Partial-Order Reduction Is Hard

Frédéric Herbreteau, Sarah Larroze-Jardiné, Igor Walukiewicz

To cite this version:

Frédéric Herbreteau, Sarah Larroze-Jardiné, Igor Walukiewicz. Partial-Order Reduction Is Hard. 36th International Conference on Concurrency Theory (CONCUR 2025), Aug 2025, Aarhus, Denmark. pp.22:1–22:20, ⟨10.4230/LIPIcs.CONCUR.2025.22⟩. ⟨hal-05399232⟩

HAL Id: hal-05399232

https://hal.science/hal-05399232v1

Submitted on 4 Dec 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Partial-Order Reduction Is Hard

Frédéric Herbreteau
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Sarah Larroze-Jardiné
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Igor Walukiewicz
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Abstract

The goal of partial-order methods is to accelerate the exploration of concurrent systems by examining only a representative subset of all possible runs. The stateful approach builds a transition system with representative runs, while the stateless method simply enumerates them. The stateless approach may be preferable if the transition system is tree-like; otherwise, the stateful method is more effective.

In the last decade, optimality has been a guiding principle for developing stateless partial-order reduction algorithms, and without doubt contributed to big progress in the field. In this paper we ask if we can get a similar principle for the stateful approach. We show that in stateful exploration, a polynomially close to optimal partial-order algorithm cannot exist unless P=NP. The result holds even for acyclic programs with just await instructions. This lower bound result justifies systematic study of heuristics for stateful partial-order reduction. We propose a notion of IFS oracle as a useful abstraction. The oracle can be used to get a very simple optimal stateless algorithm, which can then be adapted to a non-optimal stateful algorithm. While in general the oracle problem is NP-hard, we show a simple case where it can be solved in linear time.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Formal verification, Concurrent systems, Partial-order reduction, Complexity

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.22

Funding *Frédéric Herbreteau*: Supported by the French government in the framework of the France 2030 programme IdEx université de Bordeaux / RRI ROBSYS.
Igor Walukiewicz: Supported by ANR grant PaVeDys, ANR-23-CE48-0005.

1 Introduction

The goal of partial-order methods is to speed up explicit state exploration of concurrent systems. The state space of such systems grows exponentially with the number of processes. Fortunately, many runs of a concurrent system can usually be considered equivalent, so it is enough to explore only one run in each equivalence class. For example, if one process assigns $x := 2$ and another $y := 3$ then the order of execution of these two operations is usually irrelevant; the two interleavings are equivalent, and it is enough to explore only one of the two. This reduces the number of visited states as well as the exploration time. In some cases, the reductions are very substantial.

In recent years, we have seen novel applications of partial-order methods. One is proving the properties of concurrent programs [10], where equivalence between runs is not only used to reduce the number of proof objectives, but also to simplify proof objectives by choosing particular linearizations. Similarly, in symbolic executions [9] or in testing [28], partial-order can be used to limit the exploration space while still being exhaustive. Another application is verification of timed systems using recently developed local-time zones [15]. All these applications rely on explicit state enumeration, as opposed to symbolic methods such as SAT or BDDs.

© Frédéric Herbreteau, Sarah Larroze-Jardiné, and Igor Walukiewicz;
licensed under Creative Commons License CC-BY 4.0
36th International Conference on Concurrency Theory (CONCUR 2025).
Editors: Patricia Bouyer and Jaco van de Pol; Article No. 22; pp. 22:1–22:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

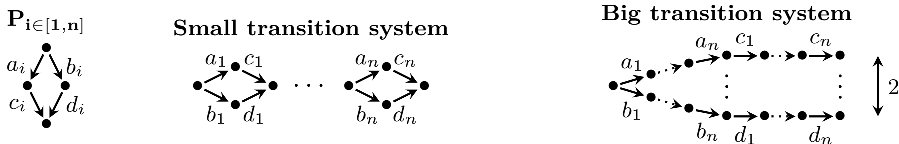


Figure 1 Consider a concurrent system consisting of processes $P_1, \ldots, P_n$, with no dependency between actions. Two trace-optimal transition systems, one of linear size and one of exponential size.

The first partial-order methods were proposed about 35 years ago under the names of stubborn sets, persistent sets, or ample sets [34, 14, 20]. In 2005, Flanagan and Godefroid introduced *stateless* dynamic partial-order reductions [12], making stateless methods a focal point of subsequent research. In 2014, Abdulla et al. proposed a notion of trace-optimality[1] [1], and a race reversal technique, initiating a new cycle of work on stateless partial-order methods [6, 5, 18, 22, 23, 3, 21, 24, 4].

In this paper, we focus on *stateful* partial-order reduction methods, that have seen relatively less progress over the last two decades [29, 37, 8, 7]. A partial-order algorithm produces a *reduced transition system* containing a representative for each equivalence class of runs of the system. A *stateless* approach produces a tree of runs, but saves memory by storing only one run at a time. In contrast, a *stateful* approach keeps all visited states in memory so that an exploration can be stopped if a state is revisited. Each of the two approaches has its advantages. When the objective is to verify existing code by running an instrumentation of it, the states are too complex to keep in memory, so stateless exploration is the only solution. When verifying pseudocode, say of a mutual exclusion protocol, the number of non-equivalent runs may be several orders of magnitude bigger than the number of states, rendering the stateless approach infeasible. The stateful approach is then necessary.

Our first question is whether we can find a guiding principle for the stateful approach similar to the trace-optimality for the stateless approach. The example in Figure 1 shows why we cannot just settle on trace optimality. The two transition systems in the figure are both trace-optimal, since no two maximal paths are equivalent. But one is of linear size and the other is of exponential size. The obvious parameter for optimality of the stateful approach is the size of the transition system. So our question is whether a state-optimal, or close to optimal, partial-order algorithm can exist.

It is not difficult to see that getting a reduced system of exactly the minimal size is NP-hard. Our main result says something much stronger: even approximating the minimal size within a polynomial factor remains NP-hard (Theorem 5). This holds even for acyclic concurrent programs using only await and write instructions. Arguably, this looks like a minimal sensible set of instructions. If we allowed cycles then we could simulate await with active wait, reducing the set of instructions to just reads and writes.

This negative result justifies starting a systematic study of heuristics for stateful partial-order reduction. We propose an approach based on a notion of *IFS oracle*. This oracle expresses an important sub-problem encountered in partial-order reduction algorithms, namely avoiding so called sleep-blocked executions. Our proposal is to concentrate on the heuristics for this oracle to improve the efficiency of stateful methods. We justify this approach by showing that in the stateless setting, the IFS oracle gives a very simple trace-

[1] We use the term trace-optimality instead of simply optimality as in op.cit. to differentiate from state-optimality we consider in this paper.

optimal algorithm. We show how to adapt this algorithm to the stateful approach. Finally, we narrow our study down to a special case, generalizing some settings considered in the stateless literature [5, 22], where the IFS oracle can be computed in linear time.

Before starting, we would like to shortly discuss the models of concurrent systems we are using. In recent years it is common to consider a model of processes with shared variables, eventually adding lock instructions. This is the model we are using for our lower bound. However, we also consider a stronger model of concurrent systems synchronizing on shared actions. This model can encode variables, and most of the synchronization mechanisms used in practice. For this reason we prefer this model for our positive results. Moreover, since the model is strong, our lower bound result is easier to prove for it, so it serves as a good intermediate step in our proof.

In this work we assume that programs are acyclic. Dealing with cycles in partial-order reduction is a separate problem from the reduction itself, and complicates the algorithms substantially. Finding a clean way of dealing with cycles in partial-order reduction is beyond the scope of this paper.

To summarize our three principal contributions are:

- A strong lower bound showing that, assuming $P \neq \mathrm{NP}$, no deterministic algorithm can construct a reduced transition system for a concurrent program $\mathbb{P}$, of polynomial size relative to the *minimal size* of a reduced system for $\mathbb{P}$, while working in time polynomial in the size of its input and its output (Theorem 5). This holds even if every process in the program is acyclic and uses only await and write instructions.
- A notion of *IFS oracle* and a simple trace-optimal stateless algorithm using this oracle. We adapt this algorithm to the stateful approach.
- A special subcase of linear fully non-blocking systems, where the IFS oracle can be computed in linear time. This case covers some situations considered in the stateless partial-order reduction literature.

1.1 Related Work

The literature on partial-order methods has expanded rapidly in the last decade. In this brief discussion, we focus only on results that are closely related to our work, primarily citing more recent papers.

Partial-order methods have been introduced around 1990 [34, 13, 20]. Applications requiring stateful partial-order methods [11] are still based on the same basic principles. Another significant concept for us here is the use of lexicographic ordering to identify representative runs [36, 19, 35]. We also highlight a Petri net unfolding-based approach [32, 8] that distinguishes itself by using prime event structures instead of transition systems.

There are already two NP-hardness results in the literature on partial-order methods. The first is the NP-hardness of computing optimal ample sets [30]. The second is NP-hardness of computing so-called alternatives in the context of unfolding based methods [8]. These results can be compared to our observation (Proposition 22) that IFS problem is NP-complete. Our main negative result is much stronger. It does not address a particular method of doing partial-order reduction, but shows that any method must be NP-hard, even when instead of optimality we just want to be polynomially close to optimal.

Stateless partial-order reduction, initiated by [12], gained momentum thanks to [1]. This approach has been the focus of extensive research in recent years, culminating in truly stateless algorithms using only polynomial-sized memory [22, 3]. Extensions of stateless techniques to systems admitting blocking, like some synchronization mechanisms, have been proposed very recently [18, 23]. Await instructions we use in our lower bound are blocking. We consider non-blocking systems in our positive result in Proposition 24.

The race reversal technique introduced in [1] has been partially adapted for use in stateful methods [37, 7]. Apart from that, the stateful methods still rely on the original stubborn/persistent set approach. The unfolding based partial-order methods have seen developments like the k-partial alternatives [8]. Although IFS oracle comes from standard concepts and problems in partial-order methods, we hope this notion will help to advance the stateful approach.

Apart from typical programs with variables, partial-order methods have been adapted to various frameworks, including actor programs [33], event driven programs [2, 17, 25], and MPI programs [31]. This motivates us to use the shared actions model in our positive results.

2 Modeling concurrent systems

For our lower bounds we will consider a very simple standard model of concurrent programs with variables and locks. Variables range over finite domains, and locks are binary. The only operations on variables are write and await. The later operation blocks until the value of a variable is in a specified set of values. We do not need a read operation in our constructions, so we do not mention it to simplify the presentation.

A *process* is a directed acyclic graph with edges labeled by instructions. The instructions come from the set:

$$\Sigma ::= w(x, i) \mid \mathit{await}(x, I) \mid \mathit{acq}(l) \mid \mathit{rel}(l)$$

Here, $w(x, i)$ sets the value of variable x to i, $\mathit{await}(x, I)$ blocks until the value of x is in the set of values I, $\mathit{acq}(l)$ acquires lock l, and $\mathit{rel}(l)$ releases it. In case of acyclic processes, it is natural to have the blocking *await* instruction. For example, all classical mutual exclusion algorithms rely on it. If we had cycles, *await* could be simulated with an active wait.

A *concurrent program* $\mathbb{P}$ is a finite set of processes, over finite sets of variables and locks. Each variable has a finite domain. All variables and locks are global, meaning that they are shared by all processes. A conditional instruction is modeled by branching in the processes. The semantics of a program $\mathbb{P}$ is given by a transition system $\mathrm{TS}(\mathbb{P})$. Its states consist of a local state for each process, and a value for each variable and lock. In the initial state, each process is in its initial local state, and each variable and lock have value 0. The transitions in $\mathrm{TS}(\mathbb{P})$ are defined according to the usual semantics of instructions from Σ. Observe that $\mathrm{TS}(\mathbb{P})$ is acyclic since each process is acyclic.

We will also consider a much more flexible and expressive model, that we call *concurrent system*. It consists of a collection of processes synchronizing on common actions. Synchronization on actions can directly encode variables, locks, and many other synchronization primitives. In the context of this paper it will be easier to show the lower bound in this model than in the model with variables and locks. But, since the model is strong the lower bound result for this model is relatively weak. This is why we will eventually refine the lower bound construction to concurrent programs without locks.

A concurrent system $\mathbb{S}$ is determined by an alphabet Σ of abstract actions, and a set of processes that are graphs with edges labeled by actions. A process must be *action deterministic*: for every $s \in S$ and $a \in \Sigma$ there is at most one t with $s \xrightarrow{a} t$. The term process is used both in the case of concurrent systems, and concurrent programs, but it will always be clear from the context which model we are using.

Every action a has its *domain*, $dom(a)$, that is a set of processes using it. We write $s_p \xrightarrow{b}_p$ if there is an outgoing b-transition from state s_p of process p. For a sequence of actions v we write $dom(v)$ for $\bigcup\{dom(a) : a \text{ appears in } v\}$, and $s_p \xrightarrow{v}_p$ if there is a path labeled v from s_p in process p.

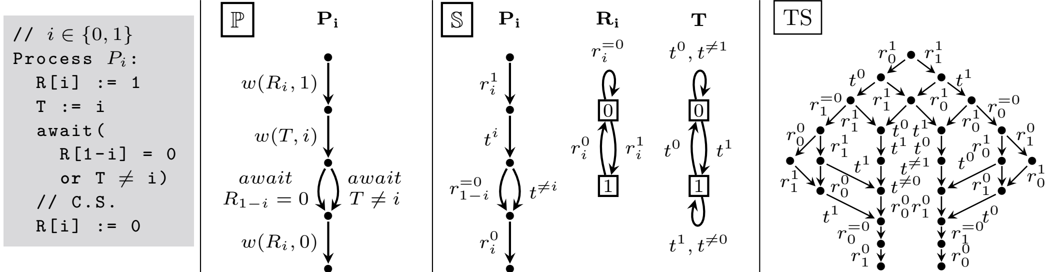


Figure 2 Peterson's mutex algorithm with 2 processes: pseudocode (left), modeled as a concurrent program $\mathbb{P}$ (middle left), as a concurrent system $\mathbb{S}$ (middle right), and their semantics TS (right).

The semantics of a concurrent system $\mathbb{S}$ is a transition system TS($\mathbb{S}$) whose states are tuples of states of process transition systems, $S = \prod_{p \in Proc} S_p$; the initial state is the tuple consisting of initial states of each process, $s^0 = \{s^0_p\}_{p \in Proc}$; every action a synchronizes processes involved in it: $s \xrightarrow{a} s'$ if $s_p \xrightarrow{a}_p s'_p$ for $p \in dom(a)$, and $s'_p = s_p$ for $p \notin dom(a)$. We write $enabled(s)$ for the set of actions labeling transitions outgoing from the global state s of TS($\mathbb{S}$).

Finally, we impose an *acyclicity condition*: for every action a, at least one of the processes in $dom(a)$ must be acyclic. This guarantees that the global transition system is acyclic.

We will need to talk about runs in a transition system TS, be it TS($\mathbb{P}$) for a concurrent program $\mathbb{P}$ or TS($\mathbb{S}$) for a concurrent system $\mathbb{S}$. A *run* is a path in the transition system TS, not necessarily from the initial state. We write $s \xrightarrow{v} t$ if there is a run labeled with a sequence of actions v from s to t. Sometimes we write just $s \xrightarrow{v}$ when t is not relevant. A *maximal run* is a run reaching a terminal state – a state with no outgoing transitions. A *full run* is a maximal run starting in the initial state.

Figure 2 on the left shows Peterson's mutual exclusion algorithm for 2 processes. The middle-left picture shows its modeling as a concurrent program $\mathbb{P} = \{P_0, P_1\}$. It has three variables: R_0, R_1 and T that range over $\{0, 1\}$ as in the pseudocode. The middle-right picture shows the same algorithm modeled as a concurrent system $\mathbb{S} = \{P_0, P_1, R_0, R_1, T\}$. Processes P_0 and P_1 implement the code while the other processes model the variables $R[0]$, $R[1]$ and T. Action r^j_i writes the value j to variable $R[i]$. Action $r^{=0}_i$ checks if the variable $R[i]$ has value 0; observe that it is enabled in the top state where $R[i]$ has value 0, but not in the bottom state where it has value 1. Similarly, for actions of T. Processes in $\mathbb{S}$ synchronize on common actions. For instance, r^1_0 synchronizes P_0 and R_0, while $t^{\neq 1}$ synchronizes P_1 with T. We write this as $dom(r^1_0) = \{P_0, R_0\}$ and $dom(t^{\neq 1}) = \{P_1, T\}$. The transition system TS is shown on the right. It represents both TS($\mathbb{P}$) and TS($\mathbb{S}$), although we have chosen to show labels from $\mathbb{S}$. Each full run corresponds to an execution of Peterson's algorithm.

3 Partial-order reduction

The initial motivation for partial-order reduction comes from the observation that some sequences of actions may be considered equivalent, and only one of the equivalent sequences needs to be explored. The equivalence relation on sequences is defined in terms of the independence relation on actions: two sequences are *por-equivalent* if one can be obtained from the other by permuting adjacent independent actions. We will use $\approx$ to denote a por-equivalence relation on sequences of actions. As we will see our arguments do not rely on a particular independence relation, as long as two actions using different processes and different resources are independent.

For concurrent programs with variables the classical independence relation says that two actions of different processes are independent if either they use different variables or they use the same variable and both of them are awaits. In particular lock operations are dependent.

For concurrent systems, the most common independence relation is the one coming from Mazurkiewicz trace theory [26]. Two actions $a, b \in \Sigma$ are *independent* if they have disjoint domains: aIb if $dom(a) \cap dom(b) = \emptyset$.

It will also be useful to use the dual concept of dependent actions. Two actions are *dependent*, written aDb, if they are not independent. For example, when $dom(a) \cap dom(b) \neq \emptyset$ in case of Mazurkiewicz independence. These notions are extended to sequences: aIv means that a is independent of all actions in v, and aDv that a is dependent on some action from v. We write Da for the set of actions dependent on a.

The goal of partial-order reduction is to construct, for a given concurrent system $\mathbb{S}$ or a concurrent program $\mathbb{P}$, a reduced transition system representing all full runs in $\mathrm{TS}(\mathbb{S})$ or $\mathrm{TS}(\mathbb{P})$ (recall that these are maximal runs from the initial state).

▶ **Definition 1.** *We say that* TS_r *is a* reduced transition system *for* TS *if it is:*

- sound*: every full run of* TS_r *is a full run in* TS*, and*
- complete*: for each full run* u *in* TS *there is a full run* v *in* TS_r *such that* $u \approx v$

A general approach for constructing a reduced transition system is to determine for each state s of TS a covering source set [1]: a subset of enabled actions that is sufficient to explore. For example, if every sequence starting from b is equivalent to a sequence starting from c then we may choose to include only b in the source set. The notion of the *first action* modulo an equivalence relation $\approx$ on sequences is central for partial-order reduction.

$$\mathit{first}(u) = \{b : \exists v.\ bv \approx u\}\ .$$

We want a source set in a state to be large enough to contain a first action of at least one representative from every equivalence class of maximal runs from the state. Using the above definition this can be formulated as: a source set in s should intersect every $\mathit{first}(u)$ for u a maximal run from s. We formalize this as follows.

▶ **Definition 2.** *For a state s of* TS *we define $\mathit{First}(s)$ as the set of first sets of all maximal runs from s:*

$$\mathit{First}(s) = \{\mathit{first}(u) : u \text{ is a maximal run from } s\}\ .$$

A set of actions B is a covering source set *in s if $B \cap F \neq \emptyset$ for every $F \in \mathit{First}(s)$.*

In particular, if B contains all enabled actions from s then B is a covering source set at s. Intuitively, smaller covering source sets should give smaller reduced transition systems. This is not always true. A bigger but incomparable w.r.t. set inclusion covering source set may give a better reduction.

Observe that the notion of a covering source set depends on the por-equivalence relation $\approx$, as the definition of *first* depends on it.

Suppose we have an assignment of a set of actions $\mathit{source}(s)$ for every state s of TS. We can use it to restrict the transition relation to transitions allowed by source sets: define $s \stackrel{a}{\Longrightarrow} t$ when $s \stackrel{a}{\longrightarrow} t$ and $a \in \mathit{source}(s)$. If every $\mathit{source}(s)$ is a covering source set in s then this restricted transition relation is enough.

▶ **Proposition 3.** *Let* TS *be a finite acyclic transition system with a transition relation $\longrightarrow$. Suppose $\Longrightarrow$ is the restricted transition relation derived from a covering source set assignment. For every state s of* TS*, and every maximal run $s \stackrel{u}{\longrightarrow}$ there is a run $v \approx u$ with $s \stackrel{v}{\Longrightarrow}$.*

This proposition allows us to construct a reduced transition system by keeping only $\Longrightarrow$ transitions and states reachable from the initial state using these transitions. We must underline though that source sets are not the only mechanism needed in partial-order reduction. Sleep-sets are another important ingredient (c.f. Remark 21).

4 Partial-order reduction is NP-hard

Recall that the goal of partial-order reduction is to construct for a given program $\mathbb{P}$ or a given system $\mathbb{S}$, a reduced system that is sound and complete. Among such systems we would ideally like to construct one with the smallest number of states. We write $minTS(\mathbb{P})$ for the smallest number of states of a sound and complete reduced transition system for $\mathbb{P}$. Similarly, we write $minTS(\mathbb{S})$ for a system $\mathbb{S}$. There may be several non-isomorphic transition systems with a minimal number of states.

Stateless algorithms use the concept of *trace-optimality*. This means that there are no two equivalent runs in the reduced transition system. As we have noted in the introduction, this concept is not very useful for our purposes. Indeed, the example from Figure 1 shows two trace-optimal transition systems, one of linear size and the other of exponential size. So in stateful partial-order reduction we should aim for a reduced transition system of small size. Even more so as there are also examples where the smallest reduced transition system is necessarily not trace-optimal.

In this section, we show that there is no polynomial-time algorithm for constructing a reduced transition system that is polynomially close to optimal. The next definition makes this precise.

▶ **Definition 4.** *We say that Alg is an* excellent POR algorithm *if there are polynomials $q(x)$ and $r(x)$ such that given a concurrent program $\mathbb{P}$, the algorithm constructs a sound and complete transition system for $\mathbb{P}$ of size bounded by $q(minTS(\mathbb{P}))$ in time $r(|\mathbb{P}| + minTS(\mathbb{P}))$.*

We use the same definition for concurrent systems $\mathbb{S}$.

The main result of this section is

▶ **Theorem 5.** *If P≠NP then there is no excellent POR algorithm, even for concurrent programs using only write and await operations.*

We will prove this theorem in three steps. First we will consider concurrent systems. The flexibility of synchronization operations makes the constructions used in the proof much simpler, and the arguments can be supported by figures of a reasonable size. In the next step we adapt the construction to concurrent programs with locks and await. In the last step we eliminate locks. The proof is structured in such a way that the same statements need to be proved in each of the three steps.

4.1 Synchronizations

The proof of Theorem 5 uses an encoding of the 3-SAT problem φ into a concurrent system $\mathbb{S}_\varphi$. If φ is not satisfiable then $\mathbb{S}_\varphi$ has a small reduced transition system. If it is satisfiable then all reduced transition systems for $\mathbb{S}_\varphi$ are big. With such an encoding we show that an excellent POR algorithm would give a polynomial-time algorithm for deciding 3-SAT.

Consider a 3-CNF formula over literals $\{x_1, \dots, x_n\} \cup \{\bar{x}_1, \dots, \bar{x}_n\}$ and with k clauses:

$$\varphi = (\alpha_1^1 \vee \alpha_2^1 \vee \alpha_3^1) \wedge \cdots \wedge (\alpha_1^k \vee \alpha_2^k \vee \alpha_3^k)$$

where each α_j^i is a literal.

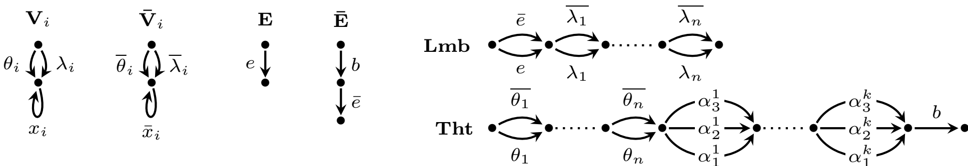


Figure 3 Concurrent system $\mathbb{S}_\varphi$ for the lower bound argument.

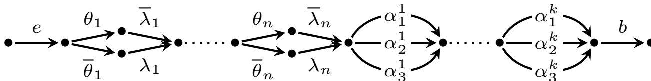


Figure 4 A sound a complete transition system for $\mathbb{S}_\varphi$ when φ is not SAT.

The concurrent system $\mathbb{S}_\varphi$ is presented in Figure 3. It has two processes for each propositional variable, V_i and $\bar{V}_i$, two special processes *Lmb*, and *Tht*, and two auxiliary processes E and $\bar{E}$. For every $i = 1, \ldots, n$ we have a process V_i corresponding to variable x_i. Process V_i has two states: top and bottom. There are two transitions from top to bottom state, the θ_i transition intuitively says that x_i should be true, and λ_i that it should be false. This choice is not encoded in the reached state though, as the two transitions go to the same bottom state where a transition on x_i is possible; interactions with special processes will make the difference. Process $\bar{V}_i$ is similar, but now we have $\bar{\theta}_i$, $\bar{\lambda}_i$, and $\bar{x}_i$ actions.

Apart from the variable processes V_i and $\bar{V}_i$, we have two special processes *Lmb* and *Tht*. Only *Tht* depends on formula φ. Process *Lmb* synchronizes on λ_i actions, and *Tht* on θ_i actions. Process *Lmb* starts with a choice between e and $\bar{e}$, while the second part of *Tht* corresponds to the clauses of the formula φ, and finishes with action b. Finally, there are two auxiliary processes E and $\bar{E}$. The first does e that is enabled in the initial state. The second can do $\bar{e}$, but only after doing b, and this in turn can happen only when *Tht* terminates.

▶ **Lemma 6.** *If φ is not SAT then all runs of $\mathbb{S}_\varphi$ start with e. In other words, $e \in \mathit{first}(w)$ for every full run w of $\mathbb{S}_\varphi$.*

Proof. Suppose w is a full run without e in $\mathit{first}(w)$. Since e is always possible until $\bar{e}$ is executed, we must have $\bar{e}$ on w. This must be preceded by b. So the first part, call it u, of w consists of synchronizations of variable processes V_i and $\bar{V}_i$ with *Tht*, without *Lmb* moving until *Tht* completes its run, that is until it does b. From the form of *Tht* it follows that the first part of u is a sequence $u_1 \ldots u_n$ where each u_i is either θ_i or $\bar{\theta}_i$. This defines a valuation v. At this stage, the variable processes corresponding to literals true in v are in their bottom states where the actions on these literals are possible. The other variable processes are in their top states where the action on the corresponding literals are impossible. So, since *Lmb* cannot move, *Tht* can get to action b iff v is a satisfying valuation. As φ is not SAT this is impossible. Hence, such w cannot exist. ◀

From the above proof we can actually see that $\bar{e}$ appears on the run iff φ is satisfiable. We do not need this fact in the rest of the argument, but it may be helpful to fix the intuitions.

▶ **Lemma 7.** *If φ is not SAT then the transition system from Figure 4 is a sound and complete transition system for $\mathbb{S}_\varphi$.*

Proof. First observe that in the state just before α's, all variable processes are in their bottom states, so all events x_i and $\bar{x}_i$ are enabled. Thus, all local paths of *Tht* on α's are feasible. Then action b is possible, but $\bar{e}$ is disabled since *Lmb* is in its bottom state. Thus, the transition system from Figure 4 is sound: every full path is a run of $\mathbb{S}_\varphi$.

It remains to verify that the transition system is complete. By Lemma 6, e is a first action of all full runs of $\mathbb{S}_\varphi$. Take such a run ew. Looking at $\mathbb{S}_\varphi$ we see that after e there are four possible actions: $\lambda_1, \bar{\lambda}_1, \theta_1, \bar{\theta}_1$. We will consider only the case when the next action is θ_1 as the argument for the other possibilities is analogous. So $w = \theta_1 u_1 \bar{\lambda}_1 u_1'$ for some sequences u_1 and u_1'; observe that $\bar{\lambda}_1$ must appear on the run, as after $e\theta_1$ there is no action that can disable it. Since the only action on which *Lmb* can synchronize is $\bar{\lambda}_1$, all actions in u_1 are synchronizations with *Tht*. Clearly they do not involve $\bar{V}_1$, as there is no way to execute $\bar{\theta}_1$ at this stage. Hence, u_1 is independent of $\bar{\lambda}_1$ giving us that w is trace equivalent to $\theta_1 \bar{\lambda}_1 u_1 u_1'$. Repeating this reasoning we obtain that w is trace equivalent to a run in the transition system from Figure 4. ◀

▶ **Lemma 8.** *If φ is SAT then there are runs containing $\bar{e}$. In every sound and complete reduced transition system for $\mathbb{S}_\varphi$ there are at least as many states as there are satisfying valuations for φ.*

Proof. For a valuation v we consider a run w_v taking θ_i if $v(x_i) = \mathit{true}$ and taking $\bar{\theta}_i$ if $v(x_i) = \mathit{false}$. For example, if x_1, and x_n hold but x_2 does not hold in v then this run would look something like:

$$\theta_1 \bar{\theta}_2 \dots \theta_n \alpha^1_{i_1} \dots \alpha^k_{i_k} b \bar{e} \bar{\lambda}_1 \lambda_2 \dots \bar{\lambda}_n$$

Here $\alpha^j_{i_j}$ is a literal that holds in the clause j. Since v is a satisfying valuation, process *Tht* can get to b.

Observe that there is no concurrency in the run w_v. All θ and $\bar{\theta}$ actions synchronize with process *Tht*. Then b and $\bar{e}$ are also dependent on each other as they happen on the same process. Action $\bar{e}$ is the first action of *Lmb*. It is followed by a sequence of actions of process *Lmb*.

Consider two different valuations $v_1 x$ and v_2 satisfying φ. Let s_1 be the state of system $\mathbb{S}_\varphi$ reached after $\bar{e}$ on the run w_{v_1}. Similarly, for s_2 and w_{v_2}. In state s_1, the variable processes that are in the bottom states are those corresponding to literals that are true in v_1 and similarly for v_2. Since v_1 and v_2 are distinct valuations, the states s_1 and s_2 are distinct. So any sound and complete transition system for $\mathbb{S}_\varphi$ must have at least as many states as there are satisfying valuations of φ. ◀

We write $|\varphi|$ for the length of φ. Observe that this is an upper bound on the number of variables as well as on the number of clauses in φ, namely, $n, k < |\varphi|$.

▶ **Corollary 9.** *If φ is not SAT then $minTS(\mathbb{S}_\varphi) \leq 6|\varphi|$ states. If φ is SAT then $minTS(\mathbb{S}_\varphi)$ is bigger than the number of satisfying valuations for φ.*

Now, we prove Theorem 5 for concurrent systems. Suppose to the contrary that *Alg* is an excellent POR algorithm. We use it to solve SAT in deterministic polynomial time.

Let $q(x)$ and $r(x)$ be the polynomials associated to *Alg*. Namely, *Alg* working in time $r(|\mathbb{S}| + |minTS(\mathbb{S})|)$ produces a sound and complete reduced transition system of size at most $q(|minTS(\mathbb{S})|)$.

Given a formula ψ, consider an integer m and a formula

$$\varphi \equiv \psi \wedge (z_1 \vee z_2) \wedge \dots \wedge (z_{2m-1} \vee z_{2m})$$

where $z_1, \dots, z_{2m}$ are new variables. Clearly φ is satisfiable iff ψ is. If ψ is satisfiable then φ has at least 2^m satisfying valuations.

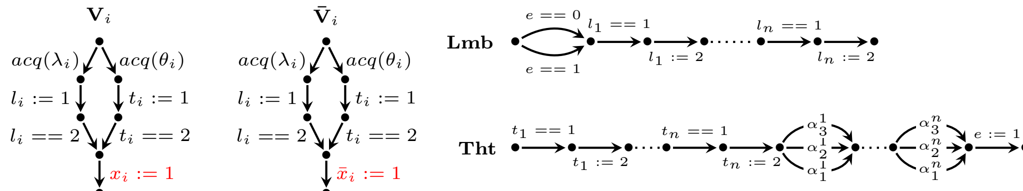


Figure 5 Processes V_i and $\bar{V}_i$ for propositional variable i, and special processes *Lmb* and *Tht*. Await instructions are written as equality checks, and writes as assignments. Instruction α_i^j tests if the corresponding variable is 1.

Now we construct our system $\mathbb{S}_\varphi$ and run *Alg* on it for $r(12|\varphi|)$ time. If φ is not SAT then, by Corollary 9, the algorithm stops and produces a sound and complete transition system. If φ is SAT then by Corollary 9 the algorithm cannot stop in this time as the smallest sound and complete transition system for $\mathbb{S}_\varphi$ has at least 2^m states, and we can choose m big enough so that $2^m > r(6|\varphi|)$.

4.2 Await and locks

In this section we show that the same phenomenon appears in concurrent programs. Here instead of direct synchronizations we have variables and locks. The only operations on variables are write and await.

The argument follows the same lines as the one for the direct synchronizations. Observe that the proof of Theorem 5 above uses Lemmas 6, 7 and 8. So it is enough to show that the three lemmas hold for concurrent programs with locks and variables to prove Theorem 5.

Given a propositional formula φ, we build a concurrent program $\mathbb{P}_\varphi$. As in the previous construction, we will have two processes for each propositional variable in φ, and two special processes: one for lambdas and one for thetas. However, we do not need processes E and $\bar{E}$ anymore. Initially all variables in $\mathbb{P}$ are set to 0.

Figure 5 shows the processes V_i and $\bar{V}_i$ for propositional variable i, as well as the two special processes *Lmb* and *Tht*. The processes V_i and $\bar{V}_i$ use two locks, λ_i and θ_i. After acquiring one of the two locks, each process sets the corresponding variable l_i or t_i to 1 in order to communicate their choice to the two special processes *Tht* and *Lmb*. Next, it awaits an acknowledgment from the special process in the form of the value of the variable l_i or t_i becoming 2. Finally, it sets x_i or $\bar{x}_i$ to 1.

The two special processes *Lmb* and *Tht* are quite similar to the previous construction. Only process *Tht* depends on the formula φ we are encoding. Process *Lmb* first tests if the value of the variable e is 0 or 1. Next it awaits for variable l_1 to become 1, and then sets it to 2. The same pattern repeats for l_2 up to l_n. Similarly, process *Tht* awaits for variable t_i to become 1, and then sets it to 2. After that, it tests if the formula is true, under the chosen valuation, and finally sets e to 1.

As in the previous section, we claim that if φ is not satisfiable, then there is a sound and complete transition system of a small size. This transition system has a similar structure to the one in the previous section (Figure 4), but is longer due to additional actions await and write. So our argument, while similar, becomes more complicated too.

▶ **Lemma 10.** *If φ is not satisfiable, then e can be set to 1 only after Lmb process executes $e == 0$.*

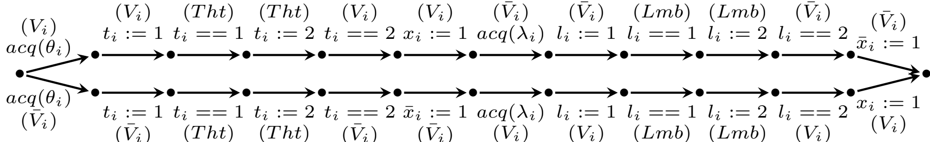


Figure 6 Block of actions in $\mathrm{TS}(\mathbb{P}_\varphi)$ when φ is not satisfiable. Await instructions are written as equality checks, and writes as assignments.

Proof. Suppose not. Then there is a run setting e to 1 without doing $e == 0$; meaning that e is set to 1 before Lmb moves. On this run all t_i must be set to 1 by processes V_i and $\bar{V}_i$, and then set to 2 by process Tht. Exactly one among processes V_i and $\bar{V}_i$ can proceed to setting x_i or $\bar{x}_i$ to 1; as the other needs to acquire λ_i, and then wait for l_i to become 2. Hence, when Tht reaches its α part, only one of x_i or $\bar{x}_i$ is set to 1. This defines a valuation, and Tht can set e to 1 if and only if this valuation satisfies φ. Hence, if φ is not satisfiable, then Tht cannot set e to 1. ◀

▶ **Lemma 11.** *If φ is not satisfiable, then there is a sound and complete transition system of size $O(n)$.*

We describe the transition system $\mathrm{TS}(\mathbb{P}_\varphi)$. By the previous lemma, if φ is not satisfiable, Lmb must perform $e == 0$ before e is set to 1. Thus, every run is equivalent to a run starting with $e == 0$. Then after the first transition, $\mathrm{TS}(\mathbb{P}_\varphi)$ is a sequence of blocks as shown in Figure 6. The upper line and the lower line depend on which process takes the θ_i lock and the λ_i lock. Then, these n blocks in $\mathrm{TS}(\mathbb{P}_\varphi)$, we have the α part of process Tht, including the last transition that sets variable e to 1. So the size of the transition system $\mathrm{TS}(\mathbb{P}_\varphi)$ is linear in the size of φ.

The proof of the third lemma is the same as for the previous construction.

▶ **Lemma 12.** *If φ is SAT then there are runs containing $e == 1$. In every sound and complete reduced transition system for $\mathbb{P}_\varphi$ there are at least as many states as there are satisfying valuations for φ.*

The three lemmas give us the proof of Theorem 5 for concurrent programs with locks and variables.

4.3 Just await

The previous construction uses locks and await instructions on variables. There is a number of classical algorithms for implementing locks with await, like Peterson, Bakery, or Szymanski. So it is natural to ask if we could eliminate locks from our construction. This is the goal of this section.

In the previous construction we used n locks, but each lock was shared between only two processes, and taken only once. We cannot just use one of the standard lock implementations out of the box because our construction is based on V_i and $\bar{V}_i$ competing for two locks. Moreover, we need to be careful with the number of states of the resulting transition system. So while there are no important new ideas in the encoding below, we face a challenge resembling a bit the classical problem of writing a program that prints itself.

Our solution is inspired by Peterson's mutual exclusion algorithm for 2 processes that uses a shared variable *turn* (T in Figure 2) to order the processes when they both request access to the critical section. Figure 7 (left) shows the two processes V_i and $\bar{V}_i$ that only use

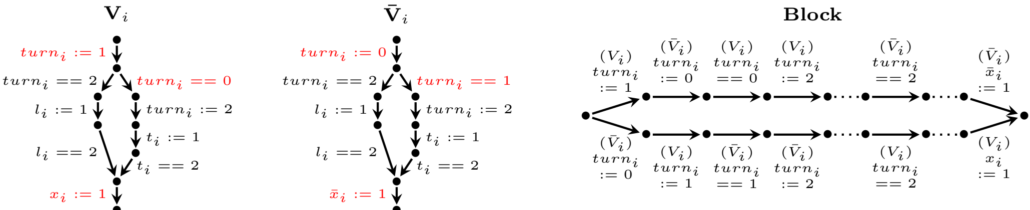


Figure 7 Processes V_i and $\bar{V}_i$ without locks for propositional variable i (left), and block of actions in $\mathrm{TS}(\mathbb{P}_\varphi)$ when φ is not satisfiable (right). Await instructions are written as equality checks, and writes as assignments.

instructions await and write. Following the principle of Peterson's algorithm, the processes V_i and $\bar{V}_i$ share a variable $turn_i$ that replaces the lock operations, and that ensures that one of V_i and $\bar{V}_i$ will set t_i while the other will set l_i. The await instructions on variable $turn_i$ are followed by the same sequences of actions on t_i or l_i as previously. The two special processes Lmb and Tht are as in Figure 5. For notational convenience, we assume that the variables $turn_i$ are initialized to 2, whereas e is initialized to 0. The differences between V_i and $\bar{V}_i$ are highlighted in Figure 7.

The proof takes the same three steps as the previous one.

▶ **Lemma 13.** *If φ is not satisfiable, then e can be set to 1 only after Lmb process performs $e == 0$.*

Proof. Suppose there is a run where $e == 0$ is not a first action. This can only happen when $e := 1$ is executed before process Lmb starts. Hence, all t_i must be set to 1 by variable processes. This means that exactly one of the processes V_i or $\bar{V}_i$ can complete its operations and set x_i or $\bar{x}_i$ to 1; while the other will not pass $l_i == 2$. If φ is not satisfiable then this is not enough for Tht process to reach the point where it can set e to 1. ◀

▶ **Lemma 14.** *If φ is not satisfiable, then there is a sound and complete transition system of size $O(n)$.*

The argument is similar to the one for the construction with locks. $\mathrm{TS}(\mathbb{P})$ has a similar structure as before, with blocks as in Figure 7 (right).

▶ **Lemma 15.** *If φ is SAT then there are runs containing $e == 1$. In every sound and complete reduced transition system for $\mathbb{P}_\varphi$ there are at least as many states as there are satisfying valuations for φ.*

The third lemma is the same as in the previous construction, and its proof is also the same. The tree lemmas give us the proof of Theorem 5 for concurrent programs using only await operations.

5 An approach to partial-order reduction

In the light of the negative results from the previous section, it is clear that practical approaches to stateful partial-order reduction should be based on heuristics, and cannot aim at optimality, or even at approximate optimality up to a polynomial factor. In this section we propose a concrete problem, that when solved with a satisfactory heuristic should advance the quality of our partial-order methods. This approach also allows to make a link with stateless partial-order methods. It leads to a simple trace-optimal algorithm for the stateless case for a restricted class of concurrent systems.

We start with a simple algorithm for constructing trace-optimal reduced transition systems. It uses three ingredients that we describe in more details later.

- Lexicographic order on sequences allowing to determine a representative run for each class of a por-equivalence relation $\approx$.
- Sleep sets giving enough information about the exploration context.
- An oracle "includes first sets", denoted *IFS*, permitting to avoid sleep-blocked runs.

Lexicographic ordering has been already used in the context of partial-order reduction [19]. Sleep sets are one of the classical concepts for partial-order methods [14]. *IFS* oracle is a direct way of avoiding sleep-blocked executions, a well known challenge in partial-order reduction [1]. Our contribution here is to make the *IFS* problem explicit. Without surprise at this stage, the *IFS* problem is NP-hard, but we show one case where it is polynomial.

Let us fix a concurrent system $\mathbb{S}$. Let us also assume that we have some linear ordering on actions of $\mathbb{S}$. This determines a lexicographic ordering on sequences of actions. Given some equivalence relation on sequences we can use the lexicographic order to define representatives for equivalence classes of $\approx$ relation.

▶ **Definition 16.** *We say that w is a* lex-sequence *if w is the smallest lexicographically among all sequences equivalent to it: that is the smallest sequence in $\{v : v \approx w\}$. A* lex-run *of* $\mathrm{TS}(\mathbb{S})$ *is a run whose labels form a lex-sequence.*

We are going to present an algorithm enumerating full lex-runs of $\mathrm{TS}(\mathbb{S})$. For this we will use the "includes first set (IFS)" oracle. The idea is as follows. Suppose the algorithm has reached a state s and produced a sleep set $\mathit{sleep}(s)$ containing actions that need not be explored from s. We would like to check if there is something left to be explored. We need to check if the exploration is not *sleep-blocked*, namely, if there is a maximal run u from s with $\mathit{first}(u) \cap \mathit{sleep}(s) = \emptyset$. If there is such run, we need to explore one of the actions in $\mathit{first}(u)$ from s.

▶ **Definition 17** (IFS). *Let s be a state of* $\mathrm{TS}(\mathbb{S})$ *and B a subset of actions. We say that B* includes a first-set in *s if there is a maximal run u from s with $\mathit{first}(u) \subseteq B$. We write $\mathit{IFS}(s, B)$ when there exists such a maximal run u.*

Listing 1 presents a very simple algorithm enumerating all full lex-runs of $\mathrm{TS}(\mathbb{S})$. The algorithm also gives us an opportunity to explain sleep sets in details. Each node n of the tree constructed by the algorithm is a pair consisting of a state of $\mathrm{TS}(\mathbb{S})$, denoted $s(n)$, and a set of actions $\mathit{sleep}(n)$. For readability, we write $\mathit{enabled}(n)$ instead of $\mathit{enabled}(s(n))$, for the set of outgoing actions from $s(n)$.

Sleep sets are a very elegant mechanism to gather some information about the exploration context. They can be computed top-down when constructing an exploration graph. At the root, the sleep set is empty. For a node n and a transition $n \xrightarrow{e} n_e$ we have $\mathit{sleep}(n_e) = (\mathit{sleep}(n) \cup \{a_1, \dots, a_k\}) - De$, where $a_1, \dots, a_k$ are labels of transitions from n created before the e-transition; and the final $-De$ term means that we remove all the actions dependent on e. The intuition behind this formula is as follows. Assume that we keep an invariant:

> (*sleep-invariant*): after exploration of a node n for every maximal run u from $s(n)$ in TS such that $\mathit{first}(u) \cap \mathit{sleep}(n) = \emptyset$ we have a path v from n with $u \approx v$.

Then the formula for $\mathit{sleep}(n_e)$ says that we need not explore from n_e a run starting, say, with a_1 if $a_1 I e$. This is because when looking from n such a run has a_1 in its first set, and it has already been explored from a_1's successor of n.

Listing 1 Lex exploration with sleep sets, constructs a tree of maximal runs.

```
1  procedure main(𝕊):
2    create node n^0 with s(n^0) = s^0 and sleep(n^0) = ∅
3    TreeExplore(n^0)
4
5  procedure TreeExplore(n):
6    Sl := sleep(n) // invariant: Sl = sleep(n) ∪ {labels of transitions outgoing from n}
7    while enabled(n) − Sl ≠ ∅:
8        choose smallest e ∈ (enabled(n) − Sl) w.r.t. linear ordering on actions
9        let s' such that s(n) --e--> s' in TS(𝕊)
10       if IFS(s', enabled(s') − (Sl − De)):
11         create node n' with s(n') = s' and sleep(n') = Sl − De
12         add edge n --e--> n'
13         TreeExplore(n')
14       Sl := Sl ∪ {e}
```

The algorithm from Listing 1 examines all enabled transitions in a node n in our fixed order on actions. For every enabled transition $s(n) \xrightarrow{e} s'$ it uses the *IFS* oracle in Line 10 to decide if it is necessary to explore it. If the answer is positive, then it creates node n' with an appropriate sleep set. If not, then it skips the transition. This guarantees that the algorithm is trace-optimal.

▶ **Lemma 18.** *The algorithm in Listing 1 constructs a tree such that: (i) every full run in the tree is a full lex-run of* $\mathrm{TS}(\mathbb{S})$*, and (ii) for every full run u of* $\mathrm{TS}(\mathbb{S})$ *there is a unique full run v in the tree with $v \sim u$.*

We could use this algorithm in stateless model-checking if we had an implementation of $\mathit{IFS}(s, B)$. In a stateful version, the algorithm is not very interesting as trees produced by this algorithm can be, and often are, orders of magnitude bigger than $\mathrm{TS}(\mathbb{S})$, the transition system of $\mathbb{S}$ without any reduction. But we can modify the algorithm to produce a graph instead of a tree. For this we need to introduce a subsumption relation between nodes.

▶ **Definition 19.** *We say that n* subsumes *n', in symbols $n \lhd n'$ if the states in n and n' are the same: $s(n) = s(n')$, and there are less sleep-blocked actions from n than from n': $\mathit{sleep}(n) \subseteq \mathit{sleep}(n')$.*

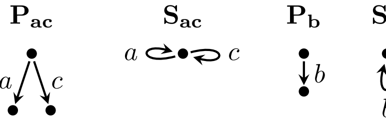
Observe that if n subsumes n', all runs that are not sleep blocked from n' are also not sleep blocked from n. Thus replacing n' by n still yields a sound and complete reduced transition system. Thus, we add subsumption to Listing 1 as shown in Listing 2. We first check if the successor s' of s is subsumed by an existing node in line 11. If yes, then, instead of creating a new node, we add an edge to the subsuming node (line 12). Otherwise, we proceed as in Listing 1. Observe that the algorithm in Listing 2 builds a directed acyclic graph (DAG) instead of a tree, hence avoiding visiting a node more than once.

The new algorithm does not satisfy a statement as in Lemma 18 because edges added due to the subsumption relation may create paths that are not lex-runs. We can get a variant of this optimality property when looking at states. Let us call a state *lex-useful* if it appears on a full lex-run.

▶ **Lemma 20.** *The algorithm with subsumption in Listing 2 builds a reduced transition system* TS_r *such that: (i) every full run in* TS_r *is a full lex-run of* $\mathrm{TS}(\mathbb{S})$*, and (ii) for every full run u of* $\mathrm{TS}(\mathbb{S})$ *there is a full run v in* TS_r *with $v \sim u$. Moreover, for every node n, its state $s(n)$ is lex-useful.*

Listing 2 Lex exploration with sleep sets and subsumption, constructs a graph.

```
1  procedure main(𝕊):
2    create node n⁰ with s(n⁰) = s⁰ and sleep(n⁰) = ∅
3    Explored := ∅
4    DAGExplore(n⁰)
5
6  procedure DAGExplore(n):
7    Sl := sleep(n) // invariant: Sl = sleep(n) ∪ {labels of transitions outgoing from n}
8    while enabled(n) − Sl ≠ ∅:
9        choose smallest e ∈ (enabled(n) − Sl) w.r.t. linear ordering on actions
10       let s′ such that s(n) −e→ s′ in TS(𝕊)
11       if ∃n″ ∈ Explored such that n″ ◁ (s′, Sl − De):
12         add edge n −e→ n″
13       else if IFS(s′, enabled(s′) − (Sl − De)):
14         create node n′ with s(n′) = s′ and sleep(n′) = Sl − De
15         add edge n −e→ n′
16         DAGExplore(n′)
17       Sl := Sl ∪ {e}
18   add n to Explored
```



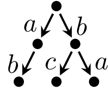


Figure 8 Sleep sets are needed for optimality.

▶ Remark 21. Sleep sets are needed for the optimality result, in a sense that some information about exploration context is needed. Consider the system of four process in Figure 8 (left) and its runs (right). The algorithm first explores the run ab, and then in the root state asks if there is a run where a is not a first action. The answer is positive because of the run bc. The algorithm then decides to explore b as it is the smallest enabled action. At this point it arrives at n_b. Without a sleep set it has no choice but to explore both a and c from n_b. But $ba \sim ab$ so this is not trace-optimal. With sleep sets we get $a \in \mathit{sleep}(n_b)$ which is exactly what is needed to block unnecessary exploration. So an idealized view suggested by Section 3 that partial-order reduction boils down to finding good covering source sets is not the whole story. Fortunately, the additional required information is easy to compute from the context.

5.1 The IFS oracle

The use of *IFS* queries gives a very clean approach to partial-order reduction, unfortunately the *IFS* problem is NP-hard.

▶ **Proposition 22.** *The following problem is* NP*-hard: given a concurrent system* $\mathbb{S}$*, its global state* s*, and a set of actions* B*, does* $\mathit{IFS}(s, B)$ *hold?*

Proof. Suppose we are given a 3CNF formula φ consisting of m clauses over variables $x_1, \ldots, x_n$. Say the j-th clause is of the form $(\alpha_j^1 \vee \alpha_j^2 \vee \alpha_j^3)$ where each α is either some variable x_i or its negation $\bar{x}_i$. We will construct a concurrent program $\mathbb{P}_\varphi$ such that φ is satisfiable if and only if $\mathit{IFS}(s_0, B)$ holds for the initial state s_0 and B some set of actions we make precise below.

$$\mathbf{P}_i^0 \bullet \xrightarrow{acq(e_i)} \bullet \xrightarrow{rel(\bar{x}_i)} \bullet \qquad \mathbf{C}_j^i \bullet \xrightarrow{acq(\alpha_j^i)} \bullet \xrightarrow{rel(\alpha_j^i)} \bullet \xrightarrow{rel(c_j)} \bullet \qquad \mathbf{F} \bullet \xrightarrow{acq(f)} \bullet$$

$$\mathbf{P}_i^1 \bullet \xrightarrow{acq(e_i)} \bullet \xrightarrow{rel(x_i)} \bullet \qquad \mathbf{D} \bullet \xrightarrow{acq(c_1)} \bullet \cdots\cdots \bullet \xrightarrow{acq(c_m)} \bullet \xrightarrow{acq(f)} \bullet$$

Figure 9 A concurrent program $\mathbb{P}_\varphi$ with locks encoding SAT(φ) for a 3CNF formula φ.

All actions in $\mathbb{P}_\varphi$ will implement taking or releasing a lock. There is one lock per propositional variable and its negation, $\{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$; one per clause, $\{c_1, \dots, c_m\}$; one additional lock for every variable $\{e_1, \dots, e_n\}$ in $\mathbb{P}_\varphi$, as well as a special lock f.

System $\mathbb{P}_\varphi$ is presented in Figure 9. For every propositional variable x_i there are two processes P_i^0 and P_i^1. For every clause $(\alpha_j^1 \vee \alpha_j^2 \vee \alpha_j^3)$ there are processes C_j^1, C_j^2, C_j^3. Additionally, there are two processes D and F.

Consider the state s where every process is in its initial state, and where the locks $\{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n, c_1, \dots, c_m\}$ are taken whereas the other locks are available. At this point the only processes enabled are F, and the processes corresponding to propositional variables, namely $P_1^0, P_1^1, \dots, P_n^0, P_n^1$. Process F can take the lock f. For each i, either process P_i^0 or process P_i^1 takes the lock e_i which amounts to choosing a value for variable x_i, as either the lock x_i or the lock $\bar{x}_i$ is released, respectively. This may allow some process corresponding to a clause to move. For example, suppose x_i was released, and α_j^1, the first literal of the j-th clause, is x_i. Then process C_j^1 can take x_i and release x_i thus testing if x_i was available. If the lock was available, then it can release c_j. Releasing c_j intuitively means that the j-th clause is satisfied. Thus, all locks $c_1, \dots, c_m$ can be released if and only if the formula φ is satisfied by the valuation determined by locks from $\{x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n\}$ that have been released. If all $c_1, \dots, c_m$ are released then process D can take all these locks and then take f. So if φ is satisfiable then there is a maximal run of the system where $acq(f)$ is not a first action. If φ is not satisfiable then there is no way to release all the locks $c_1, \dots, c_m$, so D cannot take f. In this case $acq(f)$ is in the first set of every maximal run because process F can always take f. This shows that $IFS(s, \Sigma - \{acq(f)\})$ holds iff φ is satisfiable (where Σ is the set of all actions in $\mathbb{P}_\varphi$). ◀

In our opinion it is worthwhile to study heuristics for the *IFS* problem. One possibility is to use existing static [14, 30, 34] or dynamic [12, 1, 18, 22] partial-order techniques to compute, for each node, a set of actions to visit to ensure completeness. Another intriguing approach consists in encoding $IFS(s, B)$ as a SAT problem, and then use a solver. This makes sense since the *IFS* problem is NP-hard. Designing heuristics that are efficient in practice is out of the scope of this paper. Still, we show that the *IFS* problem can actually be solved efficiently in a special case of non-blocking systems.

There is a simple setting where the *IFS* test can be done in polynomial time. Here the only operations are reads and writes, and programs are straight lines, in particular there are no conditionals. This case is indeed considered in the stateless model-checking literature [5, 22]. We generalize this situation to the case when we have variables but no operation on a variable can ever be blocked. So instead of just reads and writes we can also have operations like test-and-set, or fetch-and-add.

▶ **Definition 23.** *An* extended concurrent program *is a concurrent program $\mathbb{P}$ with arbitrary instructions on variables. It is* fully non-blocking *if in every state every first action of every process is enabled. It is* linear *if every process is a sequence of instructions, meaning there is no branching or conditionals.*

We use the term fully non-blocking to differentiate from non-blocking from [1] that is rather a forward non-blocking condition.

▶ **Proposition 24.** *If $\mathbb{P}$ is a linear, fully non-blocking extended concurrent program then IFS test can be done in polynomial time.*

Recall that $\mathit{IFS}(s, B)$ means that there is a maximal run u from s with $\mathit{first}(u) \subseteq B$. Let $\bar{B}$ denote the set of actions not in B. For a sequence v of actions we denote by $\bar{B}_v$ the set $\bar{B} - Dv$, namely the set of actions in $\bar{B}$ that are not dependent on actions in v. Then, we can check if $\mathit{IFS}(s, B)$ by constructing a run as follows. Suppose v is an already constructed run from s. Extend v with the smallest action enabled in the state reached after v, that is not in $\bar{B}_v$. When there is no such action return true if v is maximal, and false otherwise.

Thus, in the case covered by the above proposition the algorithm from Listing 1 is optimal and works in polynomial time w.r.t. size of the constructed tree. It works differently than race-reversal algorithms [1, 3].

The *IFS* test is NP-hard when one of the two conditions of Proposition 24 is lifted. This does not preclude that there is some case between fully non-blocking and blocking for which IFS is in PTIME. However, it is quite hard to imagine how to weaken the linearity condition.

6 Conclusions

Our main result is a negative one, indicating that we cannot hope for too much in terms of theoretically good partial-order algorithms. It has been known before that certain existing partial-order methods are NP-hard [30], but here we show that no method can be both polynomially close to optimal and not NP-hard.

As we have explained in the introduction, we believe that stateful partial-order methods have applications, and that there are even more applications to come. In this context our negative result gives us an argument for looking at well motivated heuristics rather than for a theoretically grounded result. We propose a new schema of a partial-order reduction algorithm based on IFS test. While in general the test is NP-complete, we hope that good heuristics for this test can translate into progress in partial-order methods. This is a direction we are actively investigating.

There is a big variety of concurrent systems, in particular depending on communication operations between processes. We have exhibited one case when IFS test can be done in PTIME. Intuitively, it captures “fully non-blocking” situations. An interesting question is what happens in the forward non-blocking case [1], or maybe to some other formalization of non-blocking in the spirit wait-free operations [16] or MPI [27].

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *Journal of the ACM*, 64(4):1–49, 2017. doi:10.1145/3073408.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Frederik Meyer Bønneland, Sarbojit Das, Bengt Jonsson, Magnus Lang, and Konstantinos Sagonas. Tailoring Stateless Model Checking for Event-Driven Multi-threaded Programs. In Étienne André and Jun Sun, editors, *Automated Technology for Verification and Analysis*, pages 176–198. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-45332-8_9.
3. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. Parsimonious optimal dynamic partial order reduction. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024,*

Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II, volume 14682 of *Lecture Notes in Computer Science*, pages 19–43. Springer, 2024. doi:10.1007/978-3-031-65630-9_2.

4 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. Trading space for simplicity in stateless model checking. In Susanne Graf, Paul Pettersson, and Bernhard Steffen, editors, *Real Time and Such - Essays Dedicated to Wang Yi to Celebrate His Scientific Career*, volume 15230 of *Lecture Notes in Computer Science*, pages 79–97. Springer, 2025. doi:10.1007/978-3-031-73751-0_8.

5 Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless Model Checking Under a Reads-Value-From Equivalence. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, volume 12759, pages 341–366. Springer International Publishing, 2021. doi:10.1007/978-3-030-81685-8_16.

6 Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2018. doi:10.1145/3158119.

7 Berk Cirisci, Constantin Enea, Azadeh Farzan, and Suha Orhun Mutluergil. A Pragmatic Approach to Stateful Partial Order Reduction. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 13881, pages 129–154. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-24950-1_7.

8 Camille Coti, Laure Petrucci, César Rodríguez, and Marcelo Sousa. Quasi-optimal partial order reduction. *Formal Methods in System Design*, 57(1):3–33, 2021. doi:10.1007/s10703-020-00350-4.

9 Frank S. De Boer, Marcello Bonsangue, Einar Broch Johnsen, Violet Ka I Pun, S. Lizeth Tapia Tarifa, and Lars Tveito. SymPaths: Symbolic Execution Meets Partial Order Reduction. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Software Verification: Future Perspectives*, volume 12345, pages 313–338. Springer International Publishing, 2020. doi:10.1007/978-3-030-64354-6_13.

10 Azadeh Farzan. Commutativity in Automated Verification. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–7. IEEE, 2023. doi:10.1109/LICS56636.2023.10175734.

11 Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Sound sequentialization for concurrent program verification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 506–521. ACM, 2022. doi:10.1145/3519939.3523727.

12 Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL'05*, 2005.

13 Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, pages 176–185. Springer, 1991. doi:10.1007/BFb0023731.

14 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems An Approach to the State-Explosion Problem.* PhD thesis, Université de Liège, 1994.

15 R. Govind, Frédéric Herbreteau, Srivathsan, and Igor Walukiewicz. Abstractions for the local-time semantics of timed automata: A foundation for partial-order methods. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–14. ACM, 2022. doi:10.1145/3531130.3533343.

16 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

17 Casper S. Jensen, Anders Moller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73. Association for Computing Machinery, 2015. doi:10.1145/2814270.2814282.

18 Bengt Jonsson, Magnus Lang, and Konstantinos Sagonas. Awaiting for Godot Stateless Model Checking that Avoids Executions where Nothing Happens. In *{22nd Formal Methods in Computer-Aided Design, {FMCAD} 2022*. IEEE, 2022.

19 Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643, pages 398–413. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-02658-4_31.

20 Shmuel Katz and Doron Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992. doi:10.1007/BF02252682.

21 Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis. Enhancing GenMC's Usability and Performance. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 66–84. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-57249-4_4.

22 Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022. doi:10.1145/3498711.

23 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. Unblocking Dynamic Partial Order Reduction. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, volume 13964, pages 230–250. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-37706-8_12.

24 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. SPORE: combining symmetry and partial order reduction. *Proc. ACM Program. Lang.*, 8(PLDI):1781–1803, 2024. doi:10.1145/3656449.

25 Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial Order Reduction for Event-Driven Multi-threaded Programs. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 680–697. Springer, 2016. doi:10.1007/978-3-662-49674-9_44.

26 Antoni W. Mazurkiewicz. Introduction to trace theory. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995. doi:10.1142/9789814261456_0001.

27 Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*, November 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

28 Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019. doi:10.1145/3360606.

29 Doron Peled. Partial-order reduction. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 173–190. Springer, 2018. doi:10.1007/978-3-319-10575-8_6.

30 Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. doi:10.1007/3-540-56922-7_34.

31 The Anh Pham. *Efficient State-Space Exploration for Asynchronous Distributed Programs: Adapting Unfolding-Based Dynamic Partial Order Reduction to MPI Programs*. These de doctorat, Rennes, Ecole normale superieure, 2019.

32 César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based Partial Order Reduction. *LIPIcs, Volume 42, CONCUR 2015*, 42:456–469, 2015. doi:10.4230/LIPICS.CONCUR.2015.456.

33 Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, pages 219–234. Springer, 2012. doi:10.1007/978-3-642-30793-5_14.

34 Antti Valmari. Stubborn sets for reduced state space generation. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and Grzegorz Rozenberg, editors, *Advances in Petri Nets 1990*, volume 483, pages 491–515. Springer Berlin Heidelberg, 1991. doi:10.1007/3-540-53863-1_36.

35 Bjorn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *2013 Formal Methods in Computer-Aided Design*, pages 210–217. IEEE, 2013. doi:10.1109/FMCAD.2013.6679412.

36 Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 382–396. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-78800-3_29.

37 Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, pages 288–305. Springer, 2008. doi:10.1007/978-3-540-85114-1_20.