Improved On-The-Fly Livelock Detection: Combining Partial Order Reduction and Parallelism for $\text{DFS}_{\text{fifo}}$

Alfons Laarman$^1$ and David Faragó$^2$

$^1$ Formal Methods and Tools, University of Twente, The Netherlands
a.w.laarman@cs.utwente.nl
$^2$ Logic and Formal Methods, Karlsruhe Institute of Technology, Germany
farago@kit.edu

Abstract. Until recently, the preferred method of livelock detection was via LTL model checking, which imposes complex constraints on partial order reduction (POR), limiting its performance and parallelization. The introduction of the $\text{DFS}_{\text{fifo}}$ algorithm by Faragó et al. showed that livelocks can theoretically be detected faster, simpler, and with stronger POR. For the first time, we implement $\text{DFS}_{\text{fifo}}$ and compare it to the LTL approach by experiments on four established case studies. They show the improvements over the LTL approach: $\text{DFS}_{\text{fifo}}$ is up to 3.2 times faster, and it makes POR up to 5 times better than with SPIN’s $\text{NDFS}$. Additionally, we propose a parallel version of $\text{DFS}_{\text{fifo}}$, which demonstrates the efficient combination of parallelization and POR. We prove parallel $\text{DFS}_{\text{fifo}}$ correct and show why it provides stronger guarantees on parallel scalability and POR compared to LTL-based methods. Experimentally, we establish almost ideal linear parallel scalability and POR close to the POR for safety checks: easily an order of magnitude better than for LTL.

1 Introduction

Context. In the automata-theoretic approach to model checking [27], the behavior of a system-under-verification is modeled, along with a property that it is expected to adhere to, in some concise specification language. This model $\mathcal{M}$ is then unfolded to yield a state space automaton $A_M$ (cf. Def. 1). Safety properties, e.g. deadlocks and invariants, can be checked directly on the states in $A_M$ as they represent all configurations of $\mathcal{M}$. This check can be done during the unfolding, on-the-fly, saving resources when a property violation is detected early on.

For more complicated properties, like liveness properties [1], $A_M$ is interpreted as an $\omega$-automaton whose language $\mathcal{L}(A_M)$ represents all infinite executions of the system. A property $\varphi$, expressed in linear temporal logic (LTL), is likewise translated to a Büchi or $\omega$-automaton $A_{\varphi}$ representing all undesired infinite executions. The intersected language $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\varphi})$ now consists of all counterexample traces, and is empty if and only if the system is correct with respect to the property. The emptiness check is reduced to the graph problem of finding cycles with designated accepting states in the cross product $A_M \otimes A_{\neg\varphi}$ (cf. Sec. 2). The nested depth-first search (NDFS) algorithm [10] solves it in time linear to the size of the product and on-the-fly as well.
Motivation. The model checking approach is limited by the so-called state space explosion problem [1], which states that $\mathcal{A}_M$ is exponential in the components of the system, and $\mathcal{A}_\varphi$ exponential in the size of $\varphi$. Luckily, several remedies exist to this problem: patience, specialization and state space reduction techniques.

State space reduction via partial order reduction (POR) prunes $\mathcal{A}_M$ by avoiding irrelevant interleavings of local components in $M$ [16,26]: only a sufficient subset of successors, the ample set, is considered in each state (cf. Sec. 2). For safety properties, the ample set can be computed locally on each state. For liveliness properties, however, an additional condition, the cycle proviso, is needed to avoid the so-called ignoring problem [9]. POR can yield exponential reductions.

Patience also pays off exponentially as Moore’s law stipulates that the number of transistors available in CPUs and memory doubles every 18 months [22]. Due to this effect, model checking capabilities have increased from handling a few thousand states to covering billions of states recently (this paper and [5]). While this trend happily continues to increase memory sizes, it recently stopped benefitting the sequential performance of CPUs because physical limitations were reached. Instead, the available parallelism on the chips is rapidly increasing. So, for runtime to benefit from Moore’s law, we must parallelize our algorithms.

Specialization towards certain subclasses of liveness properties, finally, can also help to solve them more efficiently. For instance, a limitation to the CTL and the weak-LTL fragments was shown to be efficiently parallelizable [25,8]. In this paper, we limit the discourse to livelock properties, an important subclass (used in about half of the case studies of [7] and a third of [24]) that investigates starvation, occurring if an infinite run does not make progress infinitely often. The definition of progress is up to the system designer and could for instance refer to an increase of a counter or access to a shared resource. The SPIN model checker allows the user to specify progress statements inside the specification of the model [12], which are then represented in the model by the state label ‘progress’ and referenced by the predefined progress LTL property [15]. Until 1996, SPIN used a specific livelock verification algorithm. Section 6 of [15] states that it was replaced by LTL model checking due to its incompatibility with POR.

Problem. LTL model checking can likely not be parallelized efficiently. The current state-of-the-art reveals that parallel cycle detection algorithms either raise the worst-case complexity to $L^2$ [8] or to $L \cdot P$ [8], where $L$ is the size of the LTL cross product and $P$ the number of processors. Moreover, its additional constraints on POR severely limit its reduction capabilities, even if implemented with great care (see models allocation, cs and p2p in Table 1 in the appendix of [9]). Last but not least, these constraints also limit the parallelization of POR [2].

We want to investigate whether better results can be obtained for livelocks, for which recently an efficient algorithm was proposed by Faragó et al. [11]: DFS_fifo. In theory, it has additional advantages over the LTL approach:

1. It uses the progress labels in the model directly without the definition of an LTL property; avoiding the calculation of a larger cross product.

3 PROMELA database: [http://www.albertolluch.com/research/promelamodels](http://www.albertolluch.com/research/promelamodels)
2. It requires only one pass over the state space, while the NDFS algorithm, typically used for liveness properties, requires two.
3. It eliminates the need for the expensive cycle proviso with POR. Not only is the cycle proviso a highly limiting factor in state space reduction [9], it also complicates the parallelization of the problem [2].
4. It finds the shortest counterexample with respect to progress. But DFS\textsubscript{FIFO} is yet to be implemented and evaluated experimentally, so its practical performance is unknown. Additionally, a few hypotheses stand unproven:
   1. The algorithm’s strategy to delay progress as much as possible, may also be a good heuristic for finding livelocks early, making it more on-the-fly.
   2. Its POR performance might be close to that of safety checks, because the cycle proviso is no longer required [11], and the visibility proviso (see Table 1) is also positively influenced by the postponing of progress.
   3. The use of progress transitions instead of progress states is possible, semantically more accurate, and can yield better partial order reductions.
   Furthermore, no parallelization exists for the DFS\textsubscript{FIFO} algorithm.

Contributions. We implemented the DFS\textsubscript{FIFO} algorithm in the LTSmin [21,5], with both progress states and transitions. For the latter, we extended theory, algorithms, proofs, models and implementation. We compare the runtime and POR performance to that of LTL approaches using NDFS. For DFS\textsubscript{FIFO}, we also investigate the effect of using progress transitions instead of states on POR.

Additionally, we present a parallel livelock algorithm based on DFS\textsubscript{FIFO}, together with a proof of correctness. While the algorithm builds on previous efficient parallelizations of the NDFS algorithm [8,17,19], we show that it has stronger guarantees for parallel scalability due to the nature of the underlying DFS\textsubscript{FIFO} algorithm. At the same time, it retains all the benefits of the original DFS\textsubscript{FIFO} algorithm. This entails the redundancy of the cycle proviso, hence allowing for parallel POR with almost the same reductions as for safety checks.

Our experiments confirm the theoretical expectations: using DFS\textsubscript{FIFO} on four case studies, we observed up to 3.2 times faster runtimes than with the use of an LTL property and the NDFS algorithm, even compared to measurements with the SPIN model checker. But we also confirm all hypotheses of Faragó et al.: the algorithm is more on-the-fly, and POR performance is closer to that of safety checks than the LTL approach, making it up to 5 times more effective than POR in SPIN. Our parallel version of the algorithm can work with POR and features the expected linear scalability. Its combination with POR easily outperforms other parallel approaches [3].

Overview. In Sec. 2, we recapitulate the intricacies of livelock detection via LTL and via non-progress detection, as well as POR. In Sec. 3, we introduce DFS\textsubscript{FIFO} for progress transitions with greater detail and formality than in [11], as well as its combination with POR. Thereafter, in Sec. 4 we provide a parallel version of DFS\textsubscript{FIFO} with a proof of correctness, implementation considerations, and an analysis on its scalability. Sec. 5 presents the experimental evaluation, comparing DFS\textsubscript{FIFO}’s (POR) performance and scalability against the (parallel) LTL algorithms in SPIN [13,15], DiVINE [23], and LTSmin [5,21]. We conclude in Sec. 6.
2 Preliminaries

Model checking of safety properties. Explicit-state model checking algorithms construct \( \mathcal{A}_M \) on-the-fly starting from the initial state \( s_0 \), and recursively applying the next-state function \( \text{post} \) to discover all reachable states \( \mathcal{R}_M \). This only requires storing states (no transitions). As soon as a counterexample is discovered, the exploration can terminate early, saving resources. To reason about these algorithms, it is however easier to consider \( \mathcal{A}_M \) structurally as a graph.

Definition 1 (State Space Automaton). An automaton is a quintuple \( \mathcal{A}_M = (S_M, s_0, \Sigma, T_M, L) \), with \( S_M \) a finite set of states, \( s_0 \in S_M \) an initial state, \( \Sigma \) a finite set of action labels, \( T_M \): \( S_M \times \Sigma \to S_M \) the transition relation, and \( L \): \( S_M \to 2^{AP} \) a state labeling function, over a set of atomic propositions \( AP \).

We also use the recursive application of the transition relation \( T \): \( s \xrightarrow{\alpha} s' \) iff \( \pi \) is a path in \( \mathcal{A}_M \) from \( s \) to \( s' \), or \( s \xrightarrow{\omega} s' \) if \( s = s' \). We treat a path \( \pi \) dually as a sequence of states and a sequence of actions, depending on the context. We omit the subscript \( M \) whenever it is clear from the context.

Now, we can define: the reachable states \( \mathcal{R}_M = \{ s \in S_M \mid s_0 \to^* s \} \), the function \( \text{post} \): \( S_M \to 2^\Sigma \), such that \( \text{post}(s) = \{ \alpha \in \Sigma \mid \exists s' \in S_M : (s, \alpha, s') \in T_M \} \) and \( \alpha(s) \) as the unique next-state for \( s, \alpha \) if \( \alpha \in \text{post}(s) \), i.e. the state \( t \) with \( (s, \alpha, t) \in T_M \). Note that a state \( s \in S \) comprises the variable valuations and process counters in \( M \). Hence, we can use any proposition over these values as an atomic proposition representing a state label. For example, we may write \( \text{progress} \equiv \text{Peterson}_0 = CS \) to have \( \text{progress} \in L(s) \) iff \( s \) represents a state where process instance 0 of Pet is in its critical section \( CS \). Or we can write \( \text{error} \equiv N > 1 \) to express the mutual exclusion property, with \( N \) the number of processes in \( CS \). These state labels can then be used to check safety properties using reachability, e.g., an invariant ‘\(-\text{error}\)’ to check mutual exclusion in \( M \).

LTL model checking. For an LTL property, the property \( \varphi \) is transformed to an \( \omega \)-automaton \( \mathcal{A}_\varphi \) as detailed in [27]. Structurally, the \( \omega \)-automaton extends a normal automaton [Def. 1] with dedicated accepting states [see Def. 2]. Semantically, these accepting states mark those cycles that are part of the \( \omega \)-regular language \( L(\mathcal{A}_\varphi) \) as defined in [Def. 3].

To check correctness of \( \mathcal{M} \) with respect to a property \( \varphi \), the cross product of \( \mathcal{A}_\neg \varphi \) with the state space \( \mathcal{A}_M \) is calculated: \( \mathcal{A}_{M \times \varphi} = \mathcal{A}_M \otimes \mathcal{A}_\neg \varphi \). The states of \( S_{M \times \varphi} \) are formed by tuples \( (s, s') \) with \( s \in S_M \) and \( s' \in S_{\neg \varphi} \), with \( (s, s') \in F \) iff \( s' \in F_{\neg \varphi} \). Hence, the number of possible states \( |S_{M \times \varphi}| \) equals \( |S_M| \cdot |S_{\neg \varphi}| \), whereas the number of reachable states \( |\mathcal{R}_{M \times \varphi}| \) may be smaller. The transitions in \( T_{M \times \varphi} \) are formed by synchronizing the transition labels of \( \mathcal{A}_\neg \varphi \) with the state labels in \( \mathcal{A}_M \). For an exact definition of \( T_{M \times \varphi} \), we refer to [1].

Definition 2 (Accepting states). The set of accepting states \( F \) corresponds to those states with a label \( \text{accept} \in AP \): \( F = \{ s \in S \mid \text{accept} \in L(s) \} \).

Definition 3 (Accepting run). A lasso-formed path \( s_0 \xrightarrow{\omega} s \xrightarrow{\omega} s \) in \( \mathcal{A}_M \), with \( s \in F \), constitutes an accepting run, part of the language of \( \mathcal{A} \): \( vw^2 \in L(\mathcal{A}) \).
As explained in Sec. 1, the whole procedure of finding counterexamples to $\varphi$ for $M$ is now reduced to the graph problem of finding accepting runs in $A_{M \times \varphi}$. This can be solved by the nested depth-first search (ndfs) algorithm, which does at most two explorations of all states $R_{M \times \varphi}$. Since $A_{M \times \varphi}$ can be constructed on-the-fly, ndfs saves resources when a counterexample is found early on.

**Livelock detection.** Livelocks form a specific, but important subset of the liveliness properties and can be expressed as the progress LTL property: $\Box \Diamond \text{progress}$, which states that on each infinite run, progress needs to be encountered infinitely often. As the LTL approach synchronizes the state labels of $A_M$ (see Def. 3), it requires that progress is defined on states as in Def. 4.

**Definition 4 (Progress states).** The set of progress states $S^P$ corresponds to those states with a state label $\text{progress} \in AP$: $S^P = \{ s \in S \mid \text{progress} \in L(s) \}$.

**Definition 5 (Non-progress cycle).** A reachable cycle $\pi$ in $A_M$ is a non-progress cycle ($NPcycle$) iff it contains no progress $P$.

We define $NP$ as a set of states: $NP = \{ s \in S_M \mid \exists \pi : s \xrightarrow{+} \pi \wedge \pi \cap P = \emptyset \}$. 

**Theorem 1.** Under $P = S^P$, $A_M$ contains a NPcycle iff the crossproduct with the progress property $A_{M \times \Box \Diamond \text{progress}}$ contains an accepting cycle.

Livelocks can however also be detected directly on $A_M$ if we consider for a moment that a counterexample to a livelock is formed by an infinite run that lacks progress $P$, with $P = S^P$. By proving absence of such non-progress cycles (Def. 5), we do essentially the same as via the progress LTL property, as Th. 1 shows (see [15] for the proof and details). This insight led to the proposal of dedicated algorithms in [15,11] (cf. dfsfifo in Sec. 3), requiring $|R_M|$ time units to prove livelock freedom. The automaton $A_{\neg \Box \Diamond \text{progress}}$ consists of exactly two states [15], hence $|R_M| \cdot 2 \leq |R_{M \times \varphi}|$. This, combined with the revisits of the ndfs algorithm, makes the LTL approach up to 4 times as costly as dfsfifo. 

**Partial order reduction.** To achieve the reduction as discussed in the introduction, POR replaces the post with an ample function, which computes a sufficient subset of post to explore only relevant interleavings w.r.t the property [16].

For deadlock detection, ample only needs to fulfill the emptiness proviso and dependency proviso (Table 1). The provisos can be deduced locally from $s$, post($s$), and dependency relations $D \subseteq \Sigma_M \times \Sigma_M$ that can be statically overestimated from $M$, e.g. $(\alpha, \beta) \in D$ if $\alpha$ writes to those variables that $\beta$ uses as guard [23]. For a precise definition of $D$ consult [10,26].

In general, the model checking of an LTL property (or invariant) $\varphi$ requires two additional provisos to hold: the visibility proviso ensures that traces included in $A_{\neg \varphi}$ are not pruned from $A_M$, the cycle proviso prevents the so-called ignoring problem [9]. The strong variant C3 (stronger than A4 in [1] Sec. 8.2.2)) is already hard to enforce, so often an even stronger condition, e.g. C3', is implemented. While visibility can still be checked locally, the cycle proviso is a global property, that complicates parallelization [2]. Moreover, the ndfs algorithm revisits states, which might cause different ample sets for the same states, because the procedure is non-deterministic [15]. To avoid any resulting redundant explorations, additional bookkeeping is needed to ensure a deterministic ample set.
For a cycle

We then present a new version of

In the current section, we refine the definition of progress to include transitions.

As argued in [11], progress is more naturally

We define progress transitions

Progress transitions. As argued in [11], progress is more naturally defined on transitions [Def. 6] than on states. After all, the action itself, e.g. the increase of a counter in

defined on transitions (Def. 6) than on states. After all, the action itself, e.g. the increase of a counter in

Thus the cycle

Table 1: POR provisos for the LTL model checking of \( M \) with a property \( \varphi \)

<table>
<thead>
<tr>
<th>C0</th>
<th>emptiness</th>
<th>ample(s) = ( \emptyset ) ( \iff ) post(s) = ( \emptyset )</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>dependency</td>
<td>No action ( \alpha \not\in \text{ample}(s) ) that is dependent on another ( \beta \in \text{ample}(s) ), i.e. ( (\alpha, \beta) \in D ), can be executed in the original ( A_M ) after reaching the state ( s ) and before some action in ( \text{ample}(s) ) is executed.</td>
</tr>
<tr>
<td>C2</td>
<td>visibility</td>
<td>ample(s) ( \not\in ) post(s) ( \implies ) ( \forall \alpha \in \text{ample}(s) : \alpha ) is invisible, which means that ( \alpha ) does not change a state label referred to by ( \varphi ).</td>
</tr>
<tr>
<td>C3</td>
<td>cycle</td>
<td>For a cycle ( \pi ) in ( A_M ), ( \exists s \in \pi : \text{post}(s) = \text{ample}(s) ).</td>
</tr>
<tr>
<td>C3’</td>
<td>cycle (impl.)</td>
<td>ample(s) ( \not\in ) post(s) ( \implies ) ( \exists \alpha \in \text{ample}(s) ) s.t. ( \alpha(s) ) is on the dfs stack.</td>
</tr>
</tbody>
</table>

3 Progress Transitions and DFS\(_{\text{fifo}}\) for Non-Progress

In the current section, we refine the definition of progress to include transitions. We then present a new version of DFS\(_{\text{fifo}}\), an efficient algorithm for non-progress detection by Faragó et al. [11], which supports this broader definition. We also discuss implementation considerations and the combination with POR.

**Progress transitions.** As argued in [11], progress is more naturally defined on transitions [Def. 6] than on states. After all, the action itself, e.g. the increase of a counter in \( M \), constitutes the actual progress. This becomes clear considering the semantical difference between progress transitions and progress states for livelock detection: The figure on the right shows an automaton with \( S \) transitions and progress states for livelock detection: The figure on the right shows an automaton with \( S = \{ s_1 \} \) and \( T = \{ (s_2, \alpha, s_1) \} \).

Thus the cycle \( s_2 \leftrightarrow s_3 \) exhibits only *fake progress* when progress states are used (\( P = S^P \)); the action performing the progress, \( \alpha \), is never taken. With progress transitions (\( P = T^P \)), only \( s_2 \leftrightarrow s_3 \) can be detected as NPcycle. While fake progress cycles could be hidden by enforcing strong (A-)fairness [11], Spin’s weak (A-)fairness [12] is insufficient [11]. But enforcing any kind of fairness is costly [11].

**Definition 6 (Progress transitions/actions).** We define progress transitions as: \( T^P = \{ (s, \alpha, s') \in T \mid \alpha \in \Sigma^P \} \), with \( \Sigma^P \subseteq \Sigma \) a set of progress actions.

**Theorem 2.** DFS\(_{\text{fifo}}\) ensures: \( R \cap N \mathcal{P} \neq \emptyset \iff \text{dfs-fifo}(s_0) = \text{report} \text{ NPcycle} \)

DFS\(_{\text{fifo}}\). Alg. 1 shows an adaptation of DFS\(_{\text{fifo}}\) that supports the definition of progress on both states and transitions (actions), so \( \mathcal{P} = \mathcal{S}^P \cup \Sigma^P \). Intuitively, the algorithm works by delaying progress as long as possible using a BFS and searching for NPcycles in between progress using a DFS. The correctness of this adapted algorithm follows from [Th. 2] which is implied by [Th. 4] with \( P = 1 \).

The FIFO queue \( F \) holds progress states, or immediate successors of progress transitions (which we will collectively refer to as after-progress states), with the exception of the initial state \( s_0 \). The outer dfs-fifo loop handles all after-progress states in breadth-first order. The dfs procedure, starting from a state in \( F \) then explores states up to progress, storing visited states in the set \( V \) [22], and after-progress states in \( F \) [21]. The stack of this search is maintained in a set \( S \) [13] and [23] to detect cycles at [16]. All states \( t \in S \) and their connecting transitions are non-progress by [18] except for possibly the starting state from \( F \).
Therefore, the reduced state space has a ample so
Furthermore,
Combination with POR. While the four-fold performance increase of DFS_FIFO compared to LTL (Sec. 2) is a modest gain, the algorithm provides even more potential as it relaxes conditions on POR, which, after all, might yield exponential gains. In contrast to the LTL method using NDFS, DFS_FIFO does not revisit states, simplifying the ample implementation. Moreover, Lemma 1 shows that DFS_FIFO does not require the cycle proviso using a visibility proviso from Table 2.

**Lemma 1.** Under $\mathcal{P} = \mathcal{S}^P$, $C_2^S$ implies $C_3$. Under $\mathcal{P} = \Sigma^P$, $C_2^T$ implies $C_3$.

**Proof.** If DFS_FIFO with POR traverses a cycle $C$ which makes progress, i.e. $\exists s \in C: s \in \mathcal{S}^P \land \text{ample}(s) \cap \Sigma^P \neq \emptyset$, $C_2^S / C_2^T$ guarantees full expansion of $s$, thus fulfilling $C_3$. If DFS_FIFO traverses a NPcycle, it terminates at l.16.

**Theorem 3.** Th. 2 still holds for DFS_FIFO with $C_0$, $C_1$, $C_2^S / C_2^T$.

**Proof.** Lemma 1 shows that if the $C_0$, $C_1$ and $C_2^S / C_2^T$ hold, so does $C_3$. Furthermore, $C_0$, $C_1$ and $C_2^S / C_2^T$ are independent of the path leading to $s$, so ample($s$) with DFS_FIFO retains stutter equivalence related to progress [14] p.6]. Therefore, the reduced state space has a NPcycle iff the original has one.

**Table 2:** POR visibility provisos for DFS_FIFO

| $C_2^S$ ample($s$) $\neq$ post($s$) $\implies$ s $\notin \mathcal{S}^P$ |
|-----------------|---------------|
| $C_2^T$ ample($s$) $\neq$ post($s$) $\implies \forall \alpha \in$ ample($s$) : $\alpha$ $\notin \Sigma^P$ |
4 A Parallel Liveloop Algorithm based on DFS\textsubscript{FIFO}

Algorithm 2 presents a parallel version of DFS\textsubscript{FIFO}. The algorithm does not differ much from Algorithm 1: the dfs procedure remains largely the same, and only dfs-fifo is split into parallel fifo procedures handling states from the FIFO queue F concurrently. The technique to parallelize the dfs(s, i) calls is based on successful multi-core ndfs algorithms [17,19,8]. Each worker thread i ∈ 1..P uses a local stack \( S_i \), while V and F are shared (below, we show how an efficient implementation can partially localize F). The stacks may overlap (see l.2 and l.9), but eventually diverge because we use a randomized next-state function: \( post_i \) (see l.15).

**Proof of Correctness.** Th. 4 proves correctness of Algorithm 2. We show that the propositions below hold after initialization of Algorithm 2, and inductively that they are maintained by execution of each statement in the algorithm, considering only the lines that influence the proposition. Rather than restricting progress to either transitions or states, we prove the algorithm correct under \( \mathcal{P} = \mathcal{S}P \cup \mathcal{T}P \). Hence, the dual interpretation of paths (see Def. 1) is used now and then. Note that a call to report terminates the algorithm and the callee does not return.

Lemma 2. Upon return of dfs(s, i), s is visited: \( s \in V \).

**Proof.** l.23 of dfs(s, i) adds s to V. \( \Box \)

Lemma 3. Invariantly, all direct successors of a visited state \( v \) are visited or in F: \( \forall v \in V, \alpha \in post(v) : \alpha(v) \in V \cup F \).

**Proof.** After initialization, the invariant holds trivially, as V is empty. V is only modified at l.23, where s is added after all its immediate successors t are considered at l.16–22. If \( t \in V \cup F \), we are done. Otherwise, dfs(s, i) terminates at l.17 or \( t \) is added to V at l.20 (Lemma 2) or to F at l.22. States are removed from F at l.12 but only after being added to V at l.11 (Lemma 2). \( \Box \)

**Corollary 1.** Lemma 3 holds also for a state \( v \not\in V \) in dfs(v, i) just before l.23.

---

| Algorithm 2 Parallel DFS\textsubscript{FIFO} (PDFS\textsubscript{FIFO}) |
|--------------------------|--------------------------|
| 1: procedure dfs-fifo(s\(_0\), P) | 13: procedure dfs(s, i) |
| 2: \( F := \{s\_0\} \)  \( \triangleright \) Frontier queue | 14: \( S_i := S_i \cup \{s\} \) |
| 3: \( V := \emptyset \)  \( \triangleright \) Visited set | 15: for all \( \alpha \in post_i(s) \) do |
| 4: \( S_i := \emptyset \) for all \( i \in 1..P \)  \( \triangleright \) Stacks | 16: if \( t \in S_i \land \alpha \not\in P \) then report NPcycle |
| 5: fifo(1) \( \parallel \ldots \parallel \) fifo(P) | 17: |
| 6: report progress ensured | 18: if \( t \not\in V \) then |
| 7: procedure fifo(i) | 19: if \( \alpha, t \not\in P \) then |
| 8: while \( F \not= \emptyset \) do | 20: dfs(t, i) |
| 9: \( s := \text{some } s \in F \) | 21: else if \( t \not\in F \) then |
| 10: if \( s \not\in V \) then | 22: \( F := F \cup \{t\} \) |
| 11: dfs(s, i) | 23: \( V := V \cup \{s\} \) |
| 12: \( F := F \setminus \{s\} \) | 24: \( S_i := S_i \setminus \{s\} \) |
Lemma 4. Invariantly, all paths from a visited state $v$ to a state $f \in F \setminus V$ contain progress: $\forall \pi, v \in V, f \in F \setminus V: \exists s : v \xrightarrow{\pi}s f \Rightarrow \pi \cap \pi' \neq \emptyset$.

Proof. After initialization of the sets $V$ and $F$, the lemma is trivially true. These sets are modified at l.12, l.22, and l.23 (omitting the trivial case):

Let $i$ be the first worker thread to add a state $t$ to $V$ in $\text{dfs}(s, i)$ at l.22. If some other worker $j$ adds $t$ to $V$, the invariant holds trivially, so we consider $t \notin V$. By l.19, all paths $v \rightarrow^* s \rightarrow t$ contain progress. By contradiction, we show that all other paths that do not contain $s$ also contain progress: Assume that there is a $v \in V$ such that $v \xrightarrow{\pi}s t$ and $\pi \cap \pi' = \emptyset$. By induction on the length of the path $\pi$ and Lemma 3, we obtain either $t \in V$, a contradiction, or $t \notin V$, contradicting the assumption that worker $i$ is first, or another $f \neq t$ with $f \notin F \setminus V$, for which the induction hypothesis holds.

Assume towards a contradiction that $i$ is the first worker thread to add a state $s$ to $V$ at l.23 of $\text{dfs}(s, i)$. So, we have $s \notin V$ before l.23. By Cor. 1, for all immediate predecessors $t$ of $s$, i.e. for all $t = \alpha(s)$ such that $\alpha \in \text{post}(s)$, we have $t \in V$ or $t \notin F \setminus V$. In the first case, since $s \neq t$, the induction hypothesis holds for $t$. In the second case, if $t = s$, the invariant trivially holds after l.23 and if $t \neq s$, we have $\alpha, t \in \mathcal{P}$, since otherwise $t \in V$ by l.19 and l.20 (Lemma 2). Thus the invariant holds for all paths $s \rightarrow f$.

Remark 1. Note that a state $s \in F$ might at any time be also added to $V$ by some other worker thread in two cases: (1) $s \notin \mathcal{S}^P$, i.e. it was reached via a progress transition (see l.19), but is reachable via some other non-progress path, or (2) another worker thread $j$ takes $s$ from $F$ at l.10 and completes $\text{dfs}(s, j)$.

Lemma 5. Invariantly, visited states do not lie on $\text{NPcycles}$: $V \cap \mathcal{NP} = \emptyset$.

Proof. Initially, $V = \emptyset$ and the lemma holds trivially. Let $i$ be the first worker thread to add $s$ to $V$ in $\text{dfs}(s, i)$ at l.23. So we have $s \in V$ just after l.23 of $\text{dfs}(s, i)$. Assume towards a contradiction that $s \in \mathcal{NP}$. Then there is a $\text{NPcycle}$ $s \rightarrow t \rightarrow^+ s$ with $s \neq t$ since otherwise l.17 would have reported a $\text{NPcycle}$. Now by Lemma 3, $t \in V \cup F$. By the induction hypothesis, $t \notin V$, so $t \notin F \setminus V$. Lemma 4 contradicts $s \rightarrow t$ making no progress.

Lemma 6. Upon return of $\text{dfs-fifo}$, all reachable states are visited: $\mathcal{R} \subseteq V$.

Proof. After $\text{dfs-fifo}(s_0, P)$, $F = \emptyset$ by l.18. By l.2, l.11 and Lemma 2, $s_0 \in V$. So by Lemma 3, $\mathcal{R} \subseteq V$.

Lemma 7. $\text{dfs-fifo}$ terminates and reports an $\text{NPcycle}$ or progress ensured.

Proof. Upon return of a call $\text{dfs}(s, i)$ for some $s \in F$ at l.11, $s$ has been added to $V$ (Lemma 2), removed from $F$ at l.12, and will never be added to $F$ again. Hence the set $V$ grows monotonically, but is bounded, and eventually $F = \emptyset$. Thus eventually all $\text{dfs}$ calls terminate, and $\text{dfs-fifo}(s_0, P)$ terminates too.

Lemma 8. Invariantly, the states in $S_i$ form a path without progress except for the first state: $S_i = \emptyset$ or $S_i = \pi \cap S$ for some $s \xrightarrow{\pi} s'$ and $\pi \cap \mathcal{P} \subseteq \{s_1\}$.

Proof. By induction over the recursive $\text{dfs}(s, i)$ calls, we obtain $\pi$. At l.20 we have $\alpha, t \notin \mathcal{P}$, but at l.11 we may have $s \in \mathcal{S}^P$ (by l.19 and l.22).
Theorem 4. PDFS\textsubscript{fifo} ensures: $R \cap \mathcal{NP} \neq \emptyset \iff$ dfs-fifo$(s_0, P) = \text{return NPcycle}$

Proof. We split the equivalence into two cases:

$\Leftarrow$: We have a cycle: $s \xrightarrow{a} t \xrightarrow{a} s$ s.t. $(\{a\} \cup \pi) \cap \mathcal{P} = \emptyset$ by l.16 and Lemma 8.

$\Rightarrow$: Assume that dfs-fifo$(s_0, P) \neq \text{NPcycle}$ $\land R \cap \mathcal{NP} \neq \emptyset$. However, at l.6 $R \subseteq V$ by Lemma 6 and Lemma 7, hence $R \cap \mathcal{NP} = \emptyset$ by Lemma 5.

Implementation. For a scaling implementation, the hash table storing $F$ and $V$ (see Sec. 3) is maintained in shared memory using a lockless design [20,18]. Storing also the queue $F_q$ in shared memory, however, would seriously impede scalability due to contention (recall that $F$ is maintained as both hash table and queue $F_q$). Our more efficient implementation splits $F_q$ into $P$ local queues $F_q^i$, such that $F \subseteq \bigcup_{i = 1}^{P} F_q^i$ (Remark 1 explains the $\subseteq$).

To implement load balancing, one could relax the constraint at l.21 to $s \notin F_q$, so that after-progress states end up on multiple local queues. Provided that $A_M$ is connected enough, which it usually is in model checking, this would provide good work distribution already. On the other hand, the total size of all queues $F_q^i$ would grow proportional to $P$, wasting a lot of memory on many cores.

1: procedure fifo$(i)$
2: $F_q^i := \{s_0\}$
3: while steal($F_q^i$) do
4: $F_q^i := F_q^i \setminus \{s\}$
5: if $s \notin V$ then
6: dfs$(s, i)$

Therefore, we instead opted to add explicit load balancing via work stealing. The code on the left illustrates this. If the local queue $F_q^i$ is empty, the steal function grabs states from another random queue $F_q^j$ and adds them to $F_q^i$, returning false iff it detects termination. Inspection of Lemma 3 and Lemma 7 shows that removing $s$ from $F$ is not necessary.

The proofs show that correctness of PDFS\textsubscript{fifo} does not require $F$ to be in strict FIFO order (as l.9 does not enforce any order). To optimize for scalability, we enforce a strict BFS order via synchronizations\footnote{Parallel BFS algorithms, with and without synchronization, are described in [7].} between the BFS levels only optionally\footnotemark[5]. As trade-off, counterexamples are no longer guaranteed to be the shortest with respect to progress, and the size of $F$ may increase (see Remark 1).

Analysis of scalability. Experiments with multi-core NDFS\footnote{The command line option --strict turns on strict PDFS\textsubscript{fifo} in LTSmin.} demonstrated that these parallelization techniques make the state-of-the-art for LTL model checking. Because of the BFS nature of DFS\textsubscript{fifo}, we can expect even better speedups. Moreover, in [17], additional synchronization was needed to prevent workers from early backtracking; a situation in which two workers exclude a third from part of the state space. The figure on the right illustrates this: Worker 1 can visit $s$, $v$, $t$ and $u$, and then halt. Worker 2 can visit $s$, $u$, $t$ and $v$ and backtrack over $v$. If now Worker 1 resumes and backtracks over $u$, both $v$ and $u$ are in $V$. A third worker will be excluded from visiting $t$, which might lead to a large part of the state space. Lemma 3 shows that this is impossible for PDFS\textsubscript{fifo} as the successors of visited states are either visited or in $F$ (treated in efficient parallel BFS), but never do successors lie solely on the stack $S_i$ (as in cNDFS).
5 Experimental Evaluation

In the current section, we benchmark the performance of DFS_FIFO, and its combination with POR, using both progress states and progress transitions. We compare the results against the LTL approach with progress property using, inter alia, SPIN [12]. We also investigate the scalability of PDFS_FIFO, and compare the results against the multi-core NDFS algorithm CNDFS, the state-of-the-art for parallel LTL [8,5], and the piggyback algorithm in SPIN (PB). Finally, we investigate the combination of PDFS_FIFO and POR, and compare the results with OWCY [3], which uses a topological sort to implement parallel LTL and POR [2].

We implemented PDFS_FIFO (Alg. 2 with work stealing and both strict/non-strict BFS order) in LTSMIN 2.0. LTSMIN has a frontend for PROMELA, called SPINS [12], and one for the DVE language, allowing fair comparison against SPIN 6.2.3 and DiVinE 2.5.2 [3]. To ensure similar state counts, we turned off control-flow optimizations in SPINS/SPIN, because SPIN has a more powerful optimizer, which can be, but is not yet implemented in SPINS. Only the GIOP model (described below) still yields a larger state count in SPINS/LTSMIN than in PDFS_FIFO. We still include it as it nicely features the benefits of DFS_FIFO over NDFS.

We benchmarked on a 48-core machine (a four-way AMD Opteron 6168) with 128GB of main memory, and considered 4 publicly available PROMELA models with progress labels, and adapted SPINS to interpret the labels as either progress states, as in SPIN, or progress transitions. leader is the efficient leader election protocol [10]. The Group Address Registration Protocol (GARP) is a datalink-level multicast protocol for a bridged LAN. General Inter-Orb Protocol (GIOP) models service oriented architectures. The model i-Protocol represents the GNU implementation of this protocol. We use a different leader election protocol (leader DKR) from [24] for comparison against DiVinE. For all these models, the livelock property holds under $P = S^P$ and $P = T^P$.

Performance. In theory, DFS_FIFO can be up to four times as fast as using the progress LTL formula and NDFS. To verify this, we compare DFS_FIFO to NDFS in LTSMIN and SPIN. In LTSMIN, we used the command line: prom2lts-mc --state=tree -s28 --strategy=[dfs fifo/ndfs] [model], which replaces the shared table (for F and V) by a tree table for state compression [18]. In SPIN, we used compression as well (collapse [12]): cc -O2 -DNP -DNOFAIR -DNOREDUCE -DNBOUNDCHECK -DCOLLAPSE -o pan pan.c, and pan -m100000 -l -w28, avoiding table resizes and overhead. In both tools, we also ran DFS_reachability with similar commands. We write oom for runs that overflow the main memory.

Table 3 shows the results: As expected, $|R_{\text{LTL}}|$ is 1.5 to 2 times larger than $|R|$ for both SPIN and LTSMIN; GIOP fits in memory for DFS_FIFO but the LTL cross-product overflows (NDFS). $T_{\text{NDFS}}$ is about 1.5 to 4 times larger than $T_{\text{DFS}}$ for SPIN, 2 to 5 times larger for LTSMIN (cf. Section 3). $T_{\text{DFS}}$ is 1.5 to 2 times larger than $T_{\text{DFS}}$, likely caused by its set inclusion tests on $S$ and $F$. $T_{\text{NDFS}}$ is 1.6 to 3.2 times larger than $T_{\text{DFS}}$.

---

6 LTSMIN is open source, available at: http://fmt.cs.utwente.nl/tools/ltsmin
7 Models that we modified are available at: http://doiop.com/leader4DFS_FIFO
Table 3: Runtimes (sec) of (sequential) dfs, dfs\textsubscript{fifo}, ndfs in spin and LTSmin

<table>
<thead>
<tr>
<th></th>
<th>LTSmin</th>
<th></th>
<th></th>
<th></th>
<th>SPIN</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>R</td>
<td>$</td>
<td>$</td>
<td>R\text{ltl}</td>
<td>$</td>
<td>$T_{dfs}$</td>
</tr>
<tr>
<td>leader</td>
<td>4.5E7</td>
<td>198%</td>
<td>153.7</td>
<td>233.2</td>
<td>753.6</td>
<td>4.5E7</td>
<td>198%</td>
</tr>
<tr>
<td>garp</td>
<td>1.2E8</td>
<td>150%</td>
<td>377.1</td>
<td>591.2</td>
<td>969.2</td>
<td>1.2E8</td>
<td>146%</td>
</tr>
<tr>
<td>giop</td>
<td>2.7E9</td>
<td>oom</td>
<td>21,301.4</td>
<td>43,154.3</td>
<td>oom</td>
<td>8.4E7</td>
<td>181%</td>
</tr>
<tr>
<td>i-prot</td>
<td>1.4E7</td>
<td>140%</td>
<td>28.4</td>
<td>41.4</td>
<td>70.6</td>
<td>1.4E7</td>
<td>145%</td>
</tr>
</tbody>
</table>

Table 4: Runtimes (sec) / queue sizes of the parallel algorithms: dfs, dfs\textsubscript{fifo} and cndfs in LTSmin, and PB in spin

<table>
<thead>
<tr>
<th></th>
<th>DFS</th>
<th>PDfs\textsubscript{fifo}</th>
<th>Cndfs</th>
<th>PB</th>
<th>PDfs\textsubscript{fifo}-%</th>
<th>PDfs\textsubscript{fifo}-\text{strict}</th>
<th>Cndfs</th>
</tr>
</thead>
<tbody>
<tr>
<td>leader</td>
<td>153.7</td>
<td>3.8</td>
<td>233.2</td>
<td>5.7</td>
<td>1013.6</td>
<td>58.6</td>
<td>1180.0</td>
</tr>
<tr>
<td>garp</td>
<td>377.1</td>
<td>13.1</td>
<td>501.2</td>
<td>15.3</td>
<td>1061.0</td>
<td>58.6</td>
<td>1180.0</td>
</tr>
<tr>
<td>giop</td>
<td>2.1E4</td>
<td>463.3</td>
<td>4.3E4</td>
<td>9.7E2</td>
<td>oom</td>
<td>oom</td>
<td>1.2E3</td>
</tr>
<tr>
<td>i-prot</td>
<td>28.4</td>
<td>0.7</td>
<td>41.4</td>
<td>1.1</td>
<td>76.2</td>
<td>17.7</td>
<td>1.0E6</td>
</tr>
</tbody>
</table>

Table 5: POR (%) for dfs\textsubscript{fifo}, dfs\textsubscript{fifo}, dfs and ndfs in spin and LTSmin

<table>
<thead>
<tr>
<th></th>
<th>LTSmin</th>
<th>SPIN</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DFS\textsubscript{fifo}</td>
<td>DFS\textsubscript{fifo}</td>
<td>Cndfs</td>
<td>DFS</td>
<td>NDfs</td>
<td>DFS\textsubscript{fifo}</td>
<td>Cndfs</td>
</tr>
<tr>
<td>leader</td>
<td>0.32%</td>
<td>0.49%</td>
<td>99.99%</td>
<td>99.99%</td>
<td>0.03%</td>
<td>1.15%</td>
<td></td>
</tr>
<tr>
<td>garp</td>
<td>1.90%</td>
<td>2.18%</td>
<td>4.29%</td>
<td>4.29%</td>
<td>10.56%</td>
<td>12.73%</td>
<td></td>
</tr>
<tr>
<td>giop</td>
<td>1.86%</td>
<td>1.86%</td>
<td>3.77%</td>
<td>3.77%</td>
<td>oom</td>
<td>oom</td>
<td></td>
</tr>
<tr>
<td>i-prot</td>
<td>16.14%</td>
<td>31.83%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>24.01%</td>
<td>41.37%</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1: Speedups of dfs, dfs\textsubscript{fifo} and cndfs in LTSmin, and piggyback in spin
### Parallel scalability

To compare the parallel algorithms in LTSMin, we use the options `-threads=P` and `-strategy=[dfs|fifo|cndfs]`, where `P` is the number of worker threads. In SPIN, we use `-DFSPAR`, which also turns on lossy state hashing [13], and run the `pan` binary with an option `-uP`. This turns on a parallel, linear-time, but incomplete, cycle detection algorithm called piggyback (PB) [13]. It might also be unsound due its combination with lossy hashing [4]. Fig. 1 shows the obtained speedups: As expected, reachability [20] and PDFS FIFO scale almost ideally, while CNDFS exhibits sub-linear scalability, even though it is the fastest parallel LTL solution [8]. PB also scales sub-linearly. Since LTSMin sequentially competes with SPIN (Table 4 except for GIOP), scalability can be compared.

### Parallel memory use

We expected few state duplication in `F` on local queues (see Remark 1). To verify this, we measured the total size of all local queues and hash tables using counters for strict and non-strict PDFS FIFO, and CNDFS. Table 4 shows $Q_P = \sum_{i=1..P} |F_i^q| + |S_i|$ averaged over 5 runs: Non-strict PDFS FIFO shows little difference from the strict variant, and $Q_{48}$ is at most 20% larger than $Q_1$ for all PDFS FIFO. Due to the randomness of the parallel runs, we even have $Q_{48} < Q_1$ in many cases. Revisits occurred at most 2.6% using 48 cores. In the case of CNDFS, the combined stacks typically grow because of the larger DFS searches. Accordingly, we found that PDFS FIFO's total memory use with 48 cores was between 87% and 125% compared to sequential DFS. In the worst case, PDFS FIFO (with tree compression) used 52% of the memory use of PB (collapse compression and lossy hashing) [13] – GIOP excluded as its state counts differ.

### POR performance

LTSMin's POR implementation (option `-por`) is based on stubborn sets [20], described in [24], and is competitive to SPIN's [5]. We extended it with the alternative provisos for DFS FIFO: $C2^S$ and $C2^T$. Table 5 shows the relative number of states, using the different algorithms in both tools: For all models, both LTSMin and SPIN are able to obtain reductions of multiple orders of magnitude using their DFS algorithms. We also observe that much of this benefit disappears when using the NDFS LTL algorithm due to the cycle proviso, although SPIN often performs much better than LTSMin in this respect. Also DFS FIFO with progress states (column DFS FIFO S), performs poorly: apparently, the $C2^S$ proviso is so restrictive that many states are fully expanded. But DFS FIFO with progress transitions (column DFS FIFO T) retains DFS's impressive POR with at most a factor 2 difference.

---

| $N$ | Alg. | $|\mathcal{R}|$ | $|\mathcal{T}|$ | $T_{1s}$ | $T_{cb}$ | $U$ | $|\mathcal{R}^{\text{pos}}|$ | $|\mathcal{T}^{\text{pos}}|$ | $T_{1s}^{\text{pos}}$ | $T_{cb}^{\text{pos}}$ | $U^{\text{pos}}$ |
|-----|------|--------------|--------------|----------|----------|-----|----------------|----------------|-----------------|-----------------|----------------|
| 9   | CNDFS | 3.6E7        | 2.3E8        | 502.6    | 12.0     | 41.8 | 27.9%          | 0.1%           | 211.8           | n/a             | n/a             |
| 9   | PDFS FIFO | 3.6E7 | 2.3E8 | 583.6 | 14.3 | 40.8 | 1.5% | 0.0% | 12.9 | 3.6 | 3.5 |
| 9   | OWCTY | 3.6E7        | 2.3E8        | 498.7    | 51.9     | 9.6  | 12.6%          | 0.0%           | 578.4           | 35.7            | 16.2            |
| 10  | CNDFS | 2.4E8        | 1.7E9        | 600.0    | 40.0     | 90.7 | 19.3%          | 5.4%           | 1102.7          | n/a             | n/a             |
| 10  | PDFS FIFO | 2.4E8 | 1.7E9 | 600.0 | 109.3 | 40.3 | 0.7% | 0.1% | 35.0 | 2.5 | 14.0 |
| 10  | OWCTY | 2.4E8        | 1.7E9        | 600.0    | 466.1    | 40.0 | 8.7%           | 2.2%           | 35.0            | 466.1           | 40.0            |
| 11  | PDFS FIFO | 30' | 30' | 30' | 30' | 30' | 5.1E6 | 7.1E6 | 40.0 | 109.8 | 5.3 | 20.7 |
| 11  | OWCTY | 30' | 30' | 30' | 30' | 30' | 9.3E7 | 1.7E8 | 40' | 1036.5 | 30' |
| 12  | PDFS FIFO | 30' | 30' | 30' | 30' | 30' | 1.6E7 | 2.2E7 | 369.1 | 11.2 | 33.0 |
| 13  | PDFS FIFO | 30' | 30' | 30' | 30' | 30' | 6.6E7 | 9.2E7 | 1640.5 | 38.1 | 43.0 |
| 14  | PDFS FIFO | 30' | 30' | 30' | 30' | 30' | 2.0E8 | 2.9E8 | 40' | 120.3 | 30' |
| 15  | PDFS FIFO | 30' | 30' | 30' | 30' | 30' | 8.4E8 | 1.2E9 | 40' | 527.5 | 30' |
Scalability of parallelism and POR. We created multiple instances of the leader$_{DKR}$ models by varying the number of nodes $N$ and expressed the progress LTL property in DiVinE. We start DiVinE’s state-of-the-art parallel LTL-POR algorithm, owcty, by: divine owcty [model] -wP -i30 -p. With the options described above, we turned on POR in LTSmin and ran PDFS$_{fifo}$ and CNDFS, for comparison. We limited each run to half an hour (30' indicates a timeout). Piggyback reported contradictory memory usage and far fewer states (e.g., <1%) compared to DFS with POR, although it must meet more provisos. Thus we did not compare against piggyback and suspect a bug.

Table 6 shows that PDFS$_{fifo}$ and POR complement each other rather well: Without POR (left half of the table) the almost ideal speedup ($U = \frac{T_1}{T_{48}} = 40.8$) allows to explore one model more: $N \leq 10$ instead of only $N = 9$. When enabling POR (right half of the table), we see again multiple orders of magnitude reductions, while parallel scalability reduces to $U = 3.5$ for $N = 9$, because of the small size of the reduced state space ($|\mathcal{R}_{POR}|$). When increasing the model size to $N = 13$ the speedup grows again to an almost ideal level ($U = 43$). With POR, the parallelism allows us to explore two more models within half an hour, i.e., $N \leq 15$. While owcty and NDFS also show this effect, it is less pronounced due to their cycle proviso, allowing $N \leq 11$ for owcty and $N \leq 9$ for NDFS.

As livelocks are disjoint from the class of weak LTL properties, owcty could become non-linear, but it required only one iteration for leader$_{DKR}$.

As PDFS$_{fifo}$ revisits states, the random next-state function could theoretically weaken POR (as for NDFS, see Sec. 2). But for all our 5 models, this did not occur.

On-the-fly performance. We created a leader election protocol with early (shallow) and another with late (deep) injected NPcycles (see Table 4, [10]).

The table on the right shows the average runtime in seconds ($T_1$) and counterexample length ($C_1$) over five runs. Since PDFS$_{fifo}$ finds shortest counterexamples, it outperforms CNDFS for shallow (more relevant in practice) and pays a penalty for deep. Both algorithms benefit greatly from massive parallelism (see also [19]).

6 Conclusions

We showed, in theory and in practice, that model checking livelocks, an important subset of liveness properties, can be made more efficient by specializing on it. For our PDFS$_{fifo}$ implementation with progress transitions, POR becomes significantly stronger (cf. Table 5), parallelization has linear speedup (cf. Fig. 1), and both can be combined efficiently (cf. Table 6).

Acknowledgements. We thank colleagues Mark Timmer, Mads Chr. Olesen, Christoph Scheben and Tom van Dijk for their useful comments on this paper.

References