Constrained Dynamic Partial Order Reduction [⋆]

Elvira Albert$^{1[0000-0003-0048-0705]}$, Miguel Gómez-Zamalloa$^{1[0000-0003-1557-689X]}$, Miguel Isabel$^{1[0000-0002-5474-9258]}$, and Albert Rubio$^{2[0000-0002-0501-9830]}$

[1] Complutense University of Madrid, Spain
[2] Universitat Politècnica de Catalunya, Spain

Abstract. The cornerstone of dynamic partial order reduction (DPOR) is the notion of *independence* that is used to decide whether each pair of concurrent events p and t are in a race and thus both $p \cdot t$ and $t \cdot p$ must be explored. We present *constrained* dynamic partial order reduction (CDPOR), an extension of the DPOR framework which is able to avoid redundant explorations based on the notion of *conditional independence* —the execution of p and t commutes only when certain *independence constraint*s (ICs) are satisfied. ICs can be declared by the programmer, but importantly, we present a novel SMT-based approach to automatically synthesize ICs in a static pre-analysis. A unique feature of our approach is that we have succeeded to exploit ICs within the state-of-the-art DPOR algorithm, achieving *exponential* reductions over existing implementations.

1 Introduction

Partial Order Reduction (POR) is based on the idea that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, non-conflicting *independent* execution steps. Such equivalence class is called a Mazurkiewicz trace, and POR guarantees that it is sufficient to explore one interleaving per equivalence class. Early POR algorithms [9, 11, 20] relied on static over-approximations to detect possible *future* conflicts. The Dynamic-POR (DPOR) algorithm, introduced by Flanagan and Godefroid [10] in 2005, was a breakthrough in the area because it does not need to look at the future. It keeps track of the independence races witnessed along its execution and uses them to decide the required exploration dynamically, without the need of static approximation. DPOR is nowadays considered one of the most scalable techniques for software verification. The key of DPOR algorithms is in the dynamic construction of two types of sets at each scheduling point: the *sleep set* that contains processes whose exploration has been proved to be redundant (and hence

[⋆] This work was funded partially by the Spanish MECD Salvador de Madariaga Mobility Grants PRX17/00297 and PRX17/00303, the Spanish MINECO projects TIN2015-69175-C4-2-R and TIN2015-69175-C4-3-R, and by the CM project S2013/ICE-3006.

should not be selected), and the *backtrack set* that contains the processes that have not been proved independent with previously explored steps (and hence need to be explored). Source-DPOR (SDPOR) [1, 2] improves the precision to compute backtrack sets (named *source* sets), proving optimality of the resulting algorithm for *any* number of processes w.r.t. an *unconditional independence* relation.

Challenge. When considering (S)DPOR with unconditional independence, if a pair of events is not independent in all possible executions, they are treated as potentially dependent and their interleavings explored. Unnecessary exploration can be avoided using conditional independence. E.g., two processes executing respectively the atomic instructions `if(z≥0)z=x;` and `x=x+1;` would be considered dependent even if `z≤-1` — this is indeed an *independence constraint* (IC) for these two instructions. Conditional independence was early introduced in the context of POR [12,16]. The first algorithm that has used notions of conditional independence within the state-of-the-art DPOR algorithm is Context-Sensitive DPOR (CSDPOR) [3]. However, CSDPOR does not use ICs (it rather checks state equivalence dynamically during the exploration) and exploits conditional (context-sensitive) independence only *partially* to extend the sleep sets. Our challenge is twofold: (i) extend the DPOR framework to exploit ICs during the exploration in order to both reduce the backtrack sets and expand the sleep sets as much as possible, (ii) statically synthesize ICs in an automatic pre-analysis.

Contributions. The main contributions of this work can be summarized as:

1. We introduce sufficient conditions –that can be checked dynamically– to soundly exploit ICs within the DPOR framework.
2. We extend the state-of-the-art DPOR algorithm with new forms of pruning (by means of expanding sleep sets and reducing backtrack sets).
3. We present an SMT-based approach to automatically synthesize ICs for *atomic blocks*, whose applicability goes beyond the DPOR context.
4. We experimentally show the exponential gains achieved by CDPOR on some typical concurrency benchmarks used in the DPOR literature before.

2 Background

In this section we introduce some notations, the basic notions on the POR theory and the state-of-the-art DPOR algorithm that we will extend in Sec. 3.

Our work is formalized for a general model of concurrent systems, in which a program is composed of *atomic blocks* of code. An atomic block can contain just one (global) statement that affects the global state, a sequence of local statements (that only read and write the local state of the process) followed by a global statement, or a block of code with possibly several global statements but whose execution cannot interleave with other processes because it has been implemented as atomic (e.g., using locks, semaphores, etc.). Each atomic block in the program is given a unique block identifier. We use $spawn(P[ini])$ to create a new process. Depending on the programming language, P can be the name of a method and $[ini]$ initial values for the parameters, or P can be the identifier of the initial block to execute and $[ini]$ the initialization instructions, etc., in every case with mechanisms to continue the execution from one block to the following one. Notice that the use of atomic blocks in our formalization generalizes the particular case of considering atomicity at the level of single instructions.

As previous work on DPOR [1–3], 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). Let Σ be the set of states of the system. There is a unique initial state $s_0 \in \Sigma$. The execution of a process p is represented as a partial function $execute_p : \Sigma \mapsto \Sigma$ that moves the system from one state to a subsequent state. Each application of the function $execute_p$ represents the execution of an *atomic block* of the code that p is running, denoted as *event* (or execution step) of process p. An *execution sequence* E (also called *derivation*) of a system is a finite sequence of events of its processes starting from s_0, and it is uniquely characterized by the sequence of processes that perform steps of E. For instance, $p \cdot q \cdot q$ denotes the execution sequence that first performs one step in p, followed by two steps in q. We use ϵ to denote the empty sequence. The state of the system after E is denoted by $s_{[E]}$. The set of processes enabled in state s (i.e., that can perform an execution step from s) is denoted by $enabled(s)$.

2.1 Basics of partial order reduction

An *event* e of the form (p, i) denotes the i-th occurrence of process p in an execution sequence, and $\hat{e}$ denotes the process p of event e, which is extended to sequences of events in the natural way. We write $\bar{e}$ to refer to the identifier of the atomic block of code the event e is executing. The set of events in execution sequence E is denoted by $dom(E)$. We use $e <_E e'$ to denote that event e occurs before event e' in E, s.t. $<_E$ establishes a total order between events in E, and $E \leq E'$ to denote that sequence E is a prefix of sequence E'. Let $dom_{[E]}(w)$ denote the set of events in execution sequence $E.w$ that are in sequence w, i.e., $dom(E.w) \backslash dom(E)$. If w is a single process p, we use $next_{[E]}(p)$ to denote the single event in $dom_{[E]}(p)$. If P is a set of processes, $next_{[E]}(P)$ denotes the set of $next_{[E]}(p)$ for all $p \in P$. The core concept in POR is that of the *happens-before* partial order among the events in execution sequence E, denoted by $\rightarrow_E$. This relation defines a subset of the $<_E$ total order, such that any two sequences with the same happens-before order are equivalent. Any linearization E' of $\rightarrow_E$ on $dom(E)$ is an execution sequence with exactly 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 linearizations of the same happens-before relation. The happens-before partial order has traditionally been defined in terms of a *dependency* relation between the execution steps associated to those events [11]. Intuitively, two steps p and q are *dependent* if there is at least one execution sequence E for which they do not commute, either because (i) one *enable*s the other (i.e., the execution of p leads to introducing q, or viceversa), or because (ii) $s_{[E.p.q]} \neq s_{[E.q.p]}$. We define $dep(E, e, n)$ as the subsequence containing all events e' in E that occur after e and happen-before n in $E.p$ (i.e., $e <_E e'$ and $e' \rightarrow_{E.p} n$). The unconditional dependency relation is used for defining the concept of a *race* between two events.

Algorithm 1 (Source+Context-sensitive)+Constrained DPOR algorithm

```
1: procedure EXPLORE($E$)
2:   if ($\exists p \in (enabled(s_{[E]}) \backslash sleep(E))$) then
3:     $back(E) := \{p\}$;
4:     while ($\exists p \in (back(E) \backslash sleep(E))$) do
5:       let $n = next_{[E]}(p)$;
6:       for all ($e \in dom(E)$ such that $e \precsim_{E.p} n$) do
7:         let $E' = pre(E, e)$;
8:         let $u = dep(E, e, n)$;
9:         if ($\neg(U_{\Rightarrow}(I_{\bar{e},\bar{n}}, e, n, s_{[E'.\hat{u}]}))$) then
10:          $updateBack(E, E', e, p)$;
11:          if $C(s_{[E'.\hat{u}]})$ for some $C \in I_{\bar{e},\bar{n}}$ then
12:            add $\hat{u}.p.\hat{e}$ to $sleep(E')$;
13:          else
14:            $updateSleepCS(E, E', e, p)$;
15:      $sleep(E.p) := \{x \mid x \in sleep(E),\ E \models p \diamond x\}$
16:             $\cup \{x \mid p.x \in sleep(E)\}$
17:             $\cup \{x \mid x \in sleep(E),\ |x| = 1,\ m = next_{[E]}(x),\ U_{\Rightarrow}(I_{\bar{n},\bar{m}}, n, m, s_{[E]})\}$;
18:      EXPLORE($E.p$);
19:      $sleep(E) := sleep(E) \cup \{p\}$;
```

Event e is said to be in race with event e' in execution E, if the events belong to different processes, e happens-before e' in E ($e \rightarrow_E e'$), and the two events are "concurrent", i.e. there exists an equivalent execution sequence $E' \simeq E$ where the two events are adjacent. We write $e \precsim_E e'$ to denote that e is in race with e' and that the race can be reversed (i.e., the events can be executed in reverse order). POR algorithms use this relation to reduce the number of equivalent execution sequences explored, with SDPOR ensuring that only one execution sequence in each equivalence class is explored.

2.2 State-of-the-art DPOR with unconditional independence

Algorithm 1 shows the state-of-the-art DPOR algorithm –based on the SDPOR algorithm of [1, 2],[3] which in turn is based on the original DPOR algorithm of [10]. We refer to this algorithm as DPOR in what follows. The context-sensitive extension of CSDPOR [3] (lines 14 and 16) and our extension highlighted in blue (lines 8-9, 11-13 and 17) should be ignored by now and will be described in Sec. 3.

The algorithm carries out a depth-first exploration of the execution tree using POR receiving as parameter a derivation E (initially empty). 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: $back(E')$ with the set of processes that must be explored from E', and, $sleep(E')$ with the set of sequences of processes that previous executions have determined do not need to be explored from E'. Note that in the original

[3] The extension to support *wake-up trees* [2] is deliberately not included to simplify the presentation.

DPOR the sleep set contained only single processes, but in later improvements sequences of processes are added, so our description considers this general case. The algorithm starts by selecting any process p that is enabled by the state reached after executing E and is not already in $sleep(E)$. If it does not find any such process p, it stops. Otherwise, after setting $back(E) = \{p\}$ to start the search, it explores every element in $back(E)$ that is not in $sleep(E)$. The backtrack set of E might grow as the loop progresses (due to later executions of line 10). For each such p, DPOR performs two phases: race detection (lines 6, 7 and 10) and state exploration (lines 15, 18 and 19). The race detection starts by finding all events e in $dom(E)$ such that $e \precsim_{E.p} n$, where n is the event being selected (see line 5). For each such e, it sets E' to $pre(E, e)$, i.e., to be the prefix of E up to, but not including e. Procedure *updateBack* modifies $back(E')$ in order to ensure that the race between e and n is reversed. The source-set extension of $[1, 2]$ detects cases where there is no need to modify $back(E')$ –this is done within procedure *updateBack* whose code is not shown because it is not affected by our extension. After this, the algorithm continues with the state exploration phase for $E.p$, by retaining in its sleep set any element x in $sleep(E)$ whose events in $E.p$ are independent of the next event of p in E (denoted as $E \models p \diamond x$), i.e., any x such that $next_{[E]}(p)$ would not happen-before any event in $dom(E.p.x) \backslash dom(E.p)$. Then, the algorithm explores $E.p$, and finally it adds p to $sleep(E)$ to ensure that, when backtracking on E, p is not selected until a dependent event with it is selected. All versions of the DPOR algorithm (except [3]) rely on the unconditional (or context-insensitive) dependency relation. This relation has to be over-approximated, usually by requiring that global variables accessed by one execution step are not modified by the other.

Example 1. Consider the example in Fig. 1 with 3 processes p, q, r containing a single atomic block. Since all processes have a single event, by abuse of notation, we refer to events by their process name throughout all examples in the paper. Relying on the usual over-approximation of dependency all three pairs of events are dependent. Therefore, starting with one instance per process, the algorithm has to explore 6 execution sequences, each with a different happens-before relation. The tree, including the dotted and dashed fragments, shows the exploration from the initial state $z{=}{-2}$, $x{=}{-2}$. The value of variable z is shown in brackets at each state. Essentially, in all states of the form $E.e$, the algorithm always finds a reversible race between the next event of the current selected process (p, q or r) and e, and adds it to $back(E)$. Also, when backtracking on E, none of the elements in $sleep(E)$ is propagated down, since all events are considered dependent. In the best case, considering an exact (yet unconditional) dependency relation which realizes that events p and r are independent, the algorithm will make the following reductions. In state 6, p and r will not be in race and hence p will not be added to $back(q)$. This avoids exploring the sequence $p.r$ from 5. When backtracking on state 0 with r, where $sleep(\epsilon){=}\{p, q\}$, p will be propagated down to $sleep(r)$ since $\epsilon{\models}r{\diamond}p$, hence avoiding the exploration of $p.q$ from 8. Thus, the algorithm will explore 4 sequences.

```
p: x = x+1;
q: if (z >= 0) z = x;
r: z = z+1; x = x+1;
```

$I_{\bar{p},\bar{q}} = \{z \leq -1\}$
$I_{\bar{p},\bar{r}} = \{true\}$
$I_{\bar{q},\bar{r}} = \{(z \geq 0), (z = x), (z \leq -2)\}$

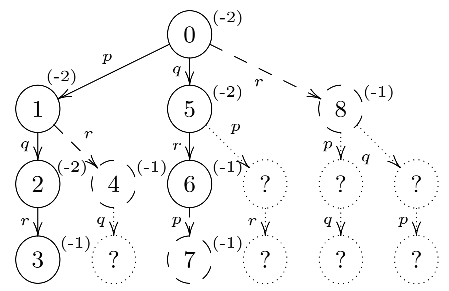

Fig. 1. Left: Code of working example (up) and ICs (down). Right: Execution tree starting from $z = -2,\ x = -2$. Full tree computed by SDPOR, dotted fragment not computed by CSDPOR, and, dashed+dotted fragment not computed by CDPOR.

3 DPOR with Conditional Independence

Our aim in CDPOR is twofold: (1) provide techniques to both infer and soundly check conditional independence, and (2) be able to exploit them at *all* points of the DPOR algorithm where dependencies are used. Sec. 3.1 reviews the notions of conditional independence and ICs, and introduces a first type of check where ICs can be directly used in the DPOR algorithm. Sec. 3.2 illustrates why ICs cannot be used at the remaining independence check points in the algorithm, and introduces sufficient conditions to soundly exploit them at those points. Finally, Sec. 3.3 presents the CDPOR algorithm that includes all types of checks.

3.1 Using precomputed ICs directly within DPOR

Conditional independence consists in checking independence at the given state.

Definition 1 (conditional independence). *Two events α and β are* independent *in state S, written indep(α, β, S) if (i1) none of them* enables *the other from S; and, (i2) if they are both enabled in S, then $S \xrightarrow{\alpha\cdot\beta} S'$ and $S \xrightarrow{\beta\cdot\alpha} S'$.*

The use of conditional independence in the POR theory was firstly studied in [16], and it has been partially applied within the DPOR algorithm in CSDPOR [3]. Function *updateSleepCS* at line 14 and the modification of *sleep* at 16 encapsulate this partial application of CSDPOR (the code of *updateSleepCS* is not shown because it is not affected by our extension). Intuitively, *updateSleepCS* works as follows: when a reversible race is found in the current sequence being explored, it builds an *alternative* sequence which corresponds to the reverse race, and then checks whether the states reached after running the two sequences are the same. If they are, it adds the alternative sequence to the corresponding *sleep* set so that this sequence is not fully explored when backtracking. Therefore, sleep sets can contain *sequences* of events which can be propagated down via the rule of line 16 (i.e., if the event being explored is the head of a sequence in the sleep set, then the tail of the sequence is propagated down). In essence, the technique to check ($i2$) in Def. 1 in CSDPOR consists in checking state equivalence with an alternative sequence in the current state (hence it is conditional) and, if the check succeeds, it is exploited in the *sleep* set only (and not in the *backtrack* set).

Example 2. Let us explain the intuition behind the reductions that CSDPOR is able to achieve w.r.t. unconditional independence-based DPOR on the example.

In state 1, when the algorithm selects q and detects the reversible race between q and p, it computes the alternative sequence $q.p$ and realizes that $s_{[p.q]} = s_{[q.p]}$, and hence adds $p.q$ to $sleep(\epsilon)$. Similarly, in state 2, it computes $p.r.q$ and realizes that $s_{[p.q.r]} = s_{[p.r.q]}$ adding $r.q$ to $sleep(p)$. Besides these two alternative sequences, it computes two more. Overall, CSDPOR explores 2 complete sequences ($p.q.r$ and $q.r.p$) and 13 states (the 9 states shown, plus 4 additional states to compute the alternative sequences).

Instead of computing state equivalence to check ($i2$) as in [3], our approach assumes precomputed *independence constraints* (ICs) for all pairs of atomic blocks in the program. ICs will be evaluated at the appropriate state to determine the independence between pairs of concurrent events executing such atomic blocks.

Definition 2 (ICs). *Consider two events α and β that execute, respectively, the atomic blocks $\bar{\alpha}$ and $\bar{\beta}$. The* independence constraints *$I_{\bar{\alpha},\bar{\beta}}$ are a set of boolean expressions (constraints) on the variables accessed by α and β (including local and global variables) s.t., if some constraint C in $I_{\bar{\alpha},\bar{\beta}}$ holds in state S, written $C(S)$, then condition (i2) of indep(α, β, S) holds.*

Our first contribution is in lines 11-13 where ICs are used within DPOR as follows. Before executing *updateSleepCS* at line 14, we check if some constraint in $I_{\bar{e},\bar{n}}$ holds in the state $s_{[E'.\hat{u}]}$, by building the sequence $E'.\hat{u}$, where $u = dep(E, e, n)$. Only if our check fails we proceed to execute *updateSleepCS*. The advantages of our check w.r.t. *updateSleepCS* are: (1) the alternative execution sequence built by *updateSleepCS* is strictly longer than ours and hence more states will be explored, and (2) *updateSleepCS* must check state equivalence while we evaluate boolean expressions. Yet, because our IC is an approximation, if we fail to prove independence we can still use *updateSleepCS*.

Example 3. Consider the ICs in Fig. 1 (down left), which provide the constraints ensuring the independence of each pair of atomic blocks, and whose synthesis is explained in Sec. 4.1. In the exploration of the example, when the algorithm detects the reversible race between q and p in state 1, instead of computing $q.p$ and then comparing $s_{[p.q]} = s_{[q.p]}$ as in CSDPOR, we would just check the constraint in $I_{\bar{p},\bar{q}}$ at state ϵ, i.e., in $z = -2$ (line 11), and since it succeeds, $q.p$ is added to $sleep(\epsilon)$. The same happens at states 2, again at 1 (when backtracking with r), and 5. This way we avoid the exploration of the additional 4 states due to the computation of the alternative sequences in Ex. 2 (namely $q.p$, $r.p$ and $r.q$ from state 0, and $r.q$ from 1). The algorithm is however still exploring many redundant derivations, namely states 4, 5, 6, 7 and 8.

3.2 Transitive uniformity: how to further exploit ICs within DPOR

The challenge now is to use ICs, and therefore conditional independence, at the remaining dependency checks performed by the DPOR algorithm, and most importantly, for the race detection (line 6). In the example, that would avoid the addition of q and r to $back(\epsilon)$ and r to $back(p)$, and hence would make the algorithm only explore the sequence $p.q.r$. Although that can be done in our example, it is unsound in general as the following counter-example illustrates.

Example 4. Consider the same example but starting from the initial state $z{=}-1$, $x{=}-2$. During the exploration of the first sequence $p.q.r$, the algorithm will not find any race since p and q are independent in $z{=}-1$, q and r are independent in $z{=}x{=}-1$, and, p and r are always independent. Therefore, no more sequences than $p.q.r$ with final result $z{=}0$ will be explored. There is however a non-equivalent sequence, $r.q.p$, which leads to a different final state $z{=}-1$.

The problem of using conditional independence within the POR theory was already identified by Katz and Peled [16]. Essentially, the main idea of POR is that the different linearizations of a partial order yield equivalent executions that can be obtained by swapping adjacent independent events. However, this is no longer true with conditional dependency. In Ex. 4, using conditional independence, the partial order of the explored derivation $p.q.r$ would be empty, which means there would be 6 possible linearizations. However $r.q.p$ is not equivalent to $p.q.r$ since q and p are dependent in $s_{[r]}$, i.e., when $z{=}0$. An extra condition, called *uniformity*, is proposed in [16] to allow using conditional independence within the POR theory. Intuitively, *uniform independence* adds a condition to Def. 1 to ensure that independence holds at all successor states for those events that are enabled and are *uniformly independent* with the two events whose independence is being proved. While this notion can be checked *a posteriori* in a given exploration, it is unclear how it could be applied in a dynamic setting where decisions are made *a priori*. Here we propose a weaker notion of uniformity, called *transitive uniformity*, for which we have been able to prove that the *dynamic*-POR framework is sound. The difference with [16] is that our extra condition ensures that independence holds at all successor states for *all* events that are enabled, which is thus a superset of the events considered in [16]. We notice that the general happens-before definition of [1,2] does not capture our transitive uniform conditional independence below (namely property seven of [1,2] does not hold), hence CDPOR cannot be seen as an instance of SDPOR but rather as an extension.

Definition 3. *The* transitive uniform *conditional independence relation, written unif*(α, β, S)*, fulfills (i1) and (i2) and, (i3) unif*$(\alpha, \beta, S_\gamma)$ *holds for all* $\gamma \notin \{\alpha, \beta\}$ *enabled in* S*, where* S_γ *is defined by* $S \xrightarrow{\gamma} S_\gamma$.

During the exploration of the sequence $p.q.r$ in Ex. 4, the algorithm will now find a reversible race between p and q, since the independence is not transitively uniform in $z{=}-1, x{=}-2$. Namely, *(i3)* does not hold since r is enabled and we have $x{=}-1$ and $z{=}0$ in $s_{[r]}$, which implies $\neg unif(p, q, s_{[r]})$ (*(i2)* does not hold).

We now introduce sufficient conditions for transitive uniformity that can be precomputed statically, and efficiently checked, in our dynamic algorithm. Condition *(i1)* is computed dynamically as usual during the exploration simply storing enabling dependencies. Condition *(i2)* is provided by the ICs. Our sufficient conditions to ensure *(i3)* are as follows. For each atomic block b, we precompute *statically* (before executing DPOR) the set $W(b)$ of the global variables that can be modified by the full execution of b, i.e., by an instruction in b or by any other block called from, or enabled by, b (transitively). To this end, we do a simple analysis which consists in: (1) First we build the call graph for the program to establish the calling relationships between the blocks in the program. Note that when we find a process creation instruction $spawn(P[ini])$ we have a calling relationship between the block in which the spawn instruction appears and P. (2) We obtain (by a fixed point computation) the largest relation fulfilling that g belongs to $W(b)$ if either g is *modified* by an instruction in b or g belongs to $W(c)$ for some block c called from b. This computation can be done with different levels of precision, and it is well-studied in the static analysis field [18]. We let $G(C)$ be the set of global variables evaluated on constraint C in I.

Definition 4 (sufficient condition for transitive uniformity, $U_{\Rightarrow}$). *Let E be a sequence, I a set of constraints, α and β be two events enabled in $s_{[E]}$, and $T = next_{[E]}(enabled(s_{[E]})) \setminus \{\alpha, \beta\}$, we define $U_{\Rightarrow}(I, \alpha, \beta, s_{[E]}) \equiv \exists C \in I : C(s_{[E]}) \wedge ((G(C) \cap \bigcup_{t \in T} W(\bar{t})) = \emptyset)$*

Intuitively, our sufficient condition ensures transitive uniformity by checking that the global variables involved in the constraint C of the IC used to ensure the uniformity condition are not modified by other enabled events in the state.

Theorem 1. *Given a sequence E and two events α and β enabled in $s_{[E]}$, we have that $U_{\Rightarrow}(I_{\bar{\alpha},\bar{\beta}}, \alpha, \beta, s_{[E]}) \Rightarrow unif(\alpha, \beta, s_{[E]})$.*

3.3 The Constrained DPOR algorithm

The code highlighted in blue in Algorithm 1 provides the extension to apply conditional independence within DPOR. In addition to the pruning explained in Sec. 3.1, it achieves two further types of pruning:

1. *Back-set reduction.* The race detection is strengthened with an extra condition (line 9) so that e and n (the next event of p) are in race only if they are not conditionally independent in state $s_{[E'.u]}$ (using our sufficient condition above). Here u is the sub-sequence of events of E that occur after e and "happen-before" n. This way the conditional independence is evaluated in the state after the shortest subsequence so that the events are adjacent in an equivalent execution sequence.
2. *Sleep-set extension.* An extra condition to propagate down elements in the sleep set is added (line 17) s.t. a sequence x, with just one process, is propagated if its corresponding event is conditionally independent of n in $s_{[E]}$.

It is important to note also that the inferred conditional independencies are recorded in the happens-before relation to be later re-used for subsequent computations of the $\precsim$ and dep definitions.

Example 5. Let us describe the exploration for the example in Fig. 1 using our CDPOR. At state 1, the algorithm checks whether p and q are in race. $U_{\Rightarrow}(I_{\bar{p},\bar{q}}, p, q, S)$ does not hold in $z{=}{-2}$ since, although $(z{\leq}{-1}) \in I_{\bar{p},\bar{q}}$ holds, we have that $G(z \leq -1) \cap W(r) = \{z\} \neq \emptyset$. Process q is hence added to $back(\epsilon)$. On the other hand, since $(z{\leq}{-1}) \in I_{\bar{p},\bar{q}}$ holds in $z{=}{-2}$ (line 11), $q.p$ is added to $sleep(\epsilon)$ (line 12). At state 2 the algorithm checks the possible race between q and r after executing p. This time the transitive uniformity of the independence of q and r holds since $(z{\leq}-2) \in I_{\bar{q},\bar{r}}$ holds, and there are no enabled events out of $\{q, r\}$. Our algorithm therefore avoids the addition of r to $back(p)$ (pruning 1 above). The algorithm also checks the possible race between p and r in $z{=}-2$. Again, $true \in I_{\bar{p},\bar{r}}$ holds and is uniform since $G(true) = \emptyset$ (pruning 1). The algorithm finishes the exploration of sequence $p.q.r$ and then backtracks with q at state 0. At state 5 the algorithm selects process r (p is in the sleep set of 5 since it is propagated down from the $q.p$ in $sleep(\epsilon)$). It then checks the possible race between q and r, which is again discarded (pruning 1), since transitive uniformity of the independence of q and r can be proved: we have that $(z{\leq}-2) \in I_{\bar{q},\bar{r}}$ holds in $z{=}-2$ and $W(p) \cap G(z{\leq}-2) = \emptyset$, where p is the only enabled event out of $\{q, r\}$ and $W(p) = \{x\}$. This avoids adding r to $back(\epsilon)$. Finally, at state 5, p is propagated down in the new sleep set (pruning 2), since as before $true \in I_{\bar{p},\bar{r}}$ ensures transitive uniformity. The exploration therefore finishes at state 6.

Overall, on our working example, CDPOR has been able to explore only one complete sequence $p.q.r$ and the partial sequence $q.r$ (a total of 6 states). The latter one could be avoided if a more precise sufficient condition for uniformity is provided which, in particular, is able to detect that the independence of p and q in ϵ is transitive uniform, i.e., it still holds after r (even if r writes variable z).

Theorem 2 (soundness). *For each Mazurkiewicz trace T defined by the happens before relation, Explore($\epsilon, \emptyset$) in Alg. 1 explores a complete execution sequence T' that reaches the same final state as T.*

4 Automatic generation of ICs using SMT

Generating ICs amounts to proving (conditional) program equivalence w.r.t. the global memory. While the problem is very hard in general, proving equivalence of smaller blocks of code becomes more tractable. This section introduces a novel SMT-based approach to synthesize ICs between pairs of atomic blocks of code. Our ICs can be used within any transformation or analysis tool –beyond DPOR– which can gain accuracy or efficiency by knowing that fragments of code (conditionally) commute. Sec. 4.1 first describes the inference for basic blocks; Sec. 4.2 extends it to handle process creation and Sec. 4.3 outlines other extensions, like loops, method invocations and data structures.

4.1 The basic inference

In this section we consider blocks of code containing conditional statements and assignments using linear integer arithmetic (LIA) expressions. The first step to carry out the inference is to transform q and r into two respective *deterministic* Transition Systems (TSs), T_q and T_r (note that q and r are assumed to be deterministic), and compose them in both reverse orders $T_{q \cdot r}$ and $T_{r \cdot q}$. Consider r and q in Fig. 1 whose associated TSs are (primed variables represent the final value of the variables):

$$T_q\!: z \geq 0 \rightarrow z' = x; \quad z < 0 \rightarrow z' = z; \qquad T_r\!: true \rightarrow x' = x + 1, z' = z + 1;$$

The code to be analyzed is the composition of T_q and T_r in both orders:

$$T_{q \cdot r}\!: \begin{array}{l} z \geq 0 \rightarrow x' = x + 1, z' = x + 1; \\ z < 0 \rightarrow x' = x + 1, z' = z + 1; \end{array} \qquad T_{r \cdot q}\!: \begin{array}{l} z \geq -1 \rightarrow x' = x + 1, z' = x + 1; \\ z < -1 \rightarrow x' = x + 1, z' = z + 1; \end{array}$$

In what follows we denote by $T_{a \cdot b}$ the deterministic TS obtained from the concatenation of the blocks a and b, such that all variables are assigned in one instruction using parallel assignment. We let $A\mid_G$ be the restriction to the global memory of the assignments in A (i.e., ignoring the effect on local variables).The following definition provides an SMT formula over LIA (a boolean formula where the atoms are equalities and inequalities over linear integer arithmetic expressions) which encodes the independence between the two blocks.

Definition 5 (IC generation). *Let us consider two atomic blocks q and r and a global memory G and let $C_i \rightarrow A_i$ (resp. $C'_j \rightarrow A'_j$) be the transitions in $T_{q \cdot r}$ (resp. $T_{r \cdot q}$). We obtain $F_{q,r}$ as the SMT formula: $\bigvee_{i,j}(C_i \wedge C'_j \wedge A_i\mid_G = A'_j\mid_G)$.*

Intuitively, the SMT encoding in the above definition has as solutions all those states where both a condition C_i of a transition in $T_{q \cdot r}$ and C'_j of a transition in $T_{r \cdot q}$ hold (and hence are compatible) and the final global state after executing all instructions in the two transitions (denoted A_i and A'_j) remains the same.

Next, we generate the constraints of the independence condition $I_{q,r}$ by obtaining a compact representation of all models over linear arithmetic atoms (computed by an *allSAT* SMT solver) satisfying $F_{q,r}$. In particular, we add a constraint in $I_{q,r}$ for every obtained model.

Example 6. In the example, we have the TS with conditions and assignments:

$T_{q \cdot r}$:	$C_1{:}z \geq 0$	$A_1{:}x' = x+1, z' = x+1$	$T_{r \cdot q}$:	$C'_1{:}z \geq -1$	$A'_1{:}x' = x+1, z' = x+1$
	$C_2{:}z < 0$	$A_2{:}x' = x+1, z' = z+1$		$C'_2{:}z < -1$	$A'_2{:}x' = x+1, z' = z+1$

and we obtain a set with three constraints $I_{q,r} = \{(z \geq 0), (z = x), (z < -1)\}$ by computing all models satisfying the following resulting formula:

$$\begin{array}{l}
(\ z \geq 0 \wedge z \geq -1 \wedge x+1 = x+1 \wedge x+1 = x+1\ ) \vee \\
(\ z \geq 0 \wedge z < -1 \wedge x+1 = x+1 \wedge x+1 = z+1\ ) \vee \\
(\ z < 0 \wedge z \geq -1 \wedge x+1 = x+1 \wedge z+1 = x+1\ ) \vee \\
(\ z < 0 \wedge z < -1 \wedge x+1 = x+1 \wedge z+1 = z+1\ )
\end{array}$$

The second conjunction is unsatisfiable since there is no model with both C_1 and C'_2. On the other hand, the equalities of the first and the last conjunctions always hold, which give us the constraints $z{\geq}0$ and $z{\leq}-2$. Finally, all equalities hold when $x{=}z$, which give us the third constraint as a result for our SMT encoding.

Note that, as in this case $F_{q,r}$ describes not only a sufficient but also a necessary condition for independence, the obtained constraints IC are also a sufficient and necessary conditions for independence. This allows removing line 14 in the algorithm, since the context-sensitive check will fail if line 11 does. However, the next extensions do not ensure that the generated ICs are necessary conditions.

4.2 IC for blocks with process creation

Consider the following two methods whose body constitutes an atomic block (e.g., the lock is taken at the method start and released at the return). They are inspired by a highly concurrent computation for the Fibonacci used in the experiments. Variables `nr` and `r` are global to all processes:

```
fib(int v) {
    if (v≤1) {spawn(res(v));}
    else {spawn(fib(v-1));
            spawn(fib(v-2));}
}
```

```
res(int v) {
    if (nr>0) {nr=0; r=v; }
    else {spawn(res(r+v));
            r=0;nr=1;}
}
```

We now want to infer $I_{\mathsf{fib(v)},\mathsf{fib(v_1)}}$, $I_{\mathsf{fib(v)},\mathsf{res(v_1)}}$, $I_{\mathsf{res(v)},\mathsf{res(v_1)}}$. The first step is to obtain, for each block r, a *TS with uninterpreted functions*, denoted TS^u_r, in which transitions are of the form $C \rightarrow (A, S)$ where A are the parallel assignments as in Sec. 4.1, and S is a multiset containing calls to fresh *uninterpreted* functions associated to the processes spawned within the transition (i.e., a process creation $spawn(P)$ is associated to an uninterpreted function $spawn\_P$).

$$
\begin{aligned}
T^u_{\mathsf{fib}}&: v \leq 1 \rightarrow (skip, \{spawn\_res(v)\})\\
&\quad v > 1 \rightarrow (skip, \{spawn\_fib(v-1), spawn\_fib(v-2)\}\\
T^u_{\mathsf{res}}&: nr \geq 0 \rightarrow (nr' = 0, r' = v, \{\})\\
&\quad nr < 0 \rightarrow (nr' = 1, r' = 0, \{spawn\_res(r+v)\}
\end{aligned}
$$

The following definition extends Def. 5 to handle process creation. Intuitively, it associates a fresh variable to each different element in the multisets (mapping P' below) and enforces equality among the multisets.

Definition 6 (IC generation with process creation). *Let us consider $TS^u_{r\cdot q}$ and $TS^u_{q\cdot r}$. We define $P = \{\cup s \mid s \in S,\ with\ C \rightarrow (A, S) \in TS^u_{r\cdot q} \cup TS^u_{q\cdot r}\}$. Let P' be a mapping from the elements in P to fresh variables, and $P'(S)$ be the replacement of the elements in the multiset S applying the mapping P'. Let $C_i \rightarrow (A_i, S_i)$ (resp. $C'_j \rightarrow (A'_j, S'_j)$) be the transitions in $TS^u_{q\cdot r}$ (resp. $TS^u_{r\cdot q}$). We obtain $F_{q,r}$ as the SMT formula: $\bigvee_{i,j}(C_i \wedge C'_j \wedge A_i\mid_G = A'_j\mid_G \wedge P'(S_i) \equiv P'(S'_j))$*

For simplicity and efficiency, we consider that $\equiv$ corresponds to the syntactic equality of the multisets. However, in order to improve the precision of the encoding we apply P' to S_i and S_j replacing two process creations by the same variable if they are equal modulo associativity and commutativity (AC) of arithmetic operators and after substituting the equalities already imposed by $A_i\mid_G = A'_j$ (see example below). A more precise treatment can be achieved by using equality with uninterpreted functions (EUF) to compare the multisets of processes.

Example 7. Let us show how we apply the above definition to infer $I_{\mathsf{res(v)},\mathsf{res(v_1)}}$. We first build $T_{\mathsf{res(v)}\cdot\mathsf{res(v_1)}}$ from $T_{\mathsf{res(v)}}$ by composing it with itself:

$$
\begin{aligned}
&nr \leq 0 \rightarrow (nr'{=}0, r'{=}v_1, \{spawn\_res(r{+}v)\})\\
&nr > 0 \rightarrow (nr'{=}1, r'{=}0, \{spawn\_res(v{+}v_1)\})
\end{aligned}
$$

and $T_{\mathsf{res(v_1)}\cdot\mathsf{res(v)}}$ which is like the one above but exchanging v and $\mathsf{v_1}$. Next, we define $P' = \{spawn\_res(r+v) \mapsto x_1, spawn\_res(v+v_1) \mapsto x_2, spawn\_res(r+v_1) \mapsto x_3, spawn\_res(v_1+v) \mapsto x_4\}$ and apply it with the improvement described above

$$
\begin{aligned}
&(\ nr \leq 0 \wedge nr \leq 0 \wedge 0 = 0 \wedge v = v_1 \wedge \{x_1\} = \{x_1\}\ ) \vee\\
&(\ nr \leq 0 \wedge nr > 0 \wedge 0 = 1 \wedge v_1 = 0 \wedge \{x_1\} = \{x_4\}\ ) \vee\\
&(\ nr > 0 \wedge nr \leq 0 \wedge 1 = 0 \wedge 0 = v\ \ \wedge \{x_2\} = \{x_3\}\ ) \vee\\
&(\ nr > 0 \wedge nr > 0 \wedge 1 = 1 \wedge 0 = 0\ \ \wedge \{x_2\} = \{x_2\}\ )
\end{aligned}
$$

Note that the second and the third conjunction are unfeasible and hence can be removed from the formula. In the first one $\mathit{spawn\_res}(r + v_1)$ is replaced by x_1 (instead of x_3) since we can substitute v_1 by v as $v = v_1$ is imposed in the conjunction and in the fourth one $\mathit{spawn\_res}(v_1 + v)$ is replaced by x_2 (instead of x_4) since it is equal modulo AC to $\mathit{spawn\_res}(v + v_1)$. Then we finally have

$$(nr \leq 0 \wedge nr \leq 0 \wedge 0 = 0 \wedge v = v_1) \quad \vee \quad (nr > 0 \wedge nr > 0 \wedge 1 = 1 \wedge 0 = 0)$$

As before, $I_{\mathsf{res(v)},\mathsf{res}(v_1)} = \{(nr > 0), (v = v_1)\}$ is then obtained by computing all satisfying models. In the same way we obtain $I_{\mathsf{fib(v)},\mathsf{res(v_1)}} = I_{\mathsf{fib(v)},\mathsf{fib(v_1)}} = \{true\}$.

The following theorem states the soundness of the inference of ICs, that holds by construction of the SMT formula.

Theorem 3 (soundness of independence conditions). *Given the assumptions in Def. 6, if* $\exists C \in I_{r,q}$ *s.t.* $C(S)$ *holds, then* $S \xrightarrow{r \cdot q} S'$ *and* $S \xrightarrow{q \cdot r} S'$.

We will also get a necessary condition in those instances where the use of syntactic equality modulo AC on the multisets of created processes (as described above) is not loosing precision. This can be checked when building the encoding.

4.3 Other extensions

We abstract loops from the code of the blocks so that we can handle them as uninterpreted functions similarly to Def. 6. Basically, for each loop, we generate as many uninterpreted functions as variables it modifies (excluding local variables of the loop) plus one to express all processes created inside the loop. The functions have as arguments the variables accessed by the loop (again excluding local variables). This transformation allows us to represent that each variable might be affected by the execution of the loop over some parameters, and then check in the reverse trace whether we get to the loop over the same parameters.

Definition 7 (loop extraction for IC generation). *Let us consider a loop* L *that accesses* $x_1, \ldots, x_n$ *variables and modifies* $y_1, \ldots, y_m$ *variables (excluding local loop variables) and let* $l_1, \ldots, l_{m+1}$ *be fresh function symbol names. We replace* L *by the following code:*

$x'_1 = x_1; \ldots;\ x'_n = x_n;\ y_1 = l_1(x'_1, ..., x'_n); \ldots;\ y_m = l_m(x'_1, ..., x'_n);$
$\mathit{spawn}(f_{m+1}(x'_1, ..., x'_n));$ *(only if there are spawn operations inside the loop)*

Existing dependency analysis can be used to infer the subset of $x_1, \ldots, x_n$ that affects each y_i, achieving more precision with a small pre-computation overhead.

The treatment of method invocations (or function calls) to be executed atomically within the considered blocks can be done analogously to loops by introducing one fresh function for every (non-local) variable that is modified within the method call and one more for the result. The parameters of these new functions are the original ones plus one for each accessed (non-local) variable. After the transformations for both loops and calls described above, we have TSs with function calls that are treated as uninterpreted functions in a similar way to Def. 6. However these functions can now occur in the conditions and the assignments of the TS. To handle them, we use again a mapping P'' to remove all function calls from the TS and replace them by fresh integer variables. After that the encoding is like in Def. 6, and we obtain an SMT formula over LIA, which is again sent to the allSAT SMT solver. Once we have obtained the models we replace back the introduced fresh variables by the function calls using the mapping P''. Several simplifications on equalities involving function calls can be done before and after invoking the solver to improve the result. As a final remark, data structures like lists or maps have been handled by expressing their uses as function calls, hence obtaining constraints that include conditions on them.

5 Experiments

In this section we report on experimental results that compare the performance of three DPOR algorithms: SDPOR [1, 2], CSDPOR [3] and our proposal CDPOR. We have implemented and experimentally evaluated our method within the SYCO tool [3], a systematic testing tool for message-passing concurrent programs. SYCO can be used online through its web interface available at `http://costa.fdi.ucm.es/syco`. To generate the ICs, SYCO calls a new feature of the VeryMax program analyzer [7] which uses Barcelogic [6] as SMT solver. As benchmarks, we have borrowed the examples from [3] (available online from the previous url) that were used to compare SDPOR with CSDPOR. They are classical concurrent applications: several concurrent sorting algorithms (QS, MS, PS), concurrent Fibonacci Fib, distributed workers Pi, a concurrent registration system Reg and database DBP, and a consumer producer interaction BB. These benchmarks feature the typical concurrent programming methodology in which computations are split into smaller atomic subcomputations which concurrently interleave their executions, and which work on the same shared data. Therefore, the concurrent processes are highly interfering, and both inferring ICs and applying DPOR algorithms on them becomes challenging.

We have executed each benchmark with size increasing input parameters. A timeout of 60*sec* is used and, when reached, we write $>X$ to indicate that for the corresponding measure we encountered X units up to that point (i.e., it is at least X). Table 1 shows the results of the executions for 6 different inputs. Column Tr shows the number of traces, S the number of states that the algorithms explore, and T the time in *sec* it takes to compute them. For CDPOR, we also show the time T^{smt} of inferring the ICs (since the inference is performed once for all executions, it is only shown in the first row). Times are obtained on an Intel(R) Core(TM) i7 CPU at 2.5Ghz with 8GB of RAM (Linux Kernel 5.4.0). Columns $G^{\mathbf{s}}$ and $G^{\mathbf{cs}}$ show the time speedup of CDPOR over SDPOR and CSDPOR, respectively, computed by dividing each respective T by the time T of CDPOR. Column $G^{\mathbf{smt}}$ shows the time speedup over CSDPOR including T^{smt} in the time of CDPOR. We can see from the speedups that the gains of CDPOR increase exponentially in all examples with the size of the input. When compared with CSDPOR, we achieve reductions up to 4 orders of magnitude for the largest inputs on which CSDPOR terminates(e.g., Pi, QS). It is important to highlight that the number of non-unitary sequences stored in sleep sets is 0 in every benchmark except in BB for which it remains quite low (namely for BB(11) the peak is 22).

	SDPOR			**CSDPOR**			**CDPOR**				**Speed-up**		
Bench.	**Tr**	**S**	**T**	**Tr**	**S**	**T**	**Tr**	**S**	**T**	$\mathbf{T^{smt}}$	$\mathbf{G^{s}}$	$\mathbf{G^{cs}}$	$\mathbf{G^{smt}}$
Fib(6)	3k	26k	7.7	1	244	0.1	1	50	0.03	0.12	366	4	0.6
Fib(7)	>13k	>160k	60.0	1	551	0.3	1	82	0.05		>1364	6	1.4
Fib(8)	>8k	>101k	60.0	1	2k	0.7	1	134	0.12		>527	6	3.0
Fib(9)	>4k	>51k	60.0	1	3k	2.8	1	218	0.25		>242	12	7.5
Fib(10)	>2k	>27k	60.0	1	8k	11.5	1	354	0.69		>88	17	14.3
Fib(14)	>10	>3k	60.0	>1	>4k	60.0	1	3k	42.67		>2	>2	>1.5
QS(10)	512	9k	2.6	1	4k	1.0	1	38	0.02	11.99	199	71	0.1
QS(13)	5k	91k	29.5	1	29k	7.9	1	50	0.03		1474	395	0.7
QS(15)	>7k	>157k	60.0	1	115k	42.6	1	58	0.05		>1500	1064	3.6
QS(20)	>4k	>98k	60.0	>1	>148k	60.0	1	78	0.04		>1539	>1539	>5.0
QS(25)	>3k	>96k	60.0	>1	>133k	60.0	1	98	0.06		>1017	>1017	>5.0
QS(200)	>5	>2k	60.0	>1	>87k	60.0	1	798	4.45		>14	>14	>3.7
MS(10)	628	7k	2.9	1	187	0.1	1	42	0.02	0.12	175	6	0.7
MS(30)	>6k	>55k	60.0	1	974	1.0	1	118	0.13		>484	8	4.0
MS(65)	>2k	>16k	60.0	1	3k	3.5	1	258	0.47		>131	8	6.1
MS(100)	>2k	>15k	60.0	>1	>19k	60.0	1	398	0.97		>63	>63	>55.6
MS(150)	>2k	>21k	60.0	>1	>18k	60.0	1	598	2.21		>28	>28	>26.0
MS(220)	>341	>6k	60.0	>1	>5k	60.0	1	878	4.49		>14	>14	>13.1
Pi(7)	6k	49k	16.2	74	2k	0.4	1	23	0.02	0.05	1243	27	5.6
Pi(8)	>10k	>105k	60.0	264	5k	1.7	1	26	0.02		>4616	128	26.9
Pi(9)	>11k	>120k	60.0	2k	19k	7.0	1	29	0.02		>4000	465	108.9
Pi(10)	>10k	>128k	60.0	6k	91k	45.2	1	32	0.02		>3530	2655	683.7
Pi(12)	>9k	>122k	60.0	>7k	>128k	60.0	1	38	0.03		>2400	>2400	>810.9
Pi(20)	>5k	>101k	60.0	>5k	>115k	60.0	1	62	0.09		>723	>723	>454.6
PS(4)	288	2k	0.4	2	41	0.1	1	16	0.01	0.59	72	2	0.1
PS(5)	35k	156k	43.2	8	142	0.1	1	22	0.01		5391	5	0.1
PS(6)	>32k	>141k	60.0	72	2k	0.4	1	29	0.02		>4286	28	0.7
PS(7)	>29k	>130k	60.0	2k	28k	7.5	1	37	0.03		>2858	357	12.3
PS(9)	>25k	>109k	60.0	>11k	>165k	60.0	1	56	0.06		>1053	>1053	>92.9
PS(11)	>23k	>103k	60.0	>9k	>132k	60.0	1	79	0.09		>690	>690	>88.8
DBP(5)	243	8k	2.0	133	4k	1.0	32	193	0.08	0.09	27	14	6.2
DBP(6)	729	33k	8.2	308	11k	3.2	64	386	0.16		53	21	13.3
DBP(7)	3k	134k	36.9	699	32k	10.8	128	771	0.33		113	33	26.2
DBP(8)	>4k	>157k	60.0	2k	91k	36.1	256	2k	0.79		>77	47	41.6
DBP(10)	>6k	>116k	60.0	>4k	>125k	60.0	2k	7k	3.23		>19	>19	>18.2
DBP(12)	>9k	>79k	60.0	>8k	>111k	60.0	5k	25k	15.79		>4	>4	>3.8
BB(6)	924	4k	1.3	215	2k	0.5	64	382	0.91	0.18	2	1	0.4
BB(7)	4k	13k	4.3	580	4k	1.2	128	830	1.49		3	1	0.8
BB(8)	13k	49k	17.2	2k	11k	3.3	256	2k	2.79		7	2	1.1
BB(9)	>41k	>156k	60.0	5k	30k	9.0	512	4k	6.15		>10	2	1.5
BB(10)	>46k	>176k	60.0	12k	81k	23.6	2k	9k	12.50		>5	2	1.9
BB(11)	>44k	>169k	60.0	>44k	>169k	60.0	3k	18k	25.74		>3	>3	>2.4

Table 1. Experimental evaluation

W.r.t. SDPOR, we achieve reductions of 4 orders of magnitude even for smaller inputs for which SDPOR terminates (e.g., PS). Note that since most examples reach the timeout, the gains are at least the ones we show, thus the concrete numbers shown should not be taken into account. In some examples (e.g., BB, MS), though the gains are linear for the small inputs, when the size of the problem increases both SDPOR and CSDPOR time out, while CDPOR can still handle them efficiently.

Similar reductions are obtained for number of states explored. In this case, the system times out when it has memory problems, and the computation stops progressing (hence the number of explored states does not increase with the input any more). As regards the time to infer the annotations T^{smt}, we observe that in most cases it is negligible compared to the exploration time of the other methods. QS is the only example that needs some seconds to be solved and this is due to the presence of several nested conditional statements combined with the use of built-in functions for lists, which makes the generated SMT encoding harder for the solver and the subsequent simplification step. Note that the inference is a pre-process which does not add complexity to the actual DPOR algorithm.

6 Related work and conclusions

The notion of conditional independence in the context of POR was first introduced in [12, 16]. Also [13] provides a similar strengthened dependency definition. CSDPOR was the first approach to exploit this notion within the state-of-the-art DPOR algorithm. We advance this line of research by fully integrating conditional independence within the DPOR framework by using *independence constraints* (ICs) together with the notion of *transitive uniform* conditional independence –which ensures the ICs hold along the whole execution sequence. Both ICs and transitive uniformity can be approximated statically and checked dynamically, making them effectively applicable within the dynamic framework. The work in [15, 21] generated for the first time ICs for processes with a single instruction following some predefined patterns. This is a problem strictly simpler than our inference of ICs both in the type of IC generated (restricted to the patterns) and on the single-instruction blocks they consider. Furthermore, our approach using an AllSAT SMT solver is different from the CEGAR approach in [5]. The ICs are used in [15, 21] for SMT-based bounded model checking, an approach to model checking fundamentally different from our stateless model checking setting. As a consequence ICs are used in a different way, in our case with no bounds on number of processes, nor derivation lengths, but requiring a uniformity condition on independence in order to ensure soundness. Maximal causality reduction [14] is technically quite different from CDPOR as it integrates SMT solving within the dynamic algorithm.

Finally, data-centric DPOR (DCDPOR) [8] presents a new DPOR algorithm based on a different notion of dependency according to which the equivalence classes of derivations are based on the pairs read-write of variables. Consider the following three simple processes $\{p, q, r\}$ and the initial state $x = 0$: p: `write(x=5)`, q: `write(x=5)`, r: `read(x)`. In DCDPOR, we have only three different observation functions: (r, x) (reading the initial value), (r, p) (reading the value that p writes), (r, q) (reading the value that q writes). Therefore, this notion of relational independence is finer grained than the traditional one in DPOR. However, DCDPOR does not consider conditional dependency, i.e., it does not realize that (r, p) and (r, q) are equivalent, and hence only two explorations are required (and explored by CDPOR). In conclusion, our approach and DCDPOR can complement each other: our approach would benefit from using a dependency based on the read-write pairs as proposed in DCDPOR, and DCDPOR would benefit from using conditional independence as proposed in our work. It remains as future work to study this integration. Related to DCDPOR, [4] extends optimal DPOR with observers. For the previous example, [4] needs to explore five executions: $r.p.q$ and $r.q.p$, are equivalent because p and q do not have any observer. Another improvement orthogonal to ours is to inspect dependencies over chains of events, as in [17] and [19].

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017.
2. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal Dynamic Partial Order Reduction. In *POPL*, pages 373–384, 2014.
3. Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In *CAV*, pages 526–543, 2017.
4. Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *TACAS. Part II*, pages 229–248, 2018.
5. Kshitij Bansal, Eric Koskinen, and Omer Tripp. Commutativity condition refinement, 2015.
6. Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *CAV*, pages 294–298, 2008.
7. Cristina Borralleras, Daniel Larraz, Albert Oliveras, José Miguel Rivero, Enric Rodríguez-Carbonell, and Albert Rubio. VeryMax: Tool description for termCOMP 2016. In *WST*, 2016.
8. Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Kapil Vaidya, and Nishant Sinha. Data-centric dynamic partial order reduction. In *POPL 2018*, 2018.
9. Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.
10. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
11. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.
12. Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *CAV*, pages 438–449, 1993.
13. Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In *VMCAI*, pages 246–265, 2017.
14. Shiyou Huang and Jeff Huang. Speeding up maximal causality reduction with static dependency analysis. In *ECOOP*, pages 16:1–16:22, 2017.
15. Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, pages 398–413, 2009.
16. Shmuel Katz and Doron A. Peled. Defining conditional independence using collapses. *TCS*, 101(2):337–359, 1992.
17. Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. *CoRR*, abs/1802.03950, 2018.
18. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
19. César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *CONCUR*, pages 456–469, 2015.
20. Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets*, pages 491–515, 1990.
21. Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS*, pages 382–396, 2008.