Combining Parallel Emptiness Checks with Partial Order Reductions

Denis Poitrenaud[1,2] and Etienne Renault[3]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
[2] Université Paris Descartes, Paris, France
[3] LRDE, EPITA, Kremlin-Bicêtre, France

Abstract. In explicit state model checking of concurrent systems, multi-core emptiness checks and partial order reductions (POR) are two major techniques to handle large state spaces. The first one tries to take advantage of multi-core architectures while the second one may decrease exponentially the size of the state space to explore.
For checking LTL properties, Bloemen and van de Pol [2] shown that the best performance is currently obtained using their multi-core SCC-based emptiness check. However, combining the latest SCC-based algorithm with POR is not trivial since a condition on cycles, the *proviso*, must be enforced on an algorithm which processes collaboratively cycles. In this paper, we suggest a pessimistic approach to tackle this problem for liveness properties. For safety ones, we propose an algorithm which takes benefit from the information computed by the SCC-based algorithm.
We also present new parallel provisos for both safety and liveness properties that relies on other multi-core emptiness checks. We observe that all presented algorithms maintain good reductions and scalability.

1 Introduction and Related Work

The automata-theoretic approach to explicit Linear-time Temporal Logic (LTL) model checking explores finite Labeled Transition Systems (LTS). Unfortunately, LTS are often too large to be fully explored in reasonable time and applying sequential algorithms becomes impractical. To tackle this well-known state explosion problem, various techniques have been suggested. In this paper we focus on the combination of two of them: Partial Order Reduction (POR) and multi-core emptiness checks.

POR exploits the interleaving semantics of concurrent systems by only considering representative executions [27, 22, 15] rather than all possible permutations of the execution of n independent actions (i.e. $n!$ possible interleavings). The selection of the representative executions is performed on-the-fly while exploring the LTS: for each state, the exploration algorithm only considers a nonempty (reduced) subset of all enabled actions, such that all omitted actions are "independent" from those in the reduced set. The execution of omitted actions is then postponed to a future state. The reduced LTS is sufficient to check reachability problems (e.g. existence of a global deadlock). However, for LTL model checking,[1] only stuttering-invariant formula (e.g. not using the Next

[1] See Peled [22] for a survey of POR reductions with LTL.

operator) can be verified. In addition to this restriction on formulas that can be checked, a complementary condition, called a *proviso*, must be enforced. If the same actions are consistently ignored along a cycle, the reduction may miss some undesirable behavior. When checking liveness properties, a sufficient condition is to force every cycle of the reduced LTS to contain at least one fully expanded state i.e. a state for which all actions are considered. When checking safety properties, forcing every (non-deadlock) state to have at least one fully expanded successor (direct or indirect) is sufficient.

An emptiness check for LTL model checking is an algorithm looking for a counterexample in the state space of the system. A counterexample is simply a lasso-shaped execution, i.e. a particular cycle reachable from the initial state.

The best multi-core emptiness checks are based on a Depth-First Search (DFS) exploration [2] and can be classified into two categories: those based on a Nested Depth First Search (NDFS) [10], and those based on an enumeration of Strongly Connected Components (SCC) [24, 2]. While the algorithm of Renault et al. [24] performs a *state-based DFS exploration*, one can note that the one suggested by Bloemen and van de Pol [2] performs a DFS over SCCs rather than states which makes the detection of individual cycles more difficult. All these concurrent algorithms are based on the swarming technique [13]: multiple threads, with their own exploration order, are spawned from the initial state; each thread shares information to prune the exploration of the other threads. Bloemen and van de Pol [2] have shown recently that their SCC-based algorithm provide actually the best results. This algorithm uses a (lock-free) concurrent union-find data structure that centralizes all the shared information. This structure is adapted with a work stealing mechanism.

In a sequential setting, provisos for emptiness checks have been well studied these last years [18, 9, 7, 11, 28]. The *in-stack* proviso introduced by Peled [22] and implemented in Spin has been improved by Evangelista and Pajault [9] with several mechanisms to reduce the number of expansions during a DFS exploration. Some of these mechanisms have then been deconstructed by Duret-Lutz et al. [7] to build new provisos (for liveness properties) that outperform the previous ones. These authors also proposed original provisos that can exploit the SCC information when the underlying emptiness check computes it. Other provisos have also been suggested (but not evaluated) in the more complex context of process algebra to consider τ-transitions [11, 28]. Some works also focus on non-DFS based emptiness checks [4, 5] thus defining new ways to detect potential ignoring cycles based on quadratic algorithms.

In a multi-core setting, POR has been less studied. Barnat et al. [1] suggested an approach based on a topological sort which sounds hard to combine efficiently with state-of-the-art parallel emptiness checks (see Laarman and Faragó [14]). Lerda and Sisto [17] proposed an adaptation of the *in-stack* proviso without knowing the entire DFS stack. More recently, Laarman and Wijs [16] worked on the adaption of the *in-stack* proviso with the best multi-core NDFS-based emptiness check [10] and achieved good reductions and good scalability.

Even if it has been shown that the best current performance is obtained using multi-core SCC-based emptiness checks, these algorithms have not yet be combined with POR due to several problems. For liveness properties, multi-core SCC-based emptiness checks compute SCCs rather than particular cycles while

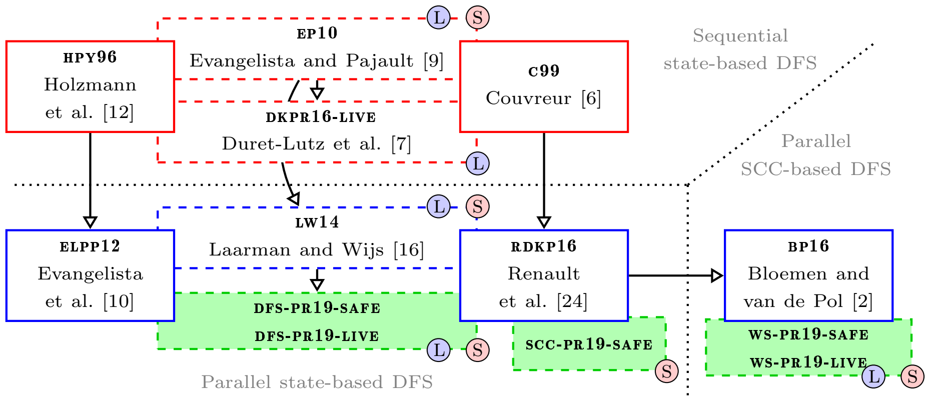

Fig. 1. Contributions of this paper are detailed in green. Red plain boxes correspond to sequential emptiness checks and blue plain boxes represent parallel emptiness checks. Dashed boxes are provisos. A proviso box is covered by a emptiness check box if the two are compatible. An edge links one box to another if the second one reuses ideas from the first one. Bullets tagged L represent liveness provisos and S the safety ones.

the proviso relies on detecting cycles. For safety properties, the expansion of a single state in each SCC without successor is enough but has never been realized in a multi-threaded context.

Figure 1 summarizes the contributions of this paper (in green). First, we aim at experimentally demonstrating that the improvements suggested in DKPR16 in a sequential setting can be shifted to multi-core one. DFS-PR19-LIVE and DFS-PR19-SAFE correspond to this adaptation. We can notice that both are compatible with the emptiness checks ELPP12 and RDKP16. After recalling necessary definitions in Section 2, we introduce in Section 3 these two new provisos and suggest a new one for safety properties SCC-PR19-SAFE. This last proviso exploits the underlying SCC computation of RDKP16. Section 4 introduces two last provisos WS-PR19-LIVE and WS-PR19-SAFE. These algorithms are the first provisos compatible with the BP16 emptiness check. Section 5 evaluates the performances of our provisos and shows that all of them achieve a reduction comparable to the state-of-the-art while maintaining a good scalability.

2 Preliminaries

A Labeled Transition System (LTS) is a tuple $L = \langle V, v_0, Act, \delta\rangle$ where V is a finite set of states, $v_0 \in V$ is a designated initial state, Act is a set of actions and $\delta \subseteq V \times Act \times V$ is a (deterministic) transition relation where each transition is labeled by an action. If $(s, \alpha, d) \in \delta$, we say that d is a *successor* of s. We denote by $post(v)$ the set of all successors of $v \in V$.

A *path* between two states $v, v' \in V$ is a finite and non-empty sequence of adjacent transitions $\rho = (v_1, \alpha_1, v_2)(v_2, \alpha_2, v_3)\dots(v_n, \alpha_n, v_{n+1}) \in \delta^+$ with $v_1 = v$ and $v_{n+1} = v'$. When $v = v'$ the path is a *cycle*. Moreover, when all the states $v_1, \dots v_n$ are distinct states, then the cycle is said *elementary*.

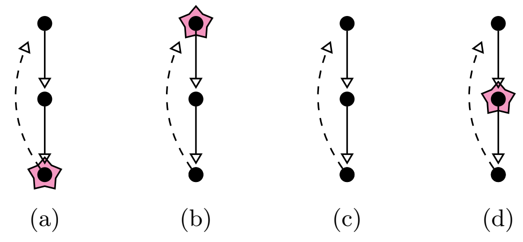

Fig. 2. Black nodes and plain edges represent the DFS stack, dashed edges represent (not yet visited) back edges, and starred states corresponds to already expanded states. In (a) and (b) CONDitional provisos do not require an expansion. In (c) and (d), the SOURCE or the DEST of the back edge should be preventively expanded. The liveness proviso of Evangelista and Pajault [9] will avoid the expansion in (d) since it is useless.

A non-empty set $C \subseteq V$ is a Strongly Connected Component (SCC) iff any two different states v, $v' \in C$ are connected by a path, and C is maximal w.r.t. inclusion. If C is not maximal we call it a *partial* SCC.

For the purpose of partial-order reductions, an LTS is equipped with a function $\mathit{reduced} : V \rightarrow 2^V$ that returns a subset of successors reachable via a reduced set of actions. For any state $v \in V$, we have $\mathit{reduced}(v) \subseteq \mathit{post}(v)$ and $\mathit{reduced}(v) = \emptyset \implies \mathit{post}(v) = \emptyset$. The *reduced* function must satisfy other conditions depending on whether we use *ample set*, *stubborn set* or *persistent set* [see 15, for a survey]. The algorithms we present do not depend on the actual technique used to compute *reduced*.

The function *reduced* preserves only two properties on the corresponding reduced LTS: the presence of deadlocks, i.e. states without successors, and the presence of an infinite sequence, i.e. a cycle. When checking more complex properties, i.e. LTL formulae (safety or liveness), some additional conditions must be enforced. The *reduced* function must be restricted to reflect the variations of the Boolean values of the atomic propositions (appearing in the property). These extra conditions can easily be integrated in the computation of *reduced*.

However, the previous conditions do not prevent from continuously ignoring the same actions (in a cycle of the reduced LTS). This is the so-called *ignoring problem*. This problem can be solved using different *provisos* depending on the nature of the property, i.e. safety or liveness. These provisos rely on the presence of a *(fully) expanded states* in some cycles. A (fully) expanded state v is simply a state for which all the successors $\mathit{post}(v)$ are considered in the reduced LTS even if $\mathit{reduced}(v)$ is strictly included in $\mathit{post}(v)$.

3 Provisos for emptiness checks applying a state-based DFS

Liveness properties. When checking liveness properties with POR and to ensure that no action will be ignored for ever, emptiness checks must ensure that *every cycle contains at least one expanded state (i.e. a state for which all actions are considered).* Notice that this property is an over-approximation but

Algorithm 1: State-based DFS equipped (highlighted in yellow) for checking liveness properties with POR

```
1  $\forall v \in V : v.status \leftarrow$ UNKNOWN
2  VISITED $\leftarrow \emptyset$
3  $\forall p \in [1 \ldots n]$ : DFS$_p \leftarrow \emptyset$
4  DFS-PR19-LIVE$_1(v_0) || \ldots ||$ DFS-PR19-LIVE$_n(v_0)$
5  Procedure DFS-PR19-LIVE$_p(v \in V)$
6      DFS$_p \leftarrow$ DFS$_p \cup \{v\}$
7      NEXT $\leftarrow reduced(v)$
8      if NEXT $= post(v)$ then cas($v.status$, UNKNOWN, EXPANDED)
9      for $v' \in$ mix$_p$(NEXT) do
10         if $v' \notin$ DFS$_p \cup$ VISITED then DFS-PR19-LIVE$_p(v')$
11         else if $v' \in$ DFS$_p \wedge v.status =$ UNKNOWN $\wedge v'.status =$ UNKNOWN then
12             cas($v'.status$, UNKNOWN, EXPANDED)
13     cas($v.status$, UNKNOWN, NOT_EXPANDED)
14     if $v.status =$ EXPANDED then
15         NEXT $\leftarrow post(v) \setminus reduced(v)$
16         for $v' \in$ mix$_p$(NEXT) do
17             if $v' \notin$ DFS$_p \cup$ VISITED then DFS-PR19-LIVE$_p(v')$
18     VISITED $\leftarrow$ VISITED $\cup \{v\}$
19     DFS$_p \leftarrow$ DFS$_p \setminus \{v\}$
```

ensures that the ignoring problem is tackled correctly. Thus it could lead to useless expansions while all actions have been seen during a particular cycle but not containing any fully expanded state.

Before diving into a multi-core setting, let us recall how this proviso property can be enforced for sequential DFS algorithms. Duret-Lutz et al. [7] suggested simple sequential provisos that are competitive with the state-of-the-art. During the DFS exploration of state v, the algorithm detects *back edges*, i.e. transitions $(v, \_, v')$ where v' is already in the DFS stack. When detecting such transitions, a cycle has been detected. Then, the algorithm (1) checks if v or v' is already expanded and if not (2) chooses to expand the source v, (exclusive) or the destination v'. In both cases, the expansion of v or v' ensures that the cycle closed by $(v, \_, v')$ contains at least one expanded state. Since all back edges discovered by the DFS cover all the elementary cycles, the property is respected. Figure 2 (a and b) describes cases where no expansion is required, while (c) describes a situation where an expansion is required (source or destination) and (d) a situation where a useless expansion is performed.

The aforementioned algorithms can be combined with a parallel swarmed exploration. Algorithm 1 (without highlighted lines) presents a swarmed exploration where all threads perform a state-based DFS exploration of the reduced state space.[2]

The n threads share a VISITED set (declared line 2) and each thread p maintains its own DFS set DFS$_p$ (line 3). The threads are spawned line 4. When a new state v is visited by a thread p, it is first added in the local set DFS$_p$ (line 6) and then a reduced set of successors is computed (line 7). These selected successors are explored in a randomized order (line 9). Each time a new state is discovered line 10, a recursive call is realized. After all the successors (not in DFS$_p$) has been

[2] All instructions (excepted recursive calls) are considered to be atomic.

inserted in VISITED, v can be itself added into VISITED and removed from DFS_p (lines 18 and 19).

Highlighted lines implement a new parallel proviso for liveness properties. It is based on the combination of two ideas: (1) the conditional destination expansion as suggested by Duret-Lutz et al. [7] since it achieves good results in sequential settings and, (2) the sharing of the state status (UNKNOWN, EXPANDED, NOT_EXPANDED) as presented by Laarman and Wijs [16].

Initially, all states are tagged UNKNOWN (line 1). When an UNKNOWN state yielding no reduction is encountered line 8, its status is fixed to EXPANDED by a compare-and-swap instruction. When a back-edge is detected between two UNKNOWN states (line 11), the destination is selected for expansion (line 12). Such an expansion is realized line 14 to 17 by considering the previously ignored successors. Before this expansion, the status of the state is checked. If this status is still UNKNOWN, no expansion is required for this state and its status can be fixed to NOT_EXPANDED (line 13).

Safety properties. When checking safety properties with POR and to ensure that no action will be ignored at all, emptiness checks must ensure that *at least one expanded state is reachable from any visited state*. Here again, this property is an over-approximation. Thus it could lead to useless expansions while all actions have been seen during a bottom SCC not containing an expanded state.

As for Algorithm 1, the highlighted lines in Algorithm 2 correspond to a proviso equipping a state-based DFS. This new proviso implements a conditional destination expansion mixed with a sharing of the state status. During the DFS exploration, the proviso of Laarman and Wijs [16] systematically expands states having all its successors on the DFS stack. Here, we expand one of its destinations and only if the other destination are not already expanded. This is the first time that a safety proviso based on the expansion of a destination is proposed.

When visiting a state, the algorithm decides to expand it if its direct successors (in the reduced set) are all in its local DFS stack. The local variable *allin* (declared line 9) tracks if this condition holds. Initially, *allin* is true and set to false when the algorithm detects a direct successor not belonging to the DFS stack (lines 12 & 15). Line 14 implements the conditional expansion: *allin* stays true if all the direct successors are on the DFS stack but have an unknown status.

When a status has been fixed (different from UNKNOWN) for a state v, either v is itself expanded or an expanded state is reachable from it. In both cases, it is not necessary to expand its predecessor. Then, lines 17 and 18 are executed only if an expansion is required: a destination is chosen randomly and marked as to be expanded (just before the DFS will backtrack this state – line 20).

In the previous algorithm, multiple expansions can occur for a given SCC (see Figure 3). The next algorithm avoids expansion in non bottom SCC and try to limit the expansions only to the entry point of each bottom SCC. In a sequential settings, this leads to have at most one expansion per bottom SCC. Recently, a state-based parallel swarmed DFS computing SCC has been proposed [24]. Here, we adapt this algorithm to implement the aforementioned idea while exploiting the status sharing as in the previous algorithm.

Algorithm 2: State-based DFS equipped (highlighted in yellow) for checking safety properties with POR

```
1  $\forall v \in V : v.status \leftarrow$ UNKNOWN
2  VISITED $\leftarrow \emptyset$
3  $\forall p \in [1 \ldots n]$ : DFS$_p \leftarrow \emptyset$
4  DFS-PR19-SAFE$_1(v_0) || \ldots ||$ DFS-PR19-SAFE$_n(v_0)$
5  Procedure DFS-PR19-SAFE$_p(v \in V)$
6      DFS$_p \leftarrow$ DFS$_p \cup \{v\}$
7      NEXT $\leftarrow reduced(v)$
8      if NEXT $= post(v)$ then cas($v.status$, UNKNOWN, EXPANDED)
9      $allin \leftarrow \top$
10     for $v' \in$ mix$_p$(NEXT) do
11         if $v' \notin$ DFS$_p \cup$ VISITED then
12             $allin \leftarrow \bot$
13             DFS-PR19-SAFE$_p(v')$
14         else if $v' \notin$ DFS$_p \lor v.status \neq$ UNKNOWN $\lor v'.status \neq$ UNKNOWN then
15             $allin \leftarrow \bot$
16     if $allin$ then
17         $v' \leftarrow$ randomlyPick($reduced(s)$)
18         cas($v'.status$, UNKNOWN, EXPANDED)
19     cas($v.status$, UNKNOWN, NOT_EXPANDED)
20     if $v.status =$ EXPANDED then
21         NEXT $\leftarrow post(v) \setminus reduced(v)$
22         for $v' \in$ mix$_p$(NEXT) do
23             if $v' \notin$ DFS$_p \cup$ VISITED then DFS-PR19-SAFE$_p(v')$
24     VISITED $\leftarrow$ VISITED $\cup \{v\}$
25     DFS$_p \leftarrow$ DFS$_p \setminus \{v\}$
```

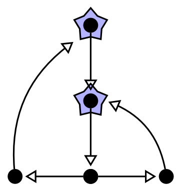

Fig. 3. Example where a useless expansion occurs in bottom-SCC for Algorithm 2. The two starred nodes will be expanded.

The unhighlighted lines of Algorithm 3 correspond to the one of Renault et al. [24]. The shared variable S maps to each state v, the set of states $S(v)$ belonging to the same (partial) SCC. The shared set DEAD contains all states belonging to fully visited SCCs. Initially, for any state v, the set $S(v) = \{v\}$. Each thread p maintains two local variables, a stack ROOTS$_p$ which contains the entry point of each traversed (partial) SCC and a set VISITED$_p$ holding each state visited by thread p. A local unique number $v.num_p$ (called the live number in the SCC computation proposed by Tarjan [26]) is associated to each state v (line 8). Each newly discovered state is considered as the root of an SCC and then inserted in the stack ROOTS$_p$ line 9. This stack as well as the mapping S are updated each time a closing edge, i.e. a transition $(v, \_, v')$ such that v' belongs t a partial SCC containing a state of the DFS stack, is detected (lines 21 to 23). The local unique number of states help to determine the effective root of the partial SCC: the stack is popped until this entry point becomes the top of the stack. The mapping S is updated to aggregate all the sets associated to popped states (line 23). Notice that the instruction line 23 must be atomic. When discovering the effective root v of a (complete) SCC (line 28), all states belonging (i.e. $S(v)$) to it are marked as DEAD atomically (line 35), thus restricting the visit by the other threads (line 15). The mapping S (and the set

Algorithm 3: State-based DFS equipped (highlighted in yellow) for checking safety properties with POR - unhighlighted lines correspond to the SCC computation algorithm as presented in Renault et al. [24]

```
1  $\forall v \in V : S(v) \leftarrow \{v\}$
2  DEAD $\leftarrow$ EXPANDED $\leftarrow \emptyset$
3  $\forall p \in [1 \ldots n]$ : VISITED$_p \leftarrow \emptyset$
4  $\forall p \in [1 \ldots n]$ : ROOTS$_p \leftarrow \emptyset$
5  SCC-PR19-SAFE$_1(v_0)$|| ... || SCC-PR19-SAFE$_n(v_0)$
6  Function SCC-PR19-SAFE($v \in V$) : Boolean
7      VISITED$_p \leftarrow$ VISITED$_p \cup \{v\}$
8      $v.num_p \leftarrow$ |VISITED$_p$|
9      ROOTS$_p$.push($v$)
10     NEXT $\leftarrow reduced(v)$
11     $isTerm \leftarrow reduced(v) \neq post(v)$            // $S(v)$ is a TSCC without exp. st.
12     if $\neg isTerm$ then
13         EXPANDED $\leftarrow$ EXPANDED $\cup \{v\}$
14     for $v' \in$ mix$_p$(NEXT) do
15         if $v' \in$ DEAD then
16             $isTerm \leftarrow \bot$
17         else if $v' \notin$ VISITED$_p$ then
18             $t \leftarrow$ SCC-PR19-SAFE($v'$)
19             $isTerm \leftarrow isTerm \wedge t$
20         else
21             while $v'.num_p <$ ROOTS$_p$.top().$num_p$ do
22                 $r \leftarrow$ ROOTS$_p$.pop()
23                 $S(r) \leftarrow S(v') \leftarrow S(r) \cup S(v')$
24     if $isTerm \wedge v \in$ EXPANDED then
25         $isTerm \leftarrow \bot$
26         NEXT $\leftarrow post(v) \setminus reduced(v)$
27         goto 14
28     if $v =$ ROOTS$_p$.top() then
29         if $isTerm$ then
30             EXPANDED $\leftarrow$ EXPANDED $\cup \{v\}$
31             $isTerm \leftarrow \bot$
32             NEXT $\leftarrow post(v) \setminus reduced(v)$
33             goto 14
34         ROOTS$_p$.pop()
35         DEAD $\leftarrow$ DEAD $\cup S(v)$
36     return $isTerm$
```

DEAD) can be efficiently implemented using a lock-free version of an union-find data structure.

To limit the number of expansions, the algorithm expands only the root of each bottom SCC not already containing an expanded state. The local variable *isTerm* tracks such an SCC (line 11, 16 and 19). When popping the root of a bottom SCC (line 28) for which no expanded state has been already discovered, an expansion is realized (lines 30–33). However, a same bottom SCC may have different entry points for different threads. To limit the number of expansions, the algorithm detects some of these situations line 24.

Notice that the two first provisos presented in this section are compatible with most of the optimizations presented by Evangelista and Pajault [9] as well as the one suggested by Duret-Lutz et al. [7]. Moreover, these provisos can be integrated in any emptiness checks based on a state-based DFS swarm exploration for instance CNDFS [10] or Renault et al. [24] algorithm. The latest presented proviso is only compatible with Renault et al. [24].

4 Provisos for SCC-based DFS emptiness checks

Provisos presented in the previous section are not compatible with the best currently known parallel emptiness check [2]. Until now, there is no proviso, neither for safety nor for liveness properties, for this model-checking algorithm. This algorithm differs from the previous ones since it does not perform a DFS in terms of states but only in terms of SCCs: in particular, the states of a same SCC may be visited (and processed) in any order. One can note that SCCs are still marked DEAD in a DFS post-order (see Algorithm 4 without highlighted lines). We denote this kind of algorithms as *SCC-based DFS emptiness-checks.*

The algorithm of Bloemen and van de Pol [2] has been introduced to tackle the main drawback of the algorithm suggested by Renault et al. [24]. Indeed, in this last algorithm, each SCC must be completely processed by the same thread before it can be marked DEAD. To improve this, Bloemen and van de Pol [2] introduced a work-stealing mechanism to allow SCCs to be cooperatively treated (see the outer loop line 9 of Algorithm 4). Notice that this mechanism induces a more complex shared union-find data-structure (see [2]).

In this approach, all threads start a DFS until they reach a (partial) SCC which is currently processed by one (or more) other thread(s). The states belonging to this SCC (aggregated by the DFS) are then distributed among the various threads (line 10) and marked DONE (line 27) when all their successors are DEAD or detected to belong to the current SCC. When the last state of an SCC is marked DONE, the SCC itself is marked DEAD (line 29).

Liveness properties. Implementing a proviso for liveness properties in this algorithm is complex since the work stealing mechanism removes all possible knowledge about cycles in this SCC. When checking liveness properties, the proviso must ensure that each elementary cycle contains (at least) one expanded state. In sequential and for such a cycle, the states can be marked DONE by the algorithm in any order. Our proviso consists to expand any state with at least one successor marked DONE. In a sequential setting, this approach ensures that all cycles or size $n > 1$ contains at least one expanded state ($n - 1$ for the worst case, $n/2$ in average).

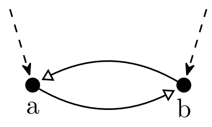

Fig. 4. Two threads cooperatively discovering a cycle. Dashed edges represent paths from the initial state while plain edges represent currently processed edges.

However, in a parallel setting, this approach is not sufficient. Let us consider the example of Figure 4 with two threads, one visiting state a, the second state b, and with a and b known to belong to the same SCC. Thread t_1 selects a line 9, while thread t_2 selects b and both a and b are not already DONE. The test line 15 prevents from a recursive call (for both t_1 and t_2). Since a and b will only be marked DONE line 27, t_1 as well as t_2 will not detect that an expansion is required. Indeed, the only successor of a (resp. b) is not DONE.

To solve this problem, we introduce the shared sets WIP_p that represent states currently processed by a thread p. Highlighted lines of Algorithm 4 detail this new proviso for liveness properties. A state is inserted into WIP_p when first discovered by a thread p (line 10) and removed either line 28 when the state

Algorithm 4: SCC-based DFS equipped (highlighted in yellow) for checking liveness properties with POR - unhighlighted lines correspond to the SCC computation algorithm as presented in Bloemen and van de Pol [2]

```
1  $\forall v \in V : S(v) \leftarrow \{v\}$
2  $\textsc{dead} \leftarrow \textsc{done} \leftarrow \textsc{expanded} \leftarrow \emptyset$
3  $\forall p \in [1 \dots n] : R_p \leftarrow \textsc{wip}_p \leftarrow \emptyset$
4  $\textsc{ws-pr19-live}_1(v_0) || \dots || \textsc{ws-pr19-live}_n(v_0)$
5  Procedure $\textsc{ws-pr19-live}_p(v \in V)$
6      $R_p.\text{push}(v)$
7      if $v \notin \textsc{expanded} \wedge post(v) = reduced(v)$ then
8          $\textsc{expanded} \leftarrow \textsc{expanded} \cup \{v\}$
9      while $v' \in (S(v) \setminus \textsc{done})$ do
10         $\textsc{wip}_p \leftarrow \textsc{wip}_p \cup \{v'\}$
11         $\textsc{next} \leftarrow v' \in \textsc{expanded}\ ?\ post(v') : reduced(v')$
12         while $\textsc{next} \neq \emptyset$ do
13             $w \leftarrow \text{randomlyPick}(\textsc{next})$
14             if $w \in \textsc{dead}$ then continue
15             else if $\nexists w' \in R_p : w \in S(w')$ then
16                 $\textsc{wip}_p \leftarrow \textsc{wip}_p \setminus \{v'\}$
17                 $\textsc{ws-pr19-live}_p(w)$
18                 goto 9
19             else
20                 if $v' \notin \textsc{expanded} \wedge w \in (\textsc{done} \cup (\bigcup_{i \in [1 \dots n]} \textsc{wip}_i)) \setminus \textsc{expanded}$ then
21                     $\textsc{expanded} \leftarrow \textsc{expanded} \cup \{w\}$
22                     $\textsc{next} \leftarrow \textsc{next} \cup (post(v') \setminus reduced(v'))$
23                 while $w \notin S(v)$ do
24                     $r \leftarrow R_p.\text{pop}()$
25                     $t \leftarrow R_p.\text{top}()$
26                     $S(r) \leftarrow S(t) \leftarrow S(r) \cup S(t)$
27         $\textsc{done} \leftarrow \textsc{done} \cup \{v'\}$
28         $\textsc{wip}_p \leftarrow \textsc{wip}_p \setminus \{v'\}$
29     if $S(v) \nsubseteq \textsc{dead}$ then $\textsc{dead} \leftarrow \textsc{dead} \cup S(v)$
30     if $v = R_p.\text{top}()$ then $R_p.\text{pop}()$
```

has been marked DONE or line 16 before performing a recursive call. Doing a recursive call ensures that state v' will not be marked by the current while loop line 12. Indeed, when backtracking from the recursive call, line 18 the thread will realize a jump to the outer loop. This jump implies that all the successors of a state must be seen without performing a recursive call before this state is marked DONE. This is the main difference between our algorithm and the one of Bloemen and van de Pol [2]. Line 20 checks when an expansion is required. A state is expanded if one of its successors is DONE or belong to a $\textsc{wip}_p$ sets and neither the source nor the destination is expanded.

Notice that the introduction of the $\textsc{wip}_p$ sets also solves the expansion of the elementary cycles of size 1. As for the previous algorithms, the EXPANDED set allows to share expansions between threads.

Safety properties. As for liveness properties, a proviso for safety properties has never been proposed for the algorithm of Bloemen and van de Pol [2]. The goal, like in Algorithm 3, is to limit expansions only to bottom SCCs and to minimize the number of expansions in such SCCs. Detecting that the SCC is a bottom one can be done as easily as for Algorithm 3. A Boolean *isTerm* is associated to each SCC and updated consequently (lines 1, 8, 15, 18 and 29 of algorithm 5). Notice that this Boolean is associated to an SCC rather than a state in order to propagate the information inside of the work stealing mechanism.

Algorithm 5: SCC-based DFS equipped (highlighted in yellow) for checking safety properties with POR - unhighlighted lines correspond to the SCC computation algorithm as presented in Bloemen et al. [3]

```
1  $\forall v \in V : S(v) \leftarrow \{v\}, S(v).isTerm \leftarrow \top$
2  $\textsc{dead} \leftarrow \textsc{done} \leftarrow \textsc{expanded} \leftarrow \emptyset$
3  $\forall p \in [1 \ldots n] : R_p \leftarrow \emptyset$
4  $\textsc{ws-pr19-safe}_1(v_0) || \ldots || \textsc{ws-pr19-safe}_n(v_0)$
5  **Procedure** $\textsc{ws-pr19-safe}_p(v \in V)$
6      **if** $v \notin \textsc{expanded} \wedge post(v) = reduced(v)$ **then**
7          $\textsc{expanded} \leftarrow \textsc{expanded} \cup \{v\}$
8          $S(v).isTerm \leftarrow \perp$
9      $R_p.\text{push}(v)$
10     **while** pick $v'$ from $(S(v) \setminus \textsc{done})$ **do**
11         $isExpanded \leftarrow v' \in \textsc{expanded}$
12         $\textsc{next} \leftarrow isExpanded\ ?\ post(v') : reduced(v')$
13         **foreach** $w \in \text{mix}_p(\textsc{next})$ **do**
14             **if** $w \in \textsc{dead}$ **then**
15                 $S(v').isTerm \leftarrow \perp$
16             **else if** $\nexists w' \in R_p : w \in S(w')$ **then**
17                 $\textsc{ws-pr19-safe}_p(w)$
18                 **if** $w \notin S(v')$ **then** $S(v').isTerm \leftarrow \perp$
19             **else**
20                 **while** $w \notin S(v)$ **do**
21                     $r \leftarrow R_p.\text{pop}()$
22                     $t \leftarrow R_p.\text{top}()$
23                     $S(r) \leftarrow S(t) \leftarrow S(r) \cup S(t)$
24         **while** $v' \notin \textsc{done}$ **do**                 // Ensure good removing or expansion
25             $nb \leftarrow |S(v') \cap \textsc{done}|$
26             // Expand the last element
27             **if** $S(v') \setminus \textsc{done} = \{v'\} \wedge S(v').isTerm$ **then**
28                 $\textsc{expanded} \leftarrow \textsc{expanded} \cup \{v'\}$
29                 $S(v').isTerm \leftarrow \perp$
30                 **break**
31             **else**
32                 // $v'$ is about to be DONE while another thread
33                 // requires an expansion
34                 **if** $\neg isExpanded \wedge v' \in \textsc{expanded}$ **then**
35                     **break**
36                 // Otherwise mark states DONE but only one by one
37                 **else**
38                     **if** $nb = |S(v') \cap \textsc{done}|$ **then** $\textsc{done} \leftarrow \textsc{done} \cup \{v'\}$
39         **if** $S(v) \nsubseteq \textsc{dead}$ **then** $\textsc{dead} \leftarrow \textsc{dead} \cup S(v)$
40         **if** $v = R_p.\text{top}()$ **then** $R_p.\text{pop}()$
```

To implement the proviso, we can exploit a property of the original algorithm: when the last state of an SCC is marked DONE, the SCC is then marked DEAD. Capturing this instant could be useful to trigger an expansion in each bottom SCC. This approach, even if satisfying, does not work in the algorithm. Indeed, the algorithm is not aware that the state is the last one to be marked DONE.

To solve this problem, we propose a pessimistic approach (as previously). When a state is about to be marked DONE (lines 24 to 38), three situations may occur. First of all (line 27), the current state v' is the last one of the bottom (partial) SCC: in this case, the state is expanded line 28 and the SCC does not require any more expansion, i.e. $isTerm$ is set to false. Second of all (line 34-35), the state is about to be marked DONE while another thread required the expansion of this state. In this case, new successors must be explored for this state, and the break line 35 will force this exploration. Finally, two (or more) states may be concurrently candidate for being DONE. Line 25 and 38 prevent concurrent multiple insertions in the set DONE (and thus potentially missing a required expansion). Each thread captures line 25 the current number of DONE states in the (partial) SCC while line 38 checks that this number has not changed in between. This leads to sequentialize the insertions in DONE. Notice that line 38 must be performed atomically even if it contains a conditional statement.

5 Evaluation

Benchmark Description. To evaluate the performance of the new provisos, we selected 21 models from the BEEM benchmark [20] that cover all types of models described by the classification of Pelánek [21]. All the models were selected such that Algorithm 1 with one thread and without applying POR would take at most 20 minutes on Intel(R) Xeon(R) @ 2.00GHz with 250GB of RAM. We fix the maximum running time to 40 minutes.[3] Here we compute only a reduced LTS explored by the algorithms presented in the previous sections. When applied in the context of a model checker, the visited reduced LTS will be larger due to the observation of visible transitions [23]. Experiments were run three times and only the median of the three values were kept.

According to Bloemen et al. [3], the performances of parallel emptiness checks may rely on the underlying graph structure. To evaluate this, the 21 models selected are divided into two categories: $\mathcal{M}_1$ (models with short cycles and many small SCCs) and $\mathcal{M}_2$ (models with long cycles and a small number of large SCCs). Bloemen et al. [3] observe that the performances for the algorithm suggested by Renault et al. [24] are degraded for $\mathcal{M}_2$ which is the motivation of the introduction of their new algorithm.

In this benchmark, we compared all the new algorithms presented in this paper with the only parallel provisos of the state-of-the-art, i.e LW14-LIVE (see Algo. 1 in Laarman and Wijs [16]),[4] LW14-SAFE (see Algo. 2 in Laarman and Wijs [16]). All the presented results have been computed using the same canvas and are then comparable. See Figure 1 for an overview of our contributions and the compatibility with existing emptiness checks.

Implementation Details. Since all the presented algorithms rely on hashtables and linked lists, they can be implemented lock-free. The *reduced* function implements the stubborn set method from Valmari [27] as described by Pater [19] but in a deterministic way, i.e. for any state s, *reduced*(s) always returns the same set. However, because the computation of a reduced set[5] of enabled transitions can be costly, we opted for its memoization using mutexes. This is

[3] For a description of our setup, including selected models, detailed results and code, see `http://www.lrde.epita.fr/~renault/benchs/ICFEM-2019/results.html`

[4] Notice that we only consider the blue DFS (without lines 33–37). When implementing an emptiness check the ignored lines could trigger complementary expansions. Thus the reported values here can be interpreted as the optimal bound (time, reduction, ...) for this algorithm.

[5] Our implementation uses *persistent sets* since a special attention must be paid when combining *ample sets* with on-the-fly exploration [25].

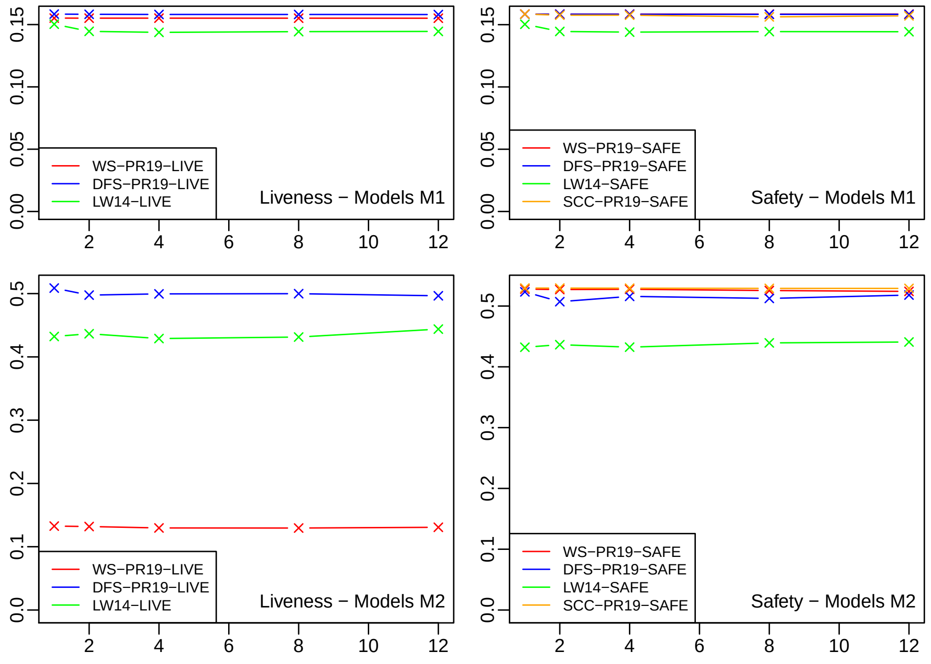

Fig. 5. Mean Reduction rates (in percent) on the whole benchmark, depicted for liveness and safety and for the two categories $\mathcal{M}_1$ and $\mathcal{M}_2$. The x-axis represents the number of threads when the y-axis the mean of the reduction rates.

an implementation choice but a pure lock-free version remains possible. All the approaches proposed here have been implemented in Spot [8]. For a given model the corresponding Kripke structure is generated on-the-fly using DiVinE 2.4 patched by the LTSmin team.[6]

Reduction rates. Figure 5 gives the mean of the reduction rates for the benchmark. It appears that the reduction rate of each algorithm is insensitive to the thread number. For $\mathcal{M}_1$, all algorithms tend to have a similar reduction rate even if LAARMAN (LIVE and SAFE) is slightly worse than the others (both for liveness and safety cases). For liveness and models $\mathcal{M}_2$, it appears that WS-PR19-LIVE significantly degrades the reduction rate. This is due to the pessimistic approach imposed by the lack of information from the DFS stack. This effect is minored for the safety case because expansions are limited to the bottom SCCs and by our strategy that minimizes the number of expansions in such SCCs.

Time analysis. Tables 1 and 2 describe the measure for the two categories of model and all the algorithms running with 1 to 12 threads. For safety, LW14-SAFE and WS-PR19-SAFE have comparable running times. Since WS-PR19-SAFE has a smaller reduction rate than LW14-SAFE, the work-stealing mechanism implemented in WS-PR19-SAFE shows its efficiency. Nonetheless, SCC-PR19-SAFE

[6] See http://fmt.cs.utwente.nl/tools/ltsmin/#divine for more details.

		LW14-LIVE			DFS-PR19-LIVE			WS-PR19-LIVE		
		States	Time	Sp.	States	Time	Sp.	States	Time(s)	Sp.
$\mathcal{M}_1$	1 th.	412.3	3 755	–	410.1	3 732	–	411.5	4 651	–
	2 th.	413.9	1 983	1.89	410.1	1 960	1.90	411.5	2 505	1.86
	4 th.	415.2	1 136	3.30	410.1	1 124	3.32	411.5	1 429	3.25
	8 th.	414.6	805	4.66	410.1	773	4.83	411.5	1 021	4.55
	12 th.	414.6	691	5.43	410.1	678	5.50	411.5	829	5.60
$\mathcal{M}_2$	1 th.	202.0	1 372	–	181.5	1 218	–	256.3	2 761	–
	2 th.	199.3	718	1.91	182.2	632	1.93	256.2	1 392	1.98
	4 th.	197.3	391	3.50	182.2	343	3.55	256.0	721	3.83
	8 th.	195.1	246	5.56	182.3	222	5.49	255.9	466	5.93
	12 th.	193.9	186	7.34	182.2	165	7.36	255.7	359	7.69

Table 1. States in 10^6, times in seconds and speedup for liveness provisos

		LW14-SAFE			DFS-PR19-SAFE			SCC-PR19-SAFE			WS-PR19-SAFE		
		State	Time	Sp.	State	Time	Sp.	State	Time	Sp.	State	Time	Sp.
$\mathcal{M}_1$	1 th.	412.3	5 124	–	410.1	3 734	–	410.1	4 179	–	410.1	5 041	–
	2 th.	413.9	2 709	1.84	410.1	1 961	1.90	410.4	2 235	1.87	410.1	2 744	1.84
	4 th.	414.6	1 527	3.38	410.1	1 124	3.32	410.3	1 372	3.04	410.1	1 493	3.38
	8 th.	414.5	1 067	4.87	410.1	803	4.65	410.7	870	4.80	410.1	1 034	4.87
	12 th.	414.7	809	6.23	410.1	641	5.82	410.5	784	5.33	410.1	809	6.23
$\mathcal{M}_2$	1 th.	202.0	1 372	–	180.2	1 214	–	179.1	1 380	–	179.1	1 935	–
	2 th.	199.6	724	1.90	180.5	633	1.92	179.2	713	1.93	179.1	997	1.94
	4 th.	197.2	391	3.50	180.7	339	3.58	179.2	336	4.10	179.1	543	3.56
	8 th.	195.1	263	5.21	180.8	214	5.67	179.5	257	5.37	179.2	362	5.35
	12 th.	194.0	187	7.32	180.7	165	7.32	179.5	205	6.72	179.2	296	6.52

Table 2. States in 10^6, times in seconds and speedup for safety provisos

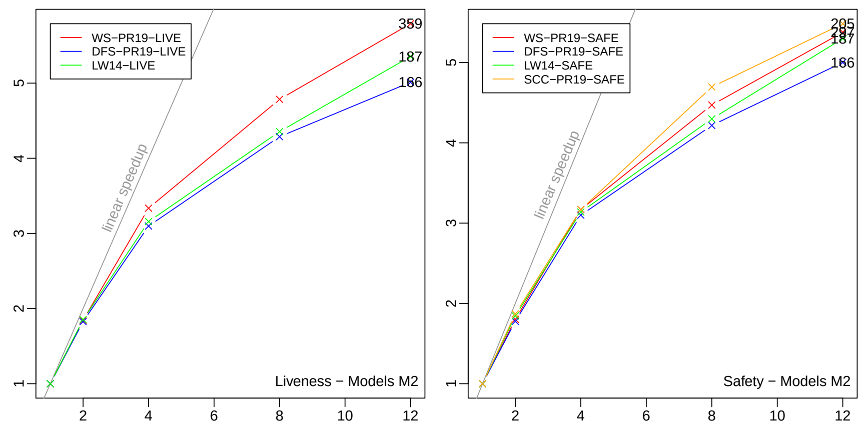

Fig. 6. Speedup on models $\mathcal{M}_2$

and DFS-PR19-SAFE perform better. However, the complex data structure of SCC-PR19-SAFE impacts negatively its running time. For liveness, LW14-LIVE performs better than WS-PR19-LIVE. DFS-PR19-LIVE performs better than the two others regardless the category of models.

Finally, we can notice that the work-stealing mechanism of WS-PR19-LIVE particularly improves the speedup for models $\mathcal{M}_2$. The Figure 6 displays the speedup curves for both liveness and safety and $\mathcal{M}_2$. For models $\mathcal{M}_1$, the speedup of all algorithms are comparable.

6 Conclusion

To our knowledge, only the work of Laarman and al. proposes provisos designed for parallel model checking. Nonetheless, in sequential settings and for liveness properties, Duret-Lutz et al. [7] empirically shown that variations on the traditional proviso could improve performances. In this paper, we demonstrate that the application of the suggested provisos (in particular DFS-PR19-LIVE and DFS-PR19-SAFE) can also benefit to the parallel emptiness check based on a state-based DFS. During this investigation, we also proposed a new proviso also based on destination expansion but dedicated to safety properties.

However, the best existing parallel emptiness checks are based on an SCC computation. For the best of them (BP16 [2]), no existing provisos can be directly applied since it is based on a work-stealing mechanism breaking the DFS post-order. In this paper, we proposed new provisos for this parallel emptiness check (for both liveness and safety properties). Moreover, we also presented a dedicated proviso for safety properties for Renault et al. [24]. Figure 1 summarizes the compatibility of the different provisos with respect to existing emptiness checks.

One of the challenging problems in parallelizing explicit state model checking is the model checking of stutter-free LTL properties on distributed systems. In this paper we propose, for the first time, several algorithms (that can be directly integrated into the bests known emptiness checks) to tackle this problem. All provisos presented and evaluated in this paper achieve comparable speedups. However, the reduction rate of WS-PR19-LIVE for models with long cycles and a small number of large SCCs is significantly degraded compared to the other proposed algorithms. An open question remains: can we develop a liveness proviso for BP16 that preserves a good reduction rate for any category of models?

References

1. J. Barnat, L. Brim, and P. Rockai. Parallel partial order reduction with topological sort proviso. In SEFM'10, pp. 222–231, Sept. 2010.
2. V. Bloemen and J. van de Pol. Multi-core scc-based ltl model checking. In HVC'16, Lecture Notes in Computer Science, pp. 18–33. Springer, Nov. 2016.
3. V. Bloemen, A. Laarman, and J. van de Pol. Multi-core on-the-fly scc decomposition. vol. 51, march 2016.
4. D. Bosnacki, S. Leue, and A. Lafuente. Partial-order reduction for general state exploring algorithms. In SPIN'06, vol. 3925 of LNCS, pp. 271–287. Springer.

5. D. Bošnaăźki, S. Leue, and A. Lluch Lafuente. Partial-order reduction for general state exploring algorithms. International Journal on Software Tools for Technology Transfer (STTT), 11(1):39–51, Feb. 2009.
6. J.-M. Couvreur. On-the-fly verification of temporal logic. In FM'99, vol. 1708 of LNCS, pp. 253–271, Sept. 1999. Springer.
7. A. Duret-Lutz, F. Kordon, D. Poitrenaud, and E. Renault. Heuristics for checking liveness properties with partial order reductions. In ATVA'16, vol. 9938 of LNCS, pp. 340–356. Springer, Oct. 2016.
8. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In ATVA'16, vol. 9938 of LNCS, pp. 122–129. Springer, Oct. 2016.
9. S. Evangelista and C. Pajault. Solving the ignoring problem for partial order reduction. International Journal on Software Tools for Technology Transfer, 12(2): 155–170, 2010.
10. S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved multi-core nested depth-first search. In ATVA'12, vol. 7561 of LNCS, pp. 269–283. Springer.
11. H. Hansen and A. Valmari. Safety property-driven stubborn sets. In Reachability Problems. Springer, 2016.
12. G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In SPIN'96, vol. 32 of DIMACS: Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, May 1996.
13. G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. IEEE Transaction on Software Engineering, 37(6):845–857, 2011.
14. A. Laarman and D. Faragó. Improved on-the-fly livelock detection. In NASA Formal Methods (NFM'13), pp. 32–47, 2013. Springer.
15. A. Laarman, E. Pater, J. Pol, and H. Hansen. Guard-based partial-order reduction. STTT, pp. 1–22, 2014.
16. A. W. Laarman and A. J. Wijs. Partial-order reduction for multi-core ltl model checking. In HVC 2014, vol. 8855 of LNCS, pp. 267–283. Springer, 2014.
17. F. Lerda and R. Sisto. Distributed-memory model checking with spin. In International SPIN Workshop on Model Checking of Software, pp. 22–39, 1999. Springer.
18. R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. FMSD, 20(1):231–247, Jan. 2002.
19. E. Pater. Partial order reduction for pins, March 2011.
20. R. Pelánek. BEEM: benchmarks for explicit model checkers. In SPIN'07, vol. 4595 of LNCS, pp. 263–267. Springer, 2007.
21. R. Pelánek. Properties of state spaces and their applications. International Journal on Software Tools for Technology Transfer (STTT), 10:443–454, 2008.
22. D. Peled. Combining partial order reductions with on-the-fly model-checking. In CAV'94, vol. 818 of LNCS, pp. 377–390. Springer, 1994.
23. D. Peled, A. Valmari, and I. Kokkarinen. Relaxed visibility enhances partial order reduction. Formal Methods in System Design, 19(3):275–289, 2001.
24. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on parallel explicit model checking for generalized Büchi automata. International Journal on Software Tools for Technology Transfer (STTT), pp. 1–21, Apr. 2016.
25. S. Stephen. What's wrong with on-the-fly partial order reduction. In CAV'19, Lecture Notes in Computer Science, pp. 478–495. Springer, 2019.
26. R. Tarjan. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.
27. A. Valmari. Stubborn sets for reduced state space generation. In ICATPN'91, vol. 618 of LNCS, pp. 491–515, 1991. Springer.
28. A. Valmari. More stubborn set methods for process algebras. In Concurrency, Security, and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday, pp. 246–271. Springer, 2017.