Partial-Order Reduction for Multi-Core LTL Model Checking

Alfons Laarman[1,2] and Anton Wijs[3]

[1] Vienna University of Technology**
[2] University of Twente
[3] RWTH Aachen University
alfons@laarman.com, awijs@cs.rwth-aachen.de

Abstract. Partial-Order Reduction (POR) is a well-known, successful technique for on-the-fly state space reduction in model checking, as evidenced by the prestigious CAV 2014 award for its pioneers. The combination of POR with LTL model checking is long known to cause the so-called ignoring problem, i.e. relevant behavior is continuously ignored and never selected for exploration. This problem has been solved with increasing sophistication over the years, using various ignoring provisos, which include all necessary actions along cycles in the state space.
However, parallel model checking algorithms still suffer from a lack of an efficient solution; the best known ones causing severe decrease in reductions. We present a new parallel ignoring proviso for POR, which solves this issue by exploiting parallel DFS-based algorithms. Its similarity to the sequential solutions allows the combination with sophisticated earlier methods solving the ignoring problem. We prove correctness of the new proviso and empirically show that it maintains good reductions, runtime performance and parallel scalability.

1 Introduction

In explicit-state model checking, the correctness of a concurrent system description M is verified with respect to a property φ. This is done by exhaustively exploring $M's$ potential behavior in the form of a *state-space graph*. Explicit-state model checking is still an indispensable technique for formal verification of software systems. However, full verification is severely limited by the need to explore and store the entire state space, which is often exponential in the size of M.

For many years, Moore's law [25] guaranteed exponential advances in computation capabilities, which for model checking meant that larger state spaces – hence more complicated systems – could be analyzed. However, since a few years, due to physical limitations, CPUs no longer deliver sequential speedups with each new generation. Instead, now, the number of *cores* on CPUs grows

** Supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.

exponentially. Only by exploiting this parallelism can one regain the previous growth trends, but parallelizing model checking algorithms is far from trivial.

Recently, it has been shown that original (sequential) verification algorithms based on depth-first search (DFS) can be parallelized efficiently for shared-memory multi-core machines [8,18,15]. These solutions *do not* attempt to parallelize DFS, but instead choose the optimistic approach to run several local DFS-based threads (workers) and lazily communicate sub-results. By randomized traversal, the workers are expected to explore different parts of the state space and communicate little. Results are typically shared in the DFS backtrack, which might not scale in theory, but in practice the algorithms have shown good speedups [8]. First, [19] demonstrated how to perform parallel LTL model checking based on the classic Nested Depth-First Search (NDFS) algorithm [7]. Since then, this technique has been improved in various forms of multi-core NDFS algorithms [8,18,20], and also employed for detecting Strongly Connected Components [24,23]. In the current paper, we focus on the currently best performing version of multi-core NDFS [8,18], which is referred to as a *Combination of MC-NDFSs* (CNDFS), because it is the state-of-the-art solution for multi-core LTL model checking (see Section 2).

Besides exploiting parallelism, another approach to handle larger state spaces is by *partial-order reduction* (POR), which prunes those concurrent interleavings of M' behavior that are irrelevant w.r.t. φ. For each reachable state of M, POR selects a subset of the locally executable transitions, based on a static analysis of the dependency relations in M' behavior. It can yield exponential reductions [31, Section 3]. Since the discovery and subsequent solving of the *ignoring problem* [29], POR can also preserve LTL properties [1]. The ignoring problem occurs when POR indefinitely postpones φ-relevant behavior by selecting similar subsets of transitions at the states along infinite execution traces of M (captured as cycles in its finite state-space graph). This problem can be solved by adding an *ignoring proviso*, i.e. a strengthening condition that limits the possible transition subsets allowed by POR. For LTL, practical ignoring provisos depend on cycles. Due to their heuristic nature and the difficulty of identifying cycles in a graph (see Section 2), efficient provisos have been studied for many years [9].

The combination of multi-core (LTL) model checking and POR benefits from both approaches. The ignoring problem however complicates matters, as its solution depends on detecting cycles in the state space, which are hard to detect in parallel algorithms. Not surprisingly, some existing parallel approaches either increase POR's time complexity and/or reduce reduction capability [3] (c.f. [15]).

In the current paper, we show how the ignoring problem can be handled efficiently using CNDFS and the novel *parallel cycle proviso*. We both mitigate the loss of reductions witnessed in previous parallel POR-enabled model checking algorithms, but also enable the use of several optimizations from [9].

The structure of the paper is as follows: in Section 2, preliminary notions are introduced, and CNDFS and POR are presented. In Section 3, we lift POR to the multi-threaded setting of CNDFS. Section 4 contains our experimental results, and **??** discusses related work. Finally, Section 5 draws conclusions.

2 Preliminaries

We choose an action-based representation of state spaces. The state-space graph $\mathcal{G}$ consists of a (finite) set of vertices or states $\mathcal{S}$, with an initial state $s_0 \in \mathcal{S}$, and a set of edges, the transitions $\mathcal{T} \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$, where Σ represents a set of actions in the system M, e.g. a statement for an imperative-language specification. We call s' a *successor* of s iff $\exists \alpha \colon (s, \alpha, s') \in \mathcal{T}$, denoted as $s \xrightarrow{\alpha} s'$, or $s \rightarrow s'$, in case α is not relevant. Furthermore, we write $s \rightarrow^+ s'$ for $(s, s') \in \mathcal{T}^+$ (transitive closure), and $s \rightarrow^* s'$ for $(s, s') \in \mathcal{T}^*$ (reflexive, transitive closure). A *path* through the state space between states s, s' is denoted by $s \overset{V}{\Rightarrow} s'$, with $V \in \Sigma^*$ a sequence of actions $\alpha_0, \ldots, \alpha_n$ ($n \in \mathbb{N}$ and Σ^* the set of all finite sequences made up of actions in Σ) such that there exist states $s_0, \ldots, s_{n+1}$ with $s_0 = s$, $s_{n+1} = s'$, and $s_i \xrightarrow{\alpha_i} s_{i+1}$ for $0 \leq i \leq n$. We define the set of reachable states as: $\mathcal{R} \equiv \{s \in \mathcal{S} \mid s_0 \rightarrow^* s\}$, i.e. a subset of $\mathcal{S}$, or all syntactically-allowed variable valuations in M.

To reflect the fact that the state space is generated *on-the-fly*, hence not known a-priori, we sometimes use a next-state function $en \colon \mathcal{S} \rightarrow 2^{\mathcal{S}}$ instead of $\mathcal{T}$ directly. On-the-fly checking procedures iteratively query all successors of all visited states, starting from the initial state.

To reason about correctness of *reactive systems*, φ may refer to infinite paths. We consider properties that are already incorporated in the state space of M. Well-known techniques exist to construct such so-called cross-products while still allowing on-the-fly verification [1, Ch.4]). Such state spaces are (finite) Büchi automata $\mathbb{B} = (\mathcal{G}, \mathcal{F})$, where $\mathcal{F} \subseteq \mathcal{S}$ is a set of accepting states. $\mathbb{B}$ accepts ω-regular words VW^ω with $V, W \in \Sigma^*$ and W^ω the infinite repetition of W. A word VW^ω is accepted by $\mathbb{B}$ iff there exists an infinite path labeled VW^ω that reaches an infinite number of accepting states. Since $\mathbb{B}$ is finite-state, this means that there must exist a path $s_0 \overset{V}{\Rightarrow} s \overset{W}{\Rightarrow} s$ and for some $XY = W$, $s \overset{X}{\Rightarrow} s'$, we have $s' \in \mathcal{F}$. Since s' is an accepting state, we call $s \overset{X}{\Rightarrow} s$ an *accepting cycle*. So, finite Büchi automata accept all traces that end in an accepting cycle, i.e. are lasso-formed. An accepted trace represents a counter-example in M to φ, hence the verification problem is reduced to finding accepting cycles that are reachable from the initial state, which can be done in time linear to the size of the state space using, for example, the well-known sequential algorithm NDFS [7].

Multi-core LTL checking. CNDFS [8] is a parallel LTL model checking algorithm, based on NDFS [7]. In NDFS, a DFS is run from s_0 to find reachable accepting states, and from each accepting state $s \in \mathcal{F}$, a *nested* DFS is launched to find a cycle containing s. Because of the order in which states are visited, NDFS runs in time linear to the state space size. For clarity, in the following, we refer to the outer DFS finding accepting states as the *blue* search, and to the nested DFS as the *red* search. These colors relate to how the searches affect the global state of the algorithm. Initially, all states are white. The blue search colors states cyan when it puts states on its stack, and blue when the state is fully explored and popped again from the stack (backtracked). The red search colors states pink when placed on its stack, and red when backtracking.

Algorithm 1 CNDFS with parallel cycle proviso (in boxed lines)

Require: $\forall s \in \mathcal{S}\colon s.prov = ?$

```
 1: procedure cndfs(s_0, P)
 2:   dfsBlue_1(s_0) || ... || dfsBlue_P(s_0)
 3:   report no-cycle
 4: procedure dfsRed_p(s)
 5:   s.pink[p] := true
 6:   R_p := R_p ∪ {s}
 7:   stack sel_p(s) := por(s)
 8:   for all s' ∈ mix_p(sel_p(s)) do
 9:     if s'.cyan[p] then
10:       report accepting cycle
11:     if s' ∉ R_p ∧ ¬s'.red then
12:       dfsRed_p(s')
13:   if sel_p(s)=por(s)∧por(s)⊂en(s) then
14:     new := ∃x ∈ por(s) : x.pink[p]
15:     cas(s.prov, ?, new)
16:     if s.prov = true then
17:       sel_p(s) := en(s)
18:       goto l.8
19:   s.pink[p] := false
20: procedure dfsBlue_p(s)
21:   s.cyan[p] := true
22:   stack sel_p(s) := por(s)
23:   for all s' ∈ mix_p(sel_p(s)) do
24:     if ¬s'.cyan[p] ∧ ¬s'.blue then
25:       dfsBlue_p(s')
26:   if sel_p(s)=por(s)∧por(s)⊂en(s) then
27:     new := ∃x ∈ por(s) : x.cyan[p]
28:     cas(s.prov, ?, new)
29:     if s.prov = true then
30:       sel_p(s) := en(s)
31:       goto l.23
32:   s.blue := true
33:   if s ∈ F then
34:     R_p := ∅
35:     dfsRed_p(s)
36:     await ∀s' ∈ R_p ∩ F \ {s}: s'.red
37:     forall s' ∈ R_p do s'.red := true
38:   s.cyan[p] := false
```

In CNDFS, several workers explore the state space mostly independently by each running a randomized NDFS; it is randomized w.r.t. the order in which the successors of each state are visited. Algorithm 1 without the boxed code (lines 13-18 and 26-31), and with $sel_p(s) = en(s)$ at l.7 and l.22, shows CNDFS. The algorithm is called for a given number of workers P. Each worker p starts by executing $dfsBlue_p(s_0)$, which starts the blue search. A local set of successor states $sel_p(s)$ is initialized to $en(s)$ at l.22. For clarity, we use a notation that distinguishes such sets for different p and s, but in practice, a stack-local variable is sufficient, i.e. the full definition of a function sel_p does not need to be maintained throughout the search. This is indicated with the **stack** keyword. Randomization of visiting successors of s is achieved through the function mix_p. If a state s is accepting, a red search is launched from it (l.35), to try to find a cycle containing s. In the red search, again local state sets are used to inspect successors (l.7). A cycle containing s is detected once a cyan state is reached (l.9). Since a cyan state is on the stack of the blue search, and accepting state s from which the red search has been launched is at the top of this stack, reaching any cyan state means that a complete cycle exists containing s.

CNDFS scales particularly well because some information is shared between workers. The blue color is shared between blue searches, hence when one worker has colored a state blue, other workers will not explore it anymore (l.24) (of course, local cyan states are also not added to the stack). In principle, one would also like to share the red color between red searches. It has been shown [8], however, that this cannot be done in a similar fashion. For correctness, one can

only share this information once a red search has completely terminated. For this reason, we use a worker-local red set $\mathcal{R}_p$, consisting of the states that have been explored by the red search of worker p, which is constructed as a red search continues, and only made globally red (l.37) once the worker knows that all out-of-order red searches in the same search region have terminated (l.36).

CNDFS's complexity can be linear in the size of the graph, and its scalability is good (although for some graphs, its performance reduces to that of sequential NDFS). CNDFS has been shown to perform better than the fixpoint-based OWCTY algorithm [2] – for many years the best known algorithm for parallel LTL model checking – which has a worst-case quadratic complexity.

Partial-order reduction. POR prunes interleavings by constraining the next-state function en. This selection should preserve the property at hand (safety or liveness), and can be performed with the *state-local* information combined with some static analysis of M (mainly involving the commutativity of its operations).

A (state-local) POR function prunes $\mathcal{G}$ *on-the-fly* during exploration by selecting in each state s a subset of the enabled transitions, the *reduced set* $por(s) \subseteq en(s)$. POR definitions often allow multiple valid reduced sets, including trivially $por(s) = en(s)$. Smaller reduced sets often lead to smaller state spaces, but this is not necessarily the case [31]. Therefore, POR is heuristic in nature.

Over the years, several techniques have been developed to select sufficient subsets of enabled transitions, such as the stubborn set technique [30]. Since the selection of subsets of enabled transitions is orthogonal to our proposed CNDFS algorithm with POR, we consider the subset selection algorithm as a given, implemented with a function $por : \mathcal{S} \to 2^{\mathcal{S}}$. For a detailed explanation of the implementation of a POR subset selection algorithm, see [21].

The ignoring problem. Valmari identified the incompleteness of POR with respect to the preservation of liveness properties [29]. As liveness properties reason over infinite paths in $\mathbb{B}$, (state-local) POR may exhibit ignoring, i.e. continuously exclude actions leading to counter examples from the reduced set. The introduction of the *ignoring proviso* solved this, by forcing the involvement of all relevant transitions when constructing reduced sets. Because these ignoring provisos depend on *global* properties of $\mathbb{B}$, i.e. cycles in its $\mathcal{G}$, we first define a dynamic function sel which relaxes the por function such that $por(s) \subseteq sel(s) \subseteq en(s)$..

The exact dynamic definition of sel will be part of the on-the-fly exploration algorithms presented in the subsequent section. The definition of the proviso now depends on the reduced state-space graph induced by sel a posteriori (after the termination of the exploration algorithm). We denote this reduced graph $\overline{\mathcal{G}} = (s_0, \overline{\mathcal{T}}, \mathcal{S})$, with $\overline{\mathcal{T}} \subseteq \mathcal{T}$ such that each $(s, \alpha, s') \in \mathcal{T}$ is also in $\overline{\mathcal{T}}$ iff $s' \in sel(s)$. Transitions in this graph are denoted $s \overline{\Rightarrow} s'$, and we define $\overline{\mathcal{R}} \equiv \{s \in \mathcal{S} \mid s_0 \overline{\Rightarrow}^* s\}$. We define $\overline{\mathbb{B}}$ as $(\overline{\mathcal{G}}, \mathcal{F})$, so that any mention of $\overline{\mathcal{G}}$ and $\overline{\mathcal{R}}$ generalizes to $\overline{\mathbb{B}}$.

The ignoring proviso can now be defined on the reduced state space. (Ignoring provisos weaker than the follwoing exist, see e.g. [32], but their refinements are orthogonal to the cycle detection problem that we aim to solve here.)

Cycle Along each cycle in $\overline{\mathcal{G}}$, at least one state s is *fully explored* ($sel(s) = en(s)$).

An implementation of the *Cycle* ignoring proviso thus needs to identify cycles on-the-fly and include all transitions of at least one state on each cycle. However, selecting the smallest set of states covering all (possibly overlapping) cycles is an NP-complete problem, known as the *vertex feedback set* in graph theory [13]. Therefore, in practice, this proviso is implemented using DFS, which guarantees full exploration of at *least one* state on each cycle, *and* can be performed efficiently [9]. In this (stronger) form, the proviso is as follows:

Stack When running a depth-first search (DFS) over $\overline{\mathcal{G}}$, each state $s \in \overline{\mathcal{R}}$ that has a successor $s' \in por(s)$ on the stack, should be fully explored.

The *Stack* proviso overestimates the amount of states to explore fully, but has been improved over the years to yield excellent reductions [9].

In the next section, we present how to detect cycles in parallel, enabling POR in that setting. The preciseness that is achieved by this method is expected to be better than in related parallel solutions (see Section 4 for experimental results).

3 Multi-Core Partial-Order Reduction

In the current section, we present a parallel cycle proviso for both safety and liveness properties, for use in parallel DFS-based algorithms. The presented algorithms indirectly implement the *sel* function on which the reduced state space $\overline{\mathcal{R}}$ was defined in the previous section. While we are mainly interested in a complete solution for LTL model checking, we commence with a solution for safety properties, in order to introduce the approach in a stepwise fashion.

3.1 Partial-Order Reduction for Safety Properties

Checking safety properties can be done through reachability analysis. To show how the ignoring problem can be solved for safety properties, we introduce a parallel algorithm that launches multiple DFS workers. We refer to this approach as *parallel DFS*. While it is not the most efficient approach to do reachability analysis – for a better approach see [16] – parallel DFS provides a nice first step towards combining POR with CNDFS.

Safety properties are preserved by a weaker version of the ignoring proviso. One such version concerns bottom Strongly Connected Components (SCCs), i.e. SCCs without outgoing transitions:

BottomSCC For all states $s \in \overline{\mathcal{R}}$ of $\overline{\mathcal{G}}$, there exists a fully explored state s' ($sel(s') = en(s')$) such that $s \Rightarrow^* s'$. (this boils down to having one fully explored state in each bottom SCC of $\overline{\mathcal{G}}$, c.f. [32]).

Our parallel DFS with POR should detect at least one state in all bottom SCCs in $\mathcal{G}$. Valmari's SCC method is optimal [29] for this purpose. However, we use the stronger *Stack'* condition, which serves our introductory purpose better as it resembles the ignoring proviso required to preserve LTL (see the *Stack* proviso in the previous section):

Algorithm 2 Parallel DFS with POR

Require: $\forall s \in \mathcal{S}\colon s.blue = false$
Require: $\forall s \in \mathcal{S}, p\colon s.cyan[p] = false$
1: **procedure** $pardfs(s_0, P)$
2: $dfs_1(s_0)$ || ... || $dfs_P(s_0)$
3: **report no-cycle**

A: $cas(s.prov, ?, \forall x \in por(s) : x.cyan[p])$
B: **if** $s.prov =$ **true then**
C: $sel_p(s) := en(s)$
D: **goto** l.7

4: **procedure** $dfs_p(s)$
5: $s.cyan[p] :=$ **true**
6: **stack** $sel_p(s) := por(s)$
7: **for all** $s' \in mix_p(sel_p(s))$ **do**
8: **if** $\neg s'.cyan[p] \wedge \neg s'.blue$ **then**
9: $dfs_p(s')$
10: **if** $sel_p(s){=}por(s){\wedge}por(s){\subset}en(s)$ **then**
11: **if** $\forall x \in por(s) : x.cyan[p]$ **then**
12: $sel_p(s) := en(s)$
13: **goto** l.7
14: $s.blue :=$ **true**
15: $s.cyan[p] :=$ **false**

Stack' When running a DFS exploration over $\overline{\mathcal{G}}$, each state $s \in \overline{\mathcal{R}}$ for which *all* successors in $por(s)$ are on the stack, should be fully explored.

Consider Algorithm 2 without the boxed lines (lines 10–13 and A–D) and with $sel_p(s) = en(s)$ at l.6. (We use a local sel_p to explain how workers communicate successor sets, the global sel is defined later.) It starts P parallel DFS workers at l.2, which initially each independently traverse the state space (see the local cyan color at l.5 and l.15, indicating that a state is currently on the DFS stack). When backtracking, the workers communicate by marking states globally as visited at l.14 (with the color blue). Clearly, this algorithm explores all reachable states, and hence terminates on finite state spaces, since a state s is only colored globally blue once all states reachable from s have either been explored (are colored blue) or are going to be explored in the future (are colored cyan).

To introduce POR for safety properties, the ignoring proviso *BottomSCC* needs to be satisfied. We show that this is done by adopting the *Stack'* proviso in parallel DFS as a parallel *Stack'* proviso at l.10-13. The resulting algorithm will find at least one state on each cycle, and this state will be fully explored by at least one worker. At l.6, $sel_p(s)$ is now actually set to $por(s)$, and l.10 checks whether this is still true. This ensures that the proviso check is performed at most once for each state s on the stack. When all successors of s are on the stack ($\forall x \in por(s) : x.cyan[p]$) (l.11), the premise of the *Stack'* proviso holds, and all successors are selected for visiting at l.12 (to satisfy the conclusion of the proviso), before restarting the for loop at l.13. (The redundant reselection of $por(s) \subset en(s)$ can be avoided, but is used here to simplify our proofs.) The second time that l.10 is reached, $sel_p(s)$ is no longer set to $por(s)$, so the check is not performed a second time. To handle the special case that $por(s) = en(s)$, we require at l.10 that $sel_p(s) \subset en(s)$. Otherwise, an infinite goto loop would occur.

In the following proofs, we assume that each line of the code is executed atomically. The global state of the algorithm is the coloring of $\overline{\mathcal{R}}$ and the program counter of each worker. We use the following notations: The sets C_p and B contain all the states colored cyan by worker p, and globally blue, respectively.

For example, $s.cyan[p] = \textbf{true}$ is expressed as $s \in C_p$. To reason on the a posteriori explored graph, we define $sel(s) = en(s)$ iff $\exists p \in \{1 \dots P\}\colon sel_p(s) = en(s)$, and $sel(s) = por(s)$ otherwise (notice that sel_p(s) only grows). Finally, we use the modal operator $s \in \Box X$ to reason about the successors of s in $\overline{\mathcal{G}}$, i.e. $\forall s' \in sel(s)\colon s' \in X$, and for local successors: $s \in \Box_p X \Rightarrow \forall s' \in sel_p(s)\colon s' \in X$. We write $F_p(s)@L$ to indicate that thread p is about to execute l.L of function F).

The first lemma shows colorings of local successors of backtracked states, while the second relates backtracked states to the coloring of global successors:

Lemma 1. *When worker p marks a state s blue, its local successors are blue or cyan local to worker p: $dfs_p(s)@14 \Rightarrow s \in \Box_p(B \cup C_p)$.*

Proof. At l.14, each local successor s' has either been skipped at l.8 (so $s' \in B \cup C_p$), or $dfs_p(s')$ had been called at l.9 leading to $t \in B$. So $s' \in (B \cup C_p)$. □

Lemma 2. *Global successors of blue states that are not cyan, are blue or cyan: $\bigcup_p (B \setminus C_p) \subseteq \Box \bigcup_p (B \cup C_p)$.*

Proof. Initially, $\bigcup_p (B \setminus C_p)$ is empty and the lemma holds. A state s is added to this set when the last worker p reaches $dfs_p(s)@l.15$. Locally, we have $s \in \Box_p(B \cup C_p)$ by Lemma 1, but since all workers backtracked $s \in \bigcup_p \Box(B \cup C_p)$ holds as well. Finally, states are never removed from $B \cup C_p$. □

To reason about states for which the proviso's conclusion holds, we consider all states s with $sel(s) = en(s)$, i.e. *inviolable* states, as belonging to a set I, and all others with $sel(s) \subset en(s)$ as belonging to a set N (violable states).

Lemma 3. *In Algorithm 2, each blue state s can reach an inviolable state s': $\forall s \in B\colon (\exists s' \in I\colon s \Rightarrow^* s')$.*

Proof. B is only modified at l.14. I and N are 'modified' right before l.14.

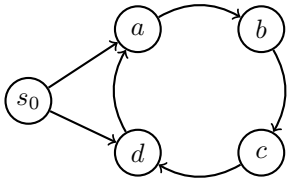
Initially, B is empty, so the lemma holds. By Lemma 1, when the first state s is marked blue, it will have blue and cyan successors. Since at that point, there are no blue states yet, all successors of s must be cyan. But then, s must be inviolable at l.14, so $s \in I$ (if $por(s) = \emptyset$, then $en(s) = \emptyset$, since POR does not introduce deadlocks). All subsequent states marked blue either are identified as inviolable, satisfying the lemma with $s \equiv s'$, or have at least one blue successor $s' \not\equiv s$, for which the theorem already holds. □

Finally, by showing that from each state reached by parallel DFS, an inviolable state is reachable, we clearly show that *BottomSCC* is satisfied.

Theorem 1. *Algorithm 2 explores all $s \in \overline{\mathcal{R}}$, and satisfies BottomSCC: $\forall s \in \overline{\mathcal{R}}\colon (\exists s' \in I\colon s \Rightarrow^* s')$.*

Proof. Due to l.10 and l.12, the goto can only be executed once per state. And since the set $B \cup \bigcup_p C_p$ grows monotonically in Algorithm 2, eventually the algorithm terminates for finite input graphs (see Section 8). By the obvious post-condition of DFS-based algorithms, we have $s_0 \in B$ at that moment. By Lemma 2, and the fact that $\bigcup_p C_p = \emptyset$ (the stacks are empty), we have $B \subseteq \Box B$. Hence, $\overline{\mathcal{R}} = B$, and it follows from Lemma 3 that *BottomSCC* is satisfied. □

Algorithm 2 has the downside that it could identify more inviolable states than strictly necessary, as the following example shows.



Example 1. The cycle in the graph on the right has multiple entrypoints. When different workers (with different search orders) enter the cycle differently, they determine a different inviolable state: a worker A entering via a will choose d (as it finds a to be cyan after traversing the cycle), while a worker B entering via d chooses c.

A coherent view of the state space. The problem that Algorithm 2 still exhibits, is that different workers obtain a different view of the state space, as they identify different inviolable states. As these are fully explored, the reduction can be limited. Therefore, we introduce synchronization between threads on their decision whether a state is inviolable or not. To realize this, we add a 3-valued variable per state called *prov*, initially set to *unknown* ('?'). This variable is global, hence workers can communicate with each other through the *prov* variables.

The boxed code at lines A–D should replace the parallel proviso check of lines 11–13. Upon backtracking, threads use the well-known atomic compare-and-swap (*cas*) operation to communicate their decision on a first-come-first-serve basis. The *cas* operation is defined as follows: $cas(x, v_1, v_2)$ atomically checks if variable x has value v_1, and if so, sets x to v_2. This solution does not completely prevent redundant inviolable states (w.r.t. to the *Stack'* proviso), but it can prevent some. For instance, in Example 1, c can be prevented from becoming inviolable, if worker A backtracks over c before worker B, marking c as violable.

Correctness of the modified algorithm follows from Lemma 4. It reasons on the states whose violability status has been determined ($s.prov \neq ?$) or is known upfront ($por(s) = en(s)$), captured by the final set: $F = \{s \in \mathcal{R} \mid s.prov \neq ? \vee por(s) = en(s)\}$. The lemma shows that when a state s is determined to be violable ($s \in F \cap N$), then s has a blue successor (for which Lemma 3 holds).

Lemma 4. *At least one global successor of a permanently violable state is blue:* $F \cap N \cap \Box B \neq \emptyset$.

Proof. A state s is added to $F \cap N$ after l.A sets $s.prov$ to **false**. By Lemma 1 and the fact that $\forall x \in por(s)\colon x.cyan[p]$ evaluated to **false**, we know there must be one blue successor (again if $por(s) = \emptyset$, then $en(s) = \emptyset$ and $s \notin N$). □

3.2 Partial-Order Reduction for Liveness Properties

For liveness properties, the *Cycle* proviso needs to hold (Section 2), which is met in finite state-space graphs if along all cycles at least one state is fully explored. In addition to this, CNDFS should search for accepting cycles, which constitute counter-examples (instead of $\mathcal{G}$, the algorithms now work on $\mathbb{B}$). In the current subsection, we show that CNDFS with POR and a novel *parallel cycle proviso*, similar to *Stack*, fulfills *Cycle*. First, we discuss how we solve a related problem.

Traditional (sequential) NDFS detects accepting cycles by launching a nested (red) DFS search from each accepting state found in the outer (blue) DFS search

(see Section 2). Combining NDFS with POR and the *Stack* proviso yields a so-called *revisiting problem* [12]; in order for NDFS with POR to be complete, it is crucial that for every state s, the selection of $sel(s)$ is deterministic. This means that two constraints must be satisfied: (1) if a state is deemed inviolable, then all searches reaching it must be aware of this and select all successors, and (2) if a state is violable, the same subset must always be selected by the *por* function.

For the NDFS algorithm in [28], the revisiting problem can solved straightforwardly, by only selecting blue and cyan successors in the red search [28, Sec. 6]. This enforces that each state reached in a red search is explored in the same way as previously done in the blue search. However, in CNDFS, this approach does not apply because different searches run out of order executions; in particular, red searches may sometimes visit white states (see the proof of [8, Prop. 3]).

However, the revisiting problem of CNDFS with POR can be solved as follows. First of all, it is crucial that the subset selection mechanism is deterministic. This can be achieved efficiently, e.g. via guard-based POR [21]. Second of all, the decisions regarding proviso status of states is made global via synchronization methods similar to those used in the previous subsection, now also indicated by the boxed code in Algorithm 1. It implements the *Stack* proviso in both the blue and the red DFS. In the blue DFS, we check for the existence of at least one cyan successor (l.27), and in the red DFS for the existence of a pink successor (l.14). The mechanism to store the results using *cas* and $s.prov$ is exactly as presented earlier for parallel DFS. This implements our parallel cycle proviso.

In the following correctness proofs, we refer with P_p to the pink states of worker p and with *Red* to the (globally) red states. We also construct a set of states backtracked in a red search: $R \equiv \bigcup_p (\mathcal{R}_p \setminus P_p) \cup Red$ (all states that are either in some local $\mathcal{R}_p$ but not on the pink stack, or globally marked red).

Now that we have $sel_p(s) = sel(s)$ at l.32, we can relate blue states to their (global) successor colorings (the proof is similar to that of Lemma 1):

Lemma 5. *Successors of blue states are blue or cyan:* $B \subseteq \Box \bigcup_p (B \cup C_p)$.

The next lemma expresses that for backtracked states, a decision has been made concerning their violability status.

Lemma 6. *Blue states and states backtracked in a red search have been considered for violability:* $B \cup R \subseteq F$.

Proof. A state s is colored blue at l.32. If $por(s) = en(s)$, then $s \in F$. If $por(s) \subset en(s)$, then l.28 has been executed, hence $s \in F$. A state s is colored red at l.37. There, we have $s \in \mathcal{R}_p \wedge s \notin P_p$, hence already $s \in R$. Also, at l.19, a state s is in R, since $s \notin P_p$. But then, either $por(s) = en(s)$, and $s \in F$, or $por(s) \subset en(s)$, and l.15 has been executed, so $s \in F$. □

The following lemmas help to prove Theorem 2, expressing that Algorithm 1 satisfies the *Stack* proviso, which implies that *Cycle* is satisfied.

Lemma 7. *Successors of states backtracked in the red search have also been backtracked in the red search or are pink:* $R \subseteq \Box \bigcup_p (R \cup P_p)$.

Proof. Since $R \equiv \bigcup_p(\mathcal{R}_p \setminus P_p) \cup \mathit{Red}$, we have $\bigcup_p(R \cup P_p) \equiv \bigcup_p((\mathcal{R}_p \setminus P_p) \cup P_p) \cup \mathit{Red}$, so we need to prove that $R \subseteq \Box\bigcup_p(\mathcal{R}_p \cup \mathit{Red})$. A state s is added to R when it is removed from P_p at l.19, since at that point $s \in \mathcal{R}_p$. At l.34, if $\mathcal{R}_p$ is non-empty, states are removed from $\mathcal{R}_p$, but those were added to *Red* at l.37 after the previous $\mathit{dfsRed}_p(s)$ terminated. Once added to *Red* (and R), states are never removed again. At l.19, all successors t have been considered at l.9–12. If $t \notin \mathcal{R}_p \cup \mathit{Red}$, then $\mathit{dfsRed}_p(t)$ is executed adding t to $\mathcal{R}_p$. So at l.19, we have $s \in \Box(\mathcal{R}_p \cup \mathit{Red})$. □

Lemma 8. *Successors of permanently violable states are blue or backtracked in a red search:* $F \cap N \subseteq \Box(B \cup R)$.

Proof. A state s is permanently marked violable before l.19 and l.32. Because the conditions at l.14, resp. l.27, do not hold there, no successor s' of s can be pink, resp. cyan. By Lemma 7 (resp. Lemma 5), all s' are in R (resp. B). □

Theorem 2. *Algorithm 1 explores all states in* $\overline{\mathcal{R}}$ *of* $\overline{\mathbb{B}}$*, and fully explores one state on each cycle in* $\overline{\mathbb{B}}$.

Proof. The termination proof is analogous to that in Theorem 1.

We prove that the proviso holds by contradiction. Assume Algorithm 1 ran to completion, and as a result some cycle $C = s_1 \to \cdots \to s_n \to s_1$ contains no inviolable state: $\forall i \in \{1 \ldots n\}\colon s_i \in N$. Take the last time that a state s_x with $x \in \{1 \ldots n\}$ on C was permanently added to N ($s_x.\mathit{prov}$ is set to **false** at that moment). At this time, some worker p must be executing either l.15 or l.28. The immediate predecessor s_y of s_x on the cycle must have been in $F \cap N$ before s_x is marked, since s_x was the last state to be permanently marked violable. Therefore, by Lemma 8, $s_x \in B \cup R$. But then, by Lemma 6, $s_x \in F$. The latter contradicts our assumption that s_x is last marked permanently violable, which can only happen if its proviso flag is still set to '?', i.e. $s_x \notin F$ or $s_x \in I$. □

4 Experimental Evaluation

Experimental Setup We implemented Algorithm 1 in the LTSmin toolset. This toolset [17] is a language-independent model checker and supports POR since version 1.6. To this end, LTSmin's Pins interface was extended with new functions in order to export the necessary static information [21]. We experimented with DVE models from the BEEM database [27] and Promela models [11]; both are supported by LTSmin via different language modules [4,17]. The selected models and properties are presented in Table 1, and include industrial case studies in Promela as well as representable instances from the large BEEM database. We focus on instances where the properties hold, because on-the-fly bug-hunting is not a bottleneck in our experience [8]. We performed experiments with version 2.1 of LTSmin.[4] All experiments were repeated 10 times on a quadruple AMD Opteron 6376 CPU with 64 cores and 512GB RAM memory.

[4] http://fmt.cs.utwente.nl/tools/ltsmin/ (see [8] for command lines)

Table 1. DVE/PROMELA models and LTL properties used (all correct)

Model (DVE)	Property
leader_filters.7	$\Diamond(\#elected \neq 0)$
elevator.3	$\Box(in \Rightarrow (\Diamond out))$
leader_election.*	$\Diamond(\#leaders \neq 0)$
anderson.6	$\Box(req \Rightarrow \Diamond CS)$

Model (PROMELA)	Property
garp	$\Box\Diamond progress$
iprotocol-2	$\Box\Diamond progress$
pacemaker_distibuted	$\Box(p \wedge (q \Rightarrow r))$
pacemaker_concurrent	$\Box((p \Rightarrow q) \wedge (r \Rightarrow s))$

The results presented here focus primarily on the efficiency of the reduction of the parallel cycle proviso in LTL model checking. The main question that is answered is whether the parallel cycle proviso introduces too many inviolable states with respect to the sequential cycle proviso. We also did some analysis on the obtained scalability of CNDFS with POR, mainly to confirm that scalability is not lost; in the past, CNDFS has shown to scale well and often better than other parallel LTL model checking algorithms [4,8]. The complete set of experimental results are available at http://fmt.cs.utwente.nl/tools/ltsmin/hvc-2014.

We would have preferred to compare the parallel cycle proviso with the topological sort proviso [3] in DIVINE (see **??**), the most sophisticated solution thus far, but the POR algorithm in DIVINE delivers less reductions than LTSMIN's stubborn set implementation making a tool-by-tool comparison senseless. We did not reimplement the topological sort proviso because it seems impossible to combine it with CNDFS. Instead, we compare our conclusions with [3].

Reduction Performance. Sequentially, the CNDFS algorithm is equal to the NDFS algorithm modulo the fact that states are not instantly colored red, but only after the nested search [8]. Similarly, the parallel cycle proviso should be equal to the stack proviso when run with one thread. With increasing parallelism, the algorithm has the potential to select more states as inviolable as

Table 2. POR reductions (percentages) without ignoring proviso, with stack proviso and with parallel cycle proviso (with multiple threads) averaged over 10 runs.

Model	$\lvert\mathcal{R}\rvert$	None	Stack	Parallel cycle proviso (threads) 1	4	8	16	32	64
leader_filters.7	26,302,351	2.35	2.35	2.35	2.35	2.35	2.35	2.35	2.35
elevator.3	495,463	92.46	92.86	94.20	94.49	94.64	94.77	94.85	94.96
leader_election.4	746,051	3.02	3.02	3.02	3.02	3.02	3.02	3.02	3.02
leader_election.6	35,773,430	0.69	0.69	0.70	0.69	0.69	0.69	0.69	0.69
anderson.6	29,315,027	15.80	33.11	48.43	52.28	52.83	52.93	52.34	51.71
garp	67,108,837	6.25	18.68	18.69	20.23	20.85	20.69	20.64	20.79
peterson4	67,108,842	14.19	15.82	15.52	15.60	15.64	15.63	15.67	15.67
iprotocol-2	18,998,110	30.95	32.24	34.80	36.31	36.71	37.10	37.46	37.91
pacemaker_distributed	67,108,832	31.13	47.89	47.81	47.86	47.94	47.98	48.16	48.26
pacemaker_concurrent	18,092,815	42.06	46.05	45.90	45.88	45.92	45.92	45.96	46.00

Table 3. CNDFS runtimes (sec.) without POR (*Full*), without ignoring (*None*), with stack proviso (*Stack*), and with parallel cycle proviso averaged over 10 runs.

Model	Full	None	Stack	Parallel cycle proviso (threads) 1	4	8	16	32	64
leader_filters.7	85.32	5.59	5.42	5.60	1.46	0.82	0.45	0.26	0.19
elevator.3	1.83	196.17	185.24	228.41	78.14	47.54	29.50	19.12	14.75
leader_election.4	5.68	6.33	6.16	6.51	2.28	1.93	1.23	1.23	1.93
leader_election.6	399.94	0.88	0.84	0.90	0.23	0.13	0.07	0.04	0.04
anderson.6	168.10	29.55	64.42	121.74	63.57	43.90	29.55	19.53	14.87
garp	426.06	15.66	52.09	62.52	28.54	18.75	12.03	7.69	5.94
peterson4	287.62	30.30	35.93	39.67	11.83	6.56	3.85	2.19	1.49
iprotocol-2	68.90	84.39	85.31	115.31	40.07	23.70	14.47	8.69	6.28
pacemaker_distributed	211.65	99.56	156.25	167.62	43.88	23.66	13.25	7.33	4.86
pacemaker_concurrent	55.16	256.92	332.50	342.65	88.68	46.01	24.72	12.88	7.97

explained in Example 1. We are interested in determining these relative differences in reductions (between the stack proviso and the parallel cycle proviso). As a measurement, we choose the total number of states stored in the hash table. Although the relation between reduced state space size and number of inviolable states is only heuristic (exploring a different, but larger subset of states fully could yield a smaller reduced state space [10]), we are unaware of a better measurement.

Table 2 shows the size of the reduced state spaces relative to the original state space. For completeness, we also included the results without any ignoring proviso (which might miss counter-examples). All models show similar parallel reductions to the stack proviso, except `anderson.6`. We suspect that this is caused by a slightly more efficient implementation of the stack proviso, in particular concerning the revisiting problem, in the sequential nested search which does not work in a parallel setting (see [28] and discussion in Section 3.2).

A slight decrease in reductions when the number of threads is increased can be observed, the effect is however minimal and often sublinear with the most increase caused by 4 threads already. Hence we can conclude that CNDFS with POR does not cause too many redundant full explorations. This is a surprising result, as the parallel benchmarks in [3] seem to show a steep decrease in reduction performance.

We cannot explain precisely why the reductions sometimes improve with more parallelism, e.g. `anderson.6` (recall that we present averages over 10 experiments). The behavior might be caused by different thread schedulings.

Runtime Performance. Table 3 shows that the runtimes of CNDFS with POR are similar to those of NDFS with stack proviso (discounting the state space difference for `anderson.6`). The overhead of the proviso bits is thus minimal.

Scalability. Figure 1 shows that CNDFS with POR exhibits good speedups for larger models (see Table 3). Comparing these speedups to those obtained earlier

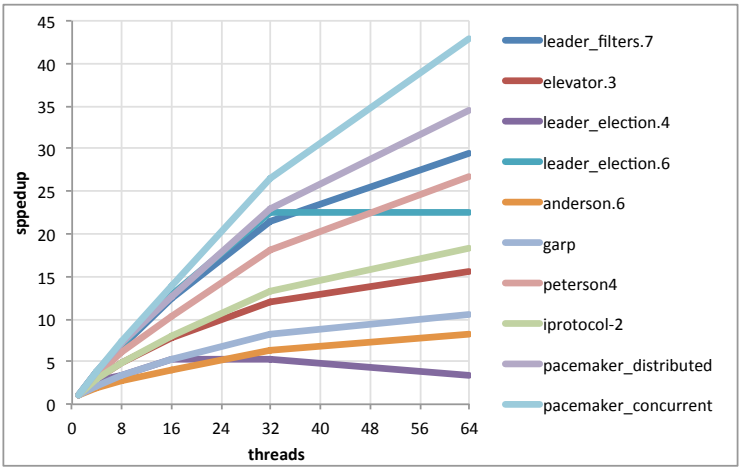


Fig. 1. Plot of parallel scalability (speedup) of CNDFS with POR.

without POR [8], we see that they are largely unaffected. It is not surprising that the smaller (reduced) `leader-election` model with only a few thousand states exhibits sublinear speedup.

5 Related Work

Applying POR when model checking liveness properties involves an ignoring proviso which may cause orders of magnitude loss in reductions (c.f. [15,9]). In a sequential setting, the use of DFS-based algorithms could mitigate these losses almost completely in the past (c.f. [15,9,29]). However, those techniques cannot be used in parallel, i.e. multi-threaded, shared memory, model checking.

In related work, several other attempts have been made to implement the ignoring proviso for similar parallel settings.

1. The *topological sort proviso* [3] uses the distributed Kahn algorithm for topological sort. When the sort is incomplete due to cycles, these nodes are removed (fully explored) and the algorithm is restarted up to fixpoint.
2. The *two-phase proviso* [26] skips new states with singleton reduced sets, trivially avoiding cycles by fully exploring other states.
3. A distributed version of SPIN [22], implements the *Stack'* proviso for safety properties, while conservatively assuming that successors maintained by other workers are on their respective stacks.
4. A stronger alternative for the *Stack* proviso is one tailored for BFS, where all states reaching queued states are fully explored [5,6].
5. Static POR identifies cycles already in the system specification [14].

All of the above methods have either shown to offer significantly less reduction than the *Stack* proviso (4 and 5), only work for safety properties (3), or have shown a degrading performance when the amount of parallelism is increased, often already noticable with 4–8 workers (1 and 2).

Finally, Evangelista and Pajault [9] further optimized the *Stack* proviso to avoid unnecessary full explorations on overlapping cycles and on cycles that already contain fully explored states. We are the first to adopt these optimizations in a parallel setting.

6 Conclusions

In this paper, we proposed how POR can be integrated in parallel DFS-based search algorithms, in particular in both a parallel DFS reachability algorithm, and CNDFS for on-the-fly LTL model checking. The used parallelization technique is very promising, since very good speedups occur in practice.

To integrate POR, the main challenge was to ensure that when confronted with cycles, the parallel threads explore beyond them, i.e. they do not continuously ignore actions that may lead to new reachable states. This is known as the ignoring problem. Furthermore, for completeness, the two DFS searches in NDFS need to agree on which transitions are explored from each state, and for CNDFS, earlier solutions for this revisiting problem are not correct. We proposed solutions for both these problems. Experimental results indicate that our solution for CNDFS does not harm the scalability of it, while reductions are achieved that are comparable when applying POR in a sequential NDFS.

For future work, the ideas from the color proviso [9] could be incorporated in the parallel cycle proviso, since both are based on a stack check. We expect similar improvements as witnessed in [9].

Acknowledgements. We thank Tom van Dijk for providing access to the 64-core machine at the FMT department of the University Twente.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
2. Barnat, J., Brim, L., Ročkai, P.: A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In: ICFEM'09. LNCS, vol. 5885, pp. 407–425. Springer (2009)
3. Barnat, J., Brim, L., Ročkai, P.: Parallel Partial Order Reduction with Topological Sort Proviso. In: SEFM'10. pp. 222–231. IEEE Computer Society (2010)
4. van der Berg, F., Laarman, A.: SpinS: Extending LTSmin with Promela through SpinJa. ENTCS 296, 95 – 105 (2013)
5. Bošnački, D., Holzmann, G.: Improving Spin's Partial-Order Reduction for Breadth-First Search. In: SPIN'05, LNCS, vol. 3639, pp. 91–105. Springer (2005)
6. Bošnački, D., Leue, S., Lluch-Lafuente, A.: Partial-Order Reduction for General State Exploring Algorithms. STTT 11(1), 39–51 (2009)
7. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: CAV'90. LNCS, vol. 531, pp. 233–242. Springer (1990)

8. Evangelista, S., Laarman, A., Petrucci, L., Pol, J.v.d.: Improved Multi-core Nested Depth-First Search. In: ATVA. LNCS, vol. 7561, pp. 269–283. Springer (2012)
9. Evangelista, S., Pajault, C.: Solving the Ignoring Problem for Partial Order Reduction. STTT 12, 155–170 (2010)
10. Geldenhuys, J., Hansen, H., Valmari, A.: Exploring the Scope for Partial Order Reduction. In: ATVA'09. pp. 39–53. LNCS, Springer (2009)
11. Holzmann, G.: The model checker SPIN. IEEE TSE 23, 279–295 (1997)
12. Holzmann, G., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: SPIN'96. pp. 23–32. American Mathematical Society (1996)
13. Karp, R.M.: Reducibility among Combinatorial Problems. In: Complexity of Computer Computations, pp. 85–103. IBM Research Symposia Series, Springer (1972)
14. Kurshan, R., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static Partial Order Reduction. In: TACAS'98. LNCS, vol. 1384, pp. 345–357. Springer (1998)
15. Laarman, A., Faragó, D.: Improved On-The-Fly Livelock Detection. In: NFM'13. LNCS, vol. 7871, pp. 32–47. Springer (2013)
16. Laarman, A., van de Pol, J., Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: FMCAD'10. pp. 247–255. IEEE-CS (2010)
17. Laarman, A., van de Pol, J., Weber, M.: Multi-Core LTSmin: Marrying Modularity and Scalability. In: NFM'11. LNCS, vol. 6617, pp. 506–511. Springer (2011)
18. Laarman, A.: Scalable Multi-Core Model Checking. Ph.D. thesis, University of Twente (2014)
19. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core Nested Depth-First Search. In: ATVA. LNCS, vol. 6996, pp. 321–335. Springer (2011)
20. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV, LNCS, vol. 8044, pp. 968–983. Springer (2013)
21. Laarman, A.W., Pater, E., Pol, J.v.d., Weber, M.: Guard-Based Partial-Order Reduction. In: SPIN'13. LNCS, vol. 7976, pp. 227–245. Springer (2013)
22. Lerda, F., Sisto, R.: Distributed-Memory Model Checking with SPIN. In: SPIN'99. LNCS, vol. 1680, pp. 22–39. Springer (1999)
23. Liu, Y., Sun, J., Dong, J.: Scalable multi-core model checking fairness enhanced systems. In: Breitman, K., Cavalcanti, A. (eds.) FMSE, LNCS, vol. 5885, pp. 426–445. Springer (2009)
24. Lowe, G.: Concurrent Depth-First Search Algorithms. In: TACAS'14. LNCS, vol. 8413, pp. 202–216. Springer (2014)
25. Moore, G.E.: Cramming more Components onto Integrated Circuits. Electronics 38(10), 114–117 (1965)
26. Nalumasu, R., Gopalakrishnan, G.: An Efficient Partial Order Reduction Algorithm with an Alternative Proviso Implementation. FMSD 20(3), 231–247 (2002)
27. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN'07. LNCS, vol. 4595, pp. 263–267. Springer (2007)
28. Schwoon, S., Esparza, J.: A Note on On-the-Fly Verification Algorithms. In: TACAS'05. LNCS, vol. 3440, pp. 174–190. Springer (2005)
29. Valmari, A.: A Stubborn Attack On State Explosion. In: CAV'91. pp. 156–165. LNCS, Springer (1991)
30. Valmari, A.: Stubborn Sets for Reduced State Space Generation. In: ICATPN/APN'89. LNCS, vol. 483, pp. 491–515. Springer (1991)
31. Valmari, A.: The State Explosion Problem. In: Petri Nets'96. LNCS, vol. 1491, pp. 429–528. Springer (1998)
32. Valmari, A.: Stubborn Set Methods for Process Algebras. In: POMIV'96. pp. 213–231. AMS Press, Inc. (1997)