Multi-core SCC-Based LTL Model Checking

Vincent Bloemen(✉) and Jaco van de Pol

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{v.bloemen,j.c.vandepol}@utwente.nl

Abstract. We investigate and improve the scalability of multi-core LTL model checking. Our algorithm, based on parallel DFS-like SCC decomposition, is able to efficiently decompose large SCCs on-the-fly, which is a difficult problem to solve in parallel.

To validate the algorithm we performed experiments on a 64-core machine. We used an extensive set of well-known benchmark collections obtained from the BEEM database and the Model Checking Contest. We show that the algorithm is competitive with the current state-of-the-art model checking algorithms. For larger models we observe that our algorithm outperforms the competitors. We investigate how graph characteristics relate to and pose limitations on the achieved speedups.

1 Introduction

The automata theoretic approach to LTL model checking involves taking the synchronized product of the negated property and the state space of the system. The resulting product is checked for emptiness by searching for an *accepting cycle*, i.e. a reachable cycle that satisfies the accepting condition [35]. If an accepting cycle is found the system is able to perform behavior that is not allowed by the original property, hence we say that a counterexample has been found.

In order to fully utilize modern hardware systems, the design of parallel algorithms has become an urgent issue. Model checking is a particularly demanding task (in both memory and time), which makes it a well-suited candidate for parallelization. *On-the-fly* model checking makes it possible to find a counterexample while only having to search through part of the state space. However, the on-the-fly restriction makes it especially difficult to design a correct and efficient parallel algorithm. In practice, this causes the algorithms to rely on *depth-first search (DFS)* exploration [32].

General Idea of the Algorithm. We present a multi-core solution for finding accepting cycles on-the-fly. It improves recent work [30] by communicating partially found *strongly connected components (SCCs)* [2]. The general idea of our algorithm is best explained using the example from Fig. 1. In Fig. 1a, two threads (or workers), which we call 'red' and 'blue', start their search from the initial state, `a`. Here, state `f` is an accepting state, and the goal is to find a reachable cycle that contains an accepting state. We assume that the workers have no prior

R. Bloem and E. Arbel (Eds.): HVC 2016, LNCS 10028, pp. 18–33, 2016.
DOI: 10.1007/978-3-319-49052-6_2

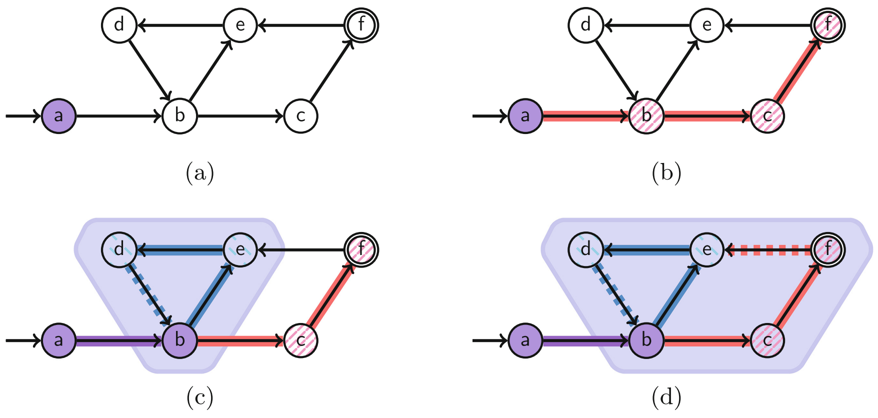


Fig. 1. Example where two workers cooperate to find a counterexample. (Color figure online)

knowledge of any other state (on-the-fly). Using a successor function `suc()`, the successor states of `a` can be computed (thus, `suc(a)={b}`).

In Fig. 1b we observe a situation where the red worker has explored the path $\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{c} \rightarrow \texttt{f}$. Suppose that the red worker halts and the blue worker starts exploring the path $\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{e} \rightarrow \texttt{d}$ and observes `suc(d)={b}` (Fig. 1c). The blue worker then finds a cycle that contains the states {`b,d,e`} and stores this information globally.

Now consider what happens in Fig. 1d. Here, the red worker continues its search and explores the edge $\texttt{f} \rightarrow \texttt{e}$. Since the states {`b,d,e`} are part of a known cycle, and the red worker has explored the path $\texttt{b} \rightarrow \texttt{c} \rightarrow \texttt{f} \rightarrow \texttt{e}$, we can thus implicitly assume that states `c` and `f` are part of the same SCC and form {`b,c,d,e,f`}. Remarkably, the algorithm can detect an accepting cycle while neither the red nor the blue worker explored the cycle $\texttt{f} \rightarrow \texttt{e} \rightarrow \texttt{d} \rightarrow \texttt{b} \rightarrow \texttt{c} \rightarrow \texttt{f}$.

We make the following contributions in this paper.

- We provide an SCC-based on-the-fly LTL model checking algorithm by extending on previous work [2] and the work from Renault et al. [30].
- We empirically compare our algorithm with state-of-the-art algorithms (all implemented in the same toolset), using an extensive set of well-known benchmark models. We show that our algorithm is competitive and even outperforms the competitors for larger models.
- We observe and discuss relations between the algorithms and scalability for models containing large SCCs.

In order to carry out the necessary experiments, we have extended the LTSmin toolset [17] to connect with the Spot v2.0 library [7] for generating Büchi Automata.

Overview. The remainder of the paper is structured as follows. In Sect. 2 we provide preliminaries on model checking. Section 3 discusses related work on parallel model checking. We present our algorithm in Sect. 4. The experiments are discussed in Sect. 5 and we conclude our work in Sect. 6.

2 Preliminaries

Directed Graph. A directed graph is a tuple $G := \langle V, E \rangle$, where V is a finite set of states, and $E \subseteq V \times V$ is a set of transitions. We denote a transition (or edge) $\langle v, w \rangle \in E$ by $v \to w$. A *path* $v_0 \to^* v_n$ is a sequence of states $v_0, \ldots, v_n \in V^*$ s.t. $\forall_{0 \leq i < n} : v_i \to v_{i+1}$, $v_0 \to^+ v_n$ denotes a path that contains at least one transition. A *cycle* is a non-empty path where $v_0 = v_n$. We say that two states v and w are *strongly connected* iff $v \to^* w$ and $w \to^* v$, written as $v \leftrightarrow w$. A *strongly connected component (SCC)* is defined as a maximal set $C \subseteq V$ s.t. $\forall v, w \in C : v \leftrightarrow w$. We call an SCC C *trivial* if $\exists v \in V : C = \{v\}$ and $v \not\to v$. We call C a *partial SCC* if all states in C are strongly connected, but C is not necessarily maximal.

Automaton Graph. The synchronized product of the negated LTL property and the state space of the system is usually represented with an *automaton graph*. There are different ways to represent an automaton graph. In practice, two common ways to describe an automaton graph is by using a *Büchi Automaton (BA)* or a *Transition-based Generalized Büchi Automaton (TGBA)*.

Definition 1 (BA). *A* BA *is a tuple* $\mathcal{B} := \langle V, E, A, v_0 \rangle$*, where* V *is a finite set of states,* $E \subseteq V \times V$ *is a set of transitions,* $A \subseteq V$ *is the set of accepting states, and* $v_0 \in V$ *is the initial state. An accepting cycle* C *on* $\mathcal{B}$ *is defined as a cycle, reachable from* v_0*, where* $C \cap A \neq \emptyset$*.*

Definition 2 (TGBA). *A* TGBA *is a tuple* $\mathcal{K} := \langle V, \delta, F, v_0 \rangle$*, where* V *is a finite set of states,* $\delta \subseteq V \times 2^F \times V$ *is a set of transitions where each transition is labeled by a subset of acceptance marks,* F *is a finite set of acceptance marks, and* $v_0 \in V$ *is the initial state. An accepting cycle* C *on* $\mathcal{K}$ *is defined as a cycle* $\langle w_0, a_0, w_1 \rangle, \ldots, \langle w_n, a_n, w_0 \rangle$*, reachable from* v_0*, where* $a_0 \cup \ldots \cup a_n = F$*. Any BA can be represented with a TGBA using the same number or fewer states and transitions* [13]*.*

For the remainder of the paper, unless stated otherwise, we consider the automaton graph to be represented as a *BA*. We make the assumption that an automaton graph is computed *on-the-fly*. This implies that an algorithm initially only has access to the initial state v_0, and can compute successor states: $\texttt{suc}(v) := \{w \in V \mid v \to w\}$.

3 Related Work

Sequential algorithms for explicit-state on-the-fly LTL model checking can be distinguished in two classes, *Nested DFS (NDFS)* and *SCC-based* algorithms.

For an excellent overview on *sequential* NDFS and SCC-based algorithms, we would like to refer the reader to the work of Schwoon and Esparza [32]. Both NDFS- and SCC-based algorithms can perform in linear time complexity on the number of edges in the graph.

NDFS Based Algorithms. NDFS, originally proposed by Courcoubetis et al. [3], performs two interleaved searches. An outer DFS to find accepting states and an inner DFS that checks for cycles around accepting states. Since its inception, several improvements have been made [11,15,32].

Multi-core NDFS. A number of multi-core variants on NDFS have been designed that scale on parallel hardware in practice [8,9,20,21]. These algorithms are based on *swarm verification* [14]. The idea is that all workers initially start from the initial state, but the list of successor states is permuted for each worker. This way, distinct workers will explore different parts of the graph with a high probability.

We consider CNDFS [8] to be the state-of-the-art NDFS-based algorithm. Independent NDFS-like instances are launched and global information is shared between the workers during (and after) the backtrack procedure. The algorithm performs in linear time.

We note that NDFS-based algorithms are explicitly based on using BA acceptance. To the best of our knowledge, no parallel NDFS-like algorithm exists for checking TGBAs.

SCC-Based Algorithms. SCC-based model checking consists of finding SCCs and detecting if the accepting criteria is met in one of these components. Tarjan's algorithm [34] is generally favored for the SCC detection procedure due to its linear time complexity and ability to perform on-the-fly. Couvreur [4], and Geldenhuys and Valmari [12] proposed modifications to more quickly recognize accepting cycles. Notably, an SCC-based model checking algorithm can be used to check for emptiness on *generalized* Büchi automata [5,32].

Multi-core SCC-Based Algorithms. There are a number of parallel algorithms that can detect SCCs in an explicitly given graph, e.g. [10,16,25,31,33]. However, none of these are applicable in the on-the-fly context since they require knowledge about a state's predecessors and/or depend on random access to the state space. There has been a lot of recent activity in finding SCCs on-the-fly. This pursuit has resulted in three new algorithms [2,23,29,30].

The algorithm by Lowe [23] is based on spawning multiple *synchronized* instances of Tarjan's algorithm. Here, each state may only be visited by one worker and a work-stealing-like procedure is used to handle conflicts that arise. Experimental evaluation shows that Lowe's algorithm performs well if the graph contains many small SCCs, but this seems to deteriorate quickly when the SCC sizes grow. Lowe's algorithm has a quadratic worst-case complexity.

The algorithm by Renault et al. [29,30] is also based on spawning multiple instances of Tarjan's (and/or Dijkstra's [6]) algorithm, but here a state may be visited by multiple workers. The approach is based on swarmed verification, where individual searches globally communicate fully explored SCCs – which are avoided by other workers from then on. To improve LTL checking, acceptance conditions are globally updated per partial SCC whenever a worker detects a cycle, making it possible to find a counterexample in a similar fashion as we present in Fig. 1. We consider this algorithm the current state-of-the-art of SCC-based parallel on-the-fly model checking and it performs in quasi-linear time complexity.

In this paper, we applied the UFSCC algorithm by Bloemen et al. [2] for LTL model checking. We discuss the algorithm extensively in Sect. 4 and show how this improves the scalability for graphs containing large SCCs.

4 Multi-core SCC Algorithm for LTL Model Checking

The main idea of the UFSCC algorithm is that it globally communicates *partially found SCCs* while maintaining a quasi-linear time complexity. This means that when a worker has locally found a cycle, it merges all states on that cycle in a global structure (implemented with concurrent union-find). In order to make efficient use of this information, the structure also tracks which workers have visited the partial SCCs to support a more lenient form of detecting back-edges. The structure also tracks which states have been fully explored and allows workers to concurrently select states that still require exploration.

A collection of disjoint sets is used for globally tracking partially discovered SCCs. This collection, $\boldsymbol{\pi} : V \rightarrow 2^V$, satisfies the following invariant: $\forall v, w \in V : w \in \boldsymbol{\pi}(v) \Leftrightarrow \boldsymbol{\pi}(v) = \boldsymbol{\pi}(w)$. In other words, the set for a specific state can be obtained from any member of the set. This also implies that every state must belong to exactly one set. A `Unite` function is used to combine two disjoint sets, while maintaining the invariant. As an example, let $\boldsymbol{\pi}(v) := \{v\}$ and $\boldsymbol{\pi}(w) := \{w, x\}$ (note $\boldsymbol{\pi}(w) = \boldsymbol{\pi}(x)$) , then $\texttt{Unite}(\boldsymbol{\pi}, v, w)$ combines $\boldsymbol{\pi}(v)$ and $\boldsymbol{\pi}(w)$, resulting in $\boldsymbol{\pi}(v) = \boldsymbol{\pi}(w) = \boldsymbol{\pi}(x) = \{v, w, x\}$ while not modifying any other mappings. These properties follow directly from an implementation with union-find.

The Algorithm. The algorithm can be found in Fig. 2. We assume that every line is executed atomically[1]. Each worker p has its own local search stack, $\mathtt{R_p}$. The global collection $\boldsymbol{\pi}$ and global sets `Dead` and `Done` are initialized in Lines 2–3. `Dead` implies that an *SCC* is fully explored and `Done` implies that a *state* is fully explored. Every worker starts exploring from the initial state $\mathtt{v_0}$. Disregarding Line 7 for the moment, Lines 8–15 describe the procedure to fully explore a state. For every successor w of v' there are three cases to be distinguished:

1. $\mathtt{w} \in \mathtt{Dead}$ (Line 9): State w is part of an already completed SCC, it may be ignored since no new information can be obtained.

[1] In practice this is not exactly true, however all necessary conditions are preserved by using a fine-grained locking structure.

```
1  $\forall$ p $\in$ [1...P] : $R_p$ := $\emptyset$
2  $\forall$ v : $\pi$(v) := {v}
3  Dead := Done := $\emptyset$
4  $UFSCC_1(v_0)$ || ... || $UFSCC_P(v_0)$
5  procedure $UFSCC_p$(v)
6    $R_p$.push(v)
7    while v' := $\pi$(v) \ Done do
8      for each w $\in$ Random(suc(v')) do
9        if w $\in$ Dead then continue
10       else if $\nexists$w'$\in$ $R_p$ : w $\in$ $\pi$(w') then $UFSCC_p$(w)
11       else while $\pi$(v) $\neq$ $\pi$(w) do
12             r := $R_p$.pop()
13             s := $R_p$.top()
14             Unite($\pi$, r, s)
15           if $\pi$(v) $\cap$ Acc $\neq \emptyset$ then report CE
16     Done := Done $\cup$ {v'}
17   if $\pi$(v) $\not\subseteq$ Dead then Dead := Dead $\cup$ $\pi$(v)
18   if v = $R_p$.top() then $R_p$.pop()
```

Fig. 2. Multi-core UFSCC algorithm for LTL model checking of BAs.

2. `w` $\notin$ `Dead` $\wedge$ $\nexists$`w'` $\in$ R_p`:` `w` $\in$ $\boldsymbol{\pi}$`(w')` (Line 10): State w is not part of the local search stack and it is also not in a partial SCC that contains a state from the local search stack (assuming that $\boldsymbol{\pi}$ correctly tracks partially found SCCs). Since the current worker has not visited w before, nor any state in $\boldsymbol{\pi}(w)$, it regards w as an undiscovered state and recursively explores it.
3. `w` $\notin$ `Dead` $\wedge$ $\exists$w'$\in$ R_p `:` `w` $\in$ $\boldsymbol{\pi}$`(w')` (Lines 11–15): Here, the worker's stack does contain a state w' that is in the same partial SCC as w (the stack may also contain w itself). This forms a cycle and thus all states on said cycle are united. We assume that partial SCCs adhere to the strong connectivity property and that the search stack sufficiently maintains a DFS order.[2]

In case a cycle is detected, Line 15 checks whether the partial SCC contains an accepting state. If this is true, we can be sure that an accepting cycle is found.

We now discuss lines 7 and 16. We maintain a mechanism to globally mark states as being fully explored. A state v is fully explored if all its outgoing transitions direct to states in already completed SCCs (the successor is part of the `Dead` set) or to states in the same partial SCC, $\boldsymbol{\pi}(v)$. In both these cases, no new information can be obtained from the successors. At Line 16, state `v'` is fully explored and is marked as such. Fully explored states are included in the `Done` set and are disregarded for exploration. Line 7 picks a state out of $\boldsymbol{\pi}(v)$ for exploration that is not fully explored. This is possible since all states in $\boldsymbol{\pi}(v)$ are strongly connected, thus no condition is violated. In case every state in $\boldsymbol{\pi}(v)$

[2] With sufficiently maintaining a DFS order we mean that for any two successive states v and w on the local search stack, we have $v \rightarrow^{+} w$; i.e. we do not require a direct edge from v to w, but w must be reachable from v.

is marked `Done`, the while loop ends and we conclude that the entire (partial) SCC has been fully explored and can be marked as complete. This is achieved by merging $\boldsymbol{\pi}(v)$ with the `Dead` set at Line 17.

For a proof of correctness, we refer the reader to Bloemen et al. [2].

Global Data Structure. The underlying global data structure satisfies the following conditions:

– Provide means to check if the current worker has previously visited a state inside the partial SCC (successor cases 2 and 3).
– Provide means to globally mark states as being fully explored, and a selection mechanism for states in a partial SCC that have not yet been fully explored.

The union-find structure uses parent pointers that direct to the representative or *root* of the set. The structure is extended to track worker IDs in the partial SCCs. The worker ID is added to the root of the set when the worker enters a (locally) new state. The set of worker IDs is updated during the `Unite` operations. This ensures that if a worker ID is set for a particular state, it remains being set if the partial SCC containing that state gets updated.

In order to mark states as being fully explored (inside a partial SCC), the structure is extended with a cyclic list which is depicted in Fig. 3 for a partial SCC. It tracks states that have not yet been marked `Done`, which we call `Busy` states, (depicted white) and removes `Done` (depicted gray) states from this list. Workers can then use the list pointers to find remaining `Busy` states of the partial SCC. At a certain point in time, the list becomes empty, i.e. every state in the partial SCC is marked `Done`. We thus conclude that all states in the partial SCC have been fully explored, which implies that the entire SCC has been fully explored and can be marked as being complete.

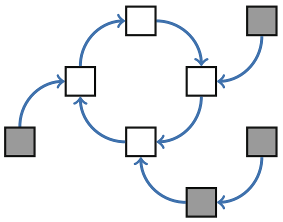


Fig. 3. Cyclic list structure.

Figure 4 depicts a consequence of using the abovementioned cyclic list for selecting states to fully explore. A worker starts from state a with the edge $a \to c$. It could then detect that state c is already marked `Done` and the worker picks a new state, d. From state d, the worker resumes its exploration. State e might also be marked `Done`, and the worker continues searching from state h.

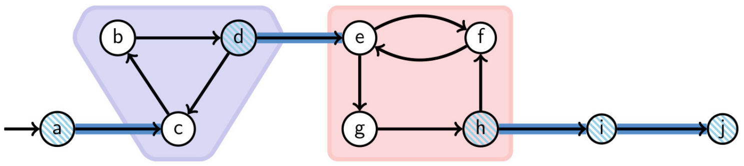


Fig. 4. Example showing a possible state traversal for one worker.

Note that this search order maintains the depth-first search order sufficiently for detecting cycles (See foot note 2).

For more details regarding the implementation of the algorithm and global data structure we refer the reader to Bloemen et al. [2].

Finding Accepting Cycles on TGBAs. The algorithm is extended to track acceptance marks in each SCC. A counterexample is then found when every acceptance mark is present in an SCC. In the algorithm from Fig. 2 this implies storing a set of acceptance marks alongside the partial SCCs and update this set with every `Unite` operation. In the implementation this is achieved by ensuring that the root of the union-find structure contains the most up-to-date acceptance set. If the acceptance set of the root contains all acceptance marks, a counterexample has been detected.

In summary, for each node we maintain a pointer towards the union-find root, and pointers to the successors in the list. For the root nodes we maitain a set of bits for the involved workers, and in case TGBAs are used also a set of bits for the acceptance marks that have been found.

5 Experiments

Experimental Setup. All experiments were performed on a machine with 4 AMD Opteron™ 6376 processors, each with 16 cores, forming a total of 64 cores. There is a total of 512 GB memory available.

Implementation. The extended UFSCC algorithm is implemented in the LTSmin toolset [17]. We furthermore extended LTSmin to use the Spot v2.0 library [7] for generating Büchi automata (both BA and TGBA) from LTL formulas.

We compare the UFSCC algorithm (implemented for BA and TGBA acceptance) with (sequential) NDFS [3], CNDFS [8] and the SCC-based algorithm by Renault et al. [29] which we further refer to as Renault. We attempt to minimize performance differences caused by effects other than those resulting from the algorithmic differences, hence each algorithm is implemented in the LTSmin toolset. All multi-core algorithms make use of LTSmin's internal shared hash tables [22], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to 2^{28} states.

Models and Formulas. We used models and LTL formulas from three existing benchmark sets and describe these as follows.

- `BEEM-orig`[3]: This consists of the complete collection of original (DVE) models and formulas from the BEEM database [26]. Additionally, a number of realistic formulas were added for several parameterized models (see Blahoudek et al. [1] for details), forming a total of over 807 formulas.

[3] Available at http://fi.muni.cz/~xstrejc/publications/spin2014.tar.gz.

- `BEEM-gen`[4]: These are the same models and LTL formulas as used by Renault et al. [29,30]. The (DVE) models are a subset of the BEEM database [26] such that every type of model from the classification of Pelánek [27] is represented. A total of 3,268 randomly generated formulas were selected such that the number of states, transitions and number of SCCs were high in the synchronized cross products.
- `MCC`[5]: We used a selection of the 2015 Model Checking Contest problems [19]. This consists of Petri net instances (specified in PNML) of both academic and industrial models, forming a total of 928 models. For each model, 48 different LTL formulas were provided that check for fireability (propositions on firing a transition) and cardinality (comparing the number of tokens in places), forming a total of $928 \times 48 = 44,544$ experiments. We performed an initial selection using UFSCC with 64 cores and selected instances taking between one second and one minute to check.[6] This resulted in 1,107 experiments.

We combined all datasets for the experiments, totaling 5,128 experiments and 2,950 contain a counterexample. Each configuration was performed at least 5 times and we computed all results by using the averages. The algorithms were not always able to successfully perform an experiment in the maximum allowed time of 10 min. When comparing two configurations, we only consider experiments where both algorithms performed successfully and within the time limit.

All results and means to reproduce the results are publicly available online at https://github.com/utwente-fmt/LTL-HVC16. We compare UFSCC with respectively NDFS, CNDFS, and Renault (all performed on BAs) in the upcoming sections. Some additional experiments follow and the results are summarized in Tables 1 and 2. In the context of validation, we are pleased to note that we won in the LTL category of the 2016 Model Checking Contest [18], where we employed the UFSCC algorithm.

5.1 Comparison with NDFS

We first compare the results of UFSCC with the sequential NDFS algorithm. Figure 5 shows the speedup of UFSCC (using 64 workers) compared to NDFS. Here, the point $(x = 10, y = 20)$ implies that the NDFS algorithm took 10 s to complete and UFSCC is 20× faster (thus taking only 0.5 s). We first consider the experiments that do not contain counterexamples (Fig. 5a).

The colored marks depict the 'origins' of the models. When relating this to time and speedup, the different classes are dispersed similarly, though the `BEEM-gen` models are more clustered.

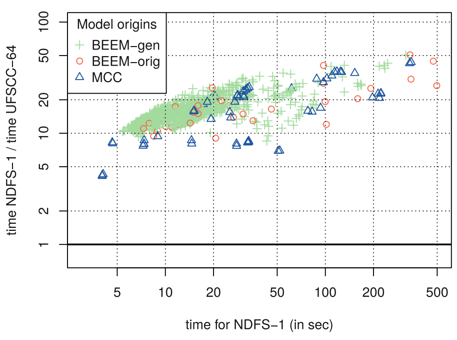
Generally, UFSCC performs at least 10× faster than NDFS. In Table 1 we observe that the average[7] speedup is 14.16. For larger models (where NDFS

[4] Available at https://www.lrde.epita.fr/~renault/benchs/TACAS-2015/results.html.

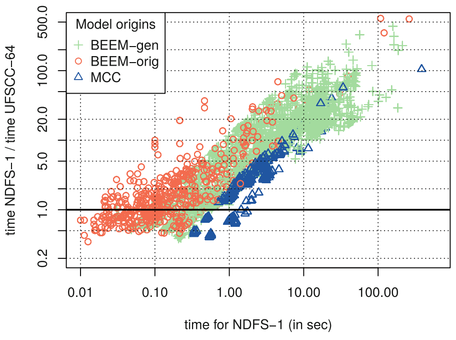
[5] Available at http://mcc.lip6.fr/2015/.

[6] The reason for this selection is to avoid unrealistic computation times, since the scalability measurements require a run of a sequential algorithm as well.

[7] When we discuss averages over the experiments, we always take the geometric mean.



(a) Without counterexamples



(b) With counterexamples

Fig. 5. Time comparison of UFSCC using 64 workers with sequential NDFS. (Color figure online)

requires more time), the speedup increases. We observed that the improvement in time to model check closely relates to the size of the model. This effect is also visible in Table 1, where the speedup grows from 13.24 to 24.35 when comparing the smallest and largest class of models.

The results are a bit different for experiments that do contain counterexamples (Fig. 5b). Here, UFSCC is actually slower than NDFS for some 'smaller' instances (where NDFS completes within 1 s). This is explained by the extra setup time required for UFSCC, combined with an additional bookkeeping on the data structures, which becomes purposeless in trivial cases.

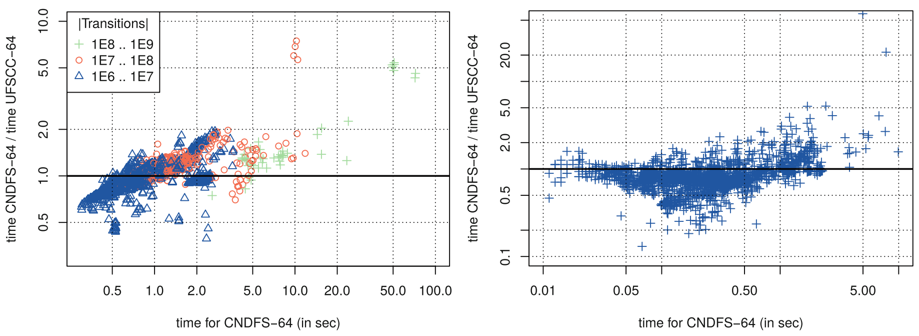
For increasingly larger models, the speedup for UFSCC improves rapidly. This speedup becomes superlinear (more than 64× faster using 64 workers), which is explained by the fact that multiple workers are more likely to find a counterexample due to randomization [8,28].

When relating the results to the origins of the models, we more clearly observe differences. NDFS performs (on average) the fastest on the `BEEM-orig` experiments, and the slowest on the `MCC` experiments. For all benchmarks with counterexamples, UFSCC performs on average 4.87 times faster than NDFS. Note that when taking the subset of models where NDFS takes more than 10 s, UFSCC is 30 times faster.

5.2 Comparison with CNDFS

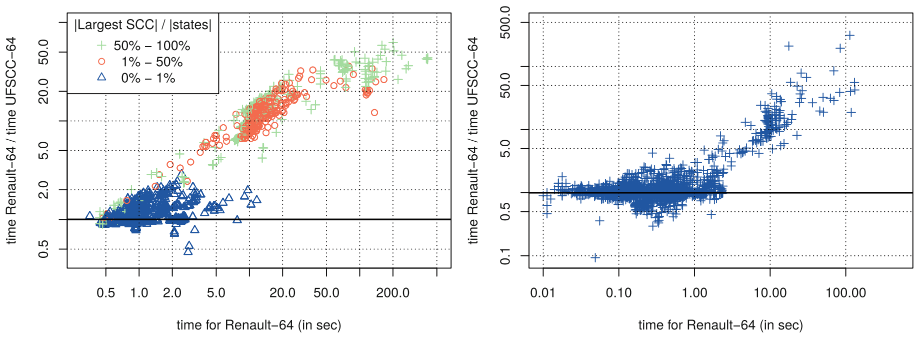
We compare the results of UFSCC with CNDFS, where both algorithms use 64 workers. Results for models without counterexamples are depicted in Fig. 6a.

In most cases, the performance of UFSCC and CNDFS is comparable, and the time difference rarely exceeds a factor of 2. The figure classifies the experiments by the number of transitions in the model. From this classification it becomes clear that UFSCC's relative performance improves for larger models. In Table 1 we find that the relative speedup of UFSCC versus CNDFS increases from 0.95 to 1.31. One can also observe in this table that UFSCC slightly outperforms CNDFS in graphs containing large SCCs.



(a) Without counterexamples (b) With counterexamples

Fig. 6. Time comparison of UFSCC with CNDFS, both using 64 workers.



(a) Without counterexamples (b) With counterexamples

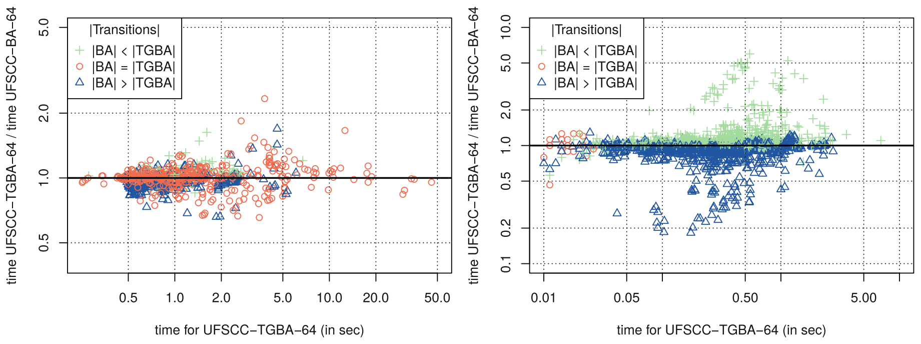
Fig. 7. Time comparison of UFSCC with Renault, both using 64 workers.

For models with counterexamples (Fig. 6b) CNDFS clearly performs better for most of the models. On average, UFSCC is 0.79 times as fast as CNDFS. However, the techniques do complement each other since UFSCC outperforms CNDFS in 14 % of the instances, in particular the experiments where CNDFS performs slowest.

5.3 Comparison with Renault

We compare the results of UFSCC with Renault, where both algorithms use 64 workers. Recall that both algorithms are SCC-based. Figure 7a depicts the results for experiments without counterexamples.

We observe a clear distinction when relating the results with the SCC characteristics. A significant speedup is observed for all models containing a largest SCC that consists of at least 1 % of the total state space. This is explained by the fact that Renault does not communicate partially found SCCs *searches* as explained in Sect. 3, whereas UFSCC does achieve this.



(a) Without counterexamples (b) With counterexamples

Fig. 8. Time comparison of UFSCC using BAs with its TGBA variant, both using 64 workers. Here, `|Transitions|` compares (a) the total number of transitions in the complete cross-product and (b) the average number of uniquely explored transitions by the algorithms.

From Table 1 we notice that UFSCC's speedup increases for larger models (19.58× speedup for the largest class of models), this can be mainly explained by the differences in SCC sizes.

For models with counterexamples (Fig. 7b), we observe that Renault performs similar to UFSCC for most of the models. This is to be expected since accepting cycles are detected in the same manner. However, the same effect concerning SCC sizes as in Fig. 7a seems present. We further analyzed some of the experiments where Renault performs relatively poor and indeed found that these instances contain large SCCs.

5.4 Experiments Using TGBA

One can consider classifying LTL formulas by using the temporal hierarchy of Manna and Pnueli [24]. For one of these classes, called *persistence*, each SCC in the automaton of the formula either fully consists of accepting states or non-accepting states. This class of problems can be reduced to a simple DFS [36]. In the dual, called *recurrence*, the automaton for the formula contains both accepting and non-accepting cycles in the same SCCs, making it necessary to perform an accepting cycle search. The combination of multiple recurrence and persistence formulas is described as a *reactive* formula, and can benefit from TGBA-acceptance by using a different accepting mark for each formula.

We made a comparison with two versions of UFSCC, where one is implemented for checking on BAs and the other for TGBAs. We found that only a few instances could be classified as persistence.

While the results do differ per model, the TGBA and BA implementations perform equally well on average in Fig. 8a and b. Also, in Fig. 8a one can observe that in most cases the size of the cross-product is equal for the BA and TGBA versions. A consequence is that a TGBA should not provide any benefit over a BA in these cases.

Table 1. Comparison of geometric mean execution times (in seconds) on models **without** counterexamples. **T** denotes the number of transitions in the state space and **S** denotes the ratio of the largest SCC size compared to the state space. The numbers between parentheses denote how many times faster UFSCC-BA is compared to the other algorithm.

		NDFS	CNDFS	Renault	UFSCC-TGBA	UFSCC-BA
T	0 .. 1E7	13.55 (13.24)	0.97 (0.95)	2.32 (2.27)	1.02 (0.99)	1.02
	1E7 .. 1E8	25.47 (18.71)	1.54 (1.13)	6.30 (4.63)	1.36 (1.00)	1.36
	1E8 .. INF	183.37 (24.35)	9.89 (1.31)	147.44 (19.58)	7.76 (1.03)	7.53
S	0% .. 1%	14.99 (13.65)	0.99 (0.91)	1.17 (1.06)	1.09 (0.99)	1.10
	1% .. 50%	18.33 (16.19)	1.38 (1.22)	11.83 (10.46)	1.13 (1.00)	1.13
	50% .. 100%	15.77 (13.69)	1.20 (1.04)	12.02 (10.44)	1.15 (1.00)	1.15
Total		15.95 (14.16)	1.11 (0.98)	3.01 (2.67)	1.12 (1.00)	1.13

Table 2. Comparison of geometric mean execution times (in seconds) on models **with** counterexamples. The numbers between parentheses denote how many times faster UFSCC-BA is compared to the other.

	NDFS	CNDFS	Renault	UFSCC-TGBA	UFSCC-BA
Total	1.52 (4.87)	0.25 (0.79)	0.37 (1.17)	0.31 (1.00)	0.31

Perhaps surprisingly, TGBAs do not benefit model checking in the experiments that we performed. Even when the TGBA version does provide a smaller cross-product, the algorithms still perform similarly. This may be explained by the additional overhead and bookkeeping for tracking acceptance sets.

5.5 Additional Results

We compare the relative maximal SCC size with the classification of transitions according to UFSCC, i.e. the number of *dead*, *visited* and *new* transitions (see Sect. 4).

The results for all experiments without counterexamples are summarized in Fig. 9. Here, the small, medium, and large SCC size cases relate to the respective three SCC classes in Table 1. The main observation is that models with large SCCs contain a high number of interconnectivity. In the 'large' class, 57.5 % of all explored transitions direct to already visited states (either locally visited or part of a globally known partial SCC).

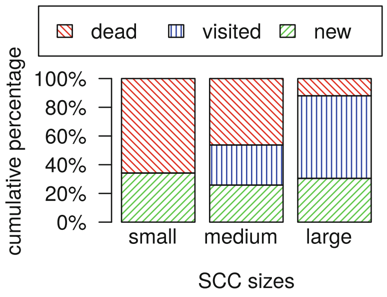


Fig. 9. Successor distribution for UFSCC using 64 workers.

Ideally, a multi-core algorithm should perfectly divide all states and transitions equally over all workers with minimal overhead. In practice, we observe that some transitions are explored by multiple workers. For UFSCC, with 64 workers, we analyzed the ratio of all explored transitions (cumulative for all workers) compared to the number of uniquely found transitions. For models without counterexamples this ratio is 141.0 %, meaning that the 64 workers perform a combined total of 41.0 % redundant explorations. Notably, for models that contain `1E7..1E8` transitions, the ratio drops to 118.8 % and 116.2 % for the largest class of models. SCC sizes do not seem to influence the re-exploration ratio.

The ratio for models with counterexamples is 182.8 %. This higher ratio is explained by the fact that only a small part of the state space is explored, which leaves few opportunities for branching the searches.

We observed that while large SCCs are generally highly interconnected in practice, the work is divided effectively since the re-exploration ratio is limited.

6 Conclusion

We showed that the UFSCC algorithm is well-suited for multi-core on-the-fly LTL model checking. The algorithm improves on related work by globally communicating partially detected SCCs, causing it to achieve good speedups on models with large SCCs. We also showed that the algorithm scales better compared to existing work when the state space increases.

Although we have considerably improved the scalability of LTL model checking, there is still room for improvement. For large models we observe a 25$\times$ speedup with 64 cores. We consider maintaining the concurrent union-find structure to be the main bottleneck. A combination with work-stealing queues or synchronizing the search instances may prove beneficial. Other directions are to extend this work to support partial-order reduction and fairness checking.

Acknowledgements. We thank Alfons Laarman for his aid with the implementation and Jeroen Meijer for his work on the Petri net benchmarks. We also thank Alexandre Duret-Lutz, Marcus Gerhold and the anonymous reviewers for their helpful comments. This work is supported by the 3TU.BSR project.

References

1. Blahoudek, F., Duret-Lutz, A., Křetínský, M., Strejček, J.: Is there a best Büchi automaton for explicit model checking? In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software, SPIN 2014, pp. 68–76. ACM (2014)
2. Bloemen, V., Laarman, A., van de Pol, J.: Multi-core on-the-fly SCC decomposition. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, pp. 8:1–8:12. ACM (2016)
3. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. In: Kurshan, R. (ed.) Computer-Aided Verification, pp. 129–142. Springer US, New York (1993)
4. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999). doi:10.1007/3-540-48119-2_16

5. Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized Büchi automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005). doi:10.1007/11537328_15
6. Dijkstra, E.W.: Finding the maximum strong components in a directed graph. In: Dijkstra, E.W. (ed.) Selected Writings on Computing: A personal Perspective. Texts and Monographs in Computer Science, pp. 22–30. Springer, New York (1982)
7. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Heidelberg (2016). doi:10.1007/978-3-319-46520-3_8
8. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved multi-core nested depth-first search. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis. LNCS, vol. 7561, pp. 269–283. Springer, Heidelberg (2012)
9. 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). doi:10.1007/978-3-642-24372-1_27
10. Fleischer, L.K., Hendrickson, B., Pınar, A.: On identifying strongly connected components in parallel. In: Rolim, J. (ed.) IPDPS 2000. LNCS, vol. 1800, pp. 505–511. Springer, Heidelberg (2000). doi:10.1007/3-540-45591-4_68
11. Gaiser, A., Schwoon, S.: Comparison of Algorithms for Checking Emptiness on Büchi Automata. CoRR abs/0910.3766 (2009)
12. Geldenhuys, J., Valmari, A.: Tarjan's algorithm makes on-the-fly LTL verification more efficient. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)
13. Giannakopoulou, D., Lerda, F.: From states to transitions: improving translation of LTL formulae to Büchi automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002). doi:10.1007/3-540-36135-9_20
14. Holzmann, G., Joshi, R., Groce, A.: Swarm verification techniques. IEEE Trans. Softw. Eng. **37**(6), 845–857 (2011)
15. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Proceedings of the Second SPIN Workshop, vol. 32, pp. 81–89 (1996)
16. Hong, S., Rodia, N., Olukotun, K.: On fast parallel detection of strongly connected components (SCC) in small-world graphs. In: 2013 International Conference High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11 (2013)
17. Kant, G., Laarman, A., Meijer, J., Pol, J., Blom, S., Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_61
18. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Chiardo, G., Hamez, A., Jezequel, L., Miner, A., Meijer, J., Paviot-Adet, E., Racordon, D., Rodriguez, C., Rohr, C., Srba, J., Thierry-Mieg, Y., Trinh, G., Wolf, K.: Complete Results for the 2016 Edition of the Model Checking Contest (2016)
19. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Linard, A., Beccuti, M., Hamez, A., Lopez-Bobeda, E., Jezequel, L., Meijer, J., Paviot-Adet, E., Rodriguez, C., Rohr, C., Srba, J., Thierry-Mieg, Y., Wolf, K.: Complete Results for the 2015 Edition of the Model Checking Contest (2015)

20. Laarman, A., Langerak, R., Pol, J., 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). doi:10.1007/978-3-642-24372-1_23
21. Laarman, A., van de Pol, J.: Variations on multi-core nested depth-first search. In: Barnat, J., Heljanko, K. (eds.) PDMC 2011. EPTCS, vol. 72, pp. 13–28 (2011)
22. Laarman, A., 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). doi:10.1007/978-3-642-20398-5_40
23. Lowe, G.: Concurrent depth-first search algorithms based on Tarjan's algorithm. Int. J. Softw. Tools Technol. Transf. **18**(2), 1–19 (2015)
24. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 1987, p. 205. ACM (1987)
25. Orzan, S.: On Distributed Verification and Verified Distribution. Ph.D. thesis (2004)
26. 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). doi:10.1007/978-3-540-73370-6_17
27. Pelánek, R.: Properties of state spaces and their applications. Int. J. Softw. Tools Technol. Transf. **10**(5), 443–454 (2008)
28. Rao, V.N., Kumar, V.: Superlinear speedup in parallel state-space search. In: Nori, K.V., Kumar, S. (eds.) Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 338, pp. 161–174. Springer, Heidelberg (1988)
29. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Parallel explicit model checking for generalized Büchi automata. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 613–627. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_56
30. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Variations on parallel explicit emptiness checks for generalized Büchi automata. Int. J. Softw. Tools Technol. Transf. 1–21 (2016). http://link.springer.com/journal/10009/onlineFirst/page/1
31. Schudy, W.: Finding strongly connected components in parallel using O(log2n) reachability queries. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 146–151. ACM (2008)
32. 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). doi:10.1007/978-3-540-31980-1_12
33. Slota, G.M., Rajamanickam, S., Madduri, K.: BFS and coloring-based parallel algorithms for strongly connected components and related problems. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 550–559 (2014)
34. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
35. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the First Symposium on Logic in Computer Science, pp. 322–331. IEEE Computer Society (1986)
36. Černá, I., Pelánek, R.: Relating hierarchy of temporal properties to model checking. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 318–327. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45138-9_26