Comparison of Algorithms for Checking Emptiness on Büchi Automata

Andreas Gaiser[1]* and Stefan Schwoon[2]

[1] Institut für Informatik, Technische Universität München, Germany
[2] LSV, CNRS, ENS de Cachan, INRIA Saclay, France
gaiser@model.in.tum.de, schwoon@lsv.ens-cachan.fr

Abstract. We re-investigate the problem of LTL model-checking for finite-state systems. Typical solutions, like in Spin, work on the fly, reducing the problem to Büchi emptiness. This can be done in linear time, and a variety of algorithms with this property exist. Nonetheless, subtle design decisions can make a great difference to their actual performance in practice, especially when used on-the-fly. We compare a number of algorithms experimentally on a large benchmark suite, measure their actual run-time performance, and propose improvements. Compared with the algorithm implemented in Spin, our best algorithm is faster by about 33 % on average. We therefore recommend that, for on-the-fly explicit-state model checking, nested DFS should be replaced by better solutions.

1 Introduction

Model checking is the problem of determining whether a given hardware or software system meets its specification. In the automata-theoretic approach, the system may have finitely many states, and the specification is an LTL formula, which is translated into a Büchi automaton, intersected with the system, and checked for emptiness. Thus, model checking becomes a graph-theoretic problem.

Because of its importance, the problem has been intensively investigated. For instance, *symbolic* algorithms use efficient data structures such as BDDs to work on sets of states; a survey of them can be found in [5]. Moreover, *parallel* model-checking algorithms have been developed [1]. The best known symbolic or parallel solutions have suboptimal asymptotic complexity ($\mathcal{O}(n \log n)$, where n is the number of states), but are often faster than that in practice.

Büchi emptiness can also be solved in $\mathcal{O}(n)$ time. All known linear algorithms are *explicit*, i.e. they construct and explore states one by one, by depth-first search (DFS). Typically, they compute some data about each state: its unique *state descriptor* and some *auxiliary data* needed for the emptiness check. Since the state descriptor is usually much larger than the auxiliary data, approximative techniques such as bitstate hashing have been developed that avoid them, storing just the auxiliary information in a hash table [13]. This entails the risk of undetectable hash collisions; however the probability of a wrong result can be

* The author was supported by the DFG Graduiertenkolleg 1480 (PUMA).

reduced below a chosen threshold by repeating the emptiness test with different hash functions. Thus they represent a trade-off between time and memory requirements. Henceforth, we shall refer to non-approximative methods that do use state descriptors as *exact* methods.

We further identify two subgroups of explicit algorithms: *Nested-DFS* methods directly look for acceptings cycle in a Büchi automaton; they need very little auxiliary memory and work well with bitstate hashing. *SCC-based* algorithms identify strongly connected components containing accepting cycles; they require more auxiliary memory but can find counterexamples more quickly.

All explicit algorithms can work "on-the-fly", i.e. the (intersected) Büchi automaton is not known at the outset. Rather, one begins with a Büchi automaton for the formula (typically small) and a compact system description and extracts the initial state from these. Successor states are computed during exploration as needed. If non-emptiness is detected, the algorithms terminate before constructing the entire intersection. Moreover, in this approach the transition relation need not be stored in memory. As we shall see, the on-the-fly nature of explicit algorithms is very significant when evaluating their performance properly.

In this paper, we investigate performance aspects of explicit, exact, on-the-fly algorithms for Büchi emptiness. The best-known example for such a tool is Spin [12], which uses the nested-DFS algorithm proposed by Holzmann et al [13], henceforth called HPY. The reasons for this choice are partly historic; the faster detection capabilities of SCC-based algorithm were not known when Spin was designed, having first been pointed out by Couvreur in 1999 [3]. Thus, the status of HPY as the best choice is questionable, all the more so since the memory advantages of nested DFS are comparatively scant in our setting. Moreover, improved nested DFS algorithms have been proposed in the meantime.

We therefore evaluate several algorithms based on their actual running time and memory usage on a large suite of benchmarks. Previous papers, especially those on SCC-based algorithms [10, 15, 4, 11], provided similar experimental results, however, experiments were few or random and unsatisfying in one important aspect: they worked from pre-computed Büchi automata, rather than truly on-the-fly. This aspect will play a significant role in our evaluation.

To summarize, this paper contains the following contributions and findings:

- We provide improvements in both subgroups, nested DFS and SCC-based. These concern the algorithms of Couvreur [3] and Schwoon/Esparza [15]. For new, self-contained proofs, see [7].
- One of the algorithms we study can be extended to generalized Büchi automata, and we investigate this aspect.
- We implemented existing and new algorithms and compare them on a large benchmark suite. We analyze the structural properties of Büchi automata that cause performance differences.

We make the following observations: The overall memory consumption of all algorithms is dominated by the state descriptors, the differences in auxiliary memory play virtually no role. The running times depend practically exclusively on the number of successor computations. When experimenting with pre-computed automata – as done in some other papers – this operation becomes cheap, which causes misleading results. Our results allow to derive detailed recommendations which algorithms to use in which circumstances. These recommendations revise those from [15]; Couvreur's algorithm which was recommended there, is shown to have weak performance; however, the modification mentioned above amends it. Moreover, our modification of Schwoon/Esparza improves the previous best nested-DFS algorithm.

We proceed as follows: Section 2 establishes preliminaries, Sections 3 and 4 present nested-DFS and SCC-based algorithms, including our modifications. Section 5 details our experimental results and concludes.

2 Preliminaries

A *Büchi automaton* (BA) is a tuple $\mathcal{B} = (S, s_I, \mathrm{post}, A)$, where S is a finite set of *states*, $s_I \in S$ is the *initial state*, $\mathrm{post}\colon S \to 2^S$ is the *successor function*, and $A \subseteq S$ are the *accepting states*. A *path* of $\mathcal{B}$ is a sequence of states $s_1 \cdots s_m$ for some $m \geq 1$ such that $s_{i+1} \in \mathrm{post}(s_i)$ for all $1 \leq i < m$. If a path from s to t exists, we write $s \to^* t$. When $m > 1$, we write $s \to^+ t$, and if additionally $s = t$, we call the path a *loop*. A *run* of $\mathcal{B}$ is an infinite sequence $(s_i)_{i \geq 0}$ such that $s_0 = s_I$ and $s_{i+1} \in \mathrm{post}(s_i)$ for all $i \geq 0$. A run is called *accepting* if $s_i \in A$ for infinitely many different i. The *emptiness problem* is to determine whether no accepting run exists. If an accepting run exists, it is also called a *counterexample*. From now on, we assume a fixed Büchi automaton $\mathcal{B}$.

Note that we omit the usual input alphabet because we are just interested in emptiness checks. Moreover, the transition relation is given as a mapping from each state to its successors, which is suitable for on-the-fly algorithms.

A *strongly connected component* (SCC) of $\mathcal{B}$ is a subset $C \subset S$ such that for each pair $s, t \in C$, we have $s \to^* t$, and moreover, no other state can be added to C without violating this property. An SCC C is called *trivial* if $|C| = 1$ and for the singleton $s \in C$, $s \notin \mathrm{post}(s)$. The following two facts are well-known:

(1) A counterexample exists iff there exists some $s \in A$ such that $s_I \to^* s$ and $s \to^+ s$. This fact is exploited by nested-DFS algorithms.
(2) A counterexample exists iff there exists a non-trivial SCC C reachable from s_I such that $C \cap A \neq \emptyset$. This fact is exploited by SCC-based algorithms.

A Büchi automaton is called *weak* if each of its SCCs is either contained in A or in $S \setminus A$. This implies the following fact:

(3) Each loop in a weak BA is entirely contained in A or in $S \setminus A$.

A *generalized Büchi automaton* (GBA) is a tuple $\mathcal{G} = (S, s_I, \mathrm{post}, \mathcal{A})$, where S, s_I, and post are as before, and $\mathcal{A} = (A_1, \ldots, A_k)$ is a *set* of acceptance conditions, i.e. $A_j \subseteq S$ for all $j = 1, \ldots, k$. Paths and runs are defined as for normal Büchi automata; a run $(s_i)_{i \geq 0}$ of $\mathcal{G}$ is called *accepting* iff for each $j = 1, \ldots, k$ there exist infinitely many different i such that $s_i \in A_j$.

GBA are generally more concise than BA: a GBA with k acceptance conditions and n states can be transformed into a BA with nk states. There is no known nested-DFS algorithm that avoids this k-fold blowup for checking emptiness of a GBA, although Tauriainen's algorithm mitigates it [17]. Some SCC-based algorithms, however, can exploit the following fact:

(4) A counterexample exists in $\mathcal{G}$ iff there exists a non-trivial SCC C reachable from s_I such that $C \cap A_j \neq \emptyset$ for all $j = 1, \ldots, k$.

3 Nested depth-first search

Nested DFS was first proposed by Courcoubetis et al [2], and all other algorithms in this subgroup still follow the same pattern. There are two DFS iterations: the "blue" DFS is the main loop and marks every newly discovered state as blue. Upon backtracing from an accepting state s, it initiates a "red" DFS that tries to find a loop back to s, marking every encountered state as red. If a loop is found, a counterexample is reported, otherwise the blue DFS continues, but the established red markings remain. Thus, both blue and red DFS visit each state at most once each. Only two bits of auxiliary data are required per state.

This pattern of searching for accepting loops in post-order ensures that multiple red searches do not interfere; states in "deep" SCCs are coloured red first, and when a red DFS terminates, red states are guaranteed not to be part of any counterexample. While being memory-efficient and simple, this has two disadvantages. First, nested DFS prefers long counterexamples over shorter ones; secondly, the blue DFS never notices that a complete counterexample has already been explored and continues exploring potentially many more states than necessary before eventually noticing the counterexample during backtracking. Also, nested DFS computes the successors of many states twice.

Several improvements have been suggested in the past, e.g. the HPY algorithm [13], implemented in Spin, and the SE algorithm [15]. We present an improvement of SE, shown in Figure 1. A self-contained presentation and proof is provided in [7]; here, we just describe the differences w.r.t. SE.

The additions to SE are in lines 4 and from 12 to 15. These exploit the fact that red states cannot be part of any counterexample; therefore a state that has only red successors cannot be either. This avoids certain initiations of the red search. The improvement is similar in spirit to [8], but avoids some unnecessary invocations of post. Like in [2], only two bits per state are used. Our experiments shall show that it performs best among the known nested DFS algorithms.

Finally, we remark that for weak automata a much simpler algorithm suffices, as observed by Černá and Pelánek [18]. Exploiting Fact (3), one can simply omit the red search because all counterexamples are bound to be reported by line 9 in Figure 1. In that case, post is only invoked once per state.

```
procedure $new\_dfs$ ()
  call $dfs\_blue(s_I)$

procedure $dfs\_blue$ ($s$)
  $allred$ := true;
  $s.colour := cyan$;
  for all $t \in \text{post}(s)$ do
    if $t.colour = cyan$
        $\wedge\ (s \in A \vee t \in A)$ then
      report cycle
    else if $t.colour = white$ then
      call $dfs\_blue$(t);
    if $t.colour \neq red$ then
      $allred$ := false;
  if $allred$ then
    $s.colour := red$
  else if $s \in A$ then
    call $dfs\_red$(s);
    $s.colour := red$
  else
    $s.colour := blue$

procedure $dfs\_red$ ($s$)
  for all $t \in \text{post}(s)$ do
    if $t.colour = cyan$ then
      report cycle
    else if $t.colour = blue$ then
      $t.colour := red$;
      call $dfs\_red$(t)
```

Fig. 1. New Nested-DFS algorithm.

4 SCC-based algorithms

An efficient algorithm for determining SCCs that works on-the-fly was first proposed by Tarjan [16]. However, for model-checking purposes Tarjan's algorithm was deemed unsuitable because it used more memory than nested DFS while offering no advantages. More recent innovations by Geldenhuys/Valmari [10] and Couvreur [3] change the picture, however: their modifications allow SCC-based analysis to report a counterexample as soon as all its states and transitions were discovered, no matter in which order. In other words, if the order in which successors are explored by the DFS is fixed, both can find a counterexample in optimal time (w.r.t. to the exploration order).

Space constraints prevent us from presenting the algorithms in detail. However, we mention a few salient points. Tarjan places all newly discovered states onto a stack (henceforth called *Tarjan stack*) and numbers them in pre-order. Certain properties of the DFS ensure that at any time during the algorithm, states belonging to the same SCC are stored consecutively on the stack and therefore also numbered consecutively. The *root* of an SCC is the state explored first during DFS, having the lowest number and being deepest on the Tarjan stack. For each state s, Tarjan computes a so-called "lowlink" number, which is identical to the number of s iff s is a root, and less than that otherwise. An SCC is completely explored when backtracking from its root, and at that point it can be identified as a complete SCC and removed from the Tarjan stack.

Geldenhuys/Valmari (GV) exploit properties of lowlinks; they remember the number of the deepest accepting state on the current search path, say k, and when a state with lowlink $\leq k$ is found, a counterexample is reported. They also propose some memory savings that are of minor importance in our context.

Couvreur (C99) omits both Tarjan stack and lowlinks but introduces a *roots stack* that stores the roots of all partially explored SCCs on the current search path. When one finds a transition to a state with number k, properties of the

```
 1  procedure $couv$ ()
 2     $count := 0$;
 3     $Roots := \emptyset$; $Active := \emptyset$;
 4     call $couv\_dfs(s_I)$
 5  procedure $couv\_dfs(s)$:
 6     $count := count + 1$;
 7     $s.dfsnum := count$;
 8     $s.current :=$ true;
 9     push$(Roots, (s, \mathcal{A}(s)))$;
10     push$(Active, s)$;
11     for all $t \in \text{post}(s)$ do
12        if $t.dfsnum = 0$ then
13           call $couv\_dfs(t)$
14        else if $t.current$ then
15           $B := \emptyset$;
16           repeat
17              $(u, C) :=$ pop$(Roots)$;
18              $B := B \cup C$;
19              if $B = K$ then report cycle
20           until $u.dfsnum \leq t.dfsnum$;
21           push$(Roots, (u, B))$;
22     if top$(Roots) = (s, ?)$ then
23        pop$(Roots)$;
24        repeat
25           u:=pop$(Active)$;
26           u.$current :=$ false
27        until $u = s$
```

Fig. 2. Amendment of Couvreur's algorithm.

numbering imply that no state with number larger than k can be a root, prompting their removal from the roots stack. This effectively merges some SCCs, and one checks whether the merger creates an SCC with the conditions from Fact (2).

Both algorithms report a counterexample after seeing the same states and transitions, provided they work with the same exploration order. However, it turns out that the removal of the Tarjan stack in C99, while more memory efficient, was a crucial oversight: when backtracking from a root, another DFS is necessary to mark these states as "removed". These extra post computations severely impede its performance. This makes GV superior to C99 in practice.

We propose to amend C99 by re-inserting the Tarjan stack.[3] This amendment makes it competitive with GV while using slightly less memory; more crucially, C99 can deal directly with GBAs, which GV cannot. Since GBAs tend to be smaller than BAs for the same LTL formula, the amended algorithm can hope to explore fewer states and be faster.

The amended algorithm, working with GBAs, is shown in Figure 2. A more detailed presentation and a proof are given in [7]. Note that in C99 accceptance conditions are annotated on the transitions, whereas here we place them on the states, which is only a minor difference. Figure 2 assumes k acceptance sets, denoting $\mathcal{A}(s) := \{\, j \mid s \in A_j \,\}$ and $K := \{1, \ldots, k\}$. Note that if k is "small", the union operation in line 18 can be implemented with bit parallelism.

5 Experiments

We implemented a framework for testing and comparing the actual performance of all the known Büchi emptiness algorithms. For practical relevance, the best framework for such an implementation would have been Spin. However, Spin

[3] The problem with C99 was first hinted at in [15]. After creating this improvement independently, we learned that similar changes were already proposed in [4] and [11].

turned out too difficult to modify for this purpose. Instead, we based our testbed on NIPS [19], a reverse-engineered Promela engine. Essentially, NIPS allows to process a Promela model, provides the initial state descriptor and a function for computing its successors. It is thus ideally suited for testing on-the-fly algorithms, and we believe that the conditions are as close to Spin as possible.

We used 266 test cases from the BEEM database [14], including many different algorithms, e.g., the Sliding Window protocol, Lamport's Bakery algorithm, Leader Election, and many others, together with various LTL properties.

Among the algorithms tested and implemented were HPY [13], GV [10], C99 [3], SE [15], and the amended algorithms presented in Sections 3 and 4, henceforth called AND and ASCC. For weak automata, we report on simple DFS (SD, see Section 3). We also implemented and tested other algorithms, notably those from [2] and [8]. However, these were always dominated by others, and we omit them in the following. Naturally, our concrete running times and memory consumptions are subject to certain implementation-specific issues. Nonetheless, we believe that the tendencies exhibited by our experiments are transferrable.

In the following, we give a summary of our results. A more detailed description of our framework, the benchmarks, and the experimental results is given in [6]; here, we just summarize the most important findings.

We first found that, in the context of exact model checking, the differences in auxiliary memory usage was basically irrelevant. Certainly, the auxiliary memory used by the various algorithms ranged from 2 bits to 12 bytes, a comparatively large difference. However, this was dwarved by the memory consumption of state descriptors, which ranged from 20 to 380 bytes, averaging at 130.

The only practical difference therefore was in the running time. Here, we found that, no matter which auxiliary data structures were employed, the running time was practically proportional to the number of post invocations (more precisely: the number of individual successor states generated by post), by far the most costly operation. In retrospect, these two observations may seem obvious; however, we find that they were consistently under-represented in previous papers, therefore it is worth re-emphasizing their relevance. The two main factors contributing to the running time were fast counterexample detection and whether an algorithm had to compute each transition at most once or twice.

Discussing individual test cases would not be very meaningful: for instance, the early-detection properties of some algorithms can cause arbitrarily large differences. Instead, we exhibit certain structural properties that occurred in many test cases and caused those differences. We first discuss algorithms working on "normal" Büchi automata, followed by a discussion of ASCC with GBAs.

First, we observe that most test cases constitute weak Büchi automata. Note that the intersection BA is weak if the BA arising from the formula is weak. Černá and Pelánek [18] estimate the proportion of weak formulae in practice to 90–95 %; indeed, we found that only 8 % of our test cases were non-weak. For weak test cases, five out of six tested algorithms (GV, C99, SE, AND, SD) detect counterexamples with minimal exploration. The three main structural effects causing performance differences (which may overlap) were as follows:

– In 86 test cases, we observed many trivial SCCs consisting of one accepting state. A typical example is the LTL property GFp, which (when negated) yields a weak automaton with a looping accepting state. Then, any non-looping part of the system necessarily yields such trivial SCCs. In these cases, GV and SD dominate, sometimes with a factor of two, whereas C99, SE, and HPY fall behind because they explore the accepting trivial SCCs twice. In our test cases, the AND algorithm had the same performance as GV and SD, although this is not guaranteed in general.
– In 98 cases, we observed non-accepting SCCs not preceded by accepting SCCs. In this case, C99 falls behind all the others.
– HPY reports counterexamples only during the red DFS, whereas SE and AND discovers some during the blue DFS. This accounts for 101 test cases in which HPY fared worst, whereas all others showed the same performance.

Non-weak automata also had these effects, affecting 18, 17, and 7 out of 21 test cases. In 7 cases, GV and C99 found counterexamples more quickly than the others, being faster by a factor of up to six. Since we used the same exploration order in all algorithms, these results are directly comparable.

algorithm	run-time
ASCC	67.0 %
GV	69.2 %
AND	69.7 %
SE	96.3 %
HPY	100.0 %
C99	128.3 %

Fig. 3. Performances

We then tested the ASCC algorithm with GBA, generated by the LTL2BA tool [9]. Most formulae yielded GBA with only one acceptance condition, meaning that the GBA had the same size as the corresponding BA. Notice that the running times of GBA with multiple conditions are not directly comparable with those of the corresponding BA. This is because using a different automaton changes the order of exploration, therefore in some "lucky" cases the BA-based algorithms may still find a counterexample more quickly.

The running times summed up over all 266 test cases are given in Figure 3, expressed as percentages of each other. Additionally, SD had the same performance as GV for the weak cases. Note that every set of benchmarks would lead to the same order among the algorithms because it reflects their different qualitative properties (e.g., quick counterexample detection or number of post calls). The concrete numbers in Figure 3 tell their quantitative effect in what we believe to be a representative set of benchmarks. We draw the following conclusions:

– Because of the dominance of weak test cases and GBAs with only one acceptance condition, the sum of running times yields small differences; only SE, HPY, and C99 clearly fall behind. The performance differences in the comparatively few other cases is very pronounced however.
– Overall, ASCC is the best algorithm if GBAs can be used. Due to the technical reasons explained above, it did not perform best in all examples.
– Among the BA-based algorithms, GV is the best for general formulae; it is never outperformed on any test case by any other BA-based algorithm. ASCC performs equally well when used with simple BAs.
– For weak formulae, SD is the best algorithm for bitstate hashing.

– For general formulae, AND is the best algorithm for bitstate hashing, improving the previous best algorithm for this setting (SE) by 28 %.
– There remains no reason to use either SE, HPY, or C99.

Acknowledgements: The authors would like to thank Michael Weber for creating and helping us use the NIPS framework.

References

1. Jiří Barnat, Luboš Brim, and Petr Ročkai. DiVinE multi-core - a parallel LTL model-checker. In *Proc. ATVA*, LNCS 5311, pages 234–239, 2008.
2. Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
3. Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *Proc. Formal Methods*, LNCS 1708, pages 253–271, 1999.
4. Jean-Michel Couvreur, Alexandre Duret-Lutz, and Denis Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. SPIN*, LNCS 3639, pages 169–184, 2005.
5. Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. TACAS*, LNCS 2031, pages 420–434, 2001.
6. Andreas Gaiser. Vergleich von Algorithmen für den Leerheitstest von Büchiautomaten. Studienarbeit, Universität Stuttgart, 2007. In German.
7. Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on Büchi automata. Technical report, arxiv.org (`arXiv:0910.3766`), 2009.
8. Paul Gastin, Pierre Moro, and Marc Zeitoun. Minimization of counterexamples in SPIN. In *Proc. 11th SPIN Workshop*, LNCS 2989, pages 92–108, 2004.
9. Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proc. CAV*, LNCS 2102, pages 53–65, 2001.
10. Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proc. TACAS*, LNCS 2988, pages 205–219, 2004.
11. Jaco Geldenhuys and Antti Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, 2005.
12. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
13. Gerard J. Holzmann, Doron A. Peled, and Mihalis Yannakakis. On nested depth first search. In *Proc. 2nd SPIN Workshop*, pages 23–32, 1996.
14. Radek Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. SPIN*, LNCS 4595, pages 263–267, 2007.
15. Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In *Proc. TACAS*, LNCS 3440, pages 174–190, 2005.
16. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
17. Heikki Tauriainen. Nested emptiness search for generalized Büchi automata. *Fundamenta Informaticae*, 70(1–2):127–154, 2006.
18. Ivana Černá and Radek Pelánek. Relating hierarchy of linear temporal properties to model checking. In *Proc. MFCS*, LNCS 2747, pages 318–327, 2003.
19. Michael Weber. An embeddable virtual machine for state space generation. In *Proc. SPIN*, LNCS 4595, pages 168–186, 2007.