Variations on Parallel Explicit Emptiness Checks for Generalized Büchi Automata

Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, Denis Poitrenaud

Noname manuscript No.
(will be inserted by the editor)

Variations on Parallel Explicit Emptiness Checks for Generalized Büchi Automata

E. Renault$^{(1)}$ · **A. Duret-Lutz**$^{(1)}$
F. Kordon$^{(2,3)}$ · **D. Poitrenaud**$^{(3,4)}$

Abstract We present new parallel explicit emptiness checks for LTL model checking. Unlike existing parallel emptiness checks, these are based on a *Strongly Connected Component* (SCC) enumeration, support generalized Büchi acceptance, and require no synchronization points nor recomputing procedures. A salient feature of our algorithms is the use of a global union-find data structure in which multiple threads share structural information about the automaton checked.

Besides these basic algorithms, we present one architectural variant isolating threads that write to the union-find, and one extension that decomposes the automaton based on the *strength* of its SCCs to use more optimized emptiness checks.

The results from an extensive experimentation of our algorithms and their variations show encouraging performances, especially when the decomposition technique is used.

1 Introduction

Automata-theoretic approach to explicit LTL model checking explores the product between two ω-automata: one automaton that represents the system, and the other that represents the negation of the property to check on this system. This product corresponds to the intersection between the executions of the system and the behaviors disallowed by the property. The property is verified if this product has no accepting executions (i.e., its language is empty).

(1) LRDE, EPITA, Kremlin-Bicêtre, France · (2) Sorbonne Universités, UPMC Univ. Paris 06, France · (3) CNRS UMR 7606, LIP6, F-75005 Paris, France · (4) Université Paris Descartes, Paris, France

Usually, the property is represented by a Büchi automaton (BA), and the system by a Kripke structure. Here we represent the property with a more concise Transition-based Generalized Büchi Automaton (TGBA), in which the Büchi acceptance condition is generalized to use multiple acceptance conditions. Furthermore, any BA can be represented by a TGBA without changing the transition structure: the TGBA-based emptiness checks we present are therefore compatible with BAs.

A BA (or TGBA) has a non-empty language iff it contains an accepting cycle reachable from the initial state (for model checking, this maps to a counterexample). An emptiness check is an algorithm that searches for such a cycle.

Most sequential explicit emptiness checks are based on a *Depth-First Search* (DFS) exploration of the automaton and can be classified in two families: those based on an enumeration of *Strongly Connected Components* (SCC), and those based on a *Nested Depth First Search* (NDFS) (see [38, 17, 35] for surveys).

Recently, parallel (or distributed) emptiness checks have been proposed [12, 2, 15, 13, 3, 5]: they are mainly based on a *Breadth First Search* (BFS) exploration that scales better than DFS [34]. Multicore adaptations of these algorithms with lock-free data structure have been discussed, but not evaluated [6].

Recent publications show that NDFS-based algorithms combined with the *swarming* technique [23] scale better in practice [20, 27, 26, 21]. As its name implies, an NDFS algorithm uses two nested DFS: a first DFS explores a BA to search for accepting states, and a second DFS is started (in post order) to find cycles around these accepting states. In these parallel setups, each thread performs the same search strategy (an NDFS) and differs only in the search order (swarming). Because each thread shares some information about its own progress in the NDFS, some mechanisms are necessary to avoid conflicts. These conflicts can be prevented using synchronization points: if a state is handled by multiple threads in the nested DFS, its status is only updated after all threads have finished. Conflicts can also be resolved a posteri using recomputing procedures that perform yet another DFS. So far, attempts to design scalable parallel DFS-based emptiness check that does not require such mechanisms have failed [21].

More recent works [28, 10, 11] focus on the parallel computation of SCCs and possible adaptations to emptiness checks. These target graphs that are composed of large and (possibly) unique SCC, also using synchronization or locking schemes to ensure correctness, even if Bloemen [10] argues that his algorithm may be implemented lockless.

This paper is an extension of our work published at TACAS'15 [37] where we proposed new parallel emptiness checks for TGBA built upon two SCC-based strategies that do not require such synchronization points nor recomputing procedures. The reason no such mechanisms are necessary is that threads only share structural information about the automaton of the form *"states x and y are in the same SCC"* or *"state x cannot be part of a counterexample"*. Since threads do not share any information about the progress of their search, we can actually mix threads with different strategies in the same emptiness check. Because the shared information can be used to partition the states of the automaton, it is stored in a global and lock-free union-find data structure [1]. As it name suggests, a union-find is a data structure that represents sets and provides efficient union, and membership-check procedures that can be implemented in near-constant time [40, 31]

In addition to the above (common with our previous paper [37]), we investigate two variants. In the first one, threads that write to the union-find are isolated, in an attempt to limit the contention on the shared data structure. Our second variant mixes the above emptiness checks with a decomposition technique we presented at TACAS'13 [36]. This decomposition is actually compatible with any parallel emptiness check: the property automaton is decomposed into three subautomata with different *strengths.* Two of them can then be checked using more efficient emptiness checks. In a parallel context, it can also be seen as an improvement of the swarming technique: the decomposition favors a more uniform distribution of the paths covered by the different threads.

The paper is organized as follows. First section 2 defines TGBAs and introduces our notations. Section 3 details the two SCC-based strategies previously presented at TACAS'15 [37] augmented with examples, and a discussion of counterexample generation. The two aforementioned variants of those algorithms are presented in Section 4. Some of the related algorithms discussed in Section 5 are finally used for comparison in the benchmarks of Section 6.

2 Preliminaries

A *TGBA* is a tuple $A = \langle AP, Q, q^0, \delta, \mathcal{F}\rangle$ where AP is a finite set of atomic propositions, Q is a finite set of states, q^0 is a designated initial state, $\mathcal{F}$ is a finite set of acceptance marks, and $\delta \subseteq Q \times 2^{\mathcal{F}} \times \mathbb{B}^{AP} \times Q$ is the (non-deterministic) transition relation where each transition is labeled by a subset of acceptance marks and an assignment of the atomic propositions.

A *path* between two states $q, q' \in Q$ is a finite and non-empty sequence of adjacent transitions $\rho = (s_1, \alpha_1, \_, s_2)(s_2, \alpha_2, \_, s_3)\ldots(s_n, \alpha_n, \_, s_{n+1}) \in \delta^+$ with $s_1 = q$ and $s_{n+1} = q'$. We denote the existence of such a path by $q \rightsquigarrow q'$. When $q = q'$ the path is a *cycle.* This cycle is *accepting* iff $\bigcup_{0<i\leq n} \alpha_i = \mathcal{F}$.

A non-empty set $S \subseteq Q$ is a Strongly Connected Component (SCC) iff $\forall s,\ s' \in S,\ s \neq s' \Rightarrow s \rightsquigarrow s'$ and S is maximal w.r.t. inclusion. If S is not maximal, we call it a *partial* SCC. An SCC S is *complete* iff $\forall s \in S, \forall \ell \in \mathbb{B}^{AP}, \exists \alpha \in 2^{\mathcal{F}}, \exists q \in S, (s, \alpha, \ell, q) \in \delta$. An SCC is *accepting* iff it contains an accepting cycle. The language of a TGBA A is non-empty iff there is a path from q^0 to an accepting SCC (denoted by $\mathscr{L}(A) \neq \emptyset$).

The automata-theoretic approach to model checking amounts to check the emptiness of the language of a TGBA that represents the product of a system (a TGBA where $\mathcal{F} = \emptyset$) with the negation of the property to verify (another TGBA).

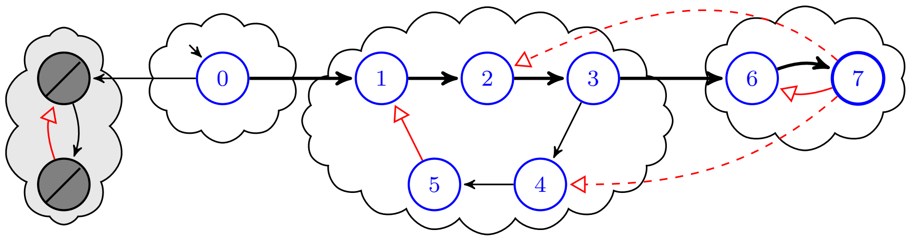
3 Generalized Parallel Emptiness Checks

In a previous work [35] we presented sequential emptiness checks for generalized Büchi automata derived from the SCC enumeration algorithms of Tarjan [39] and Dijkstra [18], and a third one using a union-find data-structure. This section adapts these algorithms to a parallel setting.

The sequential versions of the Tarjan-based and the Dijkstra-based emptiness checks both have very similar structures: they explore the automaton using a single DFS to search for an accepting SCC and maintain a partition of the states into three classes. States that have not already been visited are UNKNOWN; a state is LIVE when it is part of an SCC that has not been fully explored (i.e., it is part of an SCC that contains at least one state on the DFS stack); the other states are called DEAD. A DEAD state cannot be part of an accepting SCC. Any LIVE state can reach a state on the DFS stack, therefore a transition from the state at the top of the DFS stack leading to a LIVE state is called a *closing edge.* Figure 1 illustrates some of these concepts.

The two algorithms differ in the way they propagate information about currently visited SCCs, and in when they detect accepting SCCs. A Tarjan-based emptiness check propagates information when edges are backtracked, and may only find an accepting SCC when its *root* is popped. (The *root* of an SCC is the

Fig. 1 LIVE states are numbered by their live number, DEAD states are stroke. Clouds represents (partial) SCCs as discovered so far. The current state of the DFS is 7, and the DFS stack is represented by thick edges. All plain edges have already been explored while dashed edges are yet to be explored. Closing edges have white triangular tips.



first state encountered by the DFS when entering it.) A Dijkstra-based emptiness check propagates information every time a closing edge is detected: when this happens, a partial SCC made of all states on the cycle closed by the closing edge is immediately formed. While we have shown these two emptiness checks to be comparable [35], the Dijkstra-based algorithm reports counterexamples earlier: as soon as all the transitions belonging to an accepting cycle have been seen.

A third algorithm was a variant of Dijkstra using a union-find data structure to manage the membership of each state to its SCC. Note that this data structure could also be used for a Tarjan-based emptiness check.

Here, we parallelize the Tarjan-based and Dijkstra-based algorithms and use a (lock-free) shared union-find data structure. We rely on the *swarming* technique: each thread executes the same algorithm, but explores the automaton in a different order [23]. Furthermore, threads will use the union-find to share information about membership to SCCs, acceptance of these SCCs, and DEAD states. Note that the shared information is stable: the fact that two states belong to the same SCC, or that a state is DEAD will never change over the execution of the algorithm. All threads may therefore reuse this information freely to accelerate their exploration, and to find accepting cycles collaboratively.

3.1 Generic Canvas

Algorithm 1 presents the structure common to the Tarjan and Dijkstra-based parallel emptiness checks[1].

All threads share (1) the automaton A to explore, (2) a *stop* variable used to stop all threads as soon an accepting cycle is found or one thread detects that the whole automaton has been visited, and (3) the union-find data-structure. The union-find maintains the membership of each state to the various SCCs of the automaton, or the set of DEAD states (a state is DEAD if it belongs to the same class as the artificial *Dead* state). Furthermore, the union-find is extended to store the acceptance marks occurring in an SCC.

The union-find structure partitions the set $Q' = Q \cup \{Dead\}$ labeled with an element of $2^{\mathcal{F}}$ and offers the following methods:

- `make_set`($s \in Q'$) creates a new class containing the state s if s is not already in the union-find.
- `contains`($s \in Q'$) checks whether s is already in the union-find.
- `unite`($s_1 \in Q'$, $s_2 \in Q'$, $acc \in 2^{\mathcal{F}}$) merges the classes of s_1 and s_2, and adds the acceptance marks acc to the resulting class. This method returns the set of acceptance marks of resulting class. However, when the class constructed by `unite` contains *Dead*, this method always returns $\emptyset$. An accepting cycle can therefore be reported as soon as `unite` returns $\mathcal{F}$.
- `same_set`($s_1 \in Q'$, $s_2 \in Q'$) checks whether two states belong to the same class.

Such a union-find structure can be implemented thread-safe in many ways [9]: with fine/coarse grain locking or lock-free based either on transactional memory or on compare-and-swap operations.

The original sequential algorithms maintains a stack of LIVE states in order to mark all states of an explored SCC as DEAD. In our previous work [35], we suggested to use the union-find for this, allowing to mark all states of an SCC as dead by doing a single unite with an artificial *Dead* state. However, this notion of LIVE state (and closing edge detection) is dependent on the traversal order, and will therefore be different in each thread. Consequently, each thread has to keep track locally of its own LIVE states. Thus, each thread maintains the following local variables:

- The *dfs* stack stores elements of type *Step* composed of the current state (*src*), the acceptance mark (*acc*) for the incoming transition (or $\emptyset$ for

[1] According to our definition, transitions of the automaton should be labeled by atomic propositions (line 8), but we omit this information as it is not pertinent to emptiness check algorithms.

Algorithm 1: Main procedure

```
Shared Variables:
  $A$: TGBA of $\langle AP, Q, q^0, \delta, \mathcal{F}\rangle$
  $stop$: $boolean$
  $uf$: $union\text{-}find$ of $\langle Q \cup Dead, 2^{\mathcal{F}} \rangle$

Global Structures:
  struct $Step$ { $src$: $Q$,      $acc$: $2^{\mathcal{F}}$,
                  $pos$: $int$,    $succ$: $2^{\delta}$ }
  struct $Transition$ {$src$: $Q$, $acc$: $2^{\mathcal{F}}$, $dst$: $Q$}
  enum $Strategy$ { Mixed, Tarjan, Dijkstra}
  enum $Status$ { LIVE, DEAD, UNKNOWN}

Local Variables:
  $dfs$: $stack$ of $\langle Step\rangle$
  $live$: $stack$ of $\langle Q \rangle$
  $livenum$: $hashmap$ of $\langle Q, int \rangle$
  $pstack$: $stack$ of $\langle P \rangle$
  $str2$: $Strategy$

main($str$: $Strategy$)
  $stop \leftarrow \perp$
  $str2 \leftarrow str$
  $uf$.make_set($\langle Dead, \emptyset \rangle$)
  if str = Mixed
    $str \leftarrow$ Dijkstra
    $str2 \leftarrow$ Tarjan
  Run EC($str$, 1) ‖ ... ‖ EC($str$, $\lfloor \frac{n}{2} \rfloor$) ‖
      EC($str2$, 1+$\lfloor \frac{n}{2} \rfloor$) ‖ ... ‖ EC($str2$, $n$)
  Wait for all threads to finish

GET_STATUS($q \in Q$) $\rightarrow$ $Status$
  if $livenum$.contains($q$)
    return LIVE
  else if $uf$.contains($q$) $\wedge$
        $uf$.same_set($q$, $Dead$)
    return DEAD
  else
    return UNKNOWN

EC($str$: $Strategy$, $tid$: $int$)
  seed($tid$) // Random Number Gen.
  PUSH$_{str}$($\emptyset$, $q^0$)
  while $\neg$ $dfs$.empty() $\wedge$ $\neg$ $stop$
    $Step$ $step$ $\leftarrow$ $dfs$.top()
    if $step.succ \neq \emptyset$
      $Transition$ $t$ $\leftarrow$ randomly
        pick one off from $step.succ$
      switch GET_STATUS($t.dst$)
        case DEAD
          skip
        case LIVE
          UPDATE$_{str}$($t.acc$, $t.dst$)
        case UNKNOWN
          PUSH$_{str}$($t.acc$, $t.dst$)
    else
      POP$_{str}$()
  $stop \leftarrow \top$
```

the initial state), an identifier *pos* (whose use is different in Dijkstra and Tarjan) and the set *succ* of unvisited successors of the *src* state.

– The *live* stack stores all the LIVE states that are not on the *dfs* stack (as suggested by Nuutila and Soisalon-Soininen [30]).
– The hash map *livenum* associates each LIVE state to a (locally) unique increasing identifier.
– *pstack* holds identifiers that are used differently in the emptiness checks of this paper.

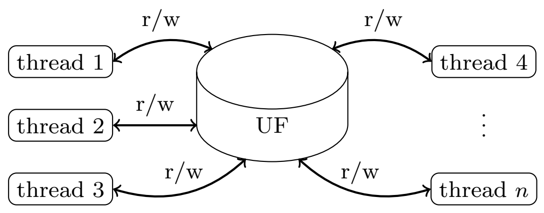
With these data structures, a thread can decide whether a state is LIVE, DEAD, or UNKNOWN (i.e., new) by first checking *livenum* (a local structure), and then *uf* (a shared structure). This test is done by `GET_STATUS`. Note that a state marked LIVE locally may have already been marked DEAD by another thread, thus leading to redundant work. However, avoiding this extra work would require more queries to the shared *uf*.

The procedure `EC` shows the generic DFS executed by all threads. The successors are ordered randomly in each thread, and the DFS stops as soon as one thread sets the *stop* flag. `GET_STATUS` is called on each reached state to decide how it has to be handled: DEAD states are ignored, UNKNOWN states are pushed on the *dfs* stack, and and LIVE states, which are reached when following a closing edge, will be handled differently by each algorithm. This generic DFS is adapted to the Tarjan and Dijkstra strategies by calling `PUSH`$_{str}$ on new states, `UPDATE`$_{str}$ on closing edges, and `POP`$_{str}$ when all the successors of a state have been visited by this thread.

Several parallel instances of this `EC` algorithm are instantiated by the `main` procedure, possibly using different strategies. Each instance is parameterized by a unique identifier *tid* and a *Strategy* selecting either Dijkstra or Tarjan. If `main` is called with the Mixed strategy, it instantiates a mix of both emptiness-checks. When one thread reports an accepting cycle or ends the exploration of the entire automaton, it sets the *stop* variable, causing all threads to terminate. The `main` procedure only has to wait for the termination of all threads.

Figure 2 presents the general architecture of the algorithm up to n threads. We observe that the union-find data structure, denoted by UF, is shared among all the threads. Since every operation on this structure may modify it, every access needs read/write permissions. With this architecture, it's clear that two

Fig. 2 Parallel emptiness check architecture.



threads can only exchange information using the union-find, and so only exchange *structural information*.

3.2 The Tarjan Strategy

Strategy 1 shows how the generic canvas is refined to implement the Tarjan strategy. Here, each new LIVE state is numbered with the actual number of LIVE states during the $\texttt{PUSH}_{Tarjan}$ operation. Furthermore each state is associated to a *lowlink*, i.e., the smallest live number of any state known to be reachable from this state. These *lowlinks*, whose purpose is to detect the root of each SCC, are only maintained for the states on the *dfs* stack, and are stored on the *pstack*.

Lowlinks are updated either when a closing edge is detected in $\texttt{UPDATE}_{Tarjan}$ (in this case the current state and the destination of the closing edge are in the same SCC) or when a non-root state is popped in $\texttt{POP}_{Tarjan}$ (in this case the current state and its predecessor on the *dfs* stack are in the same SCC). Every time a *lowlink* is updated, we therefore learn that two states belong to the same SCC and can publish this fact to the shared *uf* taking into account any acceptance mark between those two states. If the *uf* detects that the union of these acceptance marks with those already known for this SCC is $\mathcal{F}$, then the existence of an accepting cycle can be reported immediately.

$\texttt{POP}_{Tarjan}$ has two behaviors depending on whether the state being popped is a root or not. At this point, a state is a root if its *lowlink* is equal to its live number. Non-root states are transferred from the *dfs* stack to the *live* stack. When a root state is popped, we first publish that the whole SCC associated to this root is DEAD, and locally we remove all these states from *live* and *livenum* using the `markdead` function.

If there is no accepting cycle, the number of calls to `unite` performed in a single thread by this strategy is always the number of transitions in each SCC (corresponding to the lowlink updates) plus the number of SCCs (corresponding to the calls to `markdead`).

Figure 3 illustrates, on a toy example, one possible behavior of two threads executing the Tarjan strategy. This example shows a collaborative detection of a counterexample by multiple threads.

The union-find is represented by a set of pairs (state, accepting mark) and the parent of each pair is represented by an arrow. Two pairs belong to the same class if they have a common ancestor. For each thread, we also display the states on the *dfs* stack (also highlighted on the automaton) and their associated lowlink number stored in *pstack*. In this particular scenario, for a given state, the live number of a state is identical to its position in the *dfs* stack. In the general case, the correspondence between a LIVE state and its live number is given by *livenum* (not illustrated here).

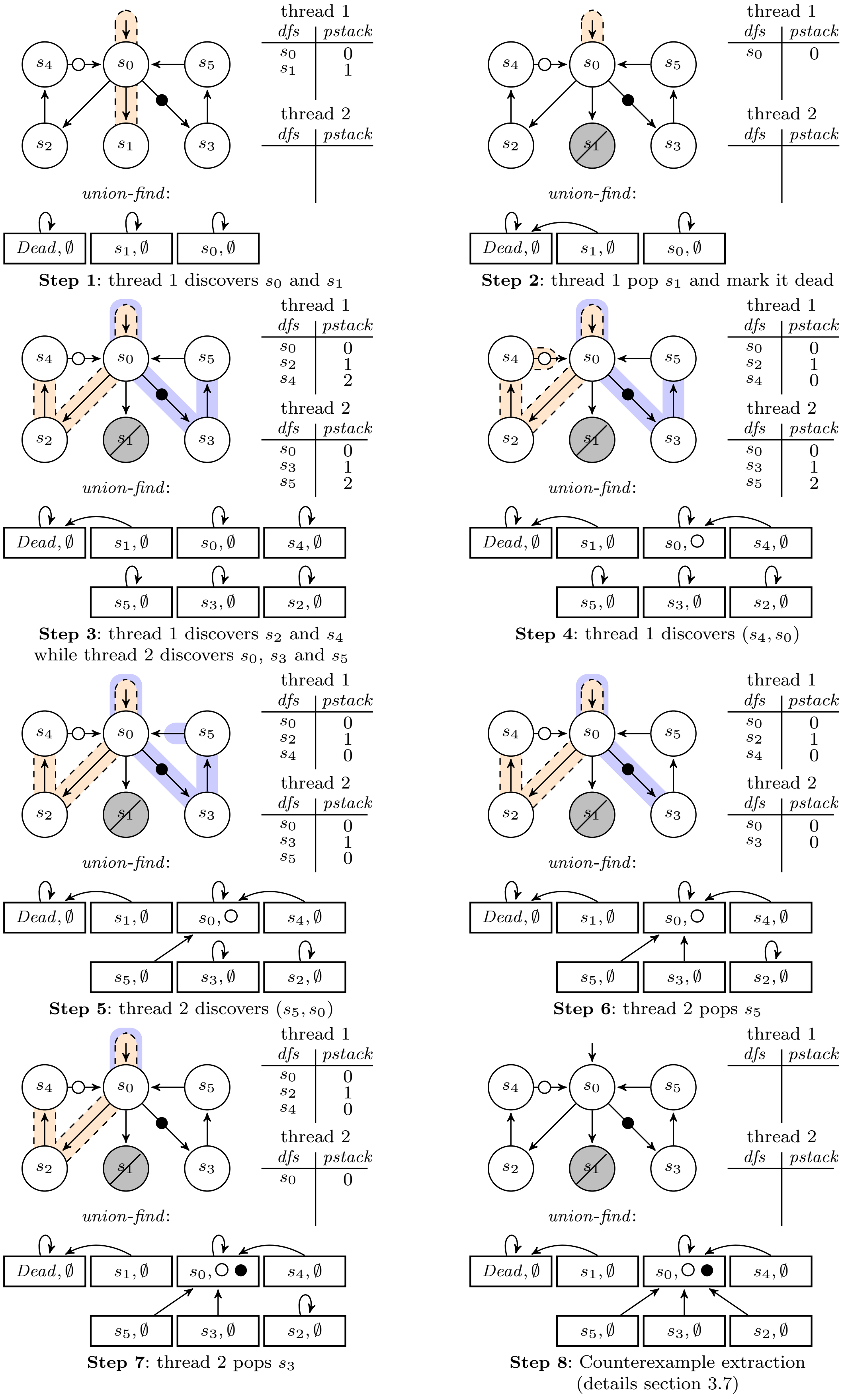
Strategy 1: Tarjan

```
struct P {p : int}

PUSH_Tarjan(acc ∈ 2^F, q ∈ Q)
    uf.make_set(q)
    p ← livenum.size()
    livenum.insert(⟨ q, p ⟩)
    pstack.push(⟨ p ⟩)
    dfs.push(⟨ q, acc, p, succ(q)⟩)
UPDATE_Tarjan(acc ∈ 2^F, d ∈ Q)
    pstack.top().p ←
        min(pstack.top().p, livenum.get(d))
    a ← uf.unite(d, dfs.top().src, acc)
    if a = F
        stop ← ⊤
        report accepting cycle found
POP_Tarjan()
    s ← dfs.pop()
    ⟨ ll ⟩ ← pstack.pop()
    if ll = s.pos
        markdead(s)
    else
        pstack.top().p ← min(pstack.top().p, ll)
        a ← uf.unite(s.src, dfs.top().src, s.acc)
        if a = F
            stop ← ⊤
            report accepting cycle found
        live.push(s.src)
// Common to all strategies.
markdead(s ∈ Step)
    uf.unite(s.src, Dead)
    livenum.remove(s.src)
    while livenum.size() > s.pos
        q ← live.pop()
        livenum.remove(q)
```

At the initial stage (also not shown here), the shared union-find data structure contains only the *Dead* class and the DFS stack of each thread is empty. In the scenario we consider, thread 1 starts from s_0, detects s_1 (step 1), marks this state as dead and backtracks to s_0 (step 2). In the union-find, we can observe that s_1 is

Fig. 3 Pure Tarjan: all threads perform a Tarjan strategy on an automaton with $\mathcal{F} = \{●,○\}$.


thread 1
dfs | pstack
thread 2
dfs | pstack
union-find:
Dead, ∅
s_0, ○ ●
Step 1: thread 1 discovers s_0 and s_1
Step 2: thread 1 pop s_1 and mark it dead
Step 3: thread 1 discovers s_2 and s_4 while thread 2 discovers s_0, s_3 and s_5
Step 4: thread 1 discovers (s_4, s_0)
Step 5: thread 2 discovers (s_5, s_0)
Step 6: thread 2 pops s_5
Step 7: thread 2 pops s_3
Step 8: Counterexample extraction (details section 3.7)

dead since it belongs to the same class as the artificial *Dead* state. Then threads 1 and 2 explore simultaneously the automaton in a different order thanks to the swarming, thus reaching step 3. In step 4, thread 1 detects the closing-edge (s_4, s_0) and the classes containing these states are merged in the union-find keeping track of the white acceptance mark on the topmost ancestor. In step 5, thread 2 discovers the closing-edge (s_5, s_0) and therefore unites s_5 with s_0: this creates the class containing s_0 and s_4 and s_5.

In step 6 and 7, the thread 2 backtracks its *dfs*. Therefore s_5, s_3 and s_0 are united. During the union of s_3 and s_0 the black acceptance mark is added to the topmost ancestor of this class. Thus, the algorithm can stop at this point because it has a proof that an accepting SCC exists.

Step 8 will illustrates counterexample detection and will be discussed later (details Section 3.7).

3.3 The Dijkstra Strategy

Strategy 2 shows how the generic canvas is refined to implement the Dijkstra strategy. The way LIVE states are numbered and marked as DEAD is identical to the previous strategy. The difference lies in the way SCC information is encoded and updated.

This algorithm maintains *pstack*, a stack of potential roots, represented (1) by their positions p in the *dfs* stack (so we can later get the incoming acceptance marks and the live number of the potential roots), and (2) the union *acc* of all the acceptance marks seen in the cycles visited around the potential root.

Here *pstack* is updated in two situations. First, line 12 removes potential roots that are discovered not to be actual root because a closing edge is detected. Second, line 22 removes states that are either an actual root or belongs to an SCC marked as DEAD by another thread. This differs from Tarjan where *pstack* is always updated as it contains all states, even non-root. When a closing edge is detected, the live number *dpos* of its destination can be used to pop all the potential roots on this cycle (those whose live number are greater than *dpos*), and merge the sets of acceptance marks along the way: this happens in $\texttt{UPDATE}_{\textit{Dijkstra}}$. Note that the *dfs* stack has to be addressable like an array during this operation.

As it is presented, $\texttt{UPDATE}_{\textit{Dijkstra}}$ calls `unite` only when a potential root is discovered not to be a root (lines 10–14). In the particular case where a closing edge does not invalidate any potential root, no `unite` operation is performed; still, the acceptance marks on this closing edge are updated locally (line 15). For

Strategy 2: Dijkstra

```
 0 struct $P$ $\{p : int,\ acc : 2^{\mathcal{F}}\}$

 1 PUSH$_{Dijkstra}$($acc \in 2^{\mathcal{F}}$, $q \in Q$)
 2     $uf$.make_set($q$)
 3     $p \leftarrow livenum$.size()
 4     $livenum$.insert($\langle q, p \rangle$)
 5     $pstack$.push($\langle dfs$.size(), $\emptyset \rangle$)
 6     $dfs$.push($\langle q, acc, p,$ succ($q$)$\rangle$)

 7 UPDATE$_{Dijkstra}$($acc \in 2^{\mathcal{F}}$, $d \in Q$)
 8     $dpos \leftarrow livenum$.get($d$)
 9     $\langle r, a \rangle \leftarrow pstack$.top()
10     $a \leftarrow a \cup acc$
11     while $dpos < dfs[r].pos$
12         $\langle r, la \rangle \leftarrow pstack$.pop()
13         $a \leftarrow a \cup dfs[r].acc \cup la$
14         $a \leftarrow uf$.unite($d$, $dfs[r].src$, $a$)
15     $pstack$.top().$acc \leftarrow a$
16     if $a = \mathcal{F}$
17         $stop \leftarrow \top$
18         report accepting cycle found

19 POP$_{Dijkstra}$()
20     $s \leftarrow dfs$.pop()
21     if $pstack$.top().$p = dfs$.size()
22         $pstack$.pop()
23         markdead($s$) // Detailed in Strategy 1
24     else
25         $live$.push($s.src$)
```

instance in Figure 1, when the closing edge $(7, 4)$ is explored, the root of the right-most SCC (containing state 7) will be popped (effectively merging the two right-most SCCs in *uf*) but when the closing edge $(7, 2)$ is later explored no pop will occur because the two states now belong to the same SCC. This strategy therefore does not share all its acceptance information with other threads. In this strategy, the acceptance accumulated in *pstack* locally are enough to detect accepting cycles. However the `unite` operation on line 14 will also return some acceptance marks discovered by other threads around this state: this additional information is also accumulated in *pstack* to speedup the detection of accepting cycles.

In this strategy, a given thread only calls `unite` to merge two disjoint sets of states belonging to the same SCC. Thus, the total number of `unite` needed to build an SCC of n states is equal to $n - 1$. This is better than the Tarjan-based version, but it also means we share less information between threads.

Figure 4 shows a possible behavior of two threads executing the Dijkstra strategy on the same example as in Figure 3. We assume that threads visit the transitions in the same order and the same interleaving as in the Tarjan scenario. The discovery of the counterexample is still collaborative in this strategy.

We reuse the same data structure for the union-find and the *dfs* stack. However the *pstack* now contains pairs describing each partial SCC: its root number and its acceptance marks. For each thread, the root number gives the index of the corresponding state in the *dfs* stack. Thus a given state could have a different root number in each thread (not the case in this scenario).

The initial stage (step 0) is the same as for Tarjan: the shared union-find data structure contains only the *Dead* class and the stacks of each thread are empty. Until step 3 the only difference is in the contents of the two *pstack*s.

In step 4, thread 1 detects the closing-edge (s_4, s_0). Since the root s_0 has 0 as DFS number, all entries of *pstack* having a DFS number greater than the one of s_0 are merged. The resulting acceptance mark ○ is the union of (1) the acceptance marks of the merged entries, (2) the in-going acceptance marks of all merged states, (3) the set of acceptance marks stored in the union-find for each class and, (4) the acceptance mark of the closing-edge. The corresponding states s_0, s_2 and s_4 are united in union-find; thus the representative s_0 of this class also holds the acceptance mark ○ previously computed.
In step 5, thread 2 discovers the closing-edge (s_5, s_0) and operates a merge similar to the previous one. This time, acceptance mark ○ is returned by `unite` and both marks are merged into the union-find and the *pstack*. Thread 2 can stop the algorithm at this point because it has a proof that an accepting SCC exists.

3.4 The Mixed Strategy

Figure 5 presents two situations on which Dijkstra and Tarjan strategies can clearly be distinguished.

The left-hand side presents a bad case for the Tarjan strategy. Regardless of the transition order chosen during the exploration, the presence of an accepting cycle is only detected when state 1 is popped. This late detection can be costly because it implies the exploration of the whole subgraph represented by a cloud.

The Dijkstra strategy will report the accepting cycle as soon as all the involved transitions have been visited. So if the transition $(1, 0)$ is visited before the transition going to the cloud, the subgraph represented by this cloud will not be visited since the counterexample will be detected before.

On the right-hand side of Fig. 5, the dotted transition represents a long path of m transitions, without acceptance marks. On this automaton, both strategies will report an accepting cycle when transition $(n, 0)$ is visited. However, the two strategies differ in their handling of transition $(m, 0)$: when Dijkstra visits this transition, it has to pop all the candidate roots $1 \dots m$, calling `unite` m times; Tarjan however only has to update the *lowlink* of m (calling `unite` once), and it delays the update of the *lowlinks* of states $0 \dots m-1$ to when these states would be popped (which will never happen because an accepting cycle is reported).

In an attempt to get the best of both worlds, the strategy called "Mixed" in Algo. 1 is a kind of *collaborative portfolio* approach: half of the available threads run the Dijkstra strategy and the other half run the Tarjan strategy. These two strategies can be combined as desired since they share the same kind of structural information.

3.5 Discussion.

All these strategies have one drawback since they use a local check to detect whether a state is alive or not: if one thread marks an SCC as DEAD, other threads already exploring the same SCC will not detect it and will continue to perform `unite` operations. Checking whether a state is DEAD in the global *uf* could be done for instance by changing the condition of line 40 of Algo. 1 into:

$$step.succ \neq \emptyset \wedge \neg uf.same\_set(step.src, Dead)$$

However such a change would be costly, as it would require as many accesses to the shared structure as there are transitions in the automaton. To avoid these additional accesses to *uf*, we propose to change the interface of `unite` so it returns an additional Boolean flag indicating that one of the two states is already marked as DEAD in *uf*. Then whenever `unite` is called and the extra bit is set, the algorithm can immediately backtrack the *dfs* stack until it finds a state that is not marked as DEAD.

3.6 Sketch of Proof

Since the Tarjan strategy is really close to the Dijkstra strategy, we only give the scheme of a proof[2] that the latter algorithm will necessarily terminate and will report a counterexample if and only if there is an accepting cycle in the automaton.

Theorem 1. For all automata A the emptiness check terminates.

[2] A complete proof can be found at: `http://www.lrde.epita.fr/~renault/publis/TACAS15.pdf`

Fig. 4 Pure Dijkstra: all threads perform a Dijkstra strategy on an automaton with $\mathcal{F} = \{\bullet, \circ\}$.

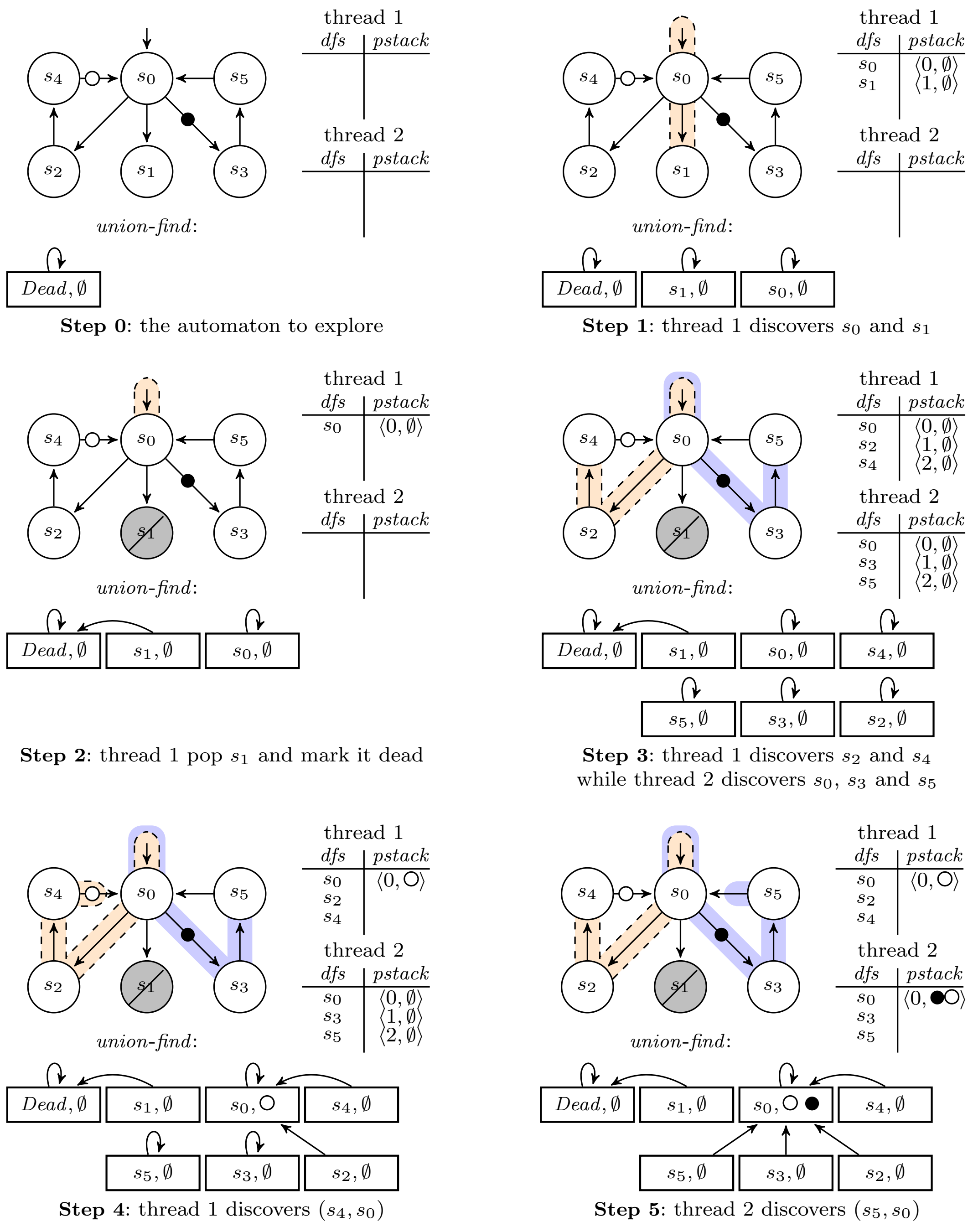
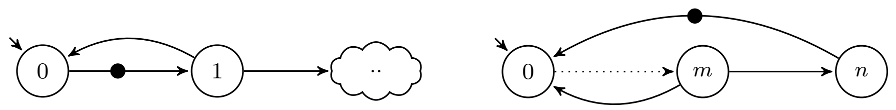


Fig. 5 Worst cases to detect accepting cycle using only one thread. The left automaton is bad for Tarjan since the accepting cycle is always found only after popping state 1. The right one disadvantages Dijkstra since the union of the states represented by dots can be costly.



Theorem 2. The emptiness check reports an accepting cycle iff $\mathscr{L}(A) \neq \emptyset$.

The theorem 1 is obvious since the emptiness check performs a DFS on a finite graph. Theorem 2 ensues from the invariants below which use the following notations. For any thread, n denotes the size of its *pstack* stack. For $0 \leq i < n$, S_i denotes the set of states in the same partial SCC represented by $pstack[i]$: $\forall i < n-1$

$$S_i = \left\{ q \in livenum \;\middle|\; \begin{array}{l} dfs[pstack[i].p].pos \leq livenum[q] \; \wedge \\ livenum[q] \leq dfs[pstack[i+1].p].pos \end{array} \right\}$$

$$S_{n-1} = \{q \in livenum \mid dfs[pstack[n-1].p].pos \leq livenum[q]\}$$

The following invariants hold for all lines of algorithm 1:
Invariant 1. *pstack* contains a subset of positions in *dfs*, in increasing order.
Invariant 2. For all $0 \leq i < n-1$, there is a transition with the acceptance marks $dfs[pstack[i+1].p].acc$ between S_i and S_{i+1}.
Invariant 3. For all $0 \leq i < n$, the subgraph induced by S_i is a partial SCC.
Invariant 4. If the class of a state inside the union-find is associated to $acc \neq \emptyset$, then the SCC containing this state has a cycle visiting *acc*. (Note: a state in the same class as *Dead* is always associated to $\emptyset$.)
Invariant 5. The first thread marking a state as DEAD has seen the full SCC containing this state.
Invariant 6. The set of DEAD states is a union of maximal SCC.
Invariant 7. If a state is DEAD it cannot be part of an accepting cycle.

These invariants establish both directions of Theorem 2: invariants 1–4 prove that when the algorithm reports a counterexample there exists a cycle visiting all acceptance marks; invariants 5–7 justify that when the algorithm exits without reporting anything, then no state can be part of a counterexample.

3.7 Counterexample extraction

An expected feature of model-checker is to report a counterexample when the property is violated. It is trivial for sequential NDFS-based emptiness checks because the stack is the counterexample [16]. For the parallel version `cndfs`, the counterexample is also given by the DFS stack of the detecting thread [21].

Extracting a counterexample from the SCC-based algorithms is harder because when the algorithm reports the existence of a counterexample we only know that some reachable partial SCC contains all acceptance marks. The DFS stack of the detecting thread gives a finite path to one state of the accepting SCC.

For all algorithms based on Dijkstra (sequential or parallel), when a counterexample is detected all the states of the accepting cycle have been marked as belonging to the same partial SCC. For instance, at step 5 of Figure 4, the detection of the closing edge (s_5, s_0) merges states s_0, s_2, s_3, s_4 and $s5$ in the same partition. Thus, we can restrict ourselves to these states to extract the accepting cycle. The procedure suggested by Couvreur et al. [17] can extract an accepting cycle by looking only at the states of this partial SCC.

When Tarjan-based algorithms report the existence of a counterexample, all the states of the accepting cycle are not necessarily united in the same partial SCC. For example, at step 7 of Figure 3, s_2 is not in the same class as states s_0, s_1, s_3, s_4 and s_5.

In order to apply Couvreur's algorithm [17] all these states must be merged in the same class. Hence, when a thread detects a counterexample, all threads must empty their DFS stack by repeatedly applying a variant of $\texttt{POP}_{Tarjan}$ without lines 18 and 22–25. The result of emptying the two stacks is shown at step 8 of Figure 3. At this point, the classes of the union-find contains at least all necessary states and the previous counterexample extraction technique can be applied.

4 Variations on Emptiness Checks

This section proposes two variations compatible with the emptiness checks strategies we presented. The first one is an architectural change that separates writers and readers of the union-find. The second one decomposes the property automaton based on the strength of its SCCs, and such that each part can be checked with the most appropriate emptiness check.

4.1 Asynchronous Emptiness Checks

Every operations on the union-find may lead to an update of the data structure. If many threads work on the same subset of states, we can therefore expect a high degree of contention. Here we present an adaptation of the previous algorithms that helps to control the number of readers and writers on the shared union-find data structure.

The main idea is to divide threads into two categories: (1) *producers* that will explore the automaton and compute structural information and, (2) *consumers* that will record this information into the union-find. *Producers* will only read the information hold by the union-find to detect DEAD states, while *consumers* will detect the existence of a counterexample as a side effect of updating the union-find. The architecture of this new algorithm is described in Figure 6.

Every time a structural information is computed by the *producer*, this information is stored into a dedicated lock-free queue [29]. In order to reduce the contention on this queue, we opted to provide one dedicated queue per *producer* while *consumers* share all queues.

With this architecture, we expect that the exploration can be faster since *producers* do not perform costly write operations on the union-find. Moreover, it also provides a canvas where we can easily adapt the number of *producers* and *consumers* according to the cost of successor computation (which could have an impact on the scalability of parallel algorithms).

The information stored in the queues is described by the *lfq_element* structure of Algorithm 2. It contains a field *info* that describes the type of operation to be performed by the *consumer*:

- UNITE means a union between two states; the *consumer* must unite states *src* and *dst* with the acceptance set *acc* inside of the shared union-find.
- MAKEDEAD means that an SCC has been fully visited. The *consumer* must unite state *src* with the artificial *Dead* state.

For the sake of clarity, let us suppose that we have an interface *lf_queue* that encapsulates all queues and offers the following methods:

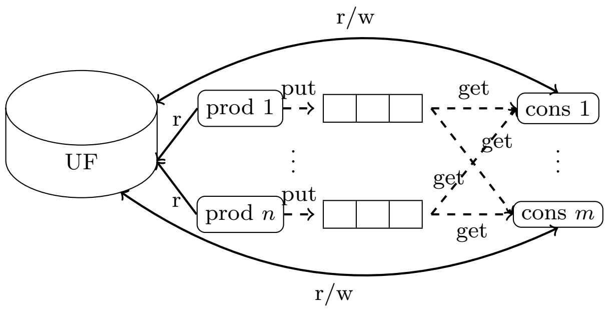
- `get()`: dequeues one *lfq_element* among all the queues. If no such element exists, this methods returns **null**.
- `put(`*lfq_element e*, *int tid*`)`: enqueues *e* to the queue associated to producer *tid*.

Algorithm 2 describes the main procedure of the one *consumer* thread. The pseudo-code is quite simple: every time the operation of line 8 returns an element different from **null**, this element is processed. At line 12–13, we can note that every state is systematically inserted into the union-find. Since producers cannot write into the union-find, this is require to ensure the validity of operation line 14.

Here we describe how to adapt Strategy 2 to this asynchronous architecture (we postpone the adaptation of Strategy 1 to the end of this section).

- The call to *uf*.`make_set(`q`)` is removed (line 2 of Strategy 2).
- The call to *uf*.`unite(`d, $dfs[r].src$, a`)` (line 14 of Strategy 2) is replaced by *queue*.`put(`$\{d$, $dfs[r].src$, UNITE, $a\}$`)` in order to transmit the structural information to *consumers*.

Fig. 6 Asynchronous parallel emptiness check architecture.



Algorithm 2: Consumer Main Procedure

```
1  Additional Shared Variables:
2    enum info { UNITE, MAKEDEAD}
3    struct lfq_element { src: Q,      dst: Q,
4                         type: info,  acc: $2^{\mathcal{F}}$ }
5  lfq_queue queue
6  EC$_{consumer}$()
7    while ¬ stop
8      lfq_element e ← queue.get()
9      if e = null
10       continue
11     if e.info = UNITE
12       uf.make_set(e.src)
13       uf.make_set(e.dst)
14       a ← uf.unite(e.src, e.dst, e.acc)
15       if a = $\mathcal{F}$
16         stop ← ⊤
17         report accepting cycle found
18     else
19       uf.unite(e.src, Dead)
```

- The call to *uf*.`unite(`*s.src*, *Dead*`)` in `markdead` (line 28 of Strategy 1) is replaced by *queue*.`put(`$\{d$, **null**, MAKEDEAD, $a\}$`)`.
- States that are removed from *livenum* by lines 29–32 of Strategy 1 are added to a local set (as described in previous work [35, Sec. 7]) to ensure that this strategy will not revisit these states in case they have not yet been united with *Dead* by a consumer.
- At line 31 of Algorithm 1, the call to `same_set` in the method `GET_STATUS` performs a write operation (for path-compression optimization) which the *producer* is not allowed to perform. Two solutions are feasible: (1) deactivate path compression for every `same_set` performed by *producers*, or (2) provide a new function `is_maybe_dead` that only look if the parent of an element is the artificial *Dead* state. We opted for the latter: while it avoid path-compression, it also decreases the load

on the structure by looking up only the parent and not following the path down to its representative. Since DEAD states are now locally available, we can only miss states already marked as DEAD by another *producer*.

Discussion. In this section we opted to present the asynchronous emptiness checks for the Dijkstra strategy. Nonetheless a similar adaptation can easily be done for the Tarjan strategy. The only difference concerns the termination: since the Tarjan strategy does not keep track of the acceptance sets of each partial SCC, *producers* cannot stop all threads when they end theirs exploration (line 52 of Algorithm 1). Indeed, if we do so, some counterexample can be missed because some structural information has yet to be processed by *consumers*. The algorithm can only end when one *producer* has finished its exploration and when its associated queue is empty.

4.2 Combination with Decomposition Technique

For some subclasses of automata, there exist emptiness check procedures that are more efficient [14]. For *terminal automata*, where an accepting run exists iff a terminal state is reachable, emptiness can be decided by a reachability search. In *weak automata* [14], there is no need to keep track of live states: states may be marked as DEAD as soon as the DFS backtracks.

In previous work [36], we suggested to exploit these dedicated algorithms by decomposing the property automaton according to the *type* of its SCCs. This decomposition produces automata of different subclasses (a.k.a. *strength*) that can be checked in parallel.

Before defining the strength of the property automaton, let us characterize the different types of SCCs. The type of an SCC is:

- *non accepting* if it does not contain any accepting cycle,
- *inherently terminal* if it contains only accepting cycles and is complete (i.e., for each letter, each state has an outgoing transition remaining in the same SCC),
- *inherently weak* if it contains only accepting cycles and it is not inherently terminal,
- *strong* if it is accepting and contains some non-accepting cycle.

These four types define a partition of the SCCs of an automaton.

We say that the strength of an automaton is:

- *inherently terminal* iff all its accepting SCCs are inherently terminal,
- *inherently weak* iff all its accepting SCCs are inherently terminal or inherently weak.
- *general* in all cases.

These three classes form a hierarchy where inherently terminal automata are inherently weak, which in turn are general. Note that the above constrains concern only accepting SCCs, but these automata may also contain non-accepting (transient) SCC.

Given a property automaton that mixes SCCs of different types, we can decompose it into three automata A_T, A_W and A_S that will catch respectively terminal, weak and strong behaviors. We denote T, W, and S, the set of all transitions belonging respectively to some terminal, weak, or strong SCC. For a set of transitions X, we denote $\mathrm{Pre}(X)$ the set of states that can reach some transition in X. We assume that $q^0 \in \mathrm{Pre}(X)$ even if X is empty or unreachable.

Property 1 *Let $A = \langle AP, Q, q^0, \delta, \{f_1, \ldots, f_n\}\rangle$ be a TGBA. Let $A_T = \langle AP, Q_T, q^0, \delta_T, \{\bullet\}\rangle$, $A_W = \langle AP, Q_W, q^0, \delta_W, \{\bullet\}\rangle$, $A_S = \langle AP, Q_S, q^0, \delta_S, \mathcal{F}\rangle$ be three automata (of respective strength: terminal, weak, and general) defined with:*

$$Q_T = \mathrm{Pre}(T)$$

$$\delta_T = \left\{ (s,c,\ell,d) \;\middle|\; \begin{array}{l} \exists (s,\_,\ell,d) \in \delta \textit{ with } s,d \in Q_T \\ \textit{and } \begin{cases} c = \{\bullet\} & \textit{if } (s,\_,\ell,d) \in T \\ c = \emptyset & \textit{otherwise} \end{cases} \end{array} \right\}$$

$$Q_W = \mathrm{Pre}(W)$$

$$\delta_W = \left\{ (s,c,\ell,d) \;\middle|\; \begin{array}{l} \exists (s,\_,\ell,d) \in \delta \textit{ with } s,d \in Q_W \\ \textit{and } \begin{cases} c = \{\bullet\} & \textit{if } (s,\_,\ell,d) \in W \\ c = \emptyset & \textit{otherwise} \end{cases} \end{array} \right\}$$

$$Q_S = \mathrm{Pre}(S)$$

$$\delta_S = \left\{ (s,c,\ell,d) \;\middle|\; \begin{array}{l} \exists\, (s,c',\ell,d) \in \delta \textit{ with } s,d \in Q_S \\ \textit{and } \begin{cases} c = c' & \textit{if } (s,\_,\ell,d) \in S \\ c = \emptyset & \textit{otherwise} \end{cases} \end{array} \right\}$$

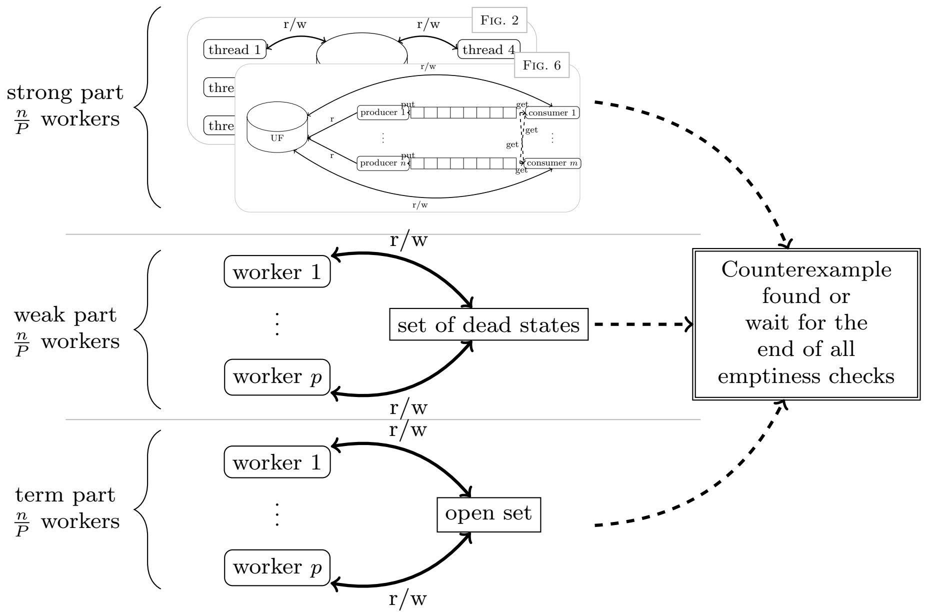
Then we have[3] *:*

$$\mathscr{L}(A) = \mathscr{L}(A_T) \cup \mathscr{L}(A_W) \cup \mathscr{L}(A_S).$$

Given a system *Sys*, we build three synchronous products ($A_T \otimes Sys$, $A_W \otimes Sys$, and $A_S \otimes Sys$) and check each of them for emptiness, using the appropriate dedicated emptiness check:

[3] Note that if A is unambiguous [7] we also have that $\mathscr{L}(A_T)$, $\mathscr{L}(A_W)$, and $\mathscr{L}(A_S)$ are pairwise disjoint.

Fig. 7 Architecture for the Strategy **S1** with n the number of threads and P the number of decomposed automata.



– For the terminal automaton and with the supposition that the system does not have deadlocks, a reachability algorithm is sufficient. In a parallel setting, such an algorithm is implemented using an open-set, that is traditionally composed of a hash set (containing all visited states) and a queue (containing all states to process).
  The algorithm starts with only the initial state in the queue and the hash table. Then, every thread can pop a state q from the queue and inserts all the successors of q not visited by any thread in both the hash table and the queue. The algorithm ends when all states have been visited, or when it reaches a state from a terminal SCC of the property automaton. In this latter case, since *Sys* does not have deadlock, a counterexample has been detected (even if it has not yet been computed).
– For the weak automaton, each thread performs a single swarmed DFS. When a state is popped, it is marked as DEAD in a shared hash table so other threads will not revisit it. When a thread detects a closing-edge reaching directly a state of its DFS stack, and if this state belongs to an accepting SCC of the property automaton, then a counterexample has been detected. The algorithm ends when a counterexample has been detected or when all states have been marked as DEAD.
– For the general automaton, we use the algorithms defined in Sections 3.2, 3.3, and 4.1.

Given N threads, we consider two strategies:

S1: if the property automaton can be decomposed into P automata, then each product will be checked using $\frac{N}{P}$ threads. Figure 7 describes the architecture of this strategy (which is a multicore adaptation of a previous work [36]).

S2: another approach aims to use the maximum number of threads for each product. In this case, the first one is checked using N threads. Then if no counterexample is found, N threads are used for the next product. In this strategy, the automata are ordered by strengths: terminal, then weak, and finally general. This way more complex emptiness checks are avoided whenever possible. Moreover since the three synchronous products do not have similar size, this strategy is expected to obtain a better speedup at each stage.

5 Related Work

We now compare our approach to recent parallel emptiness checks or parallel SCC-computations. For a more detailed overview of related work, we refer the interested reader to Bloemen's master thesis [10]. In this section, we focus on the following contenders, whose implementation is available respectively in DiVinE [4] and LTSmin [24]:

– `owcty` [15, 4]: a non-DFS based algorithm which does not work on-the-fly (this aspect has been improved for weak automata [5]). This algorithm uses a fixpoint to remove from the automaton all states that cannot lead to an accepting cycle.

Table 1 Comparison of parallel emptiness checks (or parallel SCC computation algorithms). $|T|$ is the number of transitions of the automaton to explore, $|Q|$ is the number of states, P is the number of thread used, and α is the inverse Ackermann function (which is the "almost-constant" complexity of union-find operations [40]).

Algorithm	Reference	On-the-fly	Complexity	Generalized	Lockless
`owcty`	[15, 4]	×	$O(\|T\|^2)$	?	×
`cndfs`	[21]	✓	$O(P \times \|T\|)$	×	×
`Lowe`	[28]	✓	$O(\|T\|^2)$	✓	×
`Bloemen`	[10, 11]	✓	$O(P \times \|T\| \times \|Q\|)$	✓	×
this paper	Sections 3–4	✓	$O(P \times \|T\| \times \alpha(\|Q\|))$	✓	✓

– `cndfs` [21]: an NDFS based algorithm compatible with on-the-fly exploration. The main idea is to swarm a classical, sequential NDFS. The usual blue and red colors are shared between threads, however since the coloring is highly dependent on the traversal order, synchronizations are required to ensure the correctness of the shared information.

The following two algorithms have been recently published, but no implementation have been released yet. They both rely on locks, while our implementation can be made lock-free.

– Lowe's algorithm [28]: a swarmed variant of Tarjan where each state is visited by a unique thread. A state is backtracked only once all its successors have been visited (maybe by another thread), and a special data structure is used to resolve deadlocks between threads waiting for states to be visited by other threads.
– Bloemen's algorithm (`UF-SCC`) [11]: it uses a shared union-find to perform a DFS in terms of SCCs (not states), works on-the-fly and supports generalized acceptance. It is an extension of algorithms we presented previously [37] (a preliminary version of this paper). Unlike our algorithms, UF-SCC tackles the problem of sharing LIVE states. For this purpose, the shared union-find is enriched by extra information: (1) the identities of threads currently visiting a given partial SCC, and (2) the set of states that have been fully visited for a partial SCC.

Table 1, derived from Bloemen [10], summarizes the worst-case complexity of all these algorithms. Among SCC-based algorithms, ours are those with the better complexities. They appear to be the only ones that do not require locking scheme, but Bloemen argues that his algorithm may also be implemented without lock [10].

While `cndfs` has the best worst-case complexity of Table 1, it does not support generalized acceptance conditions: a degeneralization is required, which leads to a bigger synchronized product. It is still unknown if `owcty` or `cndfs` can efficiently be extended to generalized acceptance marks in parallel settings. Indeed, the sequential versions of these algorithms exists for generalized Büchi acceptance marks [25, 41], but their complexity depends on the number of acceptance marks.

Finally, if we compare our algorithm to the two other SCC computation algorithms [28, 10], our technique is inappropriate when the automaton to explore consists in a unique, large, and non-accepting SCC. In that situation, the threads share useless information: the acceptance marks observed in the SCC may not help to discover a (non-existing) counterexample, and the SCC can only be marked as DEAD once it has been fully explored by one thread (the other threads are therefore superfluous, if not counterproductive).

6 Implementation and Benchmarks

After discussing worst-case complexities in the previous section, we now measure actual implementations.

Table 2 presents the models we use in our benchmark. The models are a subset of the BEEM benchmark [32], such that every type of model of the classification of Pelánek [33] is represented, and all synchronized products have a high number of states, transitions, and SCCs. Because there are too few LTL formulas supplied by BEEM, we opted to generate random formulas to verify on each model. We computed a total number of 3268 formulas that all require a general emptiness check.[4]

Among the 3268 formulas, 1706 result in products with the model having an empty language (the emptiness check may terminate before exploring the full product). All formulas were selected so that the sequential NDFS emptiness check of Gaiser and Schwoon [22] would take between 15 seconds and 30 minutes on an four Intel(R) Xeon(R) CPUX7460@ 2.66GHz with 128GB of RAM. This 24-core machine is also used for the following parallel experiments.

The sizes given in Table 2 are averaged over all the selected formulas, for each model.

[4] For a description of our setup, including selected models, formulas, and detailed results, see `http://www.lrde.epita.fr/~renault/benchs/STTT-2015/results.html`.

Table 2 Statistics about synchronized products having an empty language (✓) and non-empty one (×).

Model	Avg. States (✓)	Avg. States (×)	Avg. Trans. (✓)	Avg. Trans. (×)	Avg. SCCs (✓)	Avg. SCCs (×)
adding.4	5 637 711	7 720 939	10 725 851	14 341 202	5 635 309	7 716 385
bridge.3	1 702 938	3 114 566	4 740 247	8 615 971	1 701 048	3 106 797
brp.4	15 630 523	38 474 669	33 580 776	94 561 556	4 674 238	16 520 165
collision.4	30 384 332	101 596 324	82 372 580	349 949 837	347 535	22 677 968
cyclic-scheduler.3	724 400	1 364 512	6 274 289	12 368 800	453 547	711 794
elevator.4	2 371 413	3 270 061	7 001 559	9 817 617	1 327 005	1 502 808
elevator2.3	10 339 003	13 818 813	79 636 749	120 821 886	2 926 881	6 413 279
exit.3	3 664 436	8 617 173	11 995 418	29 408 340	3 659 550	8 609 674
leader-election.3	546 145	762 684	3 200 607	4 033 362	546 145	762 684
production-cell.3	2 169 112	3 908 715	7 303 450	13 470 569	1 236 881	1 925 909

All the approaches mentioned in Sections 3 and 4 have been implemented in Spot [19]. The union-find structure is lock-free and uses two common optimizations: "Immediate Parent Check", and "Path Compression" [31].

The seed used to choose a successor randomly depends on the thread identifier *tid* passed to `EC`. Thus our strategies have the same exploration order when executed sequentially; when the strategies are run in parallel, this order may be altered by information shared by other threads.

6.1 Comparison with state-of-the-art algorithms

Figure 8 presents the comparison of our prototype implementation in Spot against the `cndfs` algorithm implemented in LTSmin [24] and the `owcty` algorithm implemented in DiVinE 2.4 [4]. We selected `owcty` because it is reported to be the most efficient parallel emptiness check based on a non-DFS exploration, while `cndfs` is reported to be the most efficient based on a DFS [21]. Unfortunately, we were not able to compare our approach with the recent works of Lowe [28] and Bloemen et al. [10, 11] since no implementation have been released.

The state-spaces used in this benchmark are derived from DiVinE 2.4 in three different ways. `owcty` is the default algorithm for DiVinE, and uses DiVinE's own successor computation function. For the algorithms presented in this paper, and implemented in Spot, we use a version of DiVinE 2.4 patched by the LTSmin team[5] to compile the successor function of the system, and the product with the property automaton is performed on-the-fly. LTSmin's `cndfs` can be configured to use the very same successor function, however doing so exhibits several cases where the result of LTSmin disagrees with both Spot and DiVinE: this is apparently a bug in the on-the-fly product. To work around this issue, we precompiled the entire product for `cndfs`, as suggested by the LTSmin team.

From our original benchmark, we excluded 11 cases where `owcty` failed to answer within one hour, and 784 cases where LTSmin failed to precompile a product within one hour. The remaining 2475 cases are successfully (and consistently) solved by all algorithms within the one hour limit.

DiVinE and LTSmin implement all sorts of optimizations (like state compression, caching of successors, dedicated memory allocator...) while our implementation in Spot is still at a prototype stage. So in absolute time, the sequential version of `cndfs` is around 3 time faster. Since the implementations are different, we compare the average speedup of the parallel version of each algorithm against its sequential version.[6] The actual time can be found in the detailed results[4].

The left-hand side of Figure 8 shows those speedups, averaged for each model, for verified formulas (where the entire product has to be explored).

First, it appears that the Tarjan strategy's speedup is always lower than those of Dijkstra or Mixed for empty products. These low speedups can be explained by contention on the shared union-find data structure during `unite` operations. In an SCC of n states and m edges, a thread applying the Tarjan strategy performs m `unite` calls while applying Dijkstra one needs only $n-1$ `unite` invocations before they both mark the whole SCC as DEAD with a unique `unite` call.

Second, for all strategies we can distinguish two groups of models. For adding.4, bridge.3, exit.3, and leader-election.3, the speedups are quasi-linear. How-

[5] `http://fmt.cs.utwente.nl/tools/ltsmin/#divine`

[6] For `owcty` and our algorithms, the run time include the cost of generating the state-space, and of making the product with the property automaton; while `cndfs` is exploring a precomputed product. Although this sounds advantageous to `cndfs` in term of absolute execution time, it may actually not be the case when measuring the scalability of parallel algorithms: it is easier to obtain a good speedup if the cost of exploring the product automaton is high.

Fig. 8 Speedup of emptiness checks over the benchmark. The dashed line is the identity function, to compare our practical speedups to an imaginary algorithm whose speedup would be actual to the number of threads used.

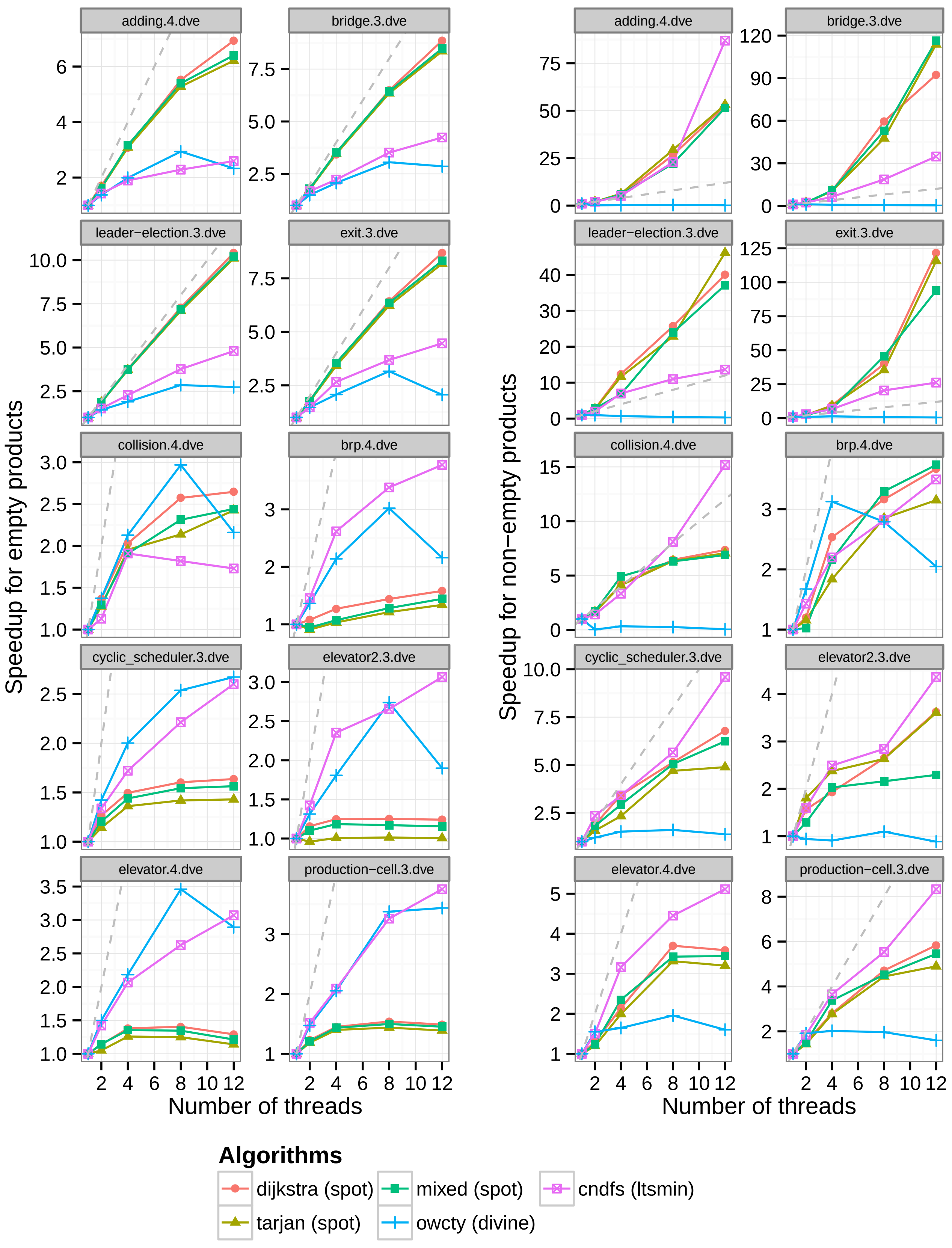


Table 3 Run times for single-threaded emptiness checks in seconds, and run times of their 12-thread version as a percentage of the single-threaded configuration. All 2475 formulas (violated and verified) were used. Note that the runtime of cndfs does not include the cost of generating the state-space[6].

	dijkstra		owcty		cndfs	
threads	1	12	1	12	1	12
adding.4	22 225s	8.6%	18 841s	47.9%	12 536s	24.4%
bridge.3	11 446s	6.4%	7 352s	58.6%	4 753s	16.6%
brp.4	18 179s	49.2%	15 425s	48.1%	3 711s	27.6%
collision.4	21 026s	24.6%	5 203s	54.7%	3 773s	40.2%
cy_sched	10 831s	40.5%	17 574s	62.6%	3 146s	28.9%
elevator.4	25 501s	54.3%	32 954s	53.4%	10 124s	27.1%
elevator2.3	26 725s	55.7%	19 032s	93.6%	4 786s	28.5%
exit.3	17 557s	6.8%	6 346s	52.4%	5 325s	15.7%
lead-el	2 031s	7.6%	1 999s	132.4%	549s	17.3%
prod-cel	21 249s	44.9%	37 831s	52.9%	8 939s	20.4%

ever for the other six models, the speedups are much more modest: it seems that adding new threads quickly yield no benefits. A look to absolute time (for the first group) shows that the Dijkstra strategy is 25% faster than `cndfs` using 12 threads where it was two time slower with only one thread.

A more detailed analysis reveals that products of the first group have many small SCC (organized in a tree shape) while products of the second group have a few big SCC. These big SCC have more closing edges: the union-find data structure is stressed at every `unite`. This confirms what we observed for the Tarjan strategy about the impact of `unite` operations.

The right-hand side of Figure 8 shows speedups for violated formulas. In these cases, the speedup can exceed the number of threads since the different threads explore the product in different orders, thus increasing the probability to report an accepting cycle earlier. The three different strategies have similar speedup for all models, however their profiles differ from `cndfs` on some models: they have better speedups on bridge.3, exit.3, and leader-election.3, but are worse on collision.4, elevator.4 and production-cell.3. The Mixed strategy exhibits speedups between those of Tarjan and Dijkstra strategies. Table 3 provides absolute run times for 1 thread, and the relative run times for 12 threads.

6.2 Impact of the variations

This section evaluates performances of the asynchronous emptiness check presented in Section 4.1 using our benchmark. We opted to implement the asynchronous variant for Dijsktra algorithm, and therefore, we compare it against the Dijkstra strategy of Section 3.3. The choice of Dijkstra over Tarjan is motivated by the better results obtained by the former strategy in Section 6.1. Moreover, basing the asynchronous check on Dijkstra allows it to stop earlier (as discussed in Section 4.1).

Figure 9 evaluates this approach with a varying number of consumer. The curves labeled by "x consumers" plot the a speedup of the asynchronous Dijkstra approach over the sequential Dijkstra strategy obtained using n threads (x consumers and $n-x$ producers). The curve labeled by "Dijkstra" plots the speedup of the parallel Dijkstra strategy for reference.

The asynchronous approach appears clearly inferior. We believe these poor results are due to more than just the overhead of handling the queues. The separation of producers and consumers can have two different consequences depending on whether the queues are mostly empty (too many consumers) or mostly full (too many producers). If there are too many consumers, most of them sit idle, waiting to empty the queue as soon as some information is produced: these idle consumers therefore lower the overall speedup. If on the other hand, there are too many producers, the delay between the time an information is produced and an information is consumed (i.e., stored in the union-find) causes more threads to produce information already present in some queue: this, in turn, yields even greater delays and overhead.

Even if our implementation of this approach is unsuccessful, it shows that the union-find data structure can be adapted to different types of architectures. We believe it might inspire the development of more successful variants.

Figures 10 and 11 evaluate the decomposition approach of Section 4.2. Among the 3268 formulas in our benchmark, 2406 generate automata with multiple SCC strengths (997 where the language of the product is empty, and 1409 where the language is non-empty). For these figures the experiments are restricted to these particular formulas.

Figure 10 shows the impact of the decomposition approaches on the speedup of parallel emptiness checks. There, we only use the 1822 formulas that all tools were able to compute. Strategies S1 and S2 outperform the speedup of Dijsktra and are competitive to `cndfs`.

For the variation S1, we observe a real improvement for both verified and violated formulas. On empty products, variation S1 is on average 33% faster than the Dijkstra strategy, while variation S2 is 23% faster. The superiority of S1 and S2 can be attributed to several factors.

Firstly, two thirds of the threads use specialized emptiness checks to explore weak and terminal auto-

Fig. 9 Impact of the number of consumers on the speedup of the asynchronous Dijkstra compared to the Dijkstra strategy with one thread.

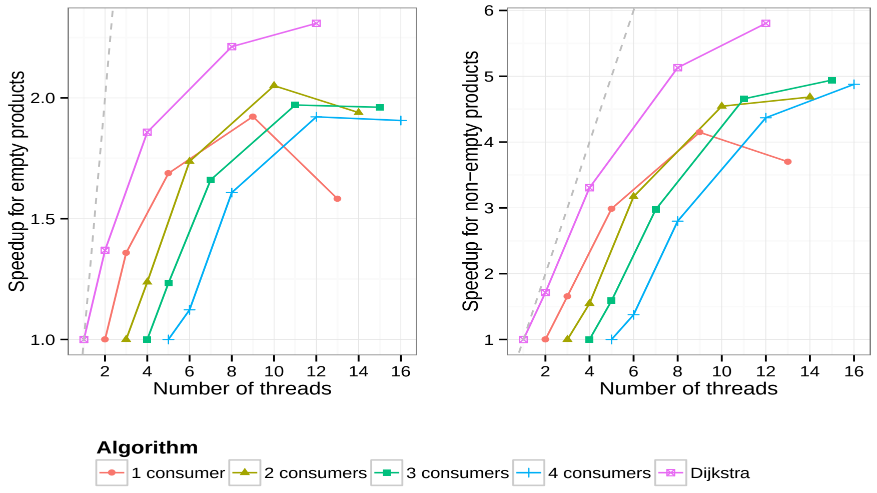
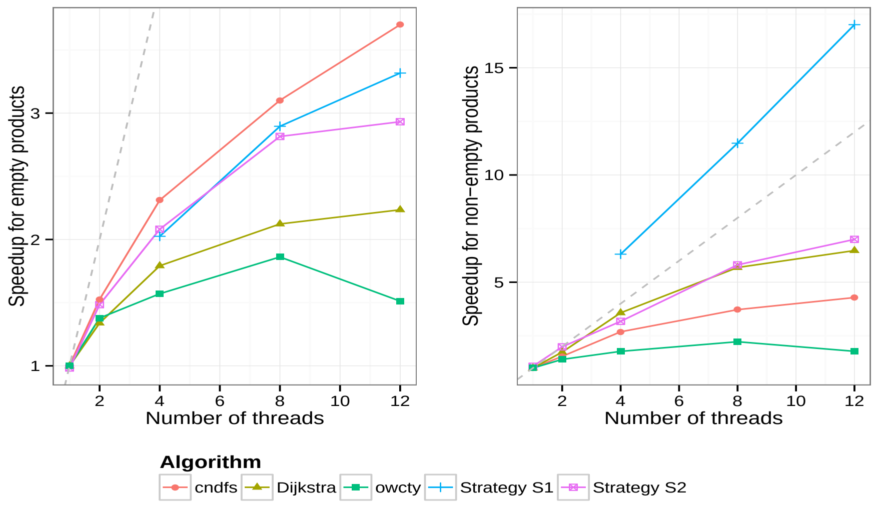


Fig. 10 Comparison of the decomposition approaches with state-of-the-art algorithms. The speedups of S1 and S2 are relative to the sequential version of Dijkstra, while the speedups of other tools are obtained by comparison to their corresponding versions with one thread.



mata. These checks are more efficient than emptiness checks that can handle the general case.

Secondly, each decomposed automaton is on the average half as big as the original one [36, Table 1], reducing both the time and memory usage of each thread.

Finally, in the case of strategy S1, the decomposition of the automaton constrains different groups of threads to explore different parts of the product. This can be seen as an improvement of the swarming, favoring a more uniform distribution of the paths covered by the different threads.

The scatter plot of Figure 11, compares S1 runtime against S2 over the 2406 formulas that generate automata with multiple SCC strengths (for 12 threads). It shows that neither S1 nor S2 provide globally better results, even if the speedup of S1 is better on the overall benchmark. A more detailed analysis would be necessary to understand the favorable cases for each strategy.

6.3 Implementation details

Besides the different algorithms and strategies that can be implemented, we have found two implementation choices having an important influence on the performance: the memory allocator, and the locking scheme used by the union-find.

Results shown in previous sections all use the native memory allocator (from the GNU Libc), and a lock-free implementation of the union-find. Table 4 evaluates the impact of two other options.

Firstly, when replacing the memory allocator by HOARD [8], an open source, multi-platform, and efficient memory allocator designed for multi-threaded programs, we observe an overall gain of 36% (up to 40% for some models like elevator2.3). Part of our observed gain is probably due to the fact that HOARD deals better with parallel allocations: in our implementation they are numerous because each thread allocate its own copy of each live state.

Secondly, while our union-find uses a lock-free implementation, Berman [9, Fig. 4.6] suggested that a fine grain locking scheme may remain efficient. Our results of using such a union-find are also reported Table 4. In our experiments, the union-find with fine-grain locking scheme performs slightly better than its lock-free implementation. Also, the implementation with fine-grain locking is much simpler and makes optimizations such as link-by-rank easier to implement.

In Figure 12, we evaluate these different setups in a setting very close to that of Bloemen [10, Fig. 6.9], where the Dijkstra strategy is used with different union-find implementations. The model `at.4` consists in a large, unique SCC, and the property automaton has been chosen so that the emptiness check will explore the entire model. In this scenario, the Dijkstra strategy can only terminate once one thread has fully explored the SCC, so the ideal run time should be independent on the number of threads. The cost of additional threads exhibited by Figure 12 is therefore caused by contention on the shared union-find, and on the memory allocation. In this example, the improvement achieved by using hoard is amplified when the fine-grain locking scheme is used. The link-by-rank optimization, although easier to implement, has little influence.

Note that these results about the locking scheme of the union-find differ from those of Bloemen [10, Fig. 6.9], where the lock-free implementation does not exhibit such an overhead, but the strategy with locks does have some overhead. A more detailed investigation would be necessary to explain these differences.

7 Conclusion

We have presented some new parallel emptiness checks based on an SCC enumeration. Our approach departs from state-of-the-art emptiness checks since it is neither BFS-based nor NDFS-based. Instead it parallelizes SCC-based emptiness checks that are built over

Fig. 11 Comparison of S1 and S2 for the decomposition

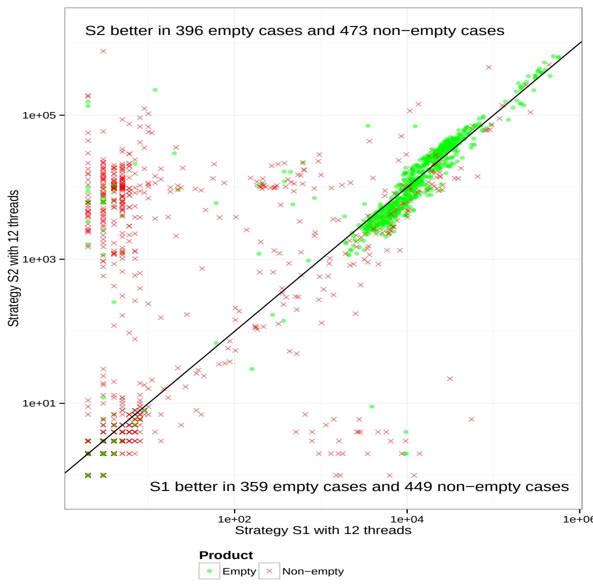
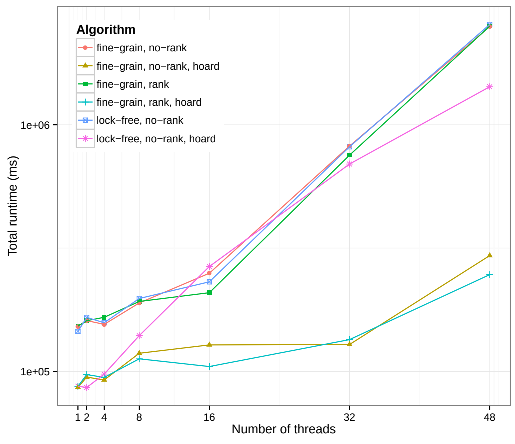


Table 4 Evaluation of implementation choices. The values give the total running time (in seconds) taken by the Dijkstra strategy for 12 threads, to process the 1706 verified formulas. (FG=Fine-grain)

Model	Dijkstra (sec)	+ FG	+ hoard	+ hoard + FG
adding.4	5 101	-7%	-31%	-29%
bridge.3	187	-2%	-22%	-29%
brp.4	6 697	+1%	-35%	-29%
collision.4	2 719	-15%	-36%	-28%
cy_sched.3	3 996	-4%	-38%	-34%
elevator.4	9 040	-4%	-33%	-32%
elevator2.3	9 637	-2%	-40%	-33%
exit.3	1 142	-3%	-24%	-31%
leader-el.3	929	-2%	-38%	-37%
prod-cell.3	4 664	-7%	-42%	-40%
Total	44 117	-4%	-36%	-32%

Fig. 12 Evaluation of various union-find implementations on the execution time of the Dijkstra strategy for the emptiness check of `at.4`.



a single DFS. Our approach supports generalized Büchi acceptance, and requires no synchronization points nor repair procedures. We therefore answer positively to the question raised by Evangelista et al. [21]: "Is the design of a scalable linear-time algorithm without repair procedures or synchronization points feasible?".

The core of our algorithms relies on a union-find data structure (possibly lock-free) to share structural information between multiple threads. The use of a union-find seems adapted to this problem, and yet it had never been used for parallel emptiness checks until the publication of the previous version of this paper at TACAS'15 [37]. Since then, Bloemen [10] explored a variation of these algorithms.

As suggested in our previous work [37], we have now investigated two variations. In the first one, we isolated threads writing in the union-find with the hope that it would reduce the contention on the shared union-find data structure. While our implementation was not successful, it shows that the versatile union-find can be easily adapted into various different architectures: this might inspire future variations using for instance job stealing or in a distributed setup. The second variant mixes these new parallel emptiness checks with the decomposition of property automaton according to its SCC strengths as suggested in a previous paper [36]. This decomposition can be seen as an improvement of the swarming technique: the decomposition favors a more uniform distribution of the paths covered by the different threads. This decomposition technique could easily be applied to other emptiness checks. Experiments showed a significant increase of performances.

In some future work, we would like to investigate more variations of our algorithms. For instance could the information shared in the union-find be used to better direct the DFS performed by the Dijkstra or Tarjan strategies and helps to balance the exploration of the automaton by the various threads? We would also like to implement Gabow's algorithm that we presented in a sequential context [35] in this same parallel setup.

References

1. R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In STC'94, pp. 370–380, 1994.
2. J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In ASE'03, pp. 106–115. IEEE Computer Society, 2003.
3. J. Barnat, L. Brim, and J. Chaloupka. From distributed memory cycle detection to parallel LTL model checking. In FMICS'05, vol. 133 of ENTCS, pp. 21–39, 2005.
4. J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core — A Parallel LTL Model-Checker. In ATVA'08, vol. 5311 of LNCS, pp. 234–239. Springer, 2008.
5. J. Barnat, L. Brim, and P. Ročkai. A time-optimal on-the-fly parallel algorithm for model checking of weak LTL properties. In ICFEM'09, vol. 5885 of LNCS, pp. 407–425, 2009. Springer.
6. J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. International Journal on Software Tools for Technology Transfer, 12(2):139–153, 2010.
7. M. Benedikt, R. Lenhardt, and J. Worrell. LTL model checking of interval markov chains. In TACAS'13, vol. 7795 of LNCS, pp. 32–46. Springer, 2013.
8. E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. Journal of the ACM, pp. 117–128, Nov. 2000.
9. I. Berman. Multicore Programming in the Face of Metamorphosis: Union-Find as an Example. Master's thesis, Tel-Aviv University, School of Computer Science, 2010.
10. V. Bloemen. On-the-fly parallel decomposition of strongly connected components. Master's thesis, University of Twente, June 2015.
11. V. Bloemen, A. Laarman, and J. van de Pol. Multi-core On-the-fly SCC Decomposition. In PPoPP'16. ACM, 2016.
12. L. Brim, I. Černá, P. Krcal, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In FSTTCS'01, pp. 96–107. Springer, 2001.
13. L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed LTL model-checking. In FMCAD'04, vol. 3312 of LNCS, pp. 352–366. Springer, November 2004.
14. I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In MFCS'03, vol. 2747 of LNCS, pp. 318–327, Aug. 2003. Springer.
15. I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In SPIN'03, vol. 2648 of LNCS, pp. 49–73. Springer, May 2003.
16. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In CAV'90, vol. 531 of LNCS, pp. 233–242. Springer, 1991.
17. J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In SPIN'05, vol. 3639 of LNCS, pp. 143–158. Springer, Aug. 2005.
18. E. W. Dijkstra. EWD 376: Finding the maximum strong components in a directed graph. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF`, May 1973.
19. A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In MASCOTS'04, pp. 76–83, Oct. 2004. IEEE Computer Society Press.
20. S. Evangelista, L. Petrucci, and S. Youcef. Parallel nested depth-first searches for LTL model checking. In ATVA'11, vol. 6996 of LNCS, pp. 381–396. Springer, 2011.
21. S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved multi-core nested depth-first search. In ATVA'12, vol. 7561 of LNCS, pp. 269–283. Springer, 2012.
22. A. Gaiser and S. Schwoon. Comparison of algorithms for checking emptiness on Büchi automata. In MEMICS'09, vol. 13 of OASICS. Schloss Dagstuhl,

Leibniz-Zentrum fuer Informatik, Germany, Nov. 2009.
23. G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. IEEE Transaction on Software Engineering, 37(6):845–857, 2011.
24. G. Kant, A. W. Laarman, J. J. G. Meijer, J. C. van de Pol, S. C. C. Blom, and T. van Dijk. Ltsmin: High-performance language-independent model checking. In Tools and Algorithms for the Construction and Analysis of Systems, vol. 9035 of LNCS, pp. 692–707. Springer, London, April 2015.
25. Y. Kesten, A. Pnueli, and L. on Raviv. Algorithmic verification of linear temporal logic specifications. In ICALP'98, vol. 1443 of LNCS, pp. 1–16. Springer, 1998.
26. A. Laarman and J. van de Pol. Variations on multi-core nested depth-first search. In PDMC'11, pp. 13–28, 2011.
27. A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-core nested depth-first search. In ATVA'11, vol. 6996 of LNCS, pp. 321–335, October 2011. Springer.
28. G. Lowe. Concurrent depth-first search algorithms based on Tarjan's algorithm. pp. 1–19. Springer, 2015.
29. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In PODC'96, pp. 267–275, 1996. ACM.
30. E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. Information Processing Letters, 49(1):9–14, Jan. 1994.
31. M. M. A. Patwary, J. R. S. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. In SEA'10, vol. 6049 of LNCS, pp. 411–423. Springer, 2010.
32. R. Pelánek. BEEM: benchmarks for explicit model checkers. In SPIN'07, vol. 4595 of LNCS, pp. 263–267. Springer, 2007.
33. R. Pelánek. Properties of state spaces and their applications. International Journal on Software Tools for Technology Transfer, 10:443–454, 2008.
34. J. H. Reif. Depth-first search is inherently sequential. Information Processing Letters, 20:229–234, 1985.
35. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In LPAR'13, vol. 8312 of LNCS, pp. 668–682. Springer, Dec. 2013.
36. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Strength-based decomposition of the property Büchi automaton for faster model checking. In TACAS'13, vol. 7795 of LNCS, pp. 580–593. Springer, 2013.
37. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Parallel explicit model checking for generalized Büchi automata. In TACAS'15, vol. 9035 of LNCS, pp. 613–627. Springer, Apr. 2015.
38. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In TACAS'05, vol. 3440 of LNCS, Apr. 2005. Springer.
39. R. Tarjan. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.
40. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. Journal of the ACM, 22(2):215–225, Apr. 1975.
41. H. Tauriainen. Nested emptiness search for generalized Büchi automata. In ACSD'04, pp. 165–174. IEEE Computer Society, June 2004.