Unfolding-based Partial Order Reduction*

César Rodríguez[1], Marcelo Sousa[2], Subodh Sharma[3], and Daniel Kroening[4]

1 **Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, France**
2,4 **Department of Computer Science, University of Oxford, UK**
3 **Indian Institute of Technology Delhi, India**

Abstract

Partial order reduction (POR) and net unfoldings are two alternative methods to tackle state-space explosion caused by concurrency. In this paper, we propose the combination of both approaches in an effort to combine their strengths. We first define, for an abstract execution model, unfolding semantics parameterized over an arbitrary independence relation. Based on it, our main contribution is a novel stateless POR algorithm that explores at most one execution per Mazurkiewicz trace, and in general, can explore exponentially fewer, thus achieving a form of *super-optimality*. Furthermore, our unfolding-based POR copes with non-terminating executions and incorporates state caching. On benchmarks with busy-waits, among others, our experiments show a dramatic reduction in the number of executions when compared to a state-of-the-art DPOR.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Partial-order reduction, unfoldings, concurrency, model checking

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2015.456

1 Introduction

Efficient exploration of the state space of a concurrent system is a fundamental problem in automated verification. Concurrent actions often interleave in intractably many ways, quickly populating the state space with many equivalent but unequal states. Existing approaches to address this problem can essentially be classified as either partial-order reduction techniques (PORs) or unfolding methods.

Conceptually, POR methods [19, 7, 6, 8, 21, 20, 2, 1] exploit the fact that executing certain transitions can be postponed because their result is independent of the execution sequence taken in their stead. They execute a provably-sufficient subset of transitions enabled at every state, computed either statically [19, 7] or dynamically [6, 2]. The latter methods, referred as dynamic PORs (DPORs), are often *stateless* (i.e., they only store one execution in memory). By contrast, unfolding approaches [14, 5, 3, 10] model execution by partial orders, bound together by a conflict relation. They construct finite, complete prefixes by a saturation procedure, and cope with non-terminating executions using cutoff events [5, 3].

POR can employ highly sophisticated decision procedures to determine a sufficient subset of the transitions to fire, and in most cases [7, 6, 8, 21, 20, 2, 1] the *commutativity of transitions* is the enabling mechanism underlying the chosen method. Commutativity, or independence, is thus a mechanism and not necessarily an irreplaceable component of a

* This research was supported by ERC project 280053 (CPROVER).

26th International Conference on Concurrency Theory (CONCUR 2015).
Editors: Luca Aceto and David de Frutos Escrig; pp. 456–469

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

POR [19, 9].[1] Conceptually, PORs that exploit commutativity establish an equivalence relation on the sequential executions of the system and explore at least one representative of each class, thus discarding equivalent executions. In this work we restrict our attention to exclusively PORs that exploit commutativity.

Despite impressive advances in the field, both unfoldings and PORs have shortcomings. We now give six of them. Current unfolding algorithms (1) need to solve an NP-complete problem when adding events to the unfolding [14], which seriously limits the performance of existing unfolders as the structure grows. They are also (2) inherently *stateful*, i.e., they cannot selectively discard visited events from memory, quickly running out of it. PORs, on the other hand, explore Mazurkiewicz traces [13], which (3) often outnumber the events in the corresponding unfolding by an exponential factor (e.g., Fig. 2 (d) gives an unfolding with $2n$ events and $\mathcal{O}(2^n)$ traces). Furthermore, DPORs often (4) explore the same states repeatedly [20], and combining them with stateful search, although achieved for non-optimal DPOR [20, 21], is difficult due to the dynamic nature of DPOR [21]. More on this in Example 1. The same holds when extending DPORs to (5) cope with non-terminating executions (note that a solution to (4) does not necessarily solve (5)). Lastly, (6) existing stateless PORs do not make full use of the available memory.

Either readily available solutions or promising directions to address these six problems can be found in, respectively, the opposite approach. PORs inexpensively add events to the current execution, contrary to unfoldings (1). They easily discard events from memory when backtracking, which addresses (2). On the other hand, while PORs explore Mazurkiewicz traces (*maximal configurations*), unfoldings explore events (*local configurations*), thus addressing (3). Explorations of repeated states and pruning of non-terminating executions is elegantly achieved in unfoldings by means of cutoff events. This solves (4) and (5).

Some of these solutions indeed seem, at present, incompatible with each other. We do not claim that the combination of POR and unfoldings immediately addresses the problems above. However, since both unfoldings and PORs share many fundamental similarities, tackling these problems in a unified framework is likely to shed light on them.

This paper lays out a DPOR algorithm on top of an unfolding structure. Our main result is a novel stateless, optimal DPOR that explores every Mazurkiewicz trace at most once, and often many fewer, owing to cutoff events. It also copes with non-terminating systems and exploits all available RAM with a *cache memory* of events, speeding up revisiting events. This provides a solution to (4), (5), (6), and a partial solution to (3). Our algorithm can alternatively be viewed as a stateless unfolding exploration, partially addressing (1) and (2).

Our result reveals DPORs as algorithms exploring an object that has richer structure than a plain directed graph. Specifically, unfoldings provide a solid notion of event *across multiple executions*, and a clear notion of conflict. Our algorithm indirectly maps important POR notions to concepts in unfolding theory.

▶ **Example 1.** We illustrate problems (3), (4), and (5), and explain how our DPOR deals with them. The following code is the skeleton of a producer-consumer program. Two concurrent producers write, resp., to `buf1` and `buf2`. The consumer accesses the buffers in sequence.

```
while (1):                    | while (1):
  lock(m1)                    |   lock(m2)
  if (buf1 < MAX): buf1++     |   if (buf2 < MAX): buf2++
  unlock(m1)                  |   unlock(m2)
```

[1] For instance, all PORs based on persistent sets [7] are based on commutativity.

```
while (1):
  lock(m1)
  if (buf1 > MIN): buf1--
  unlock(m1)
  // same for m2, buf2
```

Lock and unlock operations on both mutexes `m1` and `m2` create many Mazurkiewicz traces. However, most of them have isomorphic *suffixes*, e.g., producing two items in `buf1` and consuming one reaches the same state as only producing one. After the common state, both traces explore identical behaviours and only one needs to be explored. We use cutoff events, inherited from unfolding theory [5, 3], to dynamically stop the first trace and continue only with the second. This addresses (4) and (5), and partially deals with (3). Observe that cutoff events are a form of semantic pruning, in contrast to the syntactic pruning introduced by, e.g., bounding the depth of loops, a common technique for coping with non-terminating executions in DPOR. With cutoffs, the exploration can build unreachability *proofs*, while depth bounding renders DPOR incomplete, i.e., it limits DPOR to finding bugs.

Our first step is to formulate PORs and unfoldings in the same framework. PORs are often presented for abstract execution models, while unfoldings have mostly been considered for Petri nets, where the definition is entangled with the syntax of the net. We make a second contribution here. We define, for a general execution model, event structure semantics [16] parametric on a given independence relation.

Section 2 sets up basic notions and § 3 presents our parametric event-structure semantics. In § 4 we introduce our DPOR,§ 5 improves it with cutoff detection and discusses event caching. Experimental results are in § 6 and related work in § 7. We conclude in § 8. All lemmas cited along the paper and proofs of all stated results can be found in the extended version [17].

2 Execution Model and Partial Order Reductions

We set up notation and recall general ideas of POR. We consider an abstract model of (concurrent) computation. A *system* is a tuple $M := \langle \Sigma, T, \tilde{s} \rangle$ formed by a set Σ of *global states*, a set T of *transitions* and some *initial global state* $\tilde{s} \in \Sigma$. Each transition $t : \Sigma \to \Sigma$ in T is a *partial* function accounting for how the occurrence of t transforms the state of M.

A transition $t \in T$ is *enabled* at a state s if $t(s)$ is defined. Such t can *fire* at s, producing a new state $s' := t(s)$. We let $enabl(s)$ denote the set of transitions enabled at s. The *interleaving semantics* of M is the directed, edge-labelled graph $\mathcal{S}_M := \langle \Sigma, \to, \tilde{s} \rangle$ where Σ are the global states, $\tilde{s}$ is the initial state and $\to \subseteq \Sigma \times T \times \Sigma$ contains a triple $\langle s, t, s' \rangle$, denoted by $s \xrightarrow{t} s'$, iff t is enabled at s and $s' = t(s)$. Given two states $s, s' \in \Sigma$, and $\sigma := t_1.t_2 \dots t_n \in T^*$ (t_1 concatenated with t_2, ... until t_n), we denote by $s \xrightarrow{\sigma} s'$ the fact that there exist states $s_1, \dots, s_{n-1} \in \Sigma$ such that $s \xrightarrow{t_1} s_1$, ..., $s_{n-1} \xrightarrow{t_n} s'$.

A *run* (or *interleaving*, or *execution*) of M is any sequence $\sigma \in T^*$ such that $\tilde{s} \xrightarrow{\sigma} s$ for some $s \in \Sigma$. We denote by $state(\sigma)$ the state s that σ reaches, and by $runs(M)$ the set of runs of M, also referred to as the *interleaving space*. A state $s \in \Sigma$ is *reachable* if $s = state(\sigma)$ for some $\sigma \in runs(M)$; it is a *deadlock* if $enabl(s) = \varnothing$, and in that case σ is called *deadlocking*. We let $reach(M)$ denote the set of reachable states in M. For the rest of the paper, we fix a system $M := \langle \Sigma, T, \tilde{s} \rangle$ and assume that $reach(M)$ is finite.

The core idea behind POR[2] is that certain transitions can be seen as commutative

[2] To be completely correct we should say "*POR that exploits the independence of transitions*".

operators, i.e., changing their order of occurrence does not change the result. Given two transitions $t, t' \in T$ and one state $s \in \Sigma$, we say that t, t' *commute at* s iff

- if $t \in \mathit{enabl}(s)$ and $s \xrightarrow{t} s'$, then $t' \in \mathit{enabl}(s)$ iff $t' \in \mathit{enabl}(s')$; and
- if $t, t' \in \mathit{enabl}(s)$, then there is a state s' such that $s \xrightarrow{t.t'} s'$ and $s \xrightarrow{t'.t} s'$.

For instance, the lock operations on `m1` and `m2` (Example 1), commute on every state, as they update different variables. Commutativity of transitions at states identifies an equivalence relation on the set $\mathit{runs}(M)$. Two runs σ and σ' of the same length are *equivalent*, written $\sigma \equiv \sigma'$, if they are the same sequence modulo swapping commutative transitions. Thus equivalent runs reach the same state. POR methods explore a fragment of $\mathcal{S}_M$ that contains at least one run in the equivalence class of each run that reaches each deadlock state. This is achieved by means of a so-called *selective search* [7]. Since employing commutativity can be expensive, PORs often use *independence relations*, i.e., sound under-approximations of the commutativity relation. In this work, partially to simplify presentation, we use unconditional independence.

Formally, an *unconditional independence relation* on M is any symmetric and irreflexive relation $\diamond \subseteq T \times T$ such that if $t \diamond t'$, then t and t' commute at *every* state $s \in \mathit{reach}(M)$. If t, t' are not independent according to $\diamond$, then they are *dependent*, denoted by $t \circledast t'$.

Unconditional independence identifies an equivalence relation $\equiv_\diamond$ on the set $\mathit{runs}(M)$. Formally, $\equiv_\diamond$ is defined as the transitive closure of the relation $\equiv^1_\diamond$, which in turn is defined as $\sigma \equiv^1_\diamond \sigma'$ iff there is $\sigma_1, \sigma_2 \in T^*$ such that $\sigma = \sigma_1.t.t'.\sigma_2$, $\sigma' = \sigma_1.t'.t.\sigma_2$ and $t \diamond t'$. From the properties of $\diamond$, one can immediately see that $\equiv_\diamond$ refines $\equiv$, i.e., if $\sigma \equiv_\diamond \sigma'$, then $\sigma \equiv \sigma'$.

Given a run $\sigma \in \mathit{runs}(M)$, the equivalence class of $\equiv_\diamond$ to which σ belongs is called the *Mazurkiewicz trace* of σ [13], denoted by $\mathcal{T}_{\diamond,\sigma}$. Each trace $\mathcal{T}_{\diamond,\sigma}$ can equivalently be seen as a labelled partial order $\mathcal{D}_{\diamond,\sigma}$, traditionally called the *dependence graph* (see [13] for a formalization), satisfying that a run belongs to the trace iff it is a linearization of $\mathcal{D}_{\diamond,\sigma}$.

Sleep sets [7] are another method for state-space reduction. Unlike selective exploration, they prune successors by looking at the past of the exploration, not the future.

3 Parametric Partial Order Semantics

An unfolding is, conceptually, a tree-like structure of partial orders. In this section, given an independence relation $\diamond$ (our parameter) and a system M, we define an unfolding semantics $\mathcal{U}_{M,\diamond}$ with the following property: each constituent partial order of $\mathcal{U}_{M,\diamond}$ will correspond to one dependence graph $\mathcal{D}_{\diamond,\sigma}$, for some $\sigma \in \mathit{runs}(M)$. For the rest of this paper, let $\diamond$ be an arbitrary unconditional independence relation on M. We use prime event structures [16], a non-sequential, event-based model of concurrency, to define the unfolding $\mathcal{U}_{M,\diamond}$ of M.

▶ **Definition 2** (LES). Given a set A, an *A-labelled event structure* (A-LES, or LES in short) is a tuple $\mathcal{E} := \langle E, <, \#, h\rangle$ where E is a set of *events*, $< \subseteq E \times E$ is a strict partial order on E, called *causality relation*, $h\colon E \to A$ labels every event with an element of A, and $\# \subseteq E \times E$ is the symmetric, irreflexive *conflict relation*, satisfying

- for all $e \in E$, $\{e' \in E : e' < e\}$ is finite, and (1)
- for all $e, e', e'' \in E$, if $e \# e'$ and $e' < e''$, then $e \# e''$. (2)

The *causes* of an event $e \in E$ are the set $\lceil e \rceil := \{e' \in E : e' < e\}$ of events that need to happen before e for e to happen. A *configuration* of $\mathcal{E}$ is any finite set $C \subseteq E$ satisfying:

- (causally closed) for all $e \in C$ we have $\lceil e \rceil \subseteq C$; (3)
- (conflict free) for all $e, e' \in C$, it holds that $\neg e \# e'$. (4)

Intuitively, configurations represent partially-ordered executions. In particular, the *local configuration* of e is the $\subseteq$-minimal configuration that contains e, i.e. $\lceil e \rceil := \lceil e \rceil \cup \{e\}$.

We denote by $\mathit{conf}(\mathcal{E})$ the set of configurations of $\mathcal{E}$. Two events e, e' are in *immediate conflict*, $e \#^i e'$, iff $e \# e'$ and both $\lceil e \rceil \cup \lceil e' \rceil$ and $\lceil e \rceil \cup \lceil e' \rceil$ are configurations. Lastly, given two LESs $\mathcal{E} := \langle E, <, \#, h \rangle$ and $\mathcal{E}' := \langle E', <', \#', h' \rangle$, we say that $\mathcal{E}$ is a *prefix* of $\mathcal{E}'$, written $\mathcal{E} \trianglelefteq \mathcal{E}'$, when $E \subseteq E'$, $<$ and $\#$ are the projections of $<'$ and $\#'$ to E, and $E \supseteq \{e' \in E' : e' < e \wedge e \in E\}$.

Our semantics will unroll the system M into a LES $\mathcal{U}_{M,\diamond}$ whose events are labelled by transitions of M. Each configuration of $\mathcal{U}_{M,\diamond}$ will correspond to the dependence graph $\mathcal{D}_{\diamond,\sigma}$ of some $\sigma \in \mathit{runs}(M)$. For a LES $\langle E, <, \#, h \rangle$, we define the *interleavings* of C as $\mathit{inter}(C) := \{h(e_1), \ldots, h(e_n) : e_i, e_j \in C \wedge e_i < e_j \implies i < j\}$. Although for arbitrary LES $\mathit{inter}(C)$ may contain sequences not in $\mathit{runs}(M)$, the definition of $\mathcal{U}_{M,\diamond}$ will ensure that $\mathit{inter}(C) \subseteq \mathit{runs}(M)$. Additionally, since all sequences in $\mathit{inter}(C)$ belong to the same trace, all of them reach the same state. Abusing the notation, we define $\mathit{state}(C) := \mathit{state}(\sigma)$ if $\sigma \in \mathit{inter}(C)$. The definition is neither well-given nor unique for arbitrary LES, but will be so for the unfolding.

We now define $\mathcal{U}_{M,\diamond}$. Each event will be inductively identified by a canonical name of the form $e := \langle t, H \rangle$, where $t \in T$ is a transition of M and H a configuration of $\mathcal{U}_{M,\diamond}$. Intuitively, e represents the occurrence of t after the *history* (or the causes) $H := \lceil e \rceil$. The definition will be inductive. The base case inserts into the unfolding a special *bottom event* $\bot$ on which every event causally depends. The inductive case iteratively extends the unfolding with one event. We define the set $\mathcal{H}_{\mathcal{E},\diamond,t}$ of candidate *histories* for a transition t in an LES $\mathcal{E}$ as the set which contains exactly all configurations H of $\mathcal{E}$ such that

- transition t is enabled at $\mathit{state}(H)$, and
- either $H = \{\bot\}$ or all $<$-maximal events e in H satisfy that $h(e) \otimes t$,

where h is the labelling function in $\mathcal{E}$. Once an event e has been inserted into the unfolding, its associated transition $h(e)$ may be dependent with $h(e')$ for some e' already present and outside the history of e. Since the order of occurrence of e and e' matters, we need to prevent their occurrence within the same configuration, as configurations represent equivalent executions. We therefore introduce a conflict between e and e'. The set $\mathcal{K}_{\mathcal{E},\diamond,e}$ of *events conflicting* with $e := \langle t, H \rangle$ thus contains any event e' in $\mathcal{E}$ with $e' \notin [e]$ and $e \notin [e']$ and $t \otimes h(e')$.

Following common practice [4], the definition of $\mathcal{U}_{M,\diamond}$ proceeds in two steps. We first define (Def. 3) the collection of all prefixes of the unfolding. Then we show that there exists only one $\trianglelefteq$-maximal element in the collection, and define it to be *the* unfolding (Def. 4).

▶ **Definition 3** (Finite unfolding prefixes)**.** The set of *finite unfolding prefixes* of M under the independence relation $\diamond$ is the smallest set of LESs that satisfies the following conditions:

1. The LES having exactly one event $\bot$, empty causality and conflict relations, and $h(\bot) := \varepsilon$ is an unfolding prefix.
2. Let $\mathcal{E}$ be an unfolding prefix containing a history $H \in \mathcal{H}_{\mathcal{E},\diamond,t}$ for some transition $t \in T$. Then, the LES $\langle E, <, \#, h \rangle$ resulting from extending $\mathcal{E}$ with a new event $e := \langle t, H \rangle$ and satisfying the following constraints is also an unfolding prefix of M:
   - for all $e' \in H$, we have $e' < e$;
   - for all $e' \in \mathcal{K}_{\mathcal{E},\diamond,e}$, we have $e \# e'$; and $h(e) := t$.

Intuitively, each unfolding prefix contains the dependence graph (configuration) of one or more executions of M (of finite length). The unfolding starts from $\bot$, the "root" of the tree, and then iteratively adds events enabled by some configuration until saturation, i.e., when no more events can be added. Observe that the number of unfolding prefixes as per

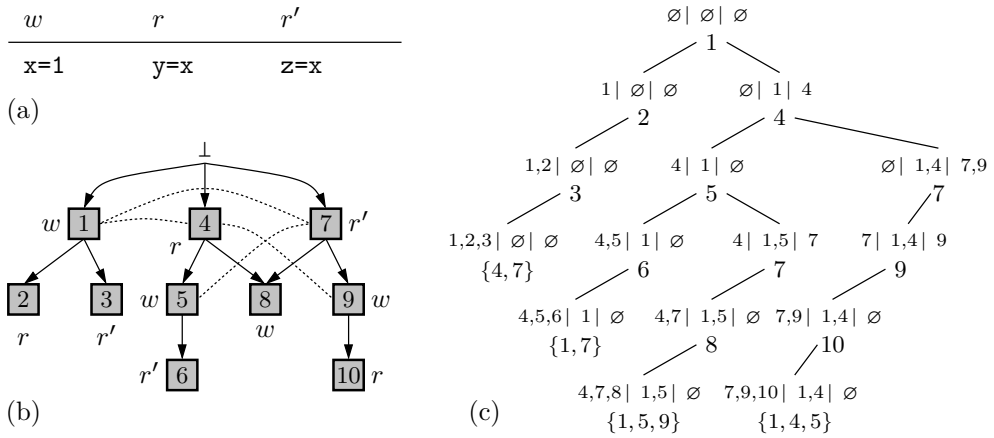

Figure 1 Running example. (a) A concurrent program; (b) its unfolding semantics. (c) The exploration performed by Alg. 1, where each node C| D| A represents one call to the function `Explore`(C, D, A). The set X underneath each leaf node is such that the value of variable U in Alg. 1 at the leaf is $U = C \cup D \cup X$. At ∅| ∅| ∅ , the alternative taken is $\{4\}$, and at 4| 1| ∅ it is $\{7\}$.

Def. 3 will be finite iff all runs of M terminate. Due to lack of space, we give the definition of *infinite* unfolding prefixes in the extended version [17], as the main ideas of this section are well conveyed using only finite prefixes. In the sequel, by *unfolding prefix* we mean a finite or infinite one.

Our first task is checking that each unfolding prefix is indeed a LES [17, Lemma 14]. Next one shows that the configurations of every unfolding prefix correspond the Mazurkiewicz traces of the system, i.e., for any configuration C, $\mathit{inter}(C) = \mathcal{T}_{\diamond,\sigma}$ for some $\sigma \in \mathit{runs}(M)$ [17, Lemma 16]. This implies that the definition of $\mathit{inter}(C)$ and $\mathit{state}(C)$ is well-given when C belongs to an unfolding prefix. The second task is defining *the* unfolding $\mathcal{U}_{M,\diamond}$ of M. Here, we prove that the set of unfolding prefixes equipped with relation $\trianglelefteq$ forms a complete join-semilattice [17, Lemma 17]. This implies the existence of a unique $\trianglelefteq$-maximal element:

▶ **Definition 4** (Unfolding)**.** The *unfolding* $\mathcal{U}_{M,\diamond}$ of M under the independence relation $\diamond$ is the *unique* $\trianglelefteq$-maximal element in the set of unfolding prefixes of M under $\diamond$.

Finally we verify that the definition is well given and that the unfolding is *complete*, i.e., every run of the system is represented by a unique configuration of the unfolding.

▶ **Theorem 5.** *The unfolding $\mathcal{U}_{M,\diamond}$ exists and is unique. Furthermore, for any non-empty run σ of M, there exists a unique configuration C of $\mathcal{U}_{M,\diamond}$ such that $\sigma \in \mathit{inter}(C)$.*

▶ **Example 6** (Programs)**.** Figure 1 (a) gives a concurrent program, where process w writes a global variable and processes r and r' read it. We can associate various semantics to it. Under an empty independence relation, the unfolding would be the computation tree, where executions would be totally ordered. Considering (the unique transition of) r and r' independent, and w dependent on them, we get the unfolding given in Fig. 1 (b).

Events are numbered from 1 to 10, and labelled with a transition. Arrows represent causality between events and dotted lines immediate conflict. The Mazurkiewicz trace of each deadlocking execution is represented by a unique $\subseteq$-maximal configuration, e.g., the run $w.r.r'$ yields configuration $\{1, 2, 3\}$, where the two possible interleavings reach the same

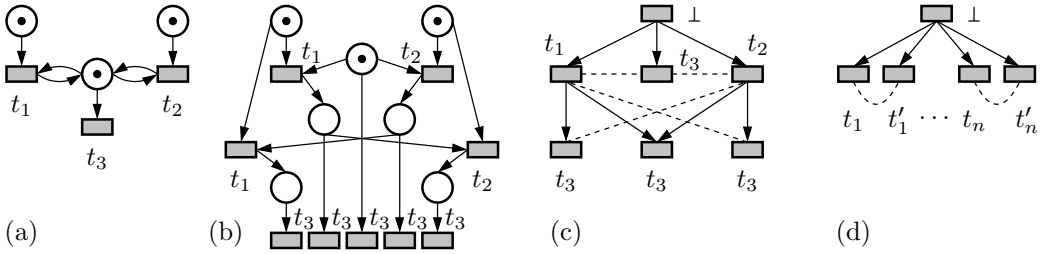

Figure 2 (a) A Petri net; (b) its classic unfolding; (c) our parametric semantics.

state. For instance, the canonic name of event 1 is $\langle w, \{\bot\}\rangle$ and of event 2 it is $\langle r, \{\bot, 1\}\rangle$. Let $\mathcal{P}$ be the unfolding prefix that contains events $\{\bot, 1, 2\}$. Definition 3 can extend it with three possible events: 3, 4, and 7. Consider transition r'. Three configurations of $\mathcal{P}$ enable r': $\{\bot\}, \{\bot, 1\}$ and $\{\bot, 1, 2\}$. But since $\neg(h(2) \otimes r')$, only the first two will be in $\mathcal{H}_{\mathcal{P},\diamond,r'}$, resulting in events $3 := \langle r', \{\bot, 1\}\rangle$ and $7 := \langle r', \{\bot\}\rangle$. Also, $\mathcal{K}_{\mathcal{P},\diamond,7}$ is $\{1\}$, as $w \otimes r'$. The four maximal configurations are $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{4, 7, 8\}$ and $\{7, 9, 10\}$, resp. reaching the states $\langle x, y, z\rangle = \langle 1, 1, 1\rangle$, $\langle 1, 0, 1\rangle$, $\langle 1, 0, 0\rangle$ and $\langle 1, 1, 0\rangle$, assuming that variables start at 0.

▶ **Example 7** (Comparison to Petri Net Unfoldings)**.** In contrast to our parametric semantics, classical unfoldings of Petri nets [5] use a fixed independence relation, specifically the complement of the following one (valid only for safe nets): given two transitions t and t',

$$t \otimes_n t' \text{ iff } (t^\bullet \cap {}^\bullet t' \neq \varnothing) \text{ or } (t'^\bullet \cap {}^\bullet t \neq \varnothing) \text{ or } ({}^\bullet t' \cap {}^\bullet t \neq \varnothing),$$

where ${}^\bullet t$ and $t^\bullet$ are respectively the *preset* and *postset* of t. Classic Petri net unfoldings (of safe nets) are therefore a specific instantiation of our semantics. A well known challenge for classic unfoldings are transitions that "*read*" places, e.g., t_1 and t_2 in Fig. 2 (a). Since $t_1 \otimes_n t_2$, the classic unfolding, Fig. 2 (b), sequentializes all their occurrences. A solution for this issue is the so-called *place replication (PR) unfolding* [15], or alternatively *contextual unfoldings* (which anyway internally are asymptotically the same size as the PR-unfolding).

This problem vanishes with our parametric unfolding. It suffices to use a dependency relation $\otimes_n' \subset \otimes_n$ that makes transitions that "*read*" common places independent. The result is that our unfolding, Fig. 2 (c), can be of the same size as the PR-unfolding, i.e., exponentially more compact than the classic unfolding. For instance, when Fig. 2 (a) is generalized to n *reading* transitions, the classic unfolding would have $\mathcal{O}(n!)$ copies of t_3, while ours would have $\mathcal{O}(2^n)$. The point here is that our semantics naturally accommodates a more suitable notion of independence without resorting to specific ad-hoc tricks.

Furthermore, although this work is restricted to *unconditional* independence, we conjecture that an adequately restricted *conditional* dependence would suffice, e.g., the one of [12]. Gains achieved in such setting would be difficult with classic unfoldings.

4 Stateless Unfolding Exploration Algorithm

We present a DPOR algorithm to explore an arbitrary event structure (e.g., the one of § 3) instead of sequential executions. Our algorithm explores one configuration at a time and organizes the exploration into a binary tree. Figure 1 (c) gives an example. The algorithm is optimal [2], in the sense that no configuration is ever visited twice in the tree.

Algorithm 1: An unfolding-based POR exploration algorithm.

```
1  Initially, set U := {⊥}, set G := ∅, and call Explore({⊥}, ∅, ∅).
2  Procedure Explore(C, D, A)
3      Extend(C)
4      if en(C) = ∅ return
5      if A = ∅
6          Choose e from en(C)
7      else
8          Choose e from A ∩ en(C)
9      Explore(C ∪ {e}, D, A ∖ {e})
10     if ∃J ∈ Alt(C, D ∪ {e})
11         Explore(C, D ∪ {e}, J ∖ C)
12     Remove(e, C, D)
13 Procedure Extend(C)
14     Add ex(C) to U
15 Procedure Remove(e, C, D)
16     Move {e} ∖ Q_{C,D,U} from U to G
17     foreach ê ∈ #^i_U(e)
18         Move [ê] ∖ Q_{C,D,U} from U to G
```

For the rest of the paper, let $\mathcal{U}_{\diamond,M} := \langle E, <, \#, h\rangle$ be the unfolding of M under $\diamond$, which we abbreviate as $\mathcal{U}$. For this section we assume that $\mathcal{U}$ is finite, i.e., that all computations of M terminate. This is only to ease presentation, and we relax this assumption in § 5.2.

We give some new definitions. Let C be a configuration of $\mathcal{U}$. The *extensions* of C, written $ex(C)$, are all those events outside C whose causes are included in C. Formally, $ex(C) := \{e \in E : e \notin C \wedge \lceil e \rceil \subseteq C\}$. We let $en(C)$ denote the set of events *enabled* by C, i.e., those corresponding to the transitions enabled at $state(C)$, formally defined as $en(C) := \{e \in ex(C) : C \cup \{e\} \in conf(\mathcal{U})\}$. All those events in $ex(C)$ that are not in $en(C)$ are the *conflicting extensions*, $cex(C) := \{e \in ex(C) : \exists e' \in C,\ e \#^i e'\}$. Clearly, sets $en(C)$ and $cex(C)$ partition the set $ex(C)$. Lastly, we define $\#^i(e) := \{e' \in E : e \#^i e'\}$, and $\#^i_U(e) := \#^i(e) \cap U$. The difference between both is that $\#^i(e)$ contains events from *anywhere* in the unfolding structure, while $\#^i_U(e)$ can only *see* events in U.

The algorithm is given as Alg. 1. The main procedure `Explore(`C, D, A`)` is given the configuration that is to be explored as parameter C. The parameter D (for *disabled*) is the set of set of events that have already been explored and prevents that `Explore()` repeats work. It can be seen as a *sleep set* [7]. The set A (for *add*) is occasionally used to guide the direction of the exploration.

Additionally, a global set U stores all events presently known to the algorithm. Whenever some event can safely be discarded from memory, `Remove` will move it from U to G (for *garbage*). Once in G, it can be discarded at any time, or be preserved in G in order to save work when it is re-inserted in U. Set G is thus our *cache memory* of events.

The key intuition for Alg. 1 is as follows. A call to `Explore(`C, D, A`)` visits all maximal configurations of $\mathcal{U}$ that contain C and do not contain D; and the first one explored will contain $C \cup A$. Figure 1 (c) gives one execution; tree nodes are of the form C| D| A .

The algorithm first updates U with all extensions of C (procedure `Extend`). If C is a maximal configuration, then there is nothing to do, and it backtracks. If not, it chooses an event in U enabled at C, using the function `en(`C`)` $:= en(C) \cap U$. If A is empty, any enabled event can be taken. If not, A needs to be explored and e must come from the intersection. Next it makes a recursive call (left subtree), where it explores *all* configurations containing all events in $C \cup \{e\}$ and no event from D. Since `Explore(`C, D, A`)` had to visit all maximal configurations containing C, it remains to visit those containing C but not e, but only if

there exists at least one! Thus, we determine whether $\mathcal{U}$ has a maximal configuration that contains C, does not contain D and does not contain e. Function `Alt` will return a set of events that witness the existence of such configuration (iff one exists). If one exists, we make a second recursive call (right subtree). Formally, we call such witness an *alternative*:

▶ **Definition 8** (Alternatives). Given a set of events $U \subseteq E$, a configuration $C \subseteq U$, and a set of events $D \subseteq U$, an *alternative* to D after C is any configuration $J \subseteq U$ satisfying that

- $C \cup J$ is a configuration (5)
- for all events $e \in D$, there is some $e' \in C \cup J$ such that $e' \in \#^i_U(e)$. (6)

Function `Alt`(X, Y) returns all alternatives (in U) to Y after X. Notice that it is called as `Alt`$(C, D \cup \{e\})$ from Alg. 1. Any returned alternative J witnesses the existence of a maximal configuration C' (constructed by arbitrarily extending $C \cup J$) where $C' \cap (D \cup \{e\}) = \varnothing$.

Although `Alt` reasons about maximal configurations of $\mathcal{U}$, thus potentially about events that have not yet been seen, it can only look at events in U. Thus, the set U needs to be large enough to contain enough *conflicting events* to satisfy (6). Perhaps surprisingly, it suffices to store only events seen (during the past exploration) in immediate conflict with C and D. Consequently, when the algorithm calls `Remove`, to clean from U events that are no longer necessary (i.e., necessary to find alternatives in the future), it needs to preserve at least those conflicting events. Specifically, `Remove` will preserve in U the following events:

$$Q_{C,D,U} := C \cup D \cup \bigcup_{e \in C \cup D, e' \in \#^i_U(e)} [e'].$$

That is, events in C, in D and events in conflict with those. An alternative definition that makes $Q_{C,D,U}$ smaller would mean that `Remove` discards more events, which could prevent a future call to `Alt` from discovering a maximal configuration that needs to be explored.

We focus now on the correctness of Alg. 1. Every call to `Explore`(C, D, A) explores a tree, where the recursive calls at lines 9 and 11 respectively explore the left and right subtrees (proof in [17, Corollary 25]). Tree nodes are tuples $\langle C, D, A\rangle$ corresponding to the arguments of calls to `Explore`, cf. Fig. 1. We refer to this object as the *call tree*. For every node, both C and $C \cup A$ are configurations, and $D \subseteq ex(C)$, cf. [17, Lemma 18]. As the algorithm goes down in the tree it monotonically increases the size of either C or D. Since $\mathcal{U}$ is finite, this implies that the algorithm terminates:

▶ **Theorem 9** (Termination). *Regardless of its input, Alg. 1 always stops.*

Next we assert that Alg. 1 never visits twice the same configuration, which is why it is called an *optimal* POR [2]. We show that for every node in the call tree, the set of configurations in the left and right subtrees are disjoint [17, Lemma 24]. This implies:

▶ **Theorem 10** (Optimality). *Let $\tilde{C}$ be a maximal configuration of $\mathcal{U}$. Then* `Explore`$(\cdot,\cdot,\cdot)$ *is called* at most once *with its first parameter being equal to $\tilde{C}$.*

Parameter A of `Explore` plays a central role in making Alg. 1 optimal. It is necessary to ensure that, once the algorithm decides to explore some alternative J, such an alternative is visited first. Not doing so makes it possible to extend C in such a way that no maximal configuration can ever avoid including events in D. Such a configuration, referred as a *sleep-set blocked* execution in [2], has already been explored before.

Finally, we ensure that Alg. 1 visits every maximal configuration of $\mathcal{U}$. This essentially reduces to showing that it makes the second recursive call, line 11, whenever there exists some unexplored maximal configuration not containing $D \cup \{e\}$. The difficulty of proving

this [17, Lemma 27] aises from the fact that Alg. 1 *only* sees events in U. Owing to space constraints, we omit an additional result on the memory consumption, see [17, Appendix B.5.].

▶ **Theorem 11** (Completeness). *Let $\tilde{C}$ be a maximal configuration of $\mathcal{U}$. Then* `Explore`$(\cdot,\cdot,\cdot)$ *is called* at least once *with its first parameter being equal to $\tilde{C}$.*

5 Improvements

5.1 State Caching

Stateless model checking algorithms explore only one configuration of $\mathcal{U}$ at a time, thus potentially under-using remaining available memory. A desirable property for an algorithm is the capacity to exploit all available memory without imposing the liability of actually requiring it. The algorithm in § 4 satisfies this property. The set G, storing events discarded from U, can be cleaned at discretion, e.g., when the memory is approaching full utilisation. Events cached in G are exploited in two different ways.

First, whenever an event in G shall be included again in U, we do not need to reconstruct it in memory (causality, conflicts, etc.). This might happen frequently. Second, using the result of the next section, cached events help prune the number of maximal configurations to visit. This means that our POR potentially visits *fewer* final states than the number of configurations of $\mathcal{U}$, thus conforming to the requirements of a *super-optimal DPOR*. The larger G is, the fewer configurations will be explored.

5.2 Non-Acyclic State Spaces

In this section we remove the assumption that $\mathcal{U}_{M,\diamond}$ is finite. We employ the notion of cutoff events [14]. While cutoffs are a standard tool for unfolding pruning, their application to our framework brings unexpected problems.

The core question here is preventing Alg. 1 from getting stuck in the exploration of an infinite configuration. We need to create the illusion that maximal configurations are finite. We achieve this by substituting procedure `Extend` in Alg. 1 with another procedure `Extend'` that operates as `Extend` except that it only adds to U an event from $e \in ex(C)$ if the predicate $\mathsf{cutoff}(e,U,G)$ evaluates to false. We define $\mathsf{cutoff}(e,U,G)$ to hold iff there exists some event $e' \in U \cup G$ such that

$$state([e]) = state([e']) \quad \text{and} \quad |[e']| < |[e]|. \tag{7}$$

We refer to e' as the *corresponding* event of e, when it exists. This definition declares e cutoff as function of U and G. This has important consequences. An event e could be declared cutoff while exploring one maximal configuration and non-cutoff while exploring the next, as the corresponding event might have disappeared from $U \cup G$. This is in stark contrast to the classic unfolding construction, where events are declared cutoffs *once and for all*. The main implication is that the standard argument [14, 5, 3] invented by McMillan for proving completeness fails. We resort to a completely different argument for proving completeness of our algorithm (see [17, Appendix C.1.]), which we are forced to skip due to lack of space.

We focus now on the correction of Alg. 1 using `Extend'` instead of `Extend`. A *causal cutoff* is any event e for which there is some $e' \in [e]$ satisfying (7). It is well known that causal cutoffs define a finite prefix of $\mathcal{U}$ as per the classic saturation definition [3]. Also, $\mathsf{cutoff}(e,U,G)$ always holds for causal cutoffs, regardless of the contents of U and G. This

means that the modified algorithm can only explore configurations from a finite prefix. It thus necessarily terminates. As for optimality, it is unaffected by the use of cutoffs, existing proofs for Alg. 1 still work. Finally, for completeness we prove the following result, stating that local reachability (e.g., fireability of transitions of M) is preserved:

▶ **Theorem 12** (Completeness). *For any reachable state $s \in reach(M)$, Alg. 1 updated with the cutoff mechanism described above explores one configuration C such that for some $C' \subseteq C$ it holds that $state(C') = s$.*

Lastly, we note that this cutoff approach imposes no liability on what events shall be kept in the prefix, set G can be cleaned at discretion. Also, redefining (7) to use adequate orders [5] is straightforward (see [17], where our proofs actually assume adequate orders).

6 Experiments

As a proof of concept, we implemented our algorithm in a new explicit-state model checker baptized Poet (Partial Order Exploration Tool).[3] Written in Haskell, a lazy functional language, it analyzes programs from a restricted fragment of the C language and supports POSIX threads. The analyzer accepts deterministic programs, implements a variant of Alg. 1 where the computation of the alternatives is memoized, and supports cutoffs events with the criteria defined in § 5.

We ran Poet on a number of multi-threaded C programs. Most of them are adapted from benchmarks of the Software Verification Competition [18]; others are used in related works [8, 20, 2]. We investigate the characteristics of average program unfoldings (depth, width, etc.) as well as the frequency and impact of cutoffs on the exploration. We also compare Poet with Nidhugg [1], a state-of-the-art stateless model checking for multi-threaded C programs that implements Source-DPOR [2], an efficient but non-optimal DPOR. All experiments were run on an Intel Xeon CPU with 2.4 GHz and 4 GB memory. Tables 1 and 2 give our experimental data for programs with acyclic and non-acyclic state spaces, respectively.

For programs with acyclic state spaces (Table 1), Poet with and without cutoffs seems to perform the same exploration when the unfolding has no cutoffs, as expected. Furthermore, the number of explored executions also coincides with Nidhugg when the latter reports 0 sleep-set blocked executions (cf., § 4), providing experimental evidence of Poet's optimality.

The unfoldings of most programs in Table 1 do not contain cutoffs. All these programs are deterministic, and many of them highly sequential (Stf, Spin08, Fib), features known to make cutoffs unlikely. Ccnf(n) are concurrent programs composed of $n-1$ threads where thread i and $i+1$ race on writing one variable, and are independent of all remaining threads. Their unfoldings resemble Fig. 2 (d), with $2^{(n-1)/2}$ traces but only $\mathcal{O}(n)$ events. Saturation-based unfolding methods would win here over both Nidhugg and Poet.

In the ssb benchmarks, Nidhugg encounters sleep-set blocked executions, thus performing sub-optimal exploration. By contrast, Poet finds many cutoff events and achieves a *super-optimal* exploration, exploring fewer traces than both Poet without cutoffs and Nidhugg. The data shows that this *super-optimality* results in substantial savings in runtime.

For non-acyclic state spaces (Table 2), unfoldings are infinite. We thus compare Poet with cutoffs and Nidhugg with a loop bound. Hence, while Nidhugg performs bounded model checking, Poet does complete verification. The benchmarks include classical mutual

[3] Source code and benchmarks available from: `http://www.cs.ox.ac.uk/people/marcelo.sousa/poet/`.

Table 1 Programs with acyclic state space. Columns are: $|P|$: nr. of threads; $|I|$: nr. of explored traces; $|B|$: nr. of sleep-set blocked executions; $t(s)$: running time; $|E|$: nr. of events in $\mathcal{U}$; $|E_{\text{cut}}|$: nr. of cutoff events; $|\Omega|$: nr. of maximal configurations; $\langle|U_\Omega|\rangle$: avg. nr. of events in U when exploring a maximal configuration. A $*$ marks programs containing bugs. `<7K` reads as "*fewer than 7000*".

Benchmark		Nidhugg			Poet (without cutoffs)				Poet (with cutoffs)				
Name	$\|P\|$	$\|I\|$	$\|B\|$	$t(s)$	$\|E\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$	$\|E\|$	$\|E_{\text{cut}}\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$
Stf	3	6	0	0.06	121	6	79	0.04	121	0	6	79	0.06
Stf$*$	3	–	–	0.05	–	–	–	0.02	–	–	–	–	0.03
Spin08	3	84	0	0.08	2974	84	1506	2.04	2974	0	84	1506	2.93
Fib	3	8953	0	3.36	<185K	8953	92878	305	<185K	0	8953	92878	704
Fib$*$	3	–	–	0.74	–	–	–	81.0	–	–	–	–	133
Ccnf(9)	9	16	0	0.05	49	16	46	0.07	49	0	16	46	0.06
Ccnf(17)	17	256	0	0.15	97	256	94	5.76	97	0	256	94	6.09
Ccnf(19)	19	512	0	0.28	109	512	106	22.5	109	0	512	106	22.0
Ssb	5	4	2	0.05	48	4	38	0.03	46	1	4	37	0.03
Ssb(1)	5	22	14	0.06	245	23	143	0.11	237	4	23	140	0.11
Ssb(3)	5	169	67	0.12	2798	172	1410	3.51	1179	48	90	618	0.90
Ssb(4)	5	336	103	0.15	<7K	340	3333	20.3	2179	74	142	1139	2.07
Ssb(8)	5	2014	327	0.85	<67K	2022	32782	4118	<12K	240	470	6267	32.1

Table 2 Programs with non-terminating executions. Column b is the loop bound. The value is chosen based on experiments described in [1].

Benchmark			Nidhugg			Poet (with cutoffs)				
Name	$\|P\|$	b	$\|I\|$	$\|B\|$	$t(s)$	$\|E\|$	$\|E_{\text{cut}}\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$
Szymanski	3	–	103	0	0.07	1121	313	159	591	0.36
Dekker	3	10	199	0	0.11	217	14	21	116	0.07
Lamport	3	10	32	0	0.06	375	28	30	208	0.12
Peterson	3	10	266	0	0.11	175	15	20	100	0.05
Pgsql	3	10	20	0	0.06	51	8	4	40	0.03
Rwlock	5	10	2174	14	0.83	<7317	531	770	3727	12.29
Rwlock(2)$*$	5	2	–	–	7.88	–	–	–	–	0.40
Prodcons	4	5	756756	0	332.62	3111	568	386	1622	5.00
Prodcons(2)	4	5	63504	0	38.49	640	25	15	374	1.61

exclusion protocols (Szymanski, Sekker, Lamport and Peterson), where Nidhugg is able to leverage an important static optimization that replaces each spin loop by a load and assume statement [1]. Hence, the number of traces and maximal configurations is not comparable. Yet Poet, which could also profit from this static optimization, achieves a significantly better reduction thanks to cutoffs alone. Cutoffs dynamically prune redundant unfolding branches and arguably constitute a more robust approach than the load and assume syntactic substitution. The substantial reduction in number of explored traces, several orders of magnitude in some cases, translates in clear runtime improvements. Finally, in our experiments, both tools were able to successfully discover assertion violations in stf$*$, fib$*$ and rwlock(2)$*$.

In our experiments, Poet's average maximal memory consumption (measured in events) is roughly half of the size of the unfolding. We also notice that most of these unfoldings are quite narrow and deep ($|E_{\text{cut}}| \div |E|$ is low) when compared with standard benchmarks for Petri nets. This suggests that they could be amenable for saturation-based unfolding verification, possibly pointing the opportunity of applying these methods in software verification.

7 Related Work

This work focuses on explicit-state POR, as opposed to symbolic POR techniques exploited inside SAT solvers, e.g., [11, 8]. Early POR statically computed the necessary transitions to fire at every state [19, 7]. Flanagan and Godefroid [6] first proposed to compute persistent sets dynamically (DPOR). However, even when combined with sleep sets [7], DPOR was still unable to explore exactly one interleaving per Mazurkiewicz trace. Abdulla et al. [2, 1] recently proposed the first solution to this, using a data structure called wakeup trees. Their DPOR is thus optimal (ODPOR) in this sense.

Unlike us, ODPOR operates on an interleaved execution model. Wakeup trees store chains of dependencies that assist the algorithm in reversing races throughly. Technically, each branch roughly correspond to one of our alternatives. According to [2], constructing and managing wakeup trees is expensive. This seems to be related with the fact that wakeup trees store canonical linearizations of configurations, and need to canonize executions before inserting them into the tree to avoid duplicates. Such checks become simple linear-time verifications when seen as partial-orders. Our alternatives are computed dynamically and exploit these partial orders, although we do not have enough experimental data to compare with wakeup trees. Finally, our algorithm is able to visit up to exponentially fewer Mazurkiewicz traces (due to cutoff events), copes with non-terminating executions, and profits from state caching. The work in [2] has none of these features.

Combining DPOR with stateful search is challenging [21]. Given a state s, DPOR relies on a complete exploration from s to determine the necessary transitions to fire from s, but such exploration could be pruned if a state is revisited, leading to unsoundness. Combining both methods requires addressing this difficulty, and two works did it [21, 20], but for non-optimal DPOR. By contrast, incorporating cutoff events into Alg. 1 was straightforward.

Classic, saturation-based unfolding algorithms are also related [14, 5, 3, 10]. They are inherently stateful, cannot discard events from memory, but explore events instead of configurations, thus may do exponentially less work. They can furthermore guarantee that the number of explored events will be at most the number of reachable states, which at present seems a difficult goal for PORs. On the other hand, finding the events to extend the unfolding is computationally harder. In [10], Kähkönen and Heljanko use unfoldings for concolic testing of concurrent programs. Unlike ours, their unfolding is not a semantics of the program, but rather a means for discovering all concurrent program paths.

While one goal of this paper is establishing an (optimal) POR exploiting the same commutativity as some non-sequential semantics, a longer-term goal is building formal connections between the latter and PORs. Hansen and Wang [9] presented a characterization of (a class of) stubborn sets [19] in terms of configuration structures, another non-sequential semantics more general than event structures. We shall clarify that while we restrict ourselves to commutativity-based PORs, they attempt a characterization of stubborn sets, which do not necessarily rely on commutativity.

8 Conclusions

In the context of commutativity-exploiting POR, we introduced an optimal DPOR that leverages on cutoff events to prune the number of explored Mazurkiewicz traces, copes with non-terminating executions, and uses state caching to speed up revisiting events. The algorithm provides a new view to DPORs as algorithms exploring an object with richer structure. In future work, we plan exploit this richer structure to further reduce the number of explored traces for both PORs and saturation-based unfoldings.

References

1 Parosh Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 9035 in LNCS, pages 353–367. Springer, 2015.

2 Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Principles of Programming Languages (POPL)*, pages 373–384. ACM, 2014.

3 Blai Bonet, Patrik Haslum, Victor Khomenko, Sylvie Thiébaux, and Walter Vogler. Recent advances in unfolding technique. *Theoretical Comp. Science*, 551:84–101, September 2014.

4 Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

5 Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

6 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages (POPL)*, pages 110–121. ACM, 2005.

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

8 Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *Model Checking Software (SPIN)*, volume 4595 of *LNCS*, pages 95–112. Springer, 2007.

9 Henri Hansen and Xu Wang. On the origin of events: branching cells as stubborn sets. In *Proc. International Conference on Application and Theory of Petri Nets and Concurrency (ICATPN)*, volume 6709 of *LNCS*, pages 248–267. Springer, 2011.

10 Kari Kähkönen and Keijo Heljanko. Testing multithreaded programs with contextual unfoldings and dynamic symbolic execution. In *Application of Concurrency to System Design (ACSD)*, pages 142–151. IEEE, 2014.

11 Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Computer Aided Verification (CAV)*, volume 5643 of *LNCS*, pages 398–413. Springer, 2009.

12 Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.

13 Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 278–324. Springer, 1987.

14 K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of async. circuits. In *Proc. CAV'92*, volume 663 of *LNCS*, pages 164–177. Springer, 1993.

15 Ugo Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32(6):545–596, 1995.

16 Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

17 César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. *CoRR*, abs/1507.00980, 2015.

18 `http://sv-comp.sosy-lab.org/2015/`.

19 Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, number 483 in LNCS, pages 491–515. Springer, 1991.

20 Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Model Checking Software (SPIN)*, volume 5156 of *LNCS*, pages 288–305. Springer, 2008.

21 Xiaodong Yi, Ji Wang, and Xuejun Yang. Stateful dynamic partial-order reduction. In *Formal Methods and Sw. Eng.*, number 4260 in LNCS, pages 149–167. Springer, 2006.