Parallel Explicit Model Checking for Generalized Büchi Automata

E. Renault[1,2,3], A. Duret-Lutz[1], F. Kordon[2,3], D. Poitrenaud[3,4]

[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

Abstract. We present new parallel emptiness checks for LTL model checking. Unlike existing parallel emptiness checks, these are based on an SCC enumeration, support generalized Büchi acceptance, and require no synchronization points nor repair 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 being checked. Our prototype implementation has encouraging performances: the new emptiness checks have better speedup than existing algorithms in half of our experiments.

1 Introduction

The 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).

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 [26, 10, 24] for surveys).

Recently, parallel (or distributed) emptiness checks have been proposed [6, 2, 9, 7, 3, 4]: they are mainly based on a *Breadth First Search* (BFS) exploration which scales better than DFS [23]. Multicore adaptations of these algorithms with lock-free data structure have been discussed, but not evaluated, by Barnat et al. [5].

Recent publications show that NDFS-based algorithms combined with the *swarming* technique [16] scale better in practice [13, 18, 17, 14]. 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, synchronization points (if a state is handled by multiple threads in the nested DFS, its status is only updated after all threads have finished) or recomputing procedures (to resolve conflicts a posteriori using yet another DFS) are required. So far, attempts to design scalable parallel DFS-based emptiness check that does not require such mechanisms have failed [14].

This paper proposes 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.

Section 2 defines TGBAs and introduces our notations. Section 3 presents our two SCC-based strategies. Finally, Section 4 compares emptiness checks based on these new strategies against existing algorithms.

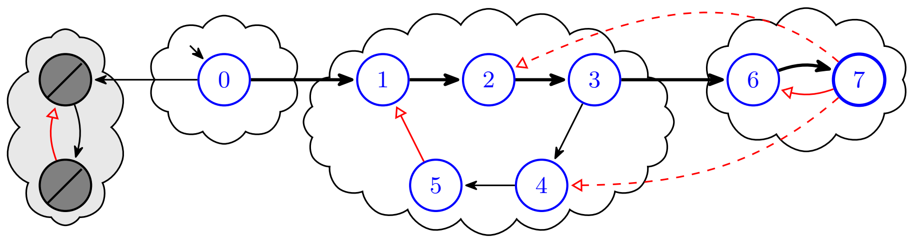
2 Preliminaries

A *TGBA* is a tuple $A = \langle Q, q^0, \delta, \mathcal{F} \rangle$ where Q is a finite set of states, q^0 is a designated initial state, $\mathcal{F}$ is a finite set of acceptance marks, and $\delta \subseteq Q \times 2^{\mathcal{F}} \times Q$ is the (non-deterministic) transition relation where each transition is labelled by a subset of acceptance marks. Let us note that in a real model checker, transitions (or states) of the automata would be labeled by atomic propositions, but we omit this information as it is not pertinent to emptiness check algorithms.

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 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, i.e. the language of A is non-empty ($\mathscr{L}(A) \neq \emptyset$).

Fig. 1. LIVE states are numbered by their live number, dead states are stroke. Clouds represents SCC 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.



3 Generalized Parallel Emptiness Checks

In a previous work [24] we presented sequential emptiness checks for generalized Büchi automata derived from the SCC enumeration algorithms of Tarjan [27] and Dijkstra [11], and a third one using a union-find data-structure. This section adapts these algorithms to a parallel setting.

The sequential versions of Tarjan-based and Dijkstra-based emptiness checks both have very similar structures: they explore the automaton using a single DFS to search for accepting SCCs 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 DFS stack leading to a LIVE state is called a *closing edge*. Figure 1 illustrates some of these concepts.

These two algorithms differ in the way they propagate information about currently visited SCCs, and when they detect accepting SCCs. A Tarjan-based emptiness check propagates information during backtrack, and may only find accepting SCC when its *root* is popped. (The *root* of an SCC is the 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 [24], 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 be used as well 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* tech-

Algorithm 1: Main procedure

```
Shared Variables:
  A: TGBA of ⟨Q, q⁰, δ, 𝓕⟩
  stop: boolean
  uf: union-find of ⟨ Q ∪ Dead, 2^𝓕 ⟩

Global Structures:
  struct Step { src: Q,      acc: 2^𝓕,
                pos: int,    succ: 2^δ }
  struct Transition {src: Q, acc: 2^𝓕
                     dst: Q}
  enum Strategy { Mixed,   Tarjan,
                  Dijkstra}
  enum Status { LIVE,    DEAD,
                UNKNOWN}
Local Variales:
  dfs: stack of ⟨Step⟩
  live: stack of ⟨ Q ⟩
  livenum: hashmap of ⟨ Q, int ⟩
  pstack: stack of ⟨ P ⟩

main(str: Strategy)
  stop ← ⊥
  uf.make_set(⟨ Dead, ∅ ⟩)
  if str ≠ Mixed
    EC(str, 1) ∥ ... ∥ EC(str, n)
  else
    str ← Dijkstra
    EC(str, 1) ∥ ... ∥ EC(str, ⌊n/2⌋)
    str ← Tarjan
    EC(str, 1+⌊n/2⌋) ∥ ... ∥ EC(str, n)
  Wait for all threads to finish

GET_STATUS(q ∈ Q) → Status
  if livenum.contains(q)
    return LIVE
  else if uf.contains(varq) ∧
          uf.same_set(q, Dead)
    return DEAD
  else
    return UNKNOWN

EC(str: Strategy, tid: int)
  seed(tid) // Random Number Gen.
  PUSH_str(∅, q⁰)
  while ¬ dfs.empty() ∧ ¬ stop
    Step step ← dfs.top()
    if step.succ ≠ ∅
      Transition t ← 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(step)
  stop ← ⊤
```

nique: each thread execute the same algorithm, but explores the automaton in a different order [16]. 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-based and Dijkstra-based parallel emptiness checks.

All threads share the automaton A to explore, 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 the union-find data-structure [20]. 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 this data structure has been 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 are in the same class.

As suggested by Anderson and Woll [1], we implement a thread safe version of this union-find structure using *compare-and-swap* since it relies on linked lists and an hash table.

The original sequential algorithms maintain a stack of LIVE states in order to mark all states of an explored SCC as DEAD. In our previous work [24], we suggested to use a union-find data structure 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 obviously 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 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 [19]).
- 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 that will be executed by all threads. The order of the successors is chosen 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 LIVE states correspond to closing edges. This generic DFS is adapted to the Tarjan and Dijkstra strategies by calling $\texttt{PUSH}_{str}$ on new states, $\texttt{UPDATE}_{str}$ on closing edges, and $\texttt{POP}_{str}$ 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 therefore only has to wait for all threads to terminate.

3.2 The Tarjan Strategy

Strategy 1 shows how the generic canvas is refined to implement the Tarjan strategy. In this algorithm, 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*.

These *lowlinks* are updated either when a closing edge is detected in the $\texttt{UPDATE}_{Tarjan}$ method (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 all the SCC associated to this root is DEAD, and also locally we remove all these states from *live* and *livenum* using the `markdead` function.

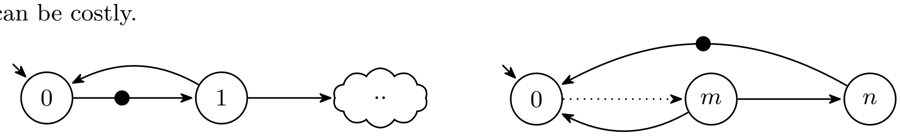
Strategy 1: Tarjan

```
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 ∈ Step)
    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)
```

Strategy 2: Dijkstra

```
struct P {p : int, acc : 2^F}

PUSH_Dijkstra(acc ∈ 2^F, q ∈ Q)
    uf.make_set(q)
    p ← livenum.size()
    livenum.insert(⟨ q, p ⟩)
    pstack.push(⟨dfs.size(), ∅ ⟩)
    dfs.push( ⟨ q, acc, p, succ(q)⟩ )

UPDATE_Dijkstra(acc ∈ 2^F, d ∈ Q)
    dpos ← livenum.get(d)
    ⟨r,a⟩ ← pstack.top()
    a ← a ∪ acc
    while dpos < dfs[r].pos
        ⟨r, la⟩ ← pstack.pop()
        a ← a ∪ dfs[r].acc ∪ la
        a ← uf.unite(d, dfs[r].src, a)
    pstack.top().acc ← a
    if a = F
        stop ← ⊤
        report accepting cycle found

POP_Dijkstra(s ∈ Step)
    dfs.pop()
    if pstack.top().p = dfs.size()
        pstack.pop()
        markdead(s)
    else
        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)
```

Fig. 2. 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.



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`). The next strategy performs fewer calls to `unite`.

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 the way states are 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 that we can later retrieve 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 only when a closing edge is detected, but not when backtracking a non-root as done in Tarjan. 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 $\texttt{UPDATE}_{Dijkstra}$. Note that the *dfs* stack has to be addressable like an array during this operation.

As it is presented, $\texttt{UPDATE}_{Dijkstra}$ calls `unite` only when a potential root is discovered not 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 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 necessarily equal to $n - 1$. This is better than the Tarjan-based version, but it also means we share less information between threads.

3.4 The Mixed Strategy

Figure 2 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. 2, 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 $1 \ldots m$, 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 $0 \ldots m-1$ 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 information.

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 43 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.

Moreover these strategies only report the existence of an accepting cycle but do not extract it. When a thread detects an accepting cycle, it can stop the others threads and can optionally launch a sequential counterexample computation [10]. Nonetheless, when performing a Dijkstra strategy the extraction can be limited to the states that are already in the union-find. The search of the accepting cycle can also be restricted to states whose projection are in the same SCC of the property automaton.

3.5 Sketch of Proof

Due to lack of space, and since the Tarjan strategy is really close to the Dijkstra strategy, we only give the scheme of a proof[1] that the latter algorithm will 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.

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*$[i]$:

$$S_i = \left\{ q \in \mathit{livenum} \;\middle|\; \begin{array}{r} \mathit{dfs}[\mathit{pstack}[i].p].\mathit{pos} \leq \mathit{livenum}[q] \\ \mathit{livenum}[q] \leq \mathit{dfs}[\mathit{pstack}[i+1].p].\mathit{pos} \end{array} \right\} \quad \text{for } i < n-1$$

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

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 $\mathit{dfs}[\mathit{pstack}[i+1].p].\mathit{acc}$ 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 $\mathit{acc} \neq \emptyset$, 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 a 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.

4 Implementation and Benchmarks

Table 1 presents the models we use in our benchmark and gives the average size of the synchronized products. The models are a subset of the BEEM benchmark [21], such that every type of model of the classification of Pelánek [22] is represented, and all synchronized products have a high number of states, transitions, and SCC. Because there are too few LTL formulas supplied by BEEM,

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

Table 1. 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-sched	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-el.3	546 145	762 684	3 200 607	4 033 362	546 145	762 684
prod-cell.3	2 169 112	3 908 715	7 303 450	13 470 569	1 236 881	1 925 909

we opted to generate random formulas to verify on each model. We computed a total number of 3268 formulas.[2]

The presented algorithms deal with any kind of generalized Büchi automata, but there exists specialized algorithms for subclasses of Büchi automata. For instance the verification of a safety property reduces to a reachability test. Similarly, persistent properties can be translated into automata where SCC cannot mix accepting cycles with non-accepting cycles [8] and for which a simpler emptiness check exists. Our benchmark contains only non-persistent properties, requiring a general emptiness check.

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 [15] 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.

All the approaches mentioned in Section 3 have been implemented in Spot [12]. The union-find structure is lock-free and uses two common optimizations: "Immediate Parent Check", and "Path Compression" [20].

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; otherwise this order may be altered by information shared by other threads.

Figure 3 presents the comparison of our prototype implementation in Spot against the `cndfs` algorithm implemented in LTSmin and the `owcty` algorithm implemented in DiVine 2.4. We selected `owcty` because it is reported to be the

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

most efficient parallel emptiness check based on a non-DFS exploration, while `cndfs` is reported to be the most efficient based on a DFS [14].

We generate the corresponding system automata using the version of DiVinE 2.4 patched by the LTSmin team.[3] For each emptiness check, we limit the execution time to one hour: all the algorithms presented in this paper proceess the 3268 synchronized products within this limit while `owcty` fails over 11 cases and `cndfs` fails over 784 cases. 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[5] than our prototype implementation which is competitive to DiVinE. Since the implementations are different, we therefore compare the average speedup of the parallel version of each algorithm against its sequential version. The actual time can be found in the detailed results[2].

The left-hand side of Figure 3 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. However 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.

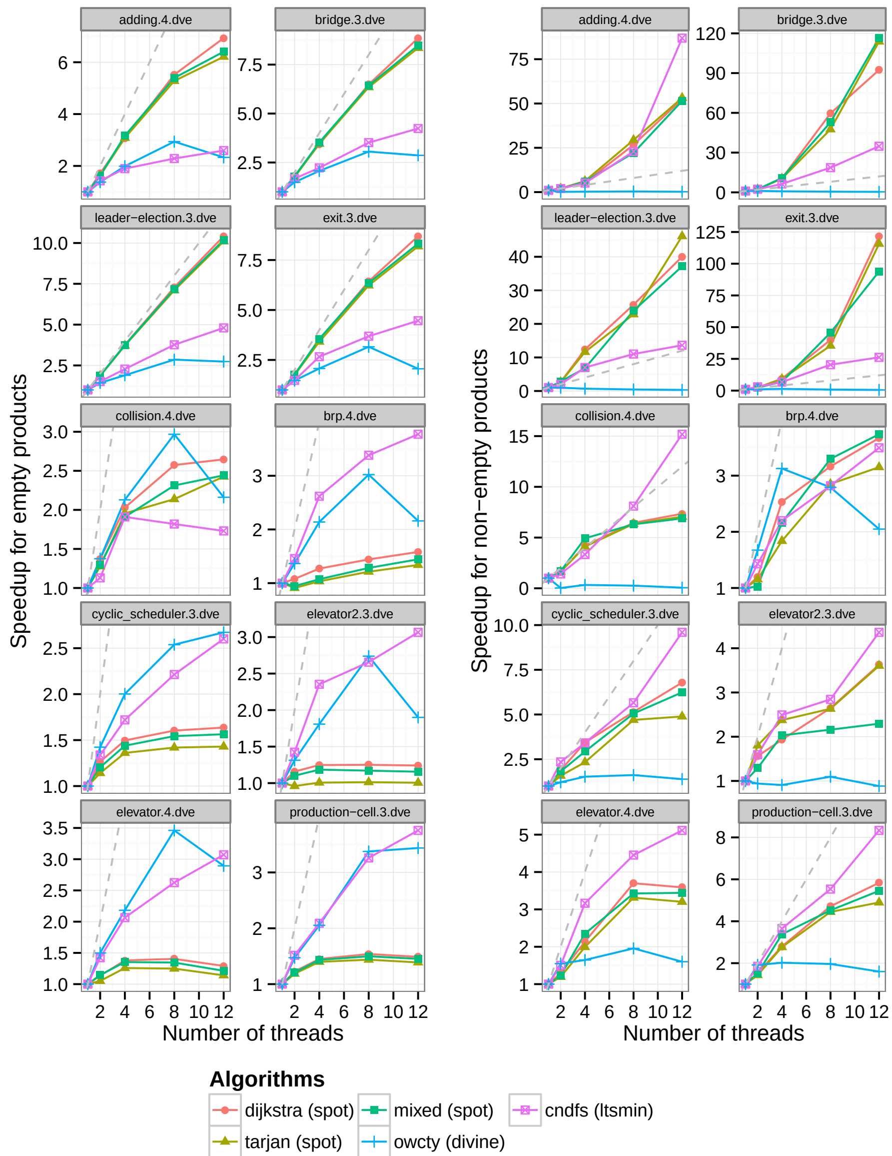
The right-hand side of Figure 3 shows speedups for violated formulas. In these cases, the speedup can exceed the number of threads since the different threads explore the product in different orders, thus increasing the probability to report an accepting cycle earlier. The three different strategies have comparable speedup for all models, however their profiles differ from `cndfs` on some models:

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

[4] This figure can be zoomed in color in the electronic version.

[5] Note that the time measured for `cndfs` does not includes the on-the-fly generation of the product (it is precalculated because doing the on-the-fly product in LTSmin exhibits a bug) while the time measured for the others includes the generation of the product.

Fig. 3. Speedup of emptiness checks over the benchmark. [4]



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 shows speedups between those of Tarjan and Dijkstra strategies.

5 Conclusion

We have presented some first and 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 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. [14]: "Is the design of a scalable linear-time algorithm without repair procedures or synchronisation points feasible?". Our prototype implementation has encouraging performances: the new emptiness checks have better speedup than existing algorithms in half of our experiments, making them suitable for portfolio approaches.

The core of our algorithms relies on a union-find (lock-free) data structure to share structural information between multiple threads. The use of a union-find seems adapted to this problem, and yet it has never been used for parallel emptiness checks (and only recently for sequential emptiness checks [24]): we believe that this first use might stimulate other researchers to derive new emptiness checks or ideas from it.

In some future work, we would like to investigate different 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 help 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 [24] in this same parallel setup. Changing the architecture, we would like to explore how the union-find data structure could be adapted to develop asynchronous algorithms where one thread could call `unite` without waiting for an answer. Another topic is to explore the use of SCC strengths [25] to improve parallel emptiness checks.

References

1. R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In Proc. 23rd ACM Symposium on Theory of Computing, 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'04, vol. 133 of ENTCS, pp. 21–39, 2005.
4. 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.
5. J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. STTT, 12(2):139–153, 2010.
6. L. Brim, I. Černá, P. Krcal, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In FSTTCS'01, pp. 96–107, 2001.

7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. IEEE Transaction on Software Engineering, 37(6):845–857, 2011.
17. A. Laarman and J. van de Pol. Variations on multi-core nested depth-first search. In PDMC, pp. 13–28, 2011.
18. 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.
19. 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.
20. 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.
21. R. Pelánek. BEEM: benchmarks for explicit model checkers. In SPIN'07, vol. 4595 of LNCS, pp. 263–267. Springer, 2007.
22. R. Pelánek. Properties of state spaces and their applications. International Journal on Software Tools for Technology Transfer (STTT), 10:443–454, 2008.
23. J. H. Reif. Depth-first search is inherently sequential. Information Processing Letters, 20:229–234, 1985.
24. 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.
25. 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.
26. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In TACAS'05, vol. 3440 of LNCS, Apr. 2005. Springer.
27. R. Tarjan. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.