Improved Multi-Core Nested Depth-First Search

Sami Evangelista[1], Alfons Laarman[2], Laure Petrucci[1], and Jaco van de Pol[2]

[1] LIPN, CNRS UMR 7030 — Université Paris 13, France
[2] Formal Methods and Tools, University of Twente, The Netherlands

Abstract. This paper presents CNDFS, a tight integration of two earlier multi-core nested depth-first search (NDFS) algorithms for LTL model checking. CNDFS combines the different strengths and avoids some weaknesses of its predecessors. We compare CNDFS to an earlier ad-hoc combination of those two algorithms and show several benefits: It has shorter and simpler code and a simpler correctness proof. It exhibits more robust performance with similar scalability, while at the same time reducing memory requirements.

The algorithm has been implemented in the multi-core backend of the LTSMIN model checker, which is now benchmarked for the first time on a 48 core machine (previously 16). The experiments demonstrate better scalability than other parallel LTL model checking algorithms, but we also investigate apparent bottlenecks. Finally, we noticed that the multi-core NDFS algorithms produce shorter counterexamples, surprisingly often shorter than their BFS-based counterparts.

1 Introduction

Model checking is a resource demanding task that can be performed by a systematic exploration of a huge directed graph representing the dynamic behaviour of the analysed system. Although memory is usually the major bottleneck, execution times can also often exceed acceptable limits. For instance the exploration of a 10^9 states graph at a high exploration rate of 10^5 states per second would take more than a day. This remains acceptable but becomes problematic when increasing the number of system configurations and properties analysed. Hence, model checking has gained a renewed interest with the advent of multi-core architectures that can help tackle this time explosion.

Some properties like safety properties rely on a complete enumeration of system states and can thus be easily parallelised since they do not ask for a specific search order. However, the problem is harder when it comes to the verification of Linear Time temporal Logic (LTL) properties. LTL model checking can be reduced to a cycle detection problem and state-of-the-art algorithms [8,9,11] proceed depth-first since cycles are more easily discovered using this search order. However, this characteristic also makes them unsuitable for parallel architectures since DFS is inherently sequential [20].

One approach to address this issue is to sacrifice the optimal linear complexity provided by DFS algorithms and switch to BFS-like algorithms, which are highly scalable both theoretically and experimentally. We compare our approach to the best representative of that family. More recently, two algorithms (LNDFS from [13] and ENDFS from [10]) adapted the well known Nested DFS (NDFS) algorithm [8] to multi-core architectures. They share the principle of launching multiple instances of NDFS that synchronise

S. Chakraborty and M. Mukund (Eds.): ATVA 2012, LNCS 7561, pp. 269–283, 2012.

themselves to avoid useless state revisits. Although they are heuristic algorithms in the sense that, in the worst case, they reduce to spawn multiple unsynchronised instances of NDFS, the experiments reported in [13,14] show good practical speedups.

The contribution of this paper is an improvement to both the LNDFS and ENDFS algorithms, called CNDFS. This new algorithm is both much simpler and uses less memory, making it more compatible with lossy compression techniques such as tree compression [17] that can compress large states down to two integers. We also pursue a thorough experimental evaluation of this algorithm on the models of the BEEM database [18] with an implementation of this algorithm on top of the LTSMIN toolset [16]. The outcome of these experiments is threefold. Firstly, CNDFS exhibits a similar speedup to its predecessors, but achieves this more robustly, with smoother speedup lines, while using less memory. Second, it combines nicely with heuristics limiting the amount of redundant work performed by individual threads. Finally, in the presence of bugs, it reports counterexamples that are usually much shorter than those reported by NDFS and, more importantly, this length tends to decrease as more working threads get involved in the verification. This property is quite appreciable from a user perspective as it eases the task of error correction.

The outline of this paper is the following. In Section 2 we formally express the LTL model checking problem and review existing (sequential and parallel) algorithms that address it. CNDFS, our new algorithm, is introduced and formally proven in Section 3. Our experimental evaluation of this algorithm is summarised in Section 4. Finally, Section 5 concludes our paper and explores some research perspectives to this work.

2 Background

We give in this section the few ingredients that are required for the understanding of this paper and briefly review existing works in the field of explicit parallel LTL model checking based on the automata theoretic approach.

2.1 The Automata Theoretic Approach to LTL Model Checking

LTL model checking is usually performed following the automata-based approach originating from [22] that proceeds in several steps. In this paper we focus only on the last step of the process that can be reduced to a graph problem: given a graph representing the synchronised product of the Büchi property automaton and the state space of the system, find a cycle containing an accepting state. Any such identified cycle determines an infinite execution of the system violating the LTL formula. In this paper we will only reason on *automaton graphs* that result from the product of a Büchi property automaton and a system graph describing the dynamic behaviour of the modelled system.

Definition 1 (Automaton graph). *An* automaton graph *is a tuple* $\mathcal{G} = (\mathcal{S}, \mathcal{T}, \mathcal{A}, s_0)$*, where* $\mathcal{S}$ *is a finite set of states;* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ *is a set of transitions;* $\mathcal{A} \subseteq \mathcal{S}$ *is the set of accepting states; and* $s_0 \in \mathcal{S}$ *is an initial state.*

Notations. Let $(\mathcal{S}, \mathcal{T}, \mathcal{A}, s_0)$ be an automaton graph. For $s \in \mathcal{S}$ the set of its *successor states* is denoted by $succ(s) = \{s' \in \mathcal{S} \mid (s,s') \in \mathcal{T}\}$. $(s,s') \in \mathcal{T}$ is also denoted by $s \to s'$.

$s \rightarrow^+ s'$ ($s \rightarrow^* s'$) denotes the (reflexive) transitive closure of $\mathcal{T}$, i.e. the fact that s' is *reachable* from s. A *path* is a state sequence $s_1, \ldots, s_n$ with $s_i \rightarrow s_{i+1}, \forall i \in \{1, \ldots, n-1\}$, a *cycle* is a path $s_1, \ldots, s_n$ with $s_1 = s_n$ and a cycle $C \equiv s_1, \ldots, s_n$ is an *accepting cycle* if $C \cap \mathcal{A} \neq \emptyset$. An *accepting run* is an accepting cycle reachable from the initial state: $s_0, \ldots, s_i, \ldots, s_n$ where $s_i = s_n$. The *LTL model checking problem* consists of finding an accepting run in an automaton graph. An LTL model checking algorithm proceeds on-the-fly if it can report an accepting run without visiting all transitions.

2.2 Sequential LTL Model Checking Algorithms

NDFS [8] was the first LTL model checking algorithm proposed. It enjoys several nice properties: an optimal linear complexity, the on-the-fly discovery of accepting cycles and a low memory consumption (2 bits per state). Two variations of Tarjan's algorithm for SCC decomposition [9,11] have also been proposed with similar characteristics but we focus here on NDFS as our new algorithm is a direct descendant of this one.

Alg. 1 NDFS [8] as presented in [21].

```
 1: dfsBlue(s_0)
 2: procedure dfsBlue(s) is
 3:     s.cyan := true
 4:     for all s' in succ(s) do
 5:         if ¬s'.blue then dfsBlue(s')
 6:     if s ∈ A then dfsRed(s)
 7:     s.cyan := false
 8:     s.blue := true
 9: procedure dfsRed(s) is
10:     s.red := true
11:     for all s' in succ(s) do
12:         if s'.cyan then exit(cycle)
13:         if ¬s'.red then dfsRed(s')
```

The pseudo-code of this algorithm is given by Alg. 1. The algorithm performs a first level DFS (the blue DFS) to discover accepting states. When such a state is backtracked from, a second level DFS (the red DFS) is launched to see whether this accepting state (now called the *seed*) is reachable from itself and is thus part of an accepting cycle. It is sufficient to find a path back to the stack of the blue DFS [21], hence the *cyan* colour in Alg. 1. Correctness depends on the fact that different invocations of the red DFS happen in post-order. The algorithm works in linear time: each state is visited at most twice, since the result of a red DFS can be reused in subsequent red DFSs; states retain their red colour.

2.3 Parallel LTL Model Checking Algorithms for Shared-Memory Architectures

In the field of parallel LTL model checking, the first algorithms designed targeted distributed memory architectures like clusters of machines. This family of algorithms includes MAP [6], OWCTY [7] and BLEDGE [2]. It is however well known that this kind of message passing algorithm can be easily ported to shared-memory architectures like multi-core computers although the specificities of these architectures must be considered to achieve good scalability [4]. Their common characteristic is to rely on some form of breadth-first search (BFS) of the graph that has the advantage of being easily parallelised, unlike depth-first search (DFS) [20]. They hence deliver excellent speed-ups but sacrifice optimality and the ability to report accepting cycles on-the-fly. A combination of OWCTY and MAP (OWCTY+MAP [3]) restores "on-the-flyness", is linear-time for the class of weak LTL properties, and maintains scalability.

SWARM verification [12] consists of spawning multiple unsynchronised instances of NDFS each exploring the graph in a random way. Accepting cycles are expected to be reported faster thanks to randomised parallel search, but in the absence of such cycles parallelisation does not help. This pragmatic strategy however targets graphs that are too large in any case to be explored in reasonable time. The purpose is then to maximise the graph coverage in a given time frame and thereby increase confidence in the model.

Two recent multi-core algorithms follow the principle of the SWARM technique but deviate from it in that working threads executing NDFS are synchronised through the sharing of some state attributes. In the first one, LNDFS [13], workers share the outcome of the red (nested) search which can then also be used to prune the blue search. Since the blue flags are not shared among threads, the red searches are still invoked in the appropriate DFS postorder. The ENDFS algorithm [10] also allows the sharing of blue flags, but a sequential emergency procedure is triggered if the appropriate invocation order of the red DFS is not respected. Moreover, to maintain correctness, information on a red DFS in progress cannot be transmitted in "real time" to other threads: the states visited by a red DFS are only marked globally red after it has returned.

A thorough experimental comparison of ENDFS and LNDFS [14] led to the main conclusion that ENDFS and LNDFS complement each other on a variety of models: the larger amount of information shared by ENDFS can potentially yield a better work distribution, but LNDFS is to be preferred when ENDFS threads often launch unfruitful emergency procedures. Since this emergency procedure launches the sequential NDFS algorithm, large portions of the graph may then be revisited, in the worst case by all workers. Hence, a combination of ENDFS and LNDFS was proposed [14] to remedy the downsides of the two algorithms. The principle of that parallel algorithm (called NMCNDFS) is to run ENDFS but replace its sequential emergency procedure by a parallel LNDFS. Experiments show that this combination pays off: NMCNDFS is always at least as fast as ENDFS or LNDFS.

While NMCNDFS combines the strengths of both earlier algorithms in terms of performance, it also conjoins their memory usage. LNDFS requires $2P + \log_2(P) + 1$ bits per state (2 local colours for all P workers, a synchronisation counter and a global red bit) and ENDFS $4P + 3$ (2 local colours plus another 2 for the repair procedure and 3 global bits: $\{dangerous, red, blue\}$). Next to more than doubling the memory usage, the conglomerated algorithm is long and complex.

3 A New Combination of Multi-Core NDFS

To mitigate the downsides of NMCNDFS, we present a new algorithm, CNDFS, shown in Alg. 2. Like the previous multi-core algorithms, it is based on the principle of SWARM worker threads (indicated by subscript p here), sharing information via colours stored in the visited states, here: *blue* and *red*. After randomly ($shuffle_p^{blue}$) visiting all successors (l.13–l.15), a state is marked blue at l.16 (meaning "globally visited") and causing the (other) blue DFS workers to lose the strict postorder property.

If the state s is accepting, as usual, a red DFS is launched at l.19 to find a cycle. At this point, state s is called "the seed". All states visited by $dfsRed_p$ are collected in $\mathcal{R}_p$. If no cycle is found in the red DFS, we can prove that none exists for the seed

Alg. 2 CNDFS, a new multi-core algorithm for LTL model checking

```
 1: procedure mcNdfs(s_0, P) is
 2:     dfsBlue_1(s_0) || ... || dfsBlue_P(s_0)
 3:     report no-cycle
 4: procedure dfsRed_p(s) is
 5:     R_p := R_p ∪ {s}
 6:     for all s' in shuffle_p^red(succ(s)) do
 7:         if s'.cyan[p] then
 8:             report cycle and terminate
 9:         if s' ∉ R_p ∧ ¬s'.red then
10:             dfsRed_p(s')
11: procedure dfsBlue_p(s) is
12:     s.cyan[p] := true
13:     for all s' in shuffle_p^blue(succ(s)) do
14:         if ¬s'.cyan[p] ∧ ¬s'.blue then
15:             dfsBlue_p(s')
16:     s.blue := true
17:     if s ∈ A then
18:         R_p := ∅
19:         dfsRed_p(s)
20:         await ∀s' ∈ R_p ∩ A : s ≠ s' ⇒ s'.red
21:         for all s' in R_p do s'.red := true
22:     s.cyan[p] := false
```

(Prop. 1). Still, because the red DFS was not necessarily called in postorder, other (non-seed, non-red) accepting states may be encountered for which we know nothing, except the fact that they are out of order and reachable from the seed. These are handled after completion of the red DFS at l.20 by simply waiting for them to become red.

Our proof shows that in this scenario there is always another worker which can colour such a state red (Prop. 3). The intuition behind this is that there has to be another worker to cause the out-of-order red search in the first place (by colouring blue) and, in the second place, this worker can continue its execution because cyclic waiting configurations can only happen for accepting cycles. These accepting cycles would however be encountered first, causing termination and a cycle report (l.8). After completion of the waiting procedure, CNDFS marks all states in $\mathcal{R}_p$ globally red, pruning other red DFSs.

The crude waiting strategy requires some justification. After reassessing the ingredients of LNDFS and ENDFS, we found that ENDFS is most effective at parallelising the blue DFS. This is absolutely necessary since the number of blue states (all reachable states) typically exceeds the number of red states (visited by the red DFS). In ENDFS, however, sharing the blue colour often led to the expensive (memory and performance wise) sequential repair procedure [10]. We were unable to construct a correct algorithm that colours both blue and red while backtracking from the respective DFS procedures. Therefore, we now want to investigate whether the intermediate solution, using a wait statement as a compromise, leaves enough parallelism to maintain scalability.

CNDFS only uses $N+2$ bits per state plus the sizes of $\mathcal{R}$. In the theoretical worst case (an accepting initial state), each worker p could collect all states in $\mathcal{R}_p$. In our vast set of experiments (cf. Sec. 4), however, we found that the set rarely contains more than one state and never more than thousands, which is still negligible compared to $|\mathcal{S}|$. Our experiments also confirmed that memory usage is close to the expected amount.

Correctness Proving correctness comprises two parts: proving the consistency of the algorithm, i.e. CNDFS reports a cycle *iff* an accepting cycle is reachable from s_0, and *termination*. The former turned out to be easier than for our previous parallel NDFS algorithms. The wait condition in combination with the late red colouring forces the accepting states to be processed in postorder. Stated differently: a worker makes the effects of its $dfsRed_p(s)$ globally visible (via the red colouring), only after all smaller (in postorder) accepting states t have been processed by some $dfsRed_{p'}(t)$. This is

expressed by Lemma 3. In Theorem 1, we finally show that, if the algorithm terminates without reporting a cycle, all accepting states must be red and consequently cannot lie on a cycle. Proof of termination was already discussed briefly and is detailed in Prop. 3.

In the following proofs, the graph colouring and the process counter of Alg. 2 are viewed as state properties of the execution. When writing $dfsBlue_p(s)@19$, we refer to the point in the execution at which a worker p is about to call *dfsRed* on a state s at l.19, within the execution of $dfsBlue_p(s)$. Graph colourings are denoted as follows: $s \in Red$ means that the *red* flag of s is set to true and similarly $s \in Blue$ means that the *blue* flag is set. For local flags we use $s \in Cyan_p$. Also, we use the modal operator $s \in \Box X$, to express $\forall s' \in succ(s): s' \in X$. We show that our propositions hold in the initial state $(\forall s \in \mathcal{S} : s \notin Red \wedge s \notin Blue \wedge \forall p \in \{1 \dots P\} : s \notin Cyan_p)$ and inductively that they are maintained by execution of each statement in the algorithm, considering only lines that can influence the truth value of the proposition. Here an important assumption is that all lines of Alg. 2 are executed atomically.

Lemma 1. *Red states have red successors: $Red \subseteq \Box Red$.*

Proof. Initially, there are no red states, hence the lemma holds.

States are coloured red when $dfsBlue_p@21$ and are never uncoloured red. The set of states $\mathcal{R}_p$ that is coloured at l.21 contains all states reachable from the seed s, but not yet red, since $dfsRed_p(s)$ performed a DFS from s over all non-red states. For the red states reachable from s, the induction hypothesis can be applied, hence there are no non-red states reachable from s that are not in $\mathcal{R}_p$. □

Lemma 2. *At l.20, the set $\mathcal{R}_p$ invariably contains (1) the seed s, (2) all non-red states reachable from s and also (3) all states in the set are reachable from the seed s: $dfsBlue_p(s)@20 \Rightarrow (s \in \mathcal{R}_p \wedge (\forall s' \notin Red : s \rightarrow^* s' \Rightarrow s' \in \mathcal{R}_p) \wedge (\forall s'' \in \mathcal{R}_p \Rightarrow s \rightarrow^* s''))$.*

Proof. At l.5, we have $s \in \mathcal{R}_p$. For the rest, see proof of Lemma 1. □

Lemma 3. *The only accepting state that can be coloured red at l.21 (for the first time) is the current seed s itself: $dfsBlue_p(s)@21 \Rightarrow (\mathcal{R}_p \cap \mathcal{A}) \setminus Red \subseteq \{s\}$.*

Proof. Assume $dfsBlue_p(s)@21$ and $\exists a \in (\mathcal{A} \setminus \{s\}) : a \in \mathcal{R}_p$. We show that $a \in Red$.

By Lemma 2, $\mathcal{R}_p$ contains at least s and the non-red states reachable from s. After l.20, all non-seed accepting states in $\mathcal{R}_p$ are red: $(\mathcal{R}_p \cap (\mathcal{A} \setminus \{s\})) \subseteq Red$. Since, $a \in \mathcal{R}_p \cap (\mathcal{A} \setminus \{s\})$, we have: $a \in Red$. □

Proposition 1. *The initial invocation of $dfsRed_p(s)$ at l.19 of Alg. 2 reports a cycle if and only if the seed s belongs to a cycle.*

Proof. $\Leftrightarrow$ is split into two cases: Case $\Rightarrow$: Every state $s' \in Cyan_p$ can reach the seed from $dfsBlue_p(s)@19$ by properties of the DFS stack. Similarly, when $dfsRed_p(s'')@8$, s'' is reachable from the seed s. Therefore, there is a cycle: $s'' \rightarrow s' \rightarrow^* s \rightarrow^* s''$.

Case $\Leftarrow$: assume $dfsRed_p(s)$ at l.19 finishes normally (without cycle report), while s lies on a cycle C. We show this leads to a contradiction. Since *dfsRed* avoids only red states (l.9), there would have to be some $r \in C \cap Red$ obstructing the search. The state r can only be coloured red at l.21 by a worker. W.l.o.g. we investigate the first worker $dfsRed_{p'}$ to have coloured r red. p' started for an $s' \in \mathcal{A}$ ($dfsBlue_{p'}(s')@19$).

Since r is not yet red, by Lemma 1 $C \cap Red = \emptyset$. Before r is coloured red, it is first stored in $\mathcal{R}_{p'}$. By Lemma 2, we also have $C \subseteq \mathcal{R}_{p'}$. Either $s' \in C$, then the cycle through s' would have been detected since $s' \in Cyan_{p'}$. Or else $s' \notin C$, and then we have $\{s\} \subseteq (\mathcal{R}_{p'} \setminus Red)$ when $dfsBlue_{p'}(s')$@21, contradicting Lemma 3. □

Proposition 2. *Red states never lie on an accepting cycle.*

Proof. Initially, there are no red states, hence the proposition holds.

When $dfsBlue_p(s)$@21, the set of states $\mathcal{R}_p$ is coloured red. The only accepting state to be colored red is the seed s (Lemma 3). By Prop. 1, this state s does not lie on an accepting cycle. Hence, Prop. 2 is preserved. □

Lemma 4. *Blue states have blue or cyan successors:* $Blue \subseteq \bigcup_p \Box(Blue \cup Cyan_p)$.

Proof. Initially there are no blue states, hence the lemma holds.

Only at l.16, states are coloured blue, after each successor t has been skipped at l.14 ($t \in Cyan \cup Blue$), or processed by $dfsBlue_p$ at l.15 (leading to $t \in Blue$). States can be uncoloured cyan (l.22), but only after they have been coloured blue (l.16). □

Lemma 5. *A blue accepting state, that is not also $Cyan_p$ for some worker p, must be red:* $\forall a \in (Blue \cap \mathcal{A}) : (\forall p \in \{1 \ldots P\} : a \notin Cyan_p) \Rightarrow a \in Red$.

Proof. Assume $s \in (\mathcal{A} \cap Blue)$ and $\forall p \in \{1 \ldots P\} : s \notin Cyan_p$. We show that $s \in Red$.

State s can only be coloured blue when $dfsBlue_p(s)$@16. There, it still retains its cyan colouring from l.12, it only loses this colour at l.22. But, since $s \in \mathcal{A}$, l.21 was reached and there $a \in \mathcal{R}_p$ by Lemma 2. Hence, $s \in Red$ at l.22. □

Proposition 3. *Algorithm 1 always terminates with a report.*

Proof. The individual DFSs cannot proceed indefinitely due to a growing set of red and blue states. So eventually a cycle (l.8) or no cycle is reported (l.3). However, progress may also halt due to the wait statement at l.20. We now assume towards a contradiction that a worker p is waiting indefinitely for a state $a \in \mathcal{A}$ to become red: $dfsBlue_p(s)$@20, $s \neq a$ and $a \in \mathcal{R}_p$. We will show that either a will be coloured red eventually, or a cycle would have been detected, contradicting the assumption that p keeps waiting.

By Lemma 2, a is reachable from s: $s \rightarrow^+ a$. And by l.16, $s \in Blue$. Induction on the path $s \rightarrow^* a$, using Lemma 4, tells us that: either all states are blue (1) or there is a cyan state on this path (2):

1. $a \in Blue \wedge \forall p \in \{1 \ldots P\} : a \notin Cyan_p$: by Lemma 5, $a \in Red$, which contradicts the assumption that p is waiting for a to become red.
2. $\exists c \in Cyan_{p'} : s \rightarrow^+ c \rightarrow^* a$, then depending on the identity of worker p', we have:
   A) $p = p'$: but then $dfsRed_p(s)$ would have terminated on cycle detection ($C \equiv s \rightarrow^+ c \rightarrow^+ s$), except when $dfsRed_p$ did not reach c in presence of a red state lying on C. However, this would contradict Prop. 2.
   B) $p \neq p'$: we show that either p' is executing or going to execute $dfsRed_{p'}(a)$. To eventually colour state a red, worker p' must not end up itself in a waiting state: $dfsBlue_{p'}(a')$@20. First, consider the case $a' \neq a$. We also have $a' \in \mathcal{R}_p$: If

$a' \in Red$, then by Prop. 2 all its reachable states are red and it cannot be waiting for a non-red reachable accepting state (Lemma 2). Therefore, $a' \notin Red$ and since also $s \rightarrow^+ c \rightarrow^* a'$ (stack $Cyan_p$), we have: $a' \in \mathcal{R}_p$ (Lemma 2). Therefore, we can assume w.l.o.g. that $a = a'$ and only consider $dfsBlue_{p'}(a)@20$. We can repeat the reasoning process of this proof, with $p \equiv p'$ and $s \equiv a$. But since there are finitely many workers, the chain of processes waiting for each other eventually terminates, except the hypothetical configuration of a cyclic waiting dependency, which we consider finally.

To exclude cyclic dependencies, assume $n \geq 2$ workers are simultaneously waiting for each other's seed to be coloured red at l.20. We have: $dfsBlue_1(s_1)@20 \wedge \ldots \wedge dfsBlue_n(s_n)@20 \wedge s_2 \in \mathcal{R}_1 \wedge \ldots \wedge s_1 \in \mathcal{R}_n$. This is only possible if $s_1 \rightarrow^+ s_n \wedge \ldots \wedge s_n \rightarrow^+ s_1$, hence there is a cycle: $s_1 \rightarrow^+ \ldots \rightarrow^+ s_n \rightarrow^+ s_1$. However, this contradicts that the red DFSs (which terminate anyway) would have detected this cycle (Prop. 1). □

Theorem 1. *Alg. 2 reports an accepting cycle if and only if one is reachable from* s_0.

Proof. By Prop. 3, the algorithm is guaranteed to terminate with some report, forming the basis for two cases: Case $\Rightarrow$: $dfsRed_p(s)@8$ implies a cycle (Prop. 1).

Case $\Leftarrow$: At l.3, we have $s_0 \in Blue$ and $Cyan = \emptyset$ by properties of DFS. Now, by Lemma 4, we have: $\forall s \in \mathcal{G} : s_0 \rightarrow^* s \Rightarrow s \in Blue$. Hence, all reachable accepting states must be red by Lemma 5 and do not lie on cycles by Prop. 2. □

4 Experimental Evaluation

Our previously reported experiments [15,14,13] were performed on 16-core machines. Meanwhile, in accordance with Moore's law applied to parallelism, we obtained access to a 48-core machine (a four-way AMD Opteron™ 6168). The added parallelism puts extra stress on the scalability of our algorithms and therefore also forces a repeat of some of our previous reachability experiments [15]. We investigated the cause for the performance difference between various algorithms: NMCNDFS [14], CNDFS (this paper), OWCTY+MAP [5] (the best representant of parallel BFS-based algorithms [13]) and reachability from [15]. Work duplication due to overlapping stacks can cause slowdowns for all multi-core NDFS variants, as can long await cycles in CNDFS. We introduced counters to measure and study these effects. Initially, we focus on models without cycles, the hardest case for these algorithms. Later we move on to show that CNDFS exhibits the same on-the-fly performance as existing multi-core NDFS variants [14].

We have used models from the BEEM database [18].[1] From each type of model, we selected the variants with more than 9 million states. Our CNDFS algorithm is implemented in the multi-core backend of the LTSMIN model checking tool set [16], based on a dedicated scalable lock-free hash table and an off-the-shelf load balancer [15]. For a fair comparison with previous algorithms, we also implemented some NDFS optimizations [13, Sec. 4.4], *all-red* and *early cycle detection*. All-red colours a state s red, if all its successors are red after l.15 of Alg. 2; correctness follows from Prop. 2. Early cycle detection detects certain accepting cycles already in the blue search.

[1] All results are available at `http://fmt.cs.utwente.nl/tools/ltsmin/atva-2012/`.

LTSMIN 1.9[2] was compiled with GCC 4.4.2 (with optimisation -O2) and ran with: `dve22lts-mc --threads=N -s28 --state=table --strategy=name`, where `name` can be `cndfs` or `endfs,lndfs`, representing the different algorithms [14]. We used DiVinE 2.5.2 [5] as OWCTY+MAP implementation, compiled and run with equivalent parameters. Since LTSMIN reuses its next-state function, both tools are comparable [15].

4.1 Models without Accepting Cycles

In [14], we showed that NMCNDFS was the best scaling LTL model checking algorithm on 16 core machines. Hence, we started comparing plain CNDFS and NMCNDFS. Table 1 shows the average runtime of both algorithms over five runs on all benchmarks, for 1, 8, 16 and 48 cores. The performance of CNDFS is on par with that of NMCNDFS, which is impressive considering the crude waiting strategy of the algorithm.

We confirmed that the time spent at the **await** statement (l.20 in Alg. 2) is indeed less than 0.01 sec on runs with 48 cores for all models in the BEEM database. This is caused by the all-red extension, which greatly reduces work in the red DFS. Without all-red, we observed high waiting times causing speeddowns with more than 8 cores.

Additionally, we made a comparison of *absolute speedups* so as to investigate the properties of the different algorithms (Fig. 1–6). For CNDFS and NMCNDFS, we included the standard deviation of the 5 runs as error bars. As the base case for the speedup of the LTL algorithms we used CNDFS: $S_n = T_1^{\text{CNDFS}}/T_n^{algo}$, for reachability we used its own base case. We included reachability from [15] to serve as a reference point for CNDFS. We were primarily interested to see whether the scalability of CNDFS keeps up with our parallel reachability implementation. After all, sequential NDFS visits each state at most twice; once in the blue DFS and possibly once in the red DFS.

Table 1. Runtimes (sec) with NMCNDFS and CNDFS for all models

	States	NMCNDFS				CNDFS			
		1	8	16	48	1	8	16	48
anderson.6.prop2	2.9E+7	144.0	46.5	31.3	23.7	146.6	47.2	31.7	23.6
anderson.6.prop4	3.6E+7	172.9	54.1	35.8	27.1	172.9	54.3	36.2	27.3
bakery.9.prop2	1.1E+8	378.9	62.4	35.5	18.9	368.9	64.6	36.9	19.9
bopdp.4.prop3	2.4E+7	74.7	11.1	6.4	3.3	74.9	11.0	6.4	3.3
elevator.5.prop3	2.1E+8	1,387.0	272.7	154.6	67.3	1,390.8	273.3	154.2	71.2
elevator2.3.prop4	1.5E+7	134.6	25.7	15.5	8.7	136.9	25.5	15.8	8.7
lamport.7.prop4	7.4E+7	299.2	61.9	35.5	23.5	297.7	60.8	35.9	22.9
leader_election.6.prop2	3.6E+7	1,495.2	189.5	194.5	31.9	1,501.9	190.1	94.5	32.2
leader_filters.6.prop2	2.1E+8	444.2	59.5	30.4	12.4	439.0	59.5	31.0	12.8
leader_filters.7.prop2	2.6E+7	73.5	9.7	6.4	2.3	73.3	9.4	5.0	2.3
lup.4.prop2	9.1E+6	19.6	4.7	2.9	2.2	19.5	4.7	2.9	2.1
mcs.5.prop4	1.2E+8	538.3	147.0	89.9	58.2	540.3	146.5	90.2	57.1
peterson.5.prop4	2.6E+8	1,186.0	229.4	135.3	84.9	1,146.5	226.2	133.0	83.6
rether.7.prop5	9.5E+6	43.0	6.2	3.8	2.7	43.6	6.3	3.9	2.6
synapse.7.prop3	1.5E+7	37.3	5.6	3.3	2.0	37.1	5.5	3.3	1.9

[2] `http://fmt.cs.utwente.nl/tools/ltsmin/` LTSmin version 2.0.

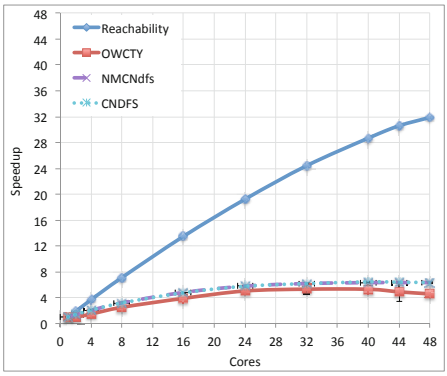

Fig. 1. Speedups of `anderson.6.prop4`

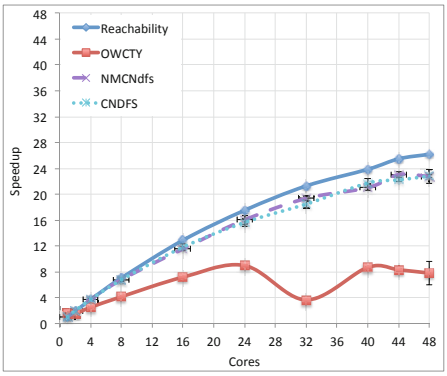

Fig. 2. Speedups of `bobp.4.prop3`

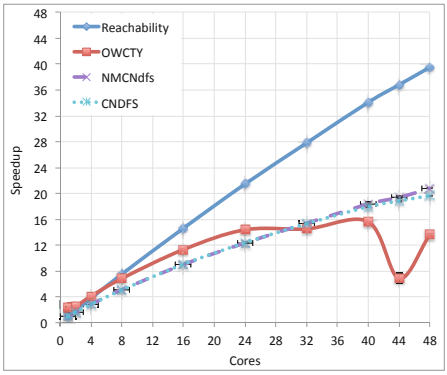

Fig. 3. Speedups of `elevator.5.prop3`

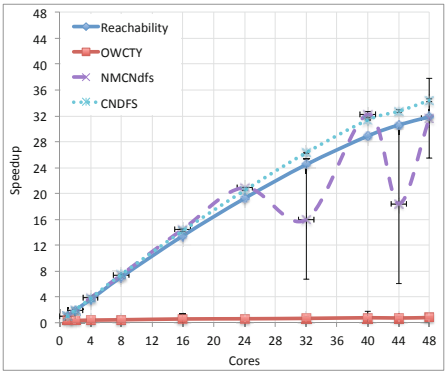

Fig. 4. Speedups of `leader_flt.6.prop2`

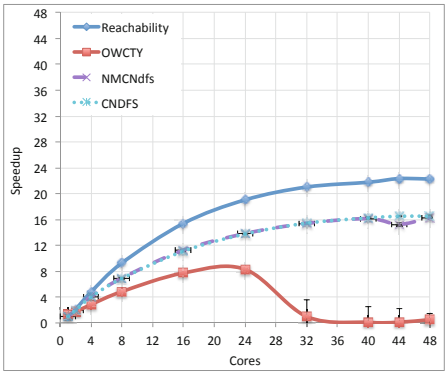

Fig. 5. Speedups of `rether.7.prop5`

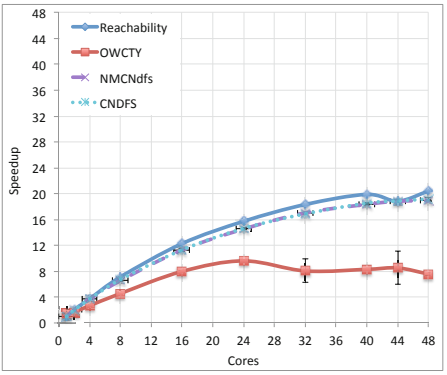

Fig. 6. Speedups of `synapse.7.prop3`

We notice that NMCNDFS and CNDFS are always faster than OWCTY+MAP. The error bars show less robust runtimes for NMCNDFS as they fluctuate greatly (e.g. `leader_filters`). Upon investigation it turned out that NMCNDFS sometimes launches a repair search even though we also fitted its ENDFS search with all-red. When only few workers enter this repair search, it cannot be parallelized. In these cases, CNDFS turns to waiting, a much better strategy, since in total it waits less than 0.01 sec. Also, reachability scales sometimes twice as good as CNDFS; `anderson` even scales 5 times better.

We investigated why the speedup of CNDFS differs from reachability. We measured the total amount of work performed by all workers. In particular, we counted for each benchmark the state count $|\mathcal{G}|$, and the numbers B_n and R_n, the total number of blue and red colourings in a run with n cores. Next, we estimate the duplicate work compared to reachability as $D_n := (R_n + B_n)/|\mathcal{G}|$. We view the reachability speedups S_n^{reach} as ideal (under the plausible assumption that maximal speedup is limited mostly by the memory bandwidth). Hence we can calculate the expected speedup $E_n^{alg} := S_n^{reach}/D_n^{alg}$ for $alg \in \{fsh, cndfs\}$ where *fsh* is CNDFS with heuristics (see below).

Table 2 compares these estimated speedups E_{48} with the actual speedups S_{48}. Note that the estimated speedups for CNDFS E_{48}^{cndfs} correspond nicely with the measured speedups S_{48}^{cndfs} for many benchmarks. Hence, we conclude that the variation in speedup is mainly caused by the degree of work duplication.

To combat work duplication, we reuse the "fresh successor heuristics" [14]. If possible, this randomly selects a successor that has not yet been visited before. It is available in the LTSMIN toolset (`--permutation=dynamic`). As a consequence, workers tend to be directed towards different regions of the state space, reducing work duplication.

These results are also shown in Table 2: D_{48}^{fsh}, E_{48}^{fsh} and S_{48}^{fsh} together with the measured amount of blue and red colourings: B_{48}^{fsh} and R_{48}^{fsh}. The heuristic approach shows quite some improvement, sometimes halving work duplication and doubling speedup

Table 2. Expected and actual speedups for CNDFS according to speedup model

	$\|\mathcal{G}\|$	B_{48}^{fsh}	R_{48}^{fsh}	S_{48}^{reach}	D_{48}^{fsh}	E_{48}^{fsh}	S_{48}^{fsh}	D_{48}^{cndfs}	E_{48}^{cndfs}	S_{48}^{cndfs}
anderson.6.prop2	3E+7	1E+8	4E+3	30.6	3.6	8.6	6.4	4.7	6.6	4.6
anderson.6.prop4	4E+7	1E+8	3E+3	31.9	3.1	10.2	6.4	4.0	8.0	5.0
bakery.9.prop2	1E+8	2E+8	4E+5	28.0	1.4	20.5	19.2	1.6	17.2	14.3
bopdp.4.prop3	2E+7	3E+7	6E+5	26.2	1.3	20.0	22.8	1.8	14.6	15.5
elevator.5.prop3	2E+8	4E+8	2E+3	39.5	1.9	21.0	19.5	3.2	12.5	9.0
elevator2.3.prop4	1E+7	3E+7	2E+6	33.2	2.0	16.3	15.8	5.3	6.3	8.0
lamport.7.prop4	7E+7	1E+8	6E+4	30.5	1.7	17.6	13.3	1.9	15.8	10.4
leader_el.6.prop2	4E+7	4E+7	4E+4	40.5	1.0	40.4	46.6	1.0	40.3	39.5
leader_filt.6.prop2	2E+8	2E+8	7E+5	31.9	1.0	31.6	34.4	1.0	30.7	29.9
leader_filt.7.prop2	3E+7	3E+7	1E+5	27.6	1.0	27.4	31.9	1.0	26.9	27.8
lup.4.prop2	9E+6	2E+7	4E+3	17.7	2.5	7.1	9.7	4.6	3.8	6.3
mcs.5.prop4	1E+8	3E+8	1E+4	34.4	2.2	15.7	9.5	2.7	12.6	7.3
peterson.5.prop4	3E+8	4E+8	8E+5	34.1	1.6	20.9	13.9	1.9	18.3	11.0
rether.7.prop5	1E+7	2E+7	1E+5	22.3	1.9	11.9	16.5	2.4	9.2	14.3
synapse.7.prop3	2E+7	2E+7	1E+2	20.4	1.1	17.9	19.2	1.2	17.0	18.6

(see `elevator`). Still we see duplications as high as 3.6 (see `anderson`). Note that the earlier speedups in Fig. 1–6 already include the benchmarks with this heuristic.

We expect that in the near future, the number of cores in many-core systems will still grow. Will this increase work duplication and put a limit on speedup of CNDFS? To give an indication, we plotted the increase of work duplication with a growing number of cores with fresh successor heuristics (Fig. 7). The increase is sub-linear, so we expect that speedups will be maintained on larger many-core systems with similar architecture and scaling bandwidth characteristics.

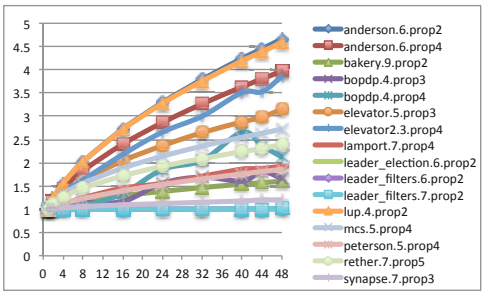

Fig. 7. Work duplication per core per model

Finally, we note that the size of the input has a small yet significant effect on the amount of work duplication; models with higher state count have less duplication.

4.2 Models with Accepting Cycles

In [14], we experimented thoroughly to investigate the "on-the-flyness" of SWARM NDFS and LNDFS. We noticed that the benefits of independent SWARM verification is limited, on average only yielding a speedup of 2-8 on 16 core machines. LNDFS however yielded speedups from 4 to 14. Combined with the fresh successor heuristic

Table 3. On-the-fly behavior of parallel LTL algorithms

		1 core	48 core				OWCTY+MAP	
		NDFS	LNDFS		CNDFS		1 core	48 core
	model	Rand.	Rand.	Fsh	Rand.	Fsh.	Static	Rand.
Runtimes (sec)	anderson.8.prop3	36.4	4.0	1.2	4.1	0.2	2858.8	1433.2
	bakery.7.prop3	3.2	0.4	0.2	0.3	0.2	2.2	5.2
	bakery.8.prop4	15.7	0.6	0.3	0.6	0.3	73.4	14.3
	elevator2.3.prop3	8.4	1.4	0.2	1.4	0.2	432.3	192.5
	extinction.4.prop2	2.2	0.1	0.1	0.1	0.1	1.8	1.7
	peterson.6.prop4	29.1	0.6	0.5	0.9	0.5	668.4	705.7
	szymanski.5.prop4	1.7	1.4	0.1	1.3	0.2	2.1	376.4
Speedups	anderson.8.prop3		9.1	31.1	8.8	175.0		2.0
	bakery.7.prop3		8.7	18.3	10.9	21.2		0.4
	bakery.8.prop4		28.3	51.1	26.2	48.9		5.1
	elevator2.3.prop3		6.0	51.5	5.9	52.1		2.2
	extinction.4.prop2		30.4	32.1	18.5	28.8		1.0
	peterson.6.prop4		46.1	59.8	33.0	62.4		0.9
	szymanski.5.prop4		1.2	12.0	1.3	10.9		0.0

speedups became often superlinear. This is not surprising [19], because we verified that in those cases there are many cycles, distributed evenly over the state space.

We performed the same experiments again with CNDFS on a 48 core machine. The results in Table 3 show that CNDFS exhibits the same desirable on-the-fly behaviour as LNDFS, scaling up to 48 cores. We conclude that our new multi-core CNDFS algorithm scales well also for models with bugs.

For completeness, we also included the runtimes and speedups with OWCTY+MAP in the table. While the heuristic on-the-fly behavior seems to work well for some models, for others it does not. It must however be mentioned that the on-the-fly capabilities of this algorithm have recently been improved by changing its exploration order to be more DFS-like [1]. In [1], performance is reported on par with the LNDFS algorithm. Unfortunately, we do not have the means (a GPGPU) to reproduce any results here.

4.3 Counterexample Length

Lengthy counterexamples are hard to study even with good model checking tools. Therefore, finding short counterexamples is quite an important property of model checking algorithms. Strict BFS algorithms deliver minimal counterexamples, while DFS algorithms can yield very long ones. Once the strict BFS/DFS order is loosened, these properties can be expected to fade. This is exactly what both OWCTY+MAP and CNDFS do. We studied the length of the counterexamples that these algorithms produce.

For this purpose, 45 models with counterexamples were selected from the BEEM database, all algorithms run 5 times, and computed the average counterexample length and standard deviation. The results are summarised in scatter plots with bars representing the standard deviation. Fig. 8 compares randomised sequential NDFS (vertical axis) against sequential OWCTY+MAP (horizontal axis). Fig. 9 compares the results of CNDFS with fresh successor heuristic (fsh) against OWCTY+MAP on 48 cores.

In the sequential case, most bars are above the equilibrium so, as expected, NDFS produces long counterexamples of variable size compared to OWCTY+MAP (which we could not randomise). The parallelism of a 48 core run, however, greatly stabilises and

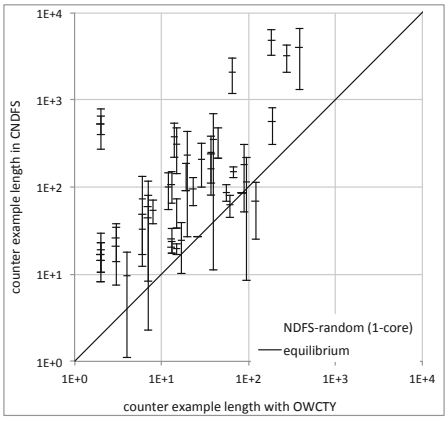

Fig. 8. NDFS vs OWCTY+MAP (1 core)

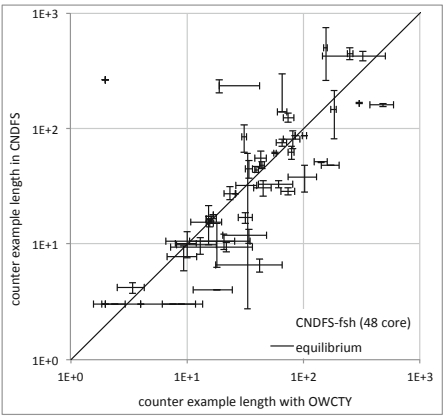

Fig. 9. Fsh vs OWCTY+MAP (48 cores)

reduces counterexample lengths for CNDFS, while the randomness added by parallelism introduces variable results for OWCTY+MAP (horizontal bars). In many cases, CNDFS counterexamples become shorter than those of OWCTY+MAP, a surprising result considering the BFS-like order of this algorithm. The one extreme outlier in this case is the `plc.4` model. All our NDFS algorithms consistently find a counterexample of length 216, while OWCTY+MAP finds one of length 2!

5 Conclusion

We presented CNDFS, a new multi-core NDFS algorithm. It can detect accepting cycles on-the-fly, and its worst case execution time is linear in the size of the input graph. We showed that CNDFS is considerably simpler than its predecessor NMCNDFS, because of the deep integration of ENDFS and LNDFS. Experiments show that CNDFS delivers performance and scalability similar to its predecessors, but achieves this more robustly. Hence CNDFS is currently the fastest multi-core LTL model checking algorithm in practice. Moreover, CNDFS halves the memory requirements per state per worker thread; an important factor since the total number of cores keeps growing.

Experiments revealed that the main bottleneck for perfect scalability of CNDFS is currently the work duplication due to overlapping stacks. Forcing workers to favour "fresh" successor states already decreases duplication. The same experiments indicate that work duplication grows only linearly in the number of cores, and decreases for larger input sizes. From this we conjecture that CNDFS will scale even beyond 48 cores.

CNDFS shares global information only during or even after backtracking, which leads to potential work duplication. In the worst case, every worker could visit the whole graph, blocking any speedup. During our extensive experiments with the entire BEEM database we have not found such cases. However, we did observe work duplication of factor 3 on 48 cores, so there is room for improvement.

Designing a provably scalable, linear-time algorithm remains an open question. Such an algorithm should cause negligible duplicate work and avoid synchronisation by await statements. So far, we have not been able to come up with a correct algorithm without await statements or a repair procedure. An improvement might be to invent a smart work stealing scheme, in which workers can cooperate instead of waiting.

Finally, we demonstrated that counterexamples in CNDFS become shorter with more parallelism, even shorter than counterexamples in parallel BFS-based OWCTY+MAP. This is an interesting and desirable property for a model checking algorithm. It is intriguing that our parallel DFS based algorithm shows good scalability and short counterexamples, usually attributed to BFS algorithms, while still maintaining the linear-time and on-the-fly properties expected from a DFS algorithm.

References

1. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. Journal of Parallel and Distributed Computing (2011)
2. Barnat, J., Brim, L., Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking. In: ASE 2003, pp. 106–115. IEEE Computer Society (2003)

3. Barnat, J., Brim, L., Ročkai, P.: A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 407–425. Springer, Heidelberg (2009)
4. Barnat, J., Brim, L., Ročkai, P.: Scalable shared memory LTL model checking. STTT 12(2), 139–153 (2010)
5. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: PDMC 2010, pp. 4–7. IEEE (2010)
6. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
7. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection (Set Based Approach). In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
8. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
9. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
10. Evangelista, S., Petrucci, L., Youcef, S.: Parallel Nested Depth-First Searches for LTL Model Checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 381–396. Springer, Heidelberg (2011)
11. Geldenhuys, J., Valmari, A.: Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)
12. Groce, A., Holzmann, G.J., Joshi, R.: Swarm Verification Techniques. Transactions on Software Engineering 37(6), 845–857 (2011)
13. Laarman, A.W., Langerak, R., van de Pol, J.C., Weber, M., Wijs, A.: Multi-Core Nested Depth-First Search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
14. Laarman, A.W., van de Pol, J.C.: Variations on multi-core nested depth-first search. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 13–28 (2011)
15. Laarman, A.W., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Sharygina, N., Bloem, R. (eds.) FMCAD 2010, Lugano, Switzerland, USA. IEEE Computer Society (October 2010)
16. Laarman, A., van de Pol, J., Weber, M.: Multi-core lTSMIN: Marrying modularity and scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
17. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011)
18. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
19. Rao, V.N., Kumar, V.: Superlinear Speedup in Parallel State-space Search. In: Kumar, S., Nori, K.V. (eds.) FSTTCS 1988. LNCS, vol. 338, pp. 161–174. Springer, Heidelberg (1988)
20. Reif, J.H.: Depth-first Search is Inherently Sequential. Information Processing Letters 20(5), 229–234 (1985)
21. Schwoon, S., Esparza, J.: A Note on On-the-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
22. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS 1986, pp. 332–344. IEEE Computer Society (1986)