Three SCC-Based Emptiness Checks for Generalized Büchi Automata

Etienne Renault[1,2], Alexandre Duret-Lutz[1],
Fabrice Kordon[2], and Denis Poitrenaud[2,3]

[1] LRDE, EPITA, Kremlin-Bicêtre, France
[2] LIP6/MoVe, Université Pierre & Marie Curie, Paris, France
[3] Université Paris Descartes, Paris, France

Abstract. The automata-theoretic approach for the verification of linear time properties involves checking the emptiness of a Büchi automaton. However generalized Büchi automata, with multiple acceptance sets, are preferred when verifying under weak fairness hypotheses. Existing emptiness checks for which the complexity is independent of the number of acceptance sets are all based on the enumeration of Strongly Connected Components (SCCs).

In this paper, we review the state of the art SCC enumeration algorithms to study how they can be turned into emptiness checks. This leads us to define two new emptiness check algorithms (one of them based on the Union-Find data structure), introduce new optimizations, and show that one of these can be of benefit to a classic SCCs enumeration algorithm. We have implemented all these variants to compare their relative performances and the overhead induced by the emptiness check compared to the corresponding SCCs enumeration algorithm. Our experiments shows that these three algorithms are comparable.

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 by the system if this product is empty.

Usually, a Büchi automaton is used to represent the property, and a Kripke structure represents the model. However, it is possible to use generalized Büchi automata (with several acceptance sets) to represent the property in a more concise way, and such generalized acceptance condition can also be used on the model to express weak fairness hypotheses on the system. In this work, we further generalize the above approach using Transition-based Generalized Büchi Automata (TGBA).

An emptiness check is an algorithm deciding whether such an automaton is empty. A Büchi automaton is non-empty if it accepts an infinite word, i.e., if it contains a lasso-shaped run: a finite prefix followed by an accepting cycle. Most

K. McMillan, A. Middeldorp, and A. Voronkov (Eds.): LPAR-19, LNCS 8312, pp. 668–682, 2013.

explicit emptiness checks are based on a DFS exploration of the automaton; they can be classified in two families. *Nested Depth First Search* algorithms [3] use a second DFS to detect the accepting cycle: if the automaton has multiple acceptance sets, this approach requires either a degeneralization, or multiple nested DFS. The second family are algorithms based on the enumeration of *Strongly Connected Components* (SCC), to find SCCs that contain accepting cycles. In these algorithms the number of times a state or transitions is visited is independent on the number of acceptance sets.

In this paper, we review the existing SCC enumeration algorithms to study how they can be adapted to become emptiness checks. To be of practical use in a model checker, we would like such emptiness checks to:

- support generalized Büchi acceptance [5, 12] (without requiring a degeneralization, or multiple passes on the automaton),
- support an on-the-fly construction of the automaton so that we do not need to construct unexplored parts of the product,
- be compatible with the bit-state hashing [15] and state-space caching [13] techniques to deal cases where memory is a critical resource.

We focus on three SCC algorithms which we shall refer to as Tarjan [19], Dijkstra [6], Gabow [8]. Tarjan is the most well-known algorithm to compute SCC and it has been extended by Geldenhuys and Valmari [11] to check the emptiness of (non-generalized) Büchi automata. Dijkstra's SCC-enumeration algorithm is a little less known, but has served as the base for several generalized emptiness checks [12, 5, 1, 10]. Essentially, both these algorithms partition the set of states according to the SCCs, and have a complexity that is linear with respect to the size of the graph. An efficient data structure to deal with the construction of a partition is the *Union-Find* [20] and Gabow [8] has suggested an algorithm to label the SCCs of a graph using such a data structure; in this context the number of Union-Find operations is linear in the size of the graph, and the amortized time-complexity of these operations is quasi-constant (related to the inverse of the Ackermann function) in the worst case. To our knowledge, this suggested algorithm, which we call Gabow[1], has never been experimented to compute SCCs, let alone to perform an emptiness check.

Our contributions are as follows. (1) We show how to adapt Tarjan's algorithm to perform a generalized emptiness check. (2) We suggest an optimization of Dijkstra's algorithm that also benefits all the emptiness checks based on this algorithm. (3) We extend Gabow's idea to implement a Union-Find-based emptiness check. (4) Moreover we show how to adjust all these algorithms to support bit-state hashing and state-space caching.

While our experiments shows that there is no algorithm that clearly outperforms the others, we believe that having the choice between these three differents schemes might prove useful to devise new extensions (such as parallel model checking).

[1] Beware! The main algorithm of Gabow's paper [8] is a reinvention of Dijkstra's algorithm. Cf. `http://www.cs.colorado.edu/~hal/Papers/DFS/pbDFShistory.html`. What we call Gabow's algorithm here is the idea evoked on page 109 of that paper.

This paper is organized as follows. Section 2 defines TGBAs and introduces our notations. Sections 3–5 successively present Tarjan's, Dijkstra's, and Gabow's algorithms and discuss how that can be extended to perform emptiness checks. Section 6 discusses the compatibility of these algorithms with the bit-state hashing and state-space caching techniques. Finally Section 7 provides experimental data to compare all these algorithms.

2 Preliminaries

Let $G = \langle Q, q^0, \delta \rangle$ be a directed graph with Q the set of states, q^0 the initial state, and $\delta \subseteq Q \times Q$ the set of transitions.

A *path* of length $n \geq 1$ between two states $q, q' \in Q$ is a finite sequence of edges $\rho = (s_1, s_2)(s_2, s_3)\dots(s_n, s_{n+1})$ 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*.

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. A *trivial SCC* is a state without self-loop.

A *TGBA* is a tuple $A = \langle Q, q^0, \delta, \mathcal{F}, f \rangle$ where $\mathcal{F}$ is a finite set of acceptance marks and $f : \delta \mapsto 2^{\mathcal{F}}$ labels each transition of the directed graph $\langle Q, q^0, \delta \rangle$ by a set 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 *degeneralization* process can transform any TGBA with n states and m acceptances marks into an equivalent TGBA with one acceptance mark and at most nm states.

An SCC $S \subseteq Q$ is *accepting* iff $\bigcup_{t \in (S \times S) \cap \delta} \{f(t)\} = \mathcal{F}$. A TGBA is non-empty iff there is a path from q^0 to an accepting SCC.

All the algorithms we consider are based on a DFS of a TGBA and we can present them by specializing the generic DFS algorithm of Algo. 1. This algorithm is slightly more complex than the average DFS, as we will use it in various settings. The *dfs* variable is the stack of the DFS algorithm and stores: a set *acc* of acceptance marks labeling the transition leading to the state *pos*, and set *succ* of the unexplored successors of this state. The state *pos* is actually represented by a *Position*, which shall be defined differently in each algorithm.

Each state is either LIVE, DEAD, or UNKNOWN. A state is UNKNOWN until it has been explored by the DFS, then it becomes LIVE. A state may only become DEAD after all the successors of the SCC it belongs to have been visited. Maintaining this status will be done by each algorithm by implementing the following methods:

- `GET_STATUS`: returns the status of a state;
- `PUSH`: called for any newly visited state, it should mark that state as LIVE;
- `UPDATE`: called every time a *back-edge* (i.e., a transition leading to a LIVE state) is found, this function detects a transition closing a cycle;
- `POP`: called every time the DFS backtracks a state. When the last state of an SCC is being popped, all the states in its SCC must be marked as DEAD

Algorithm 1. Generic DFS

```
1  Input: A TGBA A = ⟨Q, q^0, δ, F, f⟩

2  struct Step {acc: 2^F, pos: Position, succ: 2^δ}
3  struct Transition {src: Q, dst: Q}
4  dfs: stack of ⟨Step⟩

5  Position pos ← PUSH(q^0)
6  dfs.push(⟨ ∅, pos, successors(q^0)⟩)
7  while ¬ dfs.isEmpty()
8      Step step ← dfs.top()
9      if step.succ ≠ ∅
10         Transition t ← pick one from step.succ
11         switch GET_STATUS(t.dst) do
12             case DEAD
13                 skip
14             case LIVE
15                 UPDATE(f(t), t.dst)
16             case UNKNOWN
17                 pos ← PUSH(t.dst)
18                 dfs.push(⟨f(t), pos, successors(t.dst)⟩)
19     else
20         dfs.pop()
21         POP(step)
```

by POP. We call such a last state the *root* of the SCC (notice that this root may depend on the order in which the transitions are visited).

3 Tarjan-Based Algorithm

3.1 SCC Computation

In Tarjan's original algorithm [19], each state is associated to two numbers: a DFS number (indicating the order in which the states has been visited by the DFS), and a *lowlink*. Initially, this *lowlink* is equal to the DFS number, but each time a transition is backtracked (i.e., during UPDATE or POP) the *lowlink* of the source is updated to the DFS number (for UPDATE) or to the *lowlink* (for POP) of the destination if it is smaller. An SCC root is detected during POP as a state whose *lowlink* is equal to the DFS number.

A usual optimization of POP is based on the fact that when a root is popped, the (outside) states that are successors of this SCC have already been marked as DEAD. Consequently, if the set of LIVE states is stored as a stack, then all the states of the current SCC are on this stack between the position of the *root*

Algorithm 2. Tarjan's Algorithm.

```
1  struct P {lowlink: int; acc: 2^F}
2  live: hstack of ⟨Q⟩
3  dead: store of ⟨Q⟩
4  dstack: stack of ⟨P⟩

5  GET_STATUS(q ∈ Q) → Status
6    if live.get(q) ≠ -1
7      return LIVE
8    else if dead.has(q)
9      return DEAD
10   else
11     return UNKNOWN

12 UPDATE(acc ∈ 2^F, d ∈ Q)
13   dstack.top().lowlink ←
14       min(dstack.top().lowlink,
15           live.get(d))
16   dstack.top().acc ← acc ∪
17               dstack.top().acc
18   if dstack.top().acc = F
19     report counterexample found

20 PUSH(q ∈ Q) → Position
21   Position p ← live.size()
22   live.push(⟨q⟩)
23   dstack.push(⟨ p , ∅ ⟩)
24   return p

25 POP(s ∈ Step)
26   ⟨ll, acc⟩ ← dstack.pop()
27   if ll = s.pos
28     // An SCC has been found.
29     while live.size() > s.pos
30       ⟨q⟩ ← live.pop()
31       dead.add(q)
32   else
33     dstack.top().lowlink ←
34       min(dstack.top().lowlink,ll)
35   dstack.top().acc ← s.acc ∪
36       dstack.top().acc ∪ acc
37   if dstack.top().acc = F
38     report counterexample found
```

and the top of the stack. They can therefore be marked as DEAD by unwinding this stack, without exploring the graph.

Because a *lowlink* is only useful for states on *dfs*, it seems judicious to store it into a dedicated stack denoted *dstack*. This stack stores elements of the form $\langle lowlink, acc \rangle$ where *acc* is only useful when doing an emptiness check.

As the states on *dfs* are LIVE, they are simply identified by their position on *live*. We use this position instead of the DFS number when initializing *lowlink*.

To implement this *live* stack, we introduce a data structure **hstack** that stores all LIVE states and can be manipulated like a stack (with `push` and `pop`). To find the status of a state, we need to check whether it belongs to this **hstack**, therefore this structure is equipped with a `get` method that looks up a hash table to return the position associated to a given state, or -1 for missing states.

The set of DEAD states are represented by a separate data structure that support the following two operations: **add** and **has** with obvious semantics. As we shall discuss in Section 6, bit-state hashing and state-space caching can be implemented by redefining these operations.

Algorithm 2 presents our refactoring of the original Tarjan's algorithm to fit in the framework of Algorithm 1. The blue dashed boxes should be ignored on first read: they represent the parts to add to turn this SCC-enumeration algorithm into an emptiness check for TGBA.

Because LIVE and DEAD states are respectively stored in *live* and *dead*, `GET_STATUS` can easily report all other states as UNKNOWN.

As explained previously, the lowlinks are updated everytime a transition is backtracked, i.e., at lines 12–15 when backtracking a back-edge, and at 32–34 when backtracking a forward-edge inside an SCC. When `POP` detects the root of an SCC (line 27), it simply unwind *live* to mark all the SCC's states as DEAD.

3.2 Emptiness Check

Adding the blue dashed boxes will turn the SCC enumeration algorithm into an emptiness check algorithm. Each LIVE state on *dfs* is now associated to an empty set of acceptance mark at line 1. This set is updated each time an edge intern to an SCC is backtracked, at lines 16–17 and 35–36. These backtracking updates will ultimately propagate to the root, the set of all acceptance marks present in the SCC. Therefore, in the worse case, an accepting SCC will be detected when the root is popped, but it may happens earlier if one of the intermediate set is equal to $\mathcal{F}$ (hence the tests on lines 18 and 38).

To our knowledge, the only existing emptiness check based on Tarjan's algorithm has been proposed by Geldenhuys and Valmari [11]. Their algorithm targets only degeneralized Büchi automata (one acceptance mark), so they may have to explore a larger automaton that we do. However their algorithm works quite differently from this one: they maintain the *lowlink* for each LIVE state and a stack of LIVE accepting states (it would work for transition-based acceptance too) and they are therefore able to report a counterexample as soon as they close an accepting cycle, while our algorithm would have to wait for an accepting transition to be popped. This detection could be done earlier by associating an acceptance set to each element of *live*. As we target memory efficience this solution has not been retained.

4 Dijkstra-Based Algorithms

4.1 SCC Computation

Intuitively, Dijkstra's algorithm [6] maintains a stack of SCCs of the subgraph that has been explored. Everytime a back-edge is found, closing a cycle, the SCCs forming that cycle are merged.

In practice, Algorithm 3 (without the green dotted boxes) actually manages three stacks: *live*, the set of LIVE states; *dfs*, the subset of *live* that are on the DFS search path, represented—as in the previous section—by a stack of positions in *live*; and *roots*, the stack of SCC roots, stored as positions in the *dfs* stack. When given two consecutive roots, $roots[i]$ and $roots[i+1]$, the set of states belonging to the SCC rooted in $roots[i]$, are the states at positions $dfs[roots[i]].pos, \ldots, dfs[roots[i+1]].pos - 1$ in *live*. This representation makes several operations efficient. Merging consecutive SCCs can be done by simply removing elements from *roots* (lines 18 and 21). Also, it possible to decide whether

Algorithm 3. Dijkstra's Algorithm.

```
1  live: hstack of ⟨ Q ⟩
2  dead: store of ⟨ Q ⟩
3  roots: rstack of ⟨ root: int, acc: 2^F ⟩

4  GET_STATUS(q ∈ Q) → Status
5      if live.get(q) ≠ -1
6          return LIVE
7      else if dead.has(q)
8          return DEAD
9      else
10         return UNKNOWN

11 PUSH(q ∈ Q) → Position
12     Position p ← live.size()
13     live.push(q)
14     roots.push_trivial(dfs.size())
15     return p

16 UPDATE(acc ∈ 2^F, d ∈ Q)
17     dpos ← live.get(d)
18     ⟨r, a⟩ ← roots.pop()
19     a ← a ∪ acc
20     while dpos < dfs[r].pos
21         ⟨r, la⟩ ← roots.pop()
22         a ← a ∪ dfs[r].acc ∪ la
23     roots.push_non_trivial(a, r,
24                            dfs.size() - 1)
25     if a = F
26         report counterexample found

27 POP(s ∈ Step)
28     if dfs.size() = roots.top_root()
29         // An SCC has been found.
30         roots.pop()
31         while live.size() > s.pos
32             q ← live.pop()
33             dead.add(q)
```

Algorithm 4. Gabow's Algorithm.

```
1  uf: union find of
   ⟨ Q ∪ {DeadState} ⟩
2  roots: rstack of ⟨ root: int, acc: 2^F ⟩
3  uf.make_set(DeadState)

4  GET_STATUS(q ∈ Q) → Status
5      if uf.ufcontains(q)
6          if uf.same_set(q, DeadState)
7              return DEAD
8          else
9              return LIVE
10     else
11         return UNKNOWN

12 PUSH(q ∈ Q) → Position
13     uf.make_set(q)
14     roots.push_trivial(dfs.size())
15     return q

16 UPDATE(acc ∈ 2^F, d ∈ Q)
17     ⟨r, a⟩ ← roots.pop()
18     a ← a ∪ acc
19     while ¬uf.same_set(dfs[r].pos, d)
20         uf.unite(dfs[r].pos, d)
21         ⟨r, la⟩ ← roots.pop()
22         a ← a ∪ dfs[r].acc ∪ la
23     roots.push_non_trivial(a, r,
24                            dfs.size() - 1)
25     if a= F
26         report counterexample found

27 POP(s ∈ Step)
28     if dfs.size()= roots.top_root()
29         // An SCC has been found.
30         roots.pop()
31         uf.unite(s.pos, DeadState)
```

a state is a root of an SCC during POP: when the position pointed to by the top of the *roots* stack is equal to the size of *dfs* (line 28) it means the state that has already been popped by the main DFS algorithm was a root.

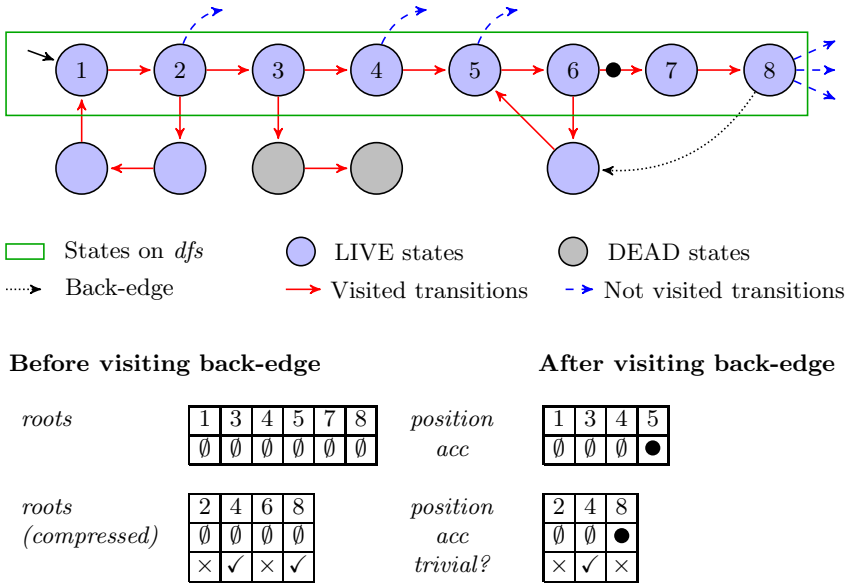

Fig. 1. Stack compression in action where numbers corresponds to DFS positions

The *roots* stack is implemented with a structure called **rstack** that supports three operations: `pop`(), `push_trivial`(*begin*) and `push_non_trivial`(*begin*,*end*). The latter two distinguish whether the SCC being pushed is trivial or not. They can be implemented as a normal `push`(*begin*), but in Section 4.2 we will see how to use these to compress the stack. Initially, any newly visited state constitutes a trivial SCC (line 14) with respect to the explored part of the automaton; non-trivial SCCs are only created when merging SCCs because of a back-edge (line 24).

DEAD states are stored in a *dead* store as in the previous algorithm, and for the same reason.

4.2 Compressing the *roots* Stack

The *roots* stack represents two kinds of SCCs: trivial and non-trivial. We suggest to compress this stack by representing ranges of consecutive trivial SCCs in a single entry. Each stack entry should have an additional Boolean indicating whether it represents of range of trivial SCCs or a non-trivial SCC, and should store the position of the last state seen before moving to the next entry. Figure 1 shows the effect of this compression.

In the worst case, it appears that we are simply adding one extra bit per entry, but as we shall see in our experiments, merging consecutive trivial SCCs is really effective.

4.3 Emptiness Checks

Dijkstra's algorithm can be turned into a emptiness check by adding the green dotted boxes. Each SCC is associated to a set of acceptance marks that have been seen inside this SCC. When some SCCs are merged, their acceptance marks are merged along with the marks of the transitions between these SCCs (line 25–26). A counterexample can be reported as soon as this union is $\mathcal{F}$.

Several authors have devised emptiness-check algorithms using this principle [1, 5, 12, 14, 10]. In this scheme, the main DFS can also be adjusted to chose the next transition to visit among all the non-visited outgoing transitions of the topmost SCC [1, 5, 14].

The algorithm proposed by Couvreur [4] is sometimes considered as a Dijkstra-based algorithm [12]: it replaces the *live* stack by a simple hash map (save a tiny bit of memory) and consequently has to rediscover the states that need to be marked DEAD during `POP` (loosing time). Nevertheless it fit perfectly into the generic canvas of Algorithm 1 and can easily be mixed with bitstate hashing and state space caching by using a *dead* store.

5 Gabow-Based Algorithms

The `POP` operation of previous algorithms is costly because it has to visit all the states in top SCC to mark them as DEAD.

If we regard Dijkstra's algorithm as partitioning of the set of states, each (live) SCC corresponds to a class in this partition, and an additional class stores all DEAD states. Merging SCCs maps to unions of LIVE classes in this partition, while popping an SCCs should incur a union with the class of dead states.

This observation is the base of Gabow's suggestion [8] to use the Union-Find data structure [20] to discover the SCCs of a graph. In this data structure, a union operation can be achieved in near constant-time (or even constant-time for this particular application [9]), without enumerating all its states.

The Union-Find structure partitions the set $Q' = Q \cup \{\mathit{DeadState}\}$ where *DeadState* represent an extra artificial DEAD state, and offers the following methods: `make_set`($s \in Q'$) creates a new class containing the state s; `unite`($s_1 \in Q'$, $s_2 \in Q'$) makes the union between two classes given by their representatives s_1 and s_2; and `same_set`($s_1 \in Q'$, $s_2 \in Q'$) checks whether two states are in the same class.

Algorithm 4 follows the same schema as Algorithm 3, except that we have replaced *live* and *dead*, by the Union-Find structure *uf*, and that *Position*s stored in *dfs* are now pointers to states. When the root of an SCC is popped (line 28), its class is merged with that of the artificial *DeadState* (line 31). `GET_STATUS` has to be updated to check deadness using this *DeadState* as well. `UPDATE` is done easily by uniting all classes representing the SCCs on the cycle.

The main difference with Dijkstra's algorithm is therefore that the use of `unite` in function `POP` dispenses from enumerating all states in the SCC. This approach remains compatible with the compression of the *roots* stack presented

in Sec. 4.2, and can be turned into an emptiness check in the same way as Dijkstra (adding purple boxes).

As-is, this algorithm is neither compatible with bit-state hashing nor state-space caching, because there is no *dead* store. Compatibility with these techniques is possible, but tricky. We discuss it in the next Section.

6 Bit-State Hashing and State-Space Caching Compatibility

Bit-state hashing [15] and state-space caching [13] are two techniques to save memory. In bit-state hashing, collisions in the hash table storing dead states are ignored, turning the algorithm into a semi-decision procedure. In state-space caching, dead state can be removed from the store at any moment, causing the algorithm to possibly revisit a state several times.

On Tarjan-based and Dijkstra-based algorithms, these techniques can be implemented by replacing the `has` and `add` methods of the *dead* store, implemented as a hash table. Note that for bit-state hashing, it is important to check the membership to *live* before the membership to *dead* in `GET_STATUS`.

When compatibility with these techniques is not required, we can forgot the use of this extra hash table, and actually store LIVE and DEAD states in the same table, using a extra bit to distinguish LIVE from DEAD. This saves a table lookup in `GET_STATUS`.

For Gabow's algorithm, compatibility with bit-state hashing and state-space caching is more tricky to achieve and we only give the intuition. First, the Union-Find data structure, which stores states in a vector, has to be made aware of what a DEAD state is: let us assume that the `unite` of line 31 is changed to `make_dead`. The first time `make_dead` is called, the states to be marked as DEAD are all at the end of the vector. The trick is to remember the frontier between LIVE and DEAD states in that vector. Then, every time a new singleton class is created with the `make_set` operation, we can reuse the slot of the first DEAD state (right after the frontier), and move that DEAD state to the DEAD store. `GET_STATUS` has to be updated as well.

Note that in this approach, the set of DEAD states is distributed in two structures: the end of the Union-Find vector, and the DEAD store, but only this store can be subject to bit-state hashing or state-space caching. However this approach still avoids the enumeration of states to mark them DEAD.

7 Implementation Issues and Benchmarks

All these approaches have been implemented in Spot [7]. The Union-Find structure of Gabow's algorithm uses common optimizations: "Immediate Parent Compression", "Link by Rank", "Path Compression", and "Memory Smart" [17].

When *dead* does not use bit-state hashing nor state-space caching techniques, an optimization consists in marking states as DEAD inside the *live* structure

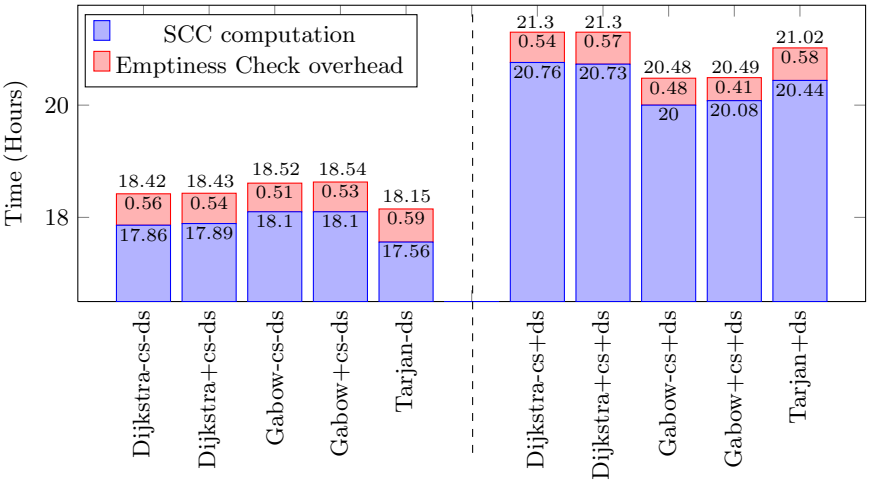

Fig. 2. Overhead of the emptiness checks over the SCC computations on 448 empty products. A total of 2.5×10^9 states, 17.3×10^9 transitions, and 10^9 SCCs were visited.

rather than transferring it into *dead* during a POP. This optimization only requires a special value to tag a state DEAD. Its use is denoted by -ds in tables, while the use of a dedicated *dead* store (as presented previously) is denoted by +ds. Similarly, +cs and -cs indicate whether the *roots* stack optimization (Sec. 4.2) is enabled or disabled.

The models we use come from the BEEM benchmark [18]. We generate the corresponding system automata using a version of DiVinE 2.4 patched by the LTSmin team.[2] Because there are too few LTL formulas supplied by the BEEM benchmark, we opted to generate random formulas for each model. We computed a total number of 860 formulas.[3]

A formula and a model generate a product that may be either empty (the formula is verified) or non-empty (a counterexample exists). To decide that a product is empty, any emptiness check has to explore all the reachable states of the product. Conversely, a non-empty product can be reported as soon as an accepting SCC is detected, avoiding the need to explore the entire product. In our implementation, all algorithms use the same generic DFS traversal and thus visit transitions in the same order.

Among our formulas, 412 result in non-empty product with the model. The remaining 448 formulas, associated to empty products, were selected so that the emptiness check algorithms would take at least 10 seconds on an Intel(R) 64-bit Xeon(R) @2.00GHz with 64GB of RAM.

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

[3] For a detailed description of our setup, including selected models and formulas, see `http://move.lip6.fr/~Etienne.Renault/benchs/LPAR-2013/results_scc.html`.

Figure 2 shows the execution time of all the emptiness check variants presented in this paper (with or without *dead* store, with or without compressed *roots* stack). To measure the overhead of the emptiness check over the SCC computation, we only focus on empty products.

For each bar the lower part represents the SCC computation time while the upper part corresponds to the overhead induced by the emptiness check. The total execution time is indicated atop the bar. The 5 rightmost bars show the emptiness check with a *dead* store enabled (+ds) while the 5 leftmost bars have it disabled (-ds).

For the same +ds/-ds setting, all execution times are very close, and the emptiness check overhead is 3% on the average.

When the *dead* store is disabled, Tarjan is slightly better than Dijkstra, which is itself slightly better than Gabow. Activating the *dead* store generate an overhead of about 15%, and is more favorable to Gabow. This latter point is due to the fact that our handling of the *dead* store for Gabow's algorithm, described in Section 6, will transfer less states from *live* to *dead*; this reduces the overhead to 10% only.

Table 1 reports the memory consumption, based on the size of the data structures used. As for time measurement, these experiments only focuses on verified formulas. The second column gives the formula that computes memory consumption at any time. The third column shows the peak we observed while running our experiments.

From that figure it appears that Dijkstra is the most memory efficient algorithm. Indeed the stack used by Dijkstra is a subset of the *dfs* stack while the *dstack* of Tarjan, storing a *lowlink* and an acceptance set for each element, follows the variations of *dfs*. Gabow's algorithm requires more memory than the two others since it has to maintain the whole structure of the Union-Find. The use of a *dead* store significantly reduces memory consumption (up to 17%).

When bit-state hashing or state-space caching are used, the size of $|dead|$ can be fixed arbitrarily, allowing an even greater reduction.

Table 2 reports the the cumulated number of transitions, states and SCC visited by each algorithm for the 412-non empty products. We use this table to compare how quickly each algorithm reports a counterexample.

Gabow's and Dijkstra's algorithms have identical results since they both report a counterexample when a cycle is closed during `UPDATE`, while Tarjan's algorithm may delay the report of a counterexample to a later `POP` and visit several states until then. Nonetheless this difference is very small in our experiment: less than 1% additional states, transitions or SCCs have been visited. This negligible difference justifies our decision not to store an additional acceptance set in each element of *live* to report counterexamples earlier in Tarjan's algorithm, as discussed at the end of Sec. 3.2.

Table 3 presents the impact of the lazy transfer into *dead* proposed for Gabow's algorithm. We observe that only half the states are transferred to *dead*; this means that the remaining states have been preserved in the DEAD part of the Union-Find structure. This explains the gain observed from Fig. 2.

Table 1. Comparison of memory consumption for emptiness check algorithms on the 448 empty products. $|roots|$ (resp. $|uf|$, $|dstack|$, $|dead|$) denotes the number of elements in *rstack* (resp. *uf*, *dstack*, *dead*). As *rstack* elements are pairs $(root, \text{acc})$, we count the memory consumption as $2|roots|$ words. The additional bit required for each element of the compressed stack is not accounted for. Since *live* is constructed using an *hashmap* and a *stack*, we distinguish these sizes with $|live_{stack}|$ and $|live_{hash}|$: they differ when no *dead* store is used.

Algorithm	**Memory consumption (words)**	**Observed peak**
Dijkstra-cs-ds	$2\|roots\| + \|live_{stack}\| + 2\|live_{hash}\|$	6 225 414 223
Dijkstra+cs-ds		6 225 411 039
Gabow-cs-ds	$2\|roots\| + 3\|uf\|$	7 364 856 119
Gabow+cs-ds		7 364 854 033
Tarjan-ds	$2\|dstack\| + \|live_{stack}\| + 2\|live_{hash}\|$	6 325 991 684
Dijkstra-cs+ds	$2\|roots\| + \|live_{stack}\| + 2\|live_{hash}\| + \|dead\|$	5 160 440 344
Dijkstra+cs+ds		5 160 435 523
Gabow-cs+ds	$2\|roots\| + 4\|uf\| + \|dead\|$	6 608 486 024
Gabow+cs+ds		6 608 482 885
Tarjan+ds	$2\|dstack\| + \|live_{stack}\| + 2\|live_{hash}\| + \|dead\|$	5 265 484 149

Table 2. Cumulated States, transitions, and SCCs visited by each emptiness check on the 412 non-empty products

	Transitions	**States**	**SCCs**
Tarjan	534 471 068	67 230 381	34 622 772
Dijkstra/Gabow	534 338 119	67 187 854	34 582 459

Table 3. Impact of the *dead* strategy of Gabow's algorithm on the 448 empty products

	Max. *dead* **peak**	**Cumulated** *dead* **peak**
Tarjan/Dijkstra (+ds)	29 098 013	2 454 950 318
Gabow (+ds)	21 430 297	1 070 440 670

Table 4. Impact of the compressed *roots* stack on the 448 empty products

	Max. *roots* **peak**	**Cumulated** *roots* **peak**
Dijkstra/Gabow (-cs)	456	98322
Dijkstra/Gabow (+cs)	119	8188

This observation also suggests that a similar optimization could be applied to Tarjan's and Dijkstra's algorithms: each time the *live* stack is reduced, the residual space (the free list) can be reused to store DEAD states temporarily.

Table 4 shows the impacts of the compression technique proposed in Sec. 4.2. It allows a tenfold memory reduction without run-time overhead according to Fig. 2. Note that such a compression technique is independent of the emptiness check layer, but may apply to Dijkstra's and Gabow's SCC computations.

In Sec. 5, we suggested that using Union-Find was an efficient way to mark all states of an SCC as DEAD in a single operation. Unfortunately, Fig. 2 reveals that these gains are offset by the inherent cost of maintaining the Union-Find structure. Our implementation of the Union-Find uses classical optimizations [17] but we have yet to investiguate wether performances could be improved by the use of a data structure dedicated to the case where each union only concern the last SCCs [9].

8 Conclusion

This paper proposed an overview of existing SCC enumeration algorithms and proposed a generic canvas to transform them into emptiness checks for TGBA.

This lead us to define two new emptiness checks. One is based on Tarjan; it differs from [11] in that it is more memory efficient and generalized. Another one is based on Gabow's suggestion to use the Union-Find data structure: our results with that data structure are mixed, but as far as we know, this is the first time this data structure is used for emptiness check.

We also introduced a couple of optimizations. For Dijkstra's and Gabow's emptiness checks we suggest to compress the *roots* stack to save some memory. Additionally, we discussed a strategy to transfer DEAD state from the Union-Find structure to the *dead* store lazily, resulting in an important gain of time, and this strategy could also be applied to the other algorithms.

We have several leads for future work. One would be to devise a compression technique for the stack of lowlink (*dstack*) used by Tarjan's algorithm to make it more competitive to Dijkstra's algorithm (currently more memory-efficient). Furthermore, the compaction of the *live* stack suggested by Nuutila and Soisalon-Soininen [16] for Tarjan's algorithm could be adapted to Dijkstra's algorithm and (with a more work) to Gabow's. Another idea would be to study the various ways to extract counterexamples from these algorithms; the procedure suggested by Couvreur et al. [5] would work for Dijkstra and Gabow but should not be difficult to adapt to Tarjan. Finally, we would like to investigate the possibility to parallelize these emptiness checks. There are very few parallel emptiness checks based on SCC computations [2], however as Tarjan and Dijkstra use different data structure than Gabow, may be one of them will be more favorable to a parallel setup.

References

[1] Alur, R., Chaudhuri, S., Etessami, K., Madhusudan, P.: On-the-fly reachability and cycle detection for recursive state machines. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 61–76. Springer, Heidelberg (2005)

[2] Černá, I., Pelánek, R.: Distributed explicit fair cycle detection (set based approach). In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
[3] Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithm for the verification of temporal properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
[4] Couvreur, J.-M.: On-the-fly verification of temporal logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
[5] Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized Büchi automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)
[6] Dijkstra, E.W.: EWD 376: Finding the maximum strong components in a directed graph (May 1973), http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF
[7] Duret-Lutz, A., Poitrenaud, D.: SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In: MASCOTS 2004, pp. 76–83. IEEE Computer Society Press (October 2004)
[8] Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Information Processing Letters 74(3-4), 107–114 (2000)
[9] Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. In: STOC 1983, pp. 246–251. ACM (1983)
[10] Gaiser, A., Schwoon, S.: Comparison of algorithms for checking emptiness on Büchi automata. In: MEMICS 2009. OASICS, vol. 13, Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik (2009)
[11] Geldenhuys, J., Valmari, A.: Tarjan's algorithm makes on-the-fly LTL verification more efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)
[12] Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theoretical Computer Science 345(1), 60–82 (2005)
[13] Godefroid, P., Holzmann, G.J., Pirottin, D.: State space caching revisited. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 178–191. Springer, Heidelberg (1993)
[14] Hansen, H., Geldenhuys, J.: Cheap and small counterexamples. In: SEFM 2008, pp. 53–62. IEEE Computer Society (November 2008)
[15] Holzmann, G.J.: On limits and possibilities of automated protocol analysis. In: PSTV 1987, pp. 339–344. North-Holland (May 1987)
[16] Nuutila, E., Soisalon-Soininen, E.: On finding the strongly connected components in a directed graph. Information Processing Letters 49(1), 9–14 (1994)
[17] Patwary, M. M.A., Blair, J., Manne, F.: Experiments on union-find algorithms for the disjoint-set data structure. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 411–423. Springer, Heidelberg (2010)
[18] Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
[19] Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
[20] Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM (JACM) 22(2), 215–225 (1975)