus, it often explores less sequences than approaches based on static approximations. As a result, DPOR is considered one of the most scalable techniques for software verification.

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

Any DPOR algorithm can thus improve its efficiency by using a more accurate independence notion [12]. Two recent approaches – DPOR\(_{cs}\) (Context-Sensitive DPOR) [3] and ODOR\(_{ob}\) (Optimal-DPOR with Observers) [7] – have achieved this by integrating orthogonal notions of conditional independence into DPOR:

- DPOR\(_{cs}\): introduced the notion of context-sensitive independence, which only requires execution steps \(p\) and \(t\) be independent in the state \(S\) where they appear. This is determined by executing sequences \(p \cdot t\) and \(t \cdot p\) in \(S\), and checking if the two states reached are equal. Consider, for example, the three concurrent processes \(p, q\) and \(r\) below, and the execution tree in Fig. 2(a), which will be explained throughout the paper.
We have implemented ODPOR with benchmarks from [3], [6] and [7]. Our experimental results give to different values to variable \( x \) in Fig. 2(a). DPOR pruning. Let us consider the leftmost branch of the execution tree not include the extension of \( \text{ODPOR} \) as defined over Source-DPOR [1]. Thus, it did not consider the notion of observability, where dependencies between execution steps \( p \) and \( t \) are conditional to the existence of future steps, called observers, which read the values modified by \( p \) and \( t \). Consider again the three concurrent processes \( p \), \( q \) and \( r \) above, and the execution tree in Fig. 2(a). Assume \( r \) is scheduled first reaching state 9, where \( x = 0 \) and the assertion holds. ODPOR\(_{cs} \) considers \( q \) and \( p \) independent since, while their interleaved execution leads to different final states, variable \( x \) is not observed later.

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

1. ODPOR\(_{cs} \) was formulated over Source-DPOR [1]. Thus, it did not include the extension of wakeup trees used by ODPOR to ensure optimality, and later used to handle observers. Our first contribution is the formulation of ODPOR\(_{cs} \) over DPOR, which we name Optimal Context-Sensitive DPOR (ODPOR\(_{cs} \)).

2. Our second contribution is to integrate observability into ODPOR\(_{cs} \), obtaining ODPOR\(_{ob} \). For this, we modify context-sensitive independence to be modulo observability, which only requires equivalence for variables affected by future observers.

We have implemented ODPOR\(_{ob} \) and experimentally evaluated it with benchmarks from [3], [6] and [7]. Our experimental results show ODPOR\(_{ob} \) can explore exponentially less sequences than either DPOR\(_{cs} \) or ODPOR\(_{ob} \).

## 2 PRELIMINARIES

### 2.1 Basics of DPOR and ODPOR

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

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

The happens-before relation is also used for defining the notion of race. Event \( e \) is said to be in race with event \( e' \) in execution \( E \), written \( e \preceq_E e' \), if the events belong to different processes, \( e \) happens-before \( e' \) in \( E \) (\( e \rightarrow_E e' \)), and the two events are "concurrent" (\( E \succeq e' \) s.t. \( E' \approx E \) and the two events are adjacent in \( E' \)). We write \( e \simeq_E e' \) 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. \( E' \approx E \) and \( e \) appears immediately before \( e' \), \( e' \) is not blocked).

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

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

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

### 2.2 ODPOR with Observers

The notion of dependency we use in this paper extends the traditional one by using the concept of observer introduced in [7].

**Definition 2.1 (observers(e, e', E)[7]).** Given an execution sequence \( E \), for all events \( e, e' \in \text{dom}(E) \) where \( e \preceq_E e' \), there exists a set \( O = \text{observers}(e, e', E) \subseteq \text{dom}(E) \) such that:

1. For all \( o \in O \), it holds that \( e \rightarrow_o o, o \notin \text{dom}(E) \).
2. For all \( o, o' \in O \), it holds that \( o \rightarrow_E o' \).
3. If \( E' \approx E \), then \( \text{observers}(e, e', E') = O \).
4. For every prefix \( E' \) of \( E \) such that \( e, e' \in \text{dom}(E') \):
   - If \( O \) is empty, then \( e \rightarrow_{E'} e' \).
   - If \( O \) is nonempty, then \( e \rightarrow_{E'} e' \) iff \( \text{dom}(E') \cap O \neq \emptyset \).
5. If \( e \simeq_E e' \), for all sequences \( w \) s.t. \( E \cdot w \) is a sequence, and all events \( e'' \in \text{dom}(E) \):
1. procedure RaceDetection(E)
2. for all e, e' \in dom(E) such that e \geq_E e' do
3. let E' = pre(E, e);
4. if observers(e, e', E) \not= \emptyset then
5. let o = max_{E}(observers(e, e', E));
6. let v = notdep'(e, e', E, o \cdot \hat{e}.\hat{\epsilon}.(notobs'(e, e', E)\hat{e} \hat{\epsilon});
7. else
8. let v = notdep'(e, e', E); \\
9. if v \notin redundant'(E, done) then
10. wut(E') := insert(E')((v, wut(E'));

Figure 1: RaceDetection of ODPOReb [7]

- If e \rightarrow_E e'', then e \notin_E, E, e''.
- If e' \rightarrow_E e', then e'' \notin_E, E, e'.

(6) For all e'' \in dom(E) such that e' \rightarrow_E E, it holds that O \cap observers(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{\delta} e'' or e' \rightarrow_{E''}, \hat{\delta} e'.

Intuitively, in the usual particular case, 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.

Thus, the happens-before relation used in this paper is the one defined in [7], based on the notion of observability discussed previously (except from Sec. 3, which uses the relation in [2]). Intuitively, two processes p and q are dependent modulo observability in execution E if either enables the other (i.e., executing E, p introduces q, or vice versa), or (ii) \|E, p, q\| \not= \|E, q, p\|, where E' \not< E, and there exists in E at least one observer reading the variable written by both of them.

The code in black of procedure Explore of Algorithm 1 (lines 11-33, excluding underlined blue parts) and the code of procedure RaceDetection of Fig. 1, corresponds to the ODPOReb algorithm [7], which extends the original ODPO [2] with the notion of observers, and is our starting point. ODPOReb carries out a depth-first exploration of the execution tree from execution sequence E (initially empty) using DPOR. Essentially, it dynamically finds reversible races and is able to backtrack at the appropriate scheduling points to reverse them. For this purpose, it keeps two sets at every prefix E' of E: the usual wake up tree wut(E'), with the execution sequences that must be explored from E', and the set done(E') of processes that have already been explored from E'.

ODPOReb starts by selecting (line 24) the leftmost process p in the wake up tree, according to its order <, that is enabled by state \eta_E (due to line 14). If there is such a process, it sets wut(E') as the subtree of wut(E) with root p (line 28), and recursively explores every sequence in wut(T') from E.p (line 31). Note that wut(E) might grow as this recursion progresses, due to later executions of line 10. After the recursion finishes, it adds p to done(E), removes from wut(E) all sequences that start from p, and iterates selecting a new p. Once a complete sequence E has been explored (E is said to be complete if enabled(\eta_E) = \emptyset), the algorithm performs the race detection phase (line 14). This starts by finding all pairs of events e and e' in dom(E) such that e \geq_E e'. For each such pair, it sets E' to pre(E, e) and checks if the race between e and e' is observed (line 4).

If the race is not observed, v is set to notdep'(e, e', E, \hat{e}'; v := \emptyset, I_{\eta_E}(E', E), E); (line 8), where notdep'(e, e', E) is the subsequence of processes e'' of E such that e'' hold e <_E e'' and e \not<_E e''. If it is observed, the race must be reversed and observed by the same observers. Thus, the last (max_{E}) observer o executed in E is selected (line 5) and used to compute v (line 6), where notobs'(e, e', E) denotes the subsequence of E containing any process e'' such that e \rightarrow_{E''} e', but e'' does not observe the race e \not<_E e' and o \not<_E e'' for any observer o of the race. There is a small change in line 5 with respect to [7]; we select o as the last (rather than an arbitrary) observer from observers(e, e', E). The reason for this will be clear in Sec. 4.1. Finally, if v is not redundant for E' (line 9), it is inserted into wut(E') (line 10). To detect if v is redundant from E', ODPOReb cannot use sleep sets because they are not sufficiently precise, in the presence of observers, for avoiding redundant explorations without missing non-redundant ones [7]. Instead, ODPOReb uses the set done: v 

\[1\]The mark \(\hat{\cdot}\) in functions notdep' and notobs' indicates they will be redefined later. Function notdep'(e, e', E) does not use parameter e; it will be used once redefined.
Thus, ODPOR is challenging due to two main problems: observers (as done in Sec. 4). Reformulating DPOR to as ODPOR even though they are dependent according to the happens-before relation in [2] with the usual syntactic approximation, since q writes global variable x and r reads it. Hence, it adds r.q to the don’t-do set of state 1. Once r is explored, q is not executed because it is in the don’t-do set of state 4, which prevents the full exploration of r.p.q.

As mentioned before, our first contribution is the reformulation of DPOR as an extension of ODPOR, rather than of Source-DPOR. This yields an optimal DPOR algorithm (see below), referred to as ODPOR, which makes it easier to integrate the notion of observers (as done in Sec. 4). Reformulating DPOR in terms of ODPOR is challenging due to two main problems:

- Problem I: While Source-Performs race detection at every state, ODPOR must delay race detections until the sequence being explored is complete.
- Problem II: As shown in [3], the effectiveness of DPOR is highly dependent on exploring don’t-do sequences as soon as possible. Indeed, DPOR 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 algorithm corresponds to the code of Algorithm 1. It is discussed in detail in Secs. 3.1 and 3.2, which explain how problems I and II, respectively, have been overcome. Finally, Sec. 3.3 discusses its correctness and optimality.

### 3.1 Overcoming Problem I

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

The new context-sensitive check corresponds to the underlined blue code in line 38 of Algorithm 1 (for now, we use the black code for v in line 37, it will be redefined in Sec. 3.2). Recall that the black code of Algorithm 1 is common to both ODPOR and ODPOR* and was explained in Sec. 2.2. Intuitively, given a reversible race event e ≤ e′ for events e and e′, the check succeeds if the state right after the race, s(pre(E, e′)], is the same as that obtained when the race is reversed, s(E′.v.succ(E, E)], which succs (E, E) is the subsequence w of E that starts with e and contains all e″ s.t. e → E e″, and w.e″ is the first execution is explored.

Consider the example of Fig. 2(a). When DPOR reaches state 3 (execution sequence p.q.r), it realizes q and r can be regarded as independent in context p, as s[p.q.r] ≠ s[p.r.q] even though they are dependent according to the happens-before relation in [2] with the usual syntactic approximation, since q writes global variable x and r reads it. Hence, it adds r.q to the don’t-do set of state 1. Once r is explored, q is not executed because it is in the don’t-do set of state 4, which prevents the full exploration of r.p.q.

Example 2.2. Consider again the processes p, q and r in Fig. 2(a). Since they all have a single event, by abuse of notation, we will refer to events by their process name. The algorithm starts at state 0 in Fig. 2(a), with both E and W∪T empty. The execution first chooses p, and explores sequence p with an empty done set to state 1. The execution then chooses q and explores sequence p.q with an empty done set to state 2. Since now only r can be chosen, the execution explores sequence p.q.r to state 3. Now, the race detection phase detects an observed race for p and q, as they both write variable x and are observed by r (line 4). It then creates sequence q.p.r in line 6, which will later lead to sequence q.p.w and will thus make r observe the value written by p. The created sequence is added to w(q(e), e) of state 0 in line 10. A race between q and r is also detected and r is added to w(q(e), e). Execution then backtracks to state 1, adding q to done(p) on the way. Next, it chooses r and continues, exploring the first five executions in Fig. 2(a). Once the fifth one is completed, it checks if p is in an observed race with q. Since it is not, as there is no observer after them, the sixth execution is not even started. Thus, ODPOR explores one sequence less than ODPOR.

### 3.2 Overcoming Problem II

3.1 Overcoming Problem I

3.2 Overcoming Problem II

3.3 Correctness and optimality

3.4 Conclusions

3.5 Related work

3.6 Future work

3.7 Acknowledgments

3.8 References

355
reversible race between $p$ and $q$, but this time $s_{|p,q|} \neq s_{|q,p|}$ and, thus, nothing is added to $\text{dnd}(\epsilon)$. After backtracking to state 1 with $r$, sequence $r.q$ is inherited down to state 4 as $q$ (line 30). Hence, this exploration is stopped at state 4 and race-detection is explicitly invoked (lines 25 and 26). For the reversible race between $p$ and $r$, the check $s_{|p,r|} = s_{|r,p|}$ also succeeds adding $r.p$ to $\text{dnd}(\epsilon)$. At this point $\text{wut}(\epsilon)$ contains $q$ and $r$. After backtracking to state 0 with $q$, $p$ and $r$ can be executed. Let us suppose that $q$.p.r is fully explored. This exploration is analogous to that of $p.q.r$. Therefore, $r.p$ will be added to $\text{dnd}(q)$, stopping the exploration at state 8 in Fig. 2(a). Due to the reversible race between $q$ and $r$, the algorithm checks $s_{|q,r|} = s_{|r,q|}$, which succeeds adding $q.r$ to $\text{dnd}(\epsilon)$. Finally, after backtracking to state 0 with $r$, sequences $r.p$ and $r.q$ are inherited down to state 9, as $p$ and $q$, respectively (line 30). Hence, the exploration stops at state 9.

### 3.2 Overcoming Problem II

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

Such a redundant trace would not have been explored by ODOR$_{cs}$. This is because ODOR$_{cs}$ (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 ODOR$_{cs}$ decisions regarding what to explore, achieving earlier and more effective context-sensitive prunings. However, wakeup trees are essential for ODOR 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 ODOR$_{cs}$.

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

### 3.3 Correctness, Optimality and Final Remarks

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

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

Note that this section focuses on the correctness and optimality theorems of ODPOR
\(_c\) and, thus, the original check [2] is used (\(sleep(E') \land W uT(E)(v) \neq \emptyset\)). However, a similar reasoning can be done for the check in [7]: \(v \in redundant(E, done)\). Proofs for the theorems in the paper can be found online in a technical report at costa.fdi.ucm.es/papers/costa/iststa19-proofs.pdf.

Lemma 3.3. If Algorithm 1 discovers that \(s(pre^+(E, e')) \equiv \emptyset\) \(E_0 \cup (s(u, succ(e)), e')\) then for any complete sequence \(E\) of the form \(E = E_0 \cup u \cdot v \cdot w\) that contains a race \(e' \prec_E e\), there is a complete sequence \(E' = pre^+(E', e'), w\) which defines a different Mazurkiewicz trace \(T' = \rightarrow_{E'}\) and leads to an identical final state.

Theorem 3.4 (Soundness of ODPOR
\(_c\)). For each Mazurkiewicz trace \(T\) defined by the happens-before relation, \(Explore(\{e\}, \emptyset)\) of Algorithm 1 explores a complete execution sequence that either implements \(T\), or reaches an identical state to one that implements \(T\).

We claim now the optimality of Algorithm 1.

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

\[
Sleep(E'') \cap W uT(E)(v'w) \neq \emptyset \Rightarrow Sleep(E''') \cap W uT(E)(v''w) \neq \emptyset
\]

Theorem 3.6 (Optimality of ODPOR
\(_c\)). Algorithm 1 never explores two complete execution sequences that are equivalent and never initiates sleep set blocked executions.

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

4 OPTIMAL CONTEXT-SENSITIVE DPOR WITH OBSERVERS

The ODPOR
\(_b\) and ODPOR
\(_c\) algorithms of Secs. 2.2 and 3 can be combined simply by joining their codes together. This also requires using the happens-before relation based on the notion of observability of [7] throughout the algorithm. The exploration performed by such a “union” algorithm would be the intersection of the explorations of ODPOR
\(_b\) and ODPOR
\(_c\), and its pruning the union of the ODPOR
\(_b\) and ODPOR
\(_c\) prunings.

This section goes beyond the union of the algorithms and proposes in Secs. 4.1 and 4.2 two enhancements that exploit the combination and the synergy between the notions of context-sensitive independence and observability. The resulting algorithm ODPOR
\(_b\) is presented in Algorithm 2. Finally, Sec. 4.3 studies the soundness of ODPOR
\(_b\).

Algorithm 2 ODPOR
\(_b\) algorithm

43: procedure \(Explore(E, W uT, DnD)\)
44: \(dnd(E) := DnD;\)
45: \(done(E) := \emptyset;\)
46: if \(enabled(s(E)) = \emptyset\) then \(RaceDetection(E);\)
47: else if \(W uT \neq \emptyset\) then \(RaceDetection(E, p);\)
48: \(wut(E) := W uT;\)
49: else if \(enabled(s(E))(dnd(E)) = \emptyset\) then
50: for each \(p \in dnd(E)\) such that \(|p| = 1:\)
51: \(RaceDetection(E, p);\)
52: else
53: choose \(p \in enabled(s(E))(dnd(E);\)
54: \(wut(E) := ((e, p), \{(p, e)\});\)
55: while \(\exists p \in wut(E)\) do
56: \(let p = min_c(p \in wut(E));\)
57: if \(p \in dnd(E)\) then
58: \(RaceDetection(E, p);\)
59: else
60: \(let W uT' = subtree(wut(E), p);\)
61: \(let DnD' :\{v | v \in dnd(E), p \notin v, E \nrightarrow p \circ v\}\)
62: \(\cup \{(u, v) | (u \cdot p \cdot v) \in dnd(E), E \nrightarrow u \circ p \circ v\};\)
63: \(Explore(E, p, W uT', DnD');\)
64: \(add p to done(E);\)
65: remove all sequences of form \(p \circ w\) from \(wut(E);\)
66: procedure \(RaceDetection(E);\)
67: for all \(e, e' \in dom(E)\) such that \(e \prec_E e'\) do
68: \(let E' = pre(E, e); let dont = e;\)
69: if \(observers(e, e') \neq \emptyset\) then
70: \(let o = max(observers(e, e'), E);\)
71: \(let v = notdep^e(e, e', E, e', \hat{e}, (notobs^e(e, e', E), \hat{e}));\)
72: \(let o' = observers(e, e', E, v \nrightarrow dnd(E, v, E));\)
73: \(if \(e' \in E, pre^e(E, e') = (\hat{e}, o', o' \in E, v \cdot succ(e, E), o')\) then \(dont := v \cdot (o' \setminus o');\)
74: else
75: \(let v = notdep^e(e, e', E, e', \hat{e}, (notobs^e(e, e', E), \hat{e}));\)
76: \(if E, pre^e(E, e') = (\hat{e}, o', o' \in E, v \cdot succ(e, E), o')\) then \(dont := v \cdot \hat{e};\)
77: \(wut(E') := insert(p', E, wut(E'));\)
78: \(add dont to dnd(E');\)


4.1 Refining the Context-Sensitive Check for Write-Write Races

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

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

The implementation of the refined check corresponds to the underlined red code in lines 72 and 73 of Algorithm 2. First, it sets \( o_\text{in} \) 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 \( o_\text{in} \) (this time treated as a set), the state after executing \( o' \) is equivalent modulo observability to the state obtained by the alternative sequence \( E' \cdot \nu_{E'}(v) \cdot v \), which contains the reversed race, followed by observer \( o \) and the remaining observers.

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

Note that all the predecessors of other observers are in \( u_{E'} \), thanks to the choice of \( o \) as \( \text{max}(observers(e, e', E)) \). Thus, we can execute \( \delta_0 \setminus \delta \) (\( \delta \) is already in \( v \)) without problem after \( E' \cdot \nu_{E'}(v) \). Note also that we cannot use \( s[p.r(e,v)] \) to perform all the checks because, after every \( o' \) of \( o_\text{in} \) has been executed, there may be another event \( e'' \) before \( o \) 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 \( s[p.r(e,v)] \) for each \( o' \) of \( o_\text{in} \) 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 \( s[noteq^*(e,e',E),\hat{e},\hat{e}',\hat{notobs}^*(e,e',E)\{\{\delta\}] \) (where the race between \( e \) and \( e' \) has not been reversed).

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

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

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

4.2 Refining the Inheritance of Don’t-Do Sequences

One could think that whenever a sequence \( w \) is added to a \( dnd(E') \) set of sequence \( E' \) due to the new refined check, then a prefix of \( w \) is also added to \( wut(E') \). Indeed, if the refined check of line 73 succeeds, the sequence \( v.(\delta_0 \setminus \delta) \) is added to \( dnd(E') \), and the sequence \( v \) is inserted to \( wut(E') \). However, it is possible for the sequence not to be added to \( wut(E') \) if it already contains an equivalent sequence (which had been added before). In such cases, the \( dnd \) sequence might not be propagated successfully during the exploration of the corresponding sequence in \( wut(E') \), resulting in unnecessary exploration.

Example 4.2. Let us consider our running example but replacing process \( r \) by \( r : o = x_\star \) and first exploring sequence \( E_1 = p.p'.q.r \), where \( p' \) is another instance of the same process \( p \). For the race between \( p \) and \( q \), the refined check builds the alternative sequence
Lemma 4.3. If Algorithm 2 discovers that $s_{\text{pre}}(E',o') = s_{E'}^{e',e''}$, for any complete sequence $E$ of the form $E=E_0,v(E_0,\delta_0),\omega'$ that contains a race $e' \geq E$ observed by $a_0$, we can immediately deduce that the race $e''$ is also observed by $a_0$. If $E'$ is a complete sequence that defines a different Mazurkiewicz trace $T' \rightarrow E'$, then $T' \rightarrow E'$ modulo observability.

Lemma 4.4 (Soundness of New Inheritance). Let $E'$ be an execution such that $q.p.u \in \text{wut}(E')$, and $q.p.u \in dnd(E')$, and $E'=q.p.u.v \Rightarrow q.p.u.v$. Let $E$ be a complete sequence such that $E$ is a complete sequence that defines a different Mazurkiewicz trace $T' \rightarrow E'$, and $T' \rightarrow E'$ modulo observability.

Theorem 4.5 (Soundness of ODPOR$^{ab}_{cs}$). For each Mazurkiewicz trace $T$ defined by the happens-before relation, $\text{Explo}(\cdot, (\cdot), \emptyset, \emptyset)$ of Algorithm 2 explores a complete execution sequence that either implements $T$, or reaches an equivalent state modulo observability as one that implements $T$.

Let us now observe the optimality of Algorithm 2.

Theorem 4.6 (Optimality of ODPOR$^{ab}_{cs}$). Algorithm 2 never explores two complete execution sequences that are equivalent.
number of states stored, which corresponds to the number of states of the longest explored trace.

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

In the second set of benchmarks DPOR\textsubscript{cs} is already optimal for Fib and MS. Hence, the addition of observers has no benefit and the slower context-sensitive checks introduce a slowdown. For DBP, observers achieve important gains and the combination with context-sensitivity gives further benefits. For QS, we obtain significant gains over both algorithms, although those over ODPORT\textsuperscript{ob}do not scale. In most benchmarks, we have been able to identify which of the extensions proposed in the paper are leading to the gains. In particular, the gains in QS are achieved due to the extension of Sec. 3. The refined context-sensitive check is fundamental for the gains achieved in FR and FR-a. Finally, the new way of inheriting the dnd sets leads to the gains of abs and DBP.

### Table 1: Experimental evaluation results

<table>
<thead>
<tr>
<th>Bench.</th>
<th>DPOR\textsubscript{cs}</th>
<th>ODPORT\textsuperscript{ob}cs</th>
<th>ODPORT\textsuperscript{ob}cs ( \times )</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR(3)</td>
<td>17 36 0.02</td>
<td>13 29 0.03</td>
<td>8 27 0.04</td>
<td>0.5x</td>
</tr>
<tr>
<td>FR(5)</td>
<td>416 865 0.35</td>
<td>81 247 0.17</td>
<td>29 141 0.14</td>
<td>2.7x</td>
</tr>
<tr>
<td>FR(7)</td>
<td>21k 42k 34.21</td>
<td>449 2k 2.28</td>
<td>68 456 0.82</td>
<td>42.1x</td>
</tr>
<tr>
<td>FR(9)</td>
<td>&gt;51k &gt;104k 120.00</td>
<td>3k 12k 29.92</td>
<td>129 2k 5.18</td>
<td>&gt;23.2k</td>
</tr>
<tr>
<td>FR-a(4)</td>
<td>24 107 0.05</td>
<td>33 86 0.05</td>
<td>1 40 0.04</td>
<td>1.3x</td>
</tr>
<tr>
<td>FR-a(6)</td>
<td>720 4k 2.17</td>
<td>193 680 0.88</td>
<td>1 119 0.25</td>
<td>9.0x</td>
</tr>
<tr>
<td>FR-a(8)</td>
<td>&gt;20k &gt;88k 120.00</td>
<td>2k 5k 10.79</td>
<td>1 270 1.18</td>
<td>&gt;101.9k</td>
</tr>
<tr>
<td>FR-a(10)</td>
<td>&gt;18k &gt;79k 120.00</td>
<td>&gt;5k &gt;26k 120.00</td>
<td>1 517 5.09</td>
<td>&gt;23.6x</td>
</tr>
<tr>
<td>LW(3)</td>
<td>6 17 0.01</td>
<td>3 10 0.01</td>
<td>1 10 0.01</td>
<td>0.9x</td>
</tr>
<tr>
<td>LW(5)</td>
<td>120 327 0.16</td>
<td>5 21 0.02</td>
<td>1 21 0.03</td>
<td>7.8x</td>
</tr>
<tr>
<td>LW(7)</td>
<td>6k 14k 8.96</td>
<td>7 36 0.06</td>
<td>1 36 0.07</td>
<td>137.8x</td>
</tr>
<tr>
<td>LW(10)</td>
<td>&gt;32k &gt;85k 120.00</td>
<td>10 66 0.32</td>
<td>1 66 0.31</td>
<td>&gt;360.6x</td>
</tr>
<tr>
<td>abs(2)</td>
<td>4 28 0.02</td>
<td>2 15 0.01</td>
<td>1 12 0.01</td>
<td>2.0x</td>
</tr>
<tr>
<td>abs(3)</td>
<td>54 577 0.30</td>
<td>2 26 0.02</td>
<td>1 23 0.02</td>
<td>19.8x</td>
</tr>
<tr>
<td>abs(4)</td>
<td>2k 33k 29.34</td>
<td>2 44 0.04</td>
<td>1 40 0.05</td>
<td>598.6x</td>
</tr>
<tr>
<td>abs(5)</td>
<td>&gt;4k &gt;109k 120.00</td>
<td>2 58 0.09</td>
<td>1 54 0.10</td>
<td>&gt;1250.0x</td>
</tr>
<tr>
<td>Lam(2)</td>
<td>37 605 0.30</td>
<td>30 470 0.59</td>
<td>26 456 0.46</td>
<td>0.7x</td>
</tr>
<tr>
<td>Fib(3)</td>
<td>1 18 0.01</td>
<td>6 22 0.01</td>
<td>1 18 0.01</td>
<td>0.8x</td>
</tr>
<tr>
<td>Fib(4)</td>
<td>1 43 0.02</td>
<td>90 250 0.18</td>
<td>1 43 0.03</td>
<td>0.7x</td>
</tr>
<tr>
<td>Fib(5)</td>
<td>1 99 0.04</td>
<td>4k 11k 22.36</td>
<td>1 99 0.09</td>
<td>0.5x</td>
</tr>
<tr>
<td>Fib(6)</td>
<td>1 228 0.13</td>
<td>&gt;2k &gt;6k 120.00</td>
<td>1 228 0.63</td>
<td>0.2x</td>
</tr>
<tr>
<td>QS(8)</td>
<td>1 763 0.31</td>
<td>4k 12k 23.67</td>
<td>1 309 0.19</td>
<td>1.7x</td>
</tr>
<tr>
<td>QS(10)</td>
<td>1 4k 1.47</td>
<td>&gt;6k &gt;31k 120.00</td>
<td>1 607 0.50</td>
<td>3.0x</td>
</tr>
<tr>
<td>QS(13)</td>
<td>1 25k 14.83</td>
<td>&gt;2k &gt;15k 120.00</td>
<td>1 2k 1.68</td>
<td>8.9x</td>
</tr>
<tr>
<td>QS(15)</td>
<td>1 99k 72.95</td>
<td>&gt;826 &gt;10k 120.00</td>
<td>1 3k 3.54</td>
<td>20.6x</td>
</tr>
<tr>
<td>MS(7)</td>
<td>1 70 0.03</td>
<td>2k 4k 5.12</td>
<td>1 68 0.06</td>
<td>0.5x</td>
</tr>
<tr>
<td>MS(9)</td>
<td>1 121 0.06</td>
<td>14k 37k 116.75</td>
<td>1 107 0.14</td>
<td>0.4x</td>
</tr>
<tr>
<td>MS(11)</td>
<td>1 172 0.09</td>
<td>&gt;4k &gt;13k 120.00</td>
<td>1 166 0.33</td>
<td>0.3x</td>
</tr>
<tr>
<td>DBP(3)</td>
<td>361 5k 2.89</td>
<td>32 210 0.24</td>
<td>4 65 0.09</td>
<td>35.2x</td>
</tr>
<tr>
<td>DBP(6)</td>
<td>2k 21k 66.59</td>
<td>64 451 0.73</td>
<td>4 73 0.14</td>
<td>479.1x</td>
</tr>
<tr>
<td>DBP(7)</td>
<td>&gt;3k &gt;26k 120.00</td>
<td>128 964 2.23</td>
<td>5 109 0.28</td>
<td>&gt;431.7x</td>
</tr>
<tr>
<td>DBP(8)</td>
<td>&gt;3k &gt;27k 120.00</td>
<td>256 3k 6.78</td>
<td>5 117 0.44</td>
<td>&gt;275.9x</td>
</tr>
<tr>
<td>BB(3)</td>
<td>11 38 0.02</td>
<td>20 49 0.03</td>
<td>5 23 0.02</td>
<td>1.0x</td>
</tr>
<tr>
<td>BB(5)</td>
<td>80 326 0.13</td>
<td>252 671 0.56</td>
<td>17 103 0.09</td>
<td>1.5x</td>
</tr>
<tr>
<td>BB(7)</td>
<td>580 3k 1.18</td>
<td>4k 10k 18.02</td>
<td>65 459 0.82</td>
<td>1.5x</td>
</tr>
<tr>
<td>BB(8)</td>
<td>5k 21k 12.49</td>
<td>&gt;10k &gt;28k 120.00</td>
<td>257 3k 15.87</td>
<td>0.8x</td>
</tr>
<tr>
<td>MapRed</td>
<td>9 162 114</td>
<td>118 856 2961</td>
<td>9 162 185</td>
<td>0.6x</td>
</tr>
<tr>
<td>SDN</td>
<td>22 242 83</td>
<td>58 287 229</td>
<td>16 83 52</td>
<td>1.6x</td>
</tr>
</tbody>
</table>

### #360
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\textsuperscript{cb} and ODPOR\textsubscript{cs} explore 9 executions in 185 ms. and 114 ms. respectively, while ODPOR\textsuperscript{ob} explores 118 executions and takes almost 3 seconds, since there is no gain in using observers in this case. For SDN, ODPOR\textsuperscript{ob} explores 16 executions in 52 ms., whereas ODPOR\textsubscript{cs} explores 22 executions in 83 ms. and ODPOR\textsuperscript{cb} 58 in 229 ms.

Regarding the memory requirements of ODPOR\textsuperscript{ob}, the results show that M\textsubscript{S} (i.e., the maximal length of the explored traces) remains low in all examples and grows linearly with the input size. The same holds for M\textsubscript{D} except for FR and FR-a, where an exponential growth can be observed. As already mentioned in Sec. 3 (and also discussed in [18] for a different algorithm), there are a number of strategies to reduce the amount of information to be stored.

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

6 CONCLUSIONS AND RELATED WORK

DPOR is one of the most scalable techniques used in the verification of concurrent systems. Recent work has introduced orthogonal notions of conditional independence into DPOR: DPOR\textsubscript{cs} [3] proposes a context-sensitive check in the current state to detect more accurately independence among processes, ODPOR\textsuperscript{ob} [7] 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\textsubscript{cs} and ODPOR\textsuperscript{cb}, via two major technical extensions to DPOR\textsubscript{cs}: (1) incorporating (and using effectively) the notion of wakeup tree used by ODPOR\textsuperscript{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 [8] focuses on the read-write of variables. It defines two traces to be observationally equivalent if every read event observes the same write event in both traces. In contrast, we use the notion of observability introduced by [7], which is based on observing interference of operations, not just individual writes. The equivalence relation used by Data-Centric is proven in [8] 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 [10]. The drawback of Data-Centric is that it is optimal only for programs with acyclic communication graphs. Instead, our work is an extension of an optimal algorithm [7].

Another approach is to generate independence constraints (ICs), which ensure the independence of each pair of processes in the program. The work in [14, 20] generated for the first time ICs for processes with a single instruction following some predefined patterns. Recently, Constrained DPOR [5] proposed to generate ICs in a pre-phase, using an SMT solver. It later used the generated ICs within DPOR in a similar way to how our context-sensitive checks are used. In addition, it can perform another type of pruning using the notion of transitive uniform conditional independence—which ensures the ICs hold along the whole execution trace (and ensures uniformity as defined in [12, 15]). The extension of Constrained DPOR with observers, to the best of our knowledge, has not been studied yet. We believe the integration of wakeup trees could be done similarly to our proposal in Sec. 3, and the enhancements in Sec. 4 would be applicable also in the Constrained DPOR framework. Still, the combination of transitive uniformity and observability remains to be investigated.

An orthogonal approach to increase scalability, introduced in Quasi-Optimal POR [17], 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 [18].

ACKNOWLEDGMENTS

This work was funded partially by the Spanish MECD FPU Grant FPU15/04313, the Spanish MINECO project TIN2015-69175-C4-2-R, the Spanish MICINN/FEDER, UE project RTI2018-094403-B-C31, the CM project S2018/TCS-4314 and the Australian ARC project DP180100151.

REFERENCES


