Kofola 1.0: A Modular Approach to ω-Regular Complementation and Inclusion Checking

Ondrej Alexaj[1], Vojtěch Havlena[1], Lukáš Holík[1,2], Ondřej Lengál[1], Yong Li[3], and Nicolas Mazzocchi[4]

[1] Brno University of Technology, Brno, Czech Republic
[2] Aalborg University, Aalborg, Denmark
[3] Key Lab. of System Software (Chinese Academy of Sciences), ISCAS, Beijing, PRC
[4] Slovak University of Technology in Bratislava, Bratislava, Slovakia
xalexa09@stud.fit.vut.cz, ihavlena@fit.vut.cz, holik@fit.vut.cz, lengal@fit.vut.cz, liyong@ios.ac.cn, nicolas.mazzocchi@stuba.sk

Abstract. We present Kofola, an efficient tool for complementation and inclusion checking of Büchi automata, two central tasks in automata-theoretic verification with applications in model checking, monitoring, and theorem proving. Kofola implements a state-of-the-art modular complementation framework that decomposes the input automaton into strongly connected components and applies to each component a complementation algorithm tailored to its structural properties. Building on this modular construction, Kofola also provides modular *inclusion checking* with new heuristics. A key ingredient is a new on-the-fly emptiness-checking algorithm for the simple generalized Rabin pair condition produced by our complementation, allowing the search to terminate as soon as the explored state space suffices. Empirical evaluation shows that Kofola is highly competitive with state-of-the-art complementation and inclusion-checking tools: it is the most robust tool in our evaluation and often outperforms competitors by several orders of magnitude on benchmarks from practical applications.

1 Introduction

Language inclusion checking for Büchi automata (BAs) is a cornerstone problem in automata-theoretic verification, with applications to model checking [51], monitoring [19], and theorem proving [42,31]. The problem is PSpace-complete [35], but it remains one of the most computationally demanding tasks in verification practice. As systems and specifications continue to grow in size and complexity, the scalability of inclusion checking has become a decisive factor in the practicality of automata-based reasoning frameworks. For two ω-regular languages L_1 and L_2, deciding whether $L_1 \subseteq L_2$ reduces to checking the emptiness of $L_1 \cap \overline{L_2}$, where $\overline{L_2}$ denotes the complement of L_2. This exposes two key challenges: (i) constructing $\overline{L_2}$ compactly, and (ii) efficiently exploring the (often enormous) state space of the product automaton for $L_1 \cap \overline{L_2}$. Addressing either of these challenges translates into performance improvements in the considered domains.

Existing tools primarily tackle these challenges by improving either the complementation of L_2 or the exploration for the product automaton. For example, tools such as Bait [21], Forklift [20], and Rabit [4,3] use Ramsey-based complementation [13,48], while improving efficiency through state-space reduction techniques—such as subsumption and simulation in Rabit or well-quasiorders in Bait and Forklift. In contrast, Spot [22] relies on determinization-based complementation [45,43,44], enhanced by lightweight simulation and specialized emptiness checking procedures for Emerson–Lei acceptance conditions [9]. We observe that state-of-the-art inclusion checkers still largely rely on classical Ramsey- or determinization-based complementation constructions, which date back more than three decades [13,45], despite substantial recent progress in Büchi complementation. This motivates an efficient and extensible framework that brings modern complementation techniques to practical inclusion checking.

Contributions. We present Kofola [2], an efficient and modular tool for BA complementation and language inclusion checking, which is based on a recent decomposition-based complementation algorithm [26]. For an inclusion query $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$, Kofola performs a fine-grained structural analysis of the complemented automaton, namely $\mathcal{A}_2$, by classifying its maximal strongly connected components (SCCs) as: (i) non-accepting components; (ii) inherently weak accepting components (IWACs), in which all cycles accept; (iii) initial almost deterministic accepting components (IADACs) and deterministic accepting components (DACs), which have only deterministic transitions (a IADAC is a deterministic SCC reachable from initial states through deterministic SCCs only, allowing nondeterminism just on cycle-free parts between SCCs); and (iv) the remaining accepting SCCs, classified as nondeterministic accepting components (NACs). Notably, IADACs are a *newly identified component type* introduced in this paper. For each component type, Kofola exploits its structural properties by applying a dedicated complementation algorithm, combining the component constructions in a synchronous manner.

For inclusion checking, Kofola constructs an ω-automaton with a single generalized Rabin pair acceptance condition, $\mathit{Fin}(0) \wedge \mathit{Inf}(1) \wedge \cdots \wedge \mathit{Inf}(k\text{-}1)$, and performs *on-the-fly* emptiness checking during the modular construction. This avoids explicitly generating the full product automaton. Although Spot can also check emptiness for generalized Rabin conditions [9], it requires the whole automaton to be constructed explicitly. To mitigate state-space explosion, Kofola implements a new emptiness-checking algorithm for the single generalized Rabin pair condition. The algorithm can be viewed as a variant of Couvreur's algorithm [18] extended with a *Fin* predicate. It is maximally lazy: the search terminates as soon as the explored states suffice to decide the result, helping to keep the generated part of the complement small, which is crucial since complementation is the main source of blow-up in inclusion checking.

We evaluated Kofola on complementation benchmarks drawn from diverse literature sources, comparing it with state-of-the-art tools Spot, Ranker [29], and an initial implementation of decomposition-based complementation [26]. Kofola outperforms all competitors both in the number of solved instances

and in the size of the resulting complement automata. We further evaluated KOFOLA on BA inclusion benchmarks arising from program termination checking, verification of concurrent systems, model checking of hyperproperties, and theorem proving. The results show that KOFOLA outperforms the state-of-the-art inclusion checkers BAIT, FORKLIFT, RABIT, and SPOT, in many cases by several orders of magnitude, and KOFOLA solves the most instances. These results establish KOFOLA as an efficient, extensible, and highly competitive tool for BA complementation and inclusion checking. More details are given in [6].

Related work. BA complementation algorithms can be broadly grouped into several families. *Ramsey-based* [13,48,12,37], *rank-based* [36,25,46,33,15,28,30,29], and *slice-based* [32,50,7] procedures encode information about word acceptance using different data structures. *Determinization-based* constructions first determinize the automaton and then complement its acceptance condition, typically producing automata with acceptance conditions that are more complex than Büchi [45,43,44,40,38]. *Learning-based* approaches [39] infer a BA for the complement through membership queries. More recent *decomposition-based* constructions [38,26] partition the automaton according to structural properties and apply specialized complementation procedures to each block. KOFOLA builds on this decomposition-based approach, introduces the new IADAC component type, and, as our experiments show, improves over previous implementations [38,26] as well as rank- and determinization-based tools such as RANKER and SPOT.

Several works also optimize BA language-inclusion testing, mostly within Ramsey-based frameworks. RABIT [4,3] uses simulation and subsumption, both within and across automata, to prune the searched state space. Well-quasiorder-based approaches [21,20] are more symbolic: to decide $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$, they perform a breadth-first fixpoint search over ultimately periodic words of $\mathcal{L}(\mathcal{A}_1)$, using a well-quasiorder to discard words subsumed by already processed ones.

2 Tool Architecture

KOFOLA is implemented in C++ and available at [2] under the GNU GPL 3.0 license. Its architecture is shown in Fig. 1. The *Frontend* uses SPOT [22] to read input automata $\mathcal{A}_1$`.hoa` and, for inclusion checking only, $\mathcal{A}_2$`.hoa`, both in the `HOA` [8] format. Next, the automata enter *Preprocessing and setup*, where (i) SPOT reduces them using either `Low` or `High` reduction level, depending on their features and (ii) KOFOLA then analyzes their structure and chooses partial complementation algorithms to be used. Then, the automaton/automata are passed to top-level procedures for *Complement* (see Section 4.1) or *Inclusion*, which orchestrate the chosen partial complementation algorithms (see Section 4.2); for inclusion checking, this includes

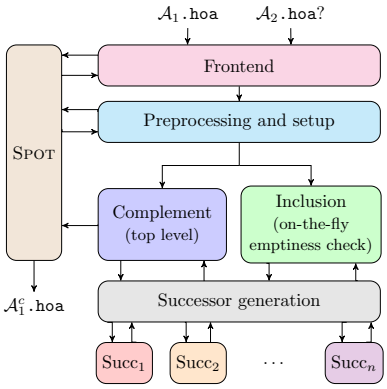

Fig. 1: Architecture of KOFOLA

a top-level on-the-fly emptiness checking algorithm (see Section 5). The main workhorse of the procedures is *Successor generation* used in generating reachable states of the complement automaton, which calls the successor functions $Succ_1, \ldots, Succ_n$ of the partial complementation algorithm and combines results. For complementation, the resulting automaton is postprocessed by Spot (reduction level `Low`) and output as $\mathcal{A}_1^c$`.hoa`.

3 Preliminaries

We assume familiarity with standard notions on ω-automata; see, e.g., [24]. We only introduce definitions specific to this paper. Fix a finite non-empty alphabet Σ, and let ω be the first infinite ordinal. An (infinite) word is a sequence $w = w_0 w_1 \ldots$ over Σ; the set of all infinite words over Σ is denoted by Σ^ω.

A (nondeterministic transition-based) *simple generalized Rabin automaton* (SGRA) over Σ is a quintuple $\mathcal{A} = (Q, \delta, I, \Gamma, \mathsf{p})$ where Q is a finite set of *states*, $\delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions* (denoted as $q \xrightarrow{a} r$), $I \subseteq Q$ is the set of *initial* states, $\Gamma = \{0, \ldots, k\text{-}1\}$ is a set of k *colours*, and $\mathsf{p}\colon \delta \to 2^\Gamma$ is a colouring function on transitions. For a set of states $C \subseteq Q$, we write $\delta|_C$ to denote the set of transitions $\{q \xrightarrow{a} r \in \delta \mid q, r \in C\}$. We call $\mathcal{A}$ *deterministic* if $|I| = 1$ and for every $a \in \Sigma$ and $q \in Q$ it holds that $|\{p \mid (q, a, p) \in \delta\}| \leq 1$. When 0 has no occurrence in p, $\mathcal{A}$ is a *generalized Büchi automaton* (GBA). When a GBA has $\Gamma = \{0, 1\}$, it is a *Büchi automaton* (BA). We denote a BA as $\mathcal{A} = (Q, \delta, I, \delta_{Acc})$ where $\delta_{Acc} = \{\tau \in \delta \mid \mathsf{p}(\tau) = \{1\}\}$. A *run* of $\mathcal{A}$ from $q \in Q$ on $w \in \Sigma^\omega$ is an infinite sequence $\rho\colon \omega \to Q$ that $\rho_0 = q$ and $\rho_i \xrightarrow{w_i} \rho_{i+1} \in \delta$ for all $i \geq 0$. Let $inf(\rho)$ denote the transitions occurring in ρ infinitely often. The run ρ is called *accepting* iff $\bigcup\{\mathsf{p}(\tau) \mid \tau \in inf(\rho)\} = \Gamma \setminus \{0\}$. Equivalently, in the Emerson–Lei formalism [23,8], SGRAs have acceptance condition $Fin(0) \wedge Inf(1) \wedge \ldots \wedge Inf(k\text{-}1)$. A word w is accepted by $\mathcal{A}$ from a state q if there exists an accepting run of $\mathcal{A}$ from q on w. We write $\mathcal{L}(\mathcal{A})$ for the language accepted from the initial states I. A state q is *useless* if no word is accepted from q.

For a BA $\mathcal{A}$, a non-empty set $C \subseteq Q$ is a *strongly connected component* (SCC) if all states in C can reach each other and C is a maximal such set. An SCC is *trivial* if it is a singleton without a self-loop, and *non-trivial* otherwise. An SCC C is *accepting* if it contains at least one 1-transition and *inherently weak* iff either (i) every run in $\mathcal{A}$ restricted to C is accepting, or (ii) no run in $\mathcal{A}$ restricted to C is accepting. An SCC C is *deterministic* iff the BA $(C, \delta|_C, \{q\}, \emptyset)$ for any $q \in C$ (obtained from $\mathcal{A}$ by removing transitions outside C) is deterministic, and it is *initial deterministic* if the BA obtained from the BA $(Q, \delta, I, \delta_{Acc}|_C)$ (i.e., $\mathcal{A}$ with accepting transitions outside C removed) by removing useless states is deterministic. In particular, a non-accepting SCC is not initial deterministic since the resulting BA has no initial states. We also generalize the notion of initial deterministic components to *initial almost deterministic components*, which are components C such that in the BA obtained from the BA $(Q, \delta, I, \delta_{Acc}|_C)$ by removing useless states, it holds that for any two transitions from the same state over the same symbol $q \xrightarrow{a} p$ and $q \xrightarrow{a} r$, either $p = r$ or the states p

and r are not in the same SCC as the state q. Hence, such components are reachable from initial states on a path going through deterministic components with nondeterminism allowed only outside non-trivial SCCs. We use the following categorization of accepting SCCs of a BA: (i) **IADACs** are initial almost deterministic accepting SCCs, (ii) **IWACs** are inherently weak accepting SCCs that are not IADACs, (iii) **DACs** are deterministic accepting SCCs not covered by the above, and (iv) **NACs** are the remaining (nondeterministic) accepting SCCs. A BA $\mathcal{A}$ is called an *elevator automaton* [30] if it contains no NAC. We assume that for all transitions $\tau = q \xrightarrow{a} r \in \delta$ not within a single SCC, we have $\mathsf{p}(\tau) = \emptyset$ (this is without loss of generality since no run can loop over such transitions). A *partition block* $P \subseteq Q$ of $\mathcal{A}$ is a nonempty union of $\mathcal{A}$'s accepting SCCs, and a *partitioning* of $\mathcal{A}$ is a sequence $P_1, \ldots, P_n$ of pairwise disjoint partition blocks of $\mathcal{A}$ that contains all accepting SCCs of $\mathcal{A}$.

4 Decomposition-based Büchi Complementation

We fix a BA $\mathcal{A} = (Q, \delta, I, \delta_{Acc})$ for the rest of this section. We first give a brief overview of the complementation framework proposed in [26] and then present our choices of partial complementation algorithms for each type of partition blocks. We also extend the framework of [26] with a new partial complementation algorithm for IADACs.

4.1 Top-level Modular Complementation Algorithm

Inspired by [38], a modular methodology was proposed in [26] for complementing BAs. In a nutshell, the modular complementation algorithm developed in [26] first classifies the SCCs in $\mathcal{A}$ into several partition blocks according to their structural properties, then performs complementation for each of the partition blocks independently, while synchronizing the complementation algorithms for all partition blocks in each step. The intuition for complementing SCCs separately is that every run of $\mathcal{A}$ eventually gets trapped in one SCC, so we only need to consider the runs staying in the given SCC. For those runs that exit the given SCC at some point, we handle them as discontinued runs since they must eventually stay in another SCC. The complement automaton $\mathcal{A}^c$ accepts a word over which all runs of $\mathcal{A}$ are rejecting, so we only need to make sure that all runs in *accepting* SCCs over the word do not visit accepting transitions infinitely often. That is, we check only the runs in IADACs, IWACs, DACs, and NACs. Runs in rejecting SCCs do not require checking acceptance, so we just track them using traditional subset construction. We choose to decompose $\mathcal{A}$ into partition blocks where one partition block contains all IADACs, one contains all IWACs, one contains all DACs, and each NAC has a separate partition block. However, thanks to the modularity of the complementation framework, we can decompose $\mathcal{A}$ in other ways and even use different partial complementation algorithms for partition blocks of the same type. The modular complementation algorithm uses a partial algorithm $\texttt{Alg}^P$ for each partition block P. Each partial algoritm $\texttt{Alg}^P$ specifies the

following: (i) the type of macrostates produced by the algorithm, (ii) the set of initial macrostates, (iii) a function `Succ` returning the successors of a macrostate, possibly together with a transition color, and (iv) the acceptance condition for the partial algorithm.

The input of the top-level algorithm are, except the automaton itself, also partitions $P_1, \ldots, P_k$ of the input automaton and the partial algorithms $\texttt{Alg}^{P_1}, \ldots, \texttt{Alg}^{P_k}$. The macrostates of the top-level algoritm are then of the form $(S, M_1, \ldots, M_k)$, where S is the set of reachable states and $M_1, \ldots, M_k$ are macrostates of the partial algorithms. The successors of a macrostate are built from the results of partial algorithms' successor functions (and colors from partial successor functions are gathered on the transition) as shown in Fig. 2. The acceptance condition of $\mathcal{A}^c$ is given as a conjunction of acceptance conditions of the partial algorithms.

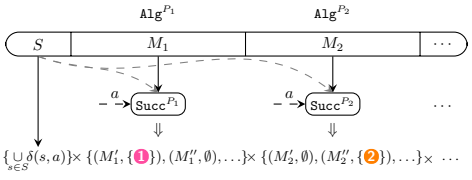

Fig. 2: Visualization of macrostate's successor computation over symbol a.

4.2 Partial Complementation Algorithms

Next, we briefly describe partial algorithms to complement partition blocks with IWACs, DACs, and NACs. Details and formal definitions can be found in [26]. Finally, we also present a new partial complementation algorithm for partition blocks with IADACs only. The sole source of the $Fin(\mathbf{0})$ acceptance condition comes from IADACs, making the resulting automaton an SGRA.

IWACs. The algorithm for the partition block $\mathcal{W}$ that contains all IWACs exploits the classical *Miyano-Hayashi* construction [41], generating the acceptance condition $Inf(\mathbf{1})$.

DACs. For the partition block with DACs, we use a modification of the NCSB algorithm [11,16,29], which generates the acceptance condition $Inf(\mathbf{2})$.

NACs. For partition blocks containing a NAC $\mathcal{N}$, Kofola supports two algorithms: (i) determinization-based (based on the Safra-Piterman's [45,43] algorithm, specifically its version from [38]), which translates $\mathcal{N}$ into a deterministic parity automaton, and (ii) a slice-based approach from [7] (this approach generates for a given NAC the acceptance condition $Inf(\mathbf{k})$ with $k > 2$; we note that $\mathbf{0}$, $\mathbf{1}$, and $\mathbf{2}$ are reserved for the procedures for IADACs, IWACs, and DACs respectively). Each of the algorithms is used in a different setting. The determinization-based algorithm is used in *complementation*, since it usually produces smaller results. On the other hand, the slice-based algorithm is used in *inclusion checking*, since it is more suitable for on-the-fly product emptiness test.

IADACs. Let us now generalize the partial algorithm for complementing initial deterministic components (IDCs) from [26] to IADACs. We show the construction on a BA consisting of IADACs only. Putting the construction into the formal framework of [26] is straightforward. Given a BA $\mathcal{A} = (Q, \delta, I, \delta_{Acc})$ consisting of IADACs only, the complement SGRA with the acceptance condition $Fin(\mathbf{0})$ is given as $\mathcal{B} = (2^Q, \Delta, \{I\}, \{\mathbf{0}\}, \mathsf{p})$ where $\Delta = \{R \xrightarrow{a} S \mid \bigcup_{r \in R} \delta(r, a) = S\}$ and

for a transition $t = R \xrightarrow{a} S$ the coloring function p is given as $\mathsf{p}(t) = \{\textbf{0}\}$ if $\bigcup_{r \in R} \delta_{Acc}(r, a) \neq \emptyset$ otherwise $\mathsf{p}(t) = \emptyset$.

Intuitively, the algorithm determinizes the IADACs using a subset construction and emits a rejecting (due to the *Fin* condition) mark **0** whenever some tracked run sees an accepting transition. The key argument for correctness is that the number of runs in IADACs is bounded (the directed acyclic graph representing the run—called the *run DAG*—has a finite number of branches), due to IADACs having non-deterministic branching only outside of (non-trivial) SCCs. (On the other hand, observe that a DAC having non-deterministic branching between another DAC and itself would generate unboundedly many runs.)

Theorem 1. $\mathcal{L}(\mathcal{A}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.

Proof (Sketch). Since all IADAC runs are deterministic, deciding whether w is accepted amounts to checking whether the runs on w visit δ_{Acc}-transitions only finitely often. By construction, w is not accepted in the IADAC iff every run of $\mathcal{A}$ over w emits only finitely many **0**'s; equivalently, the corresponding run in $\mathcal{B}$ emits **0** only finitely often, as captured by the acceptance condition $\mathit{Fin}(\textbf{0})$. □

5 Efficient Büchi Inclusion Checking

On the top level, our algorithm for testing the inclusion $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ for BAs $\mathcal{A}_1$ and $\mathcal{A}_2$ checks emptiness of the SGRA $\mathcal{A}$ constructed as the intersection of $\mathcal{A}_1$ and the SGRA $\mathcal{A}_2^c$ for the complement of $\mathcal{A}_2$. Note that $\mathcal{A}$ can be explored by performing the product construction of $\mathcal{A}_1$ with $\mathcal{A}_2^c$ on the fly, extending the colours of $\mathcal{A}_2^c$ with one more colour that represents the acceptance of $\mathcal{A}_1$. Since the biggest asymptotic state space explosion comes from generating the complement $\mathcal{A}_2^c$, our goal is to have an emptiness checking algorithm that is maximally lazy in the sense that it stops generating new states of $\mathcal{A}_2^c$ as soon as the already generated states are sufficient for establishing the emptiness of $\mathcal{A}$. This precludes the use of algorithms such as [9], which require the whole automaton to identify its SCCs. To achieve this goal, below, we describe a new language emptiness algorithm for SGRAs that satisfies the laziness condition.

Algorithm 1 performs on-the-fly emptiness test for SGRAs. It uses the notation $\mathtt{src}(t)/\mathtt{tgt}(t)$ for the source/target state of a transition t, respectively, $\mathtt{outgoing}(q)$ to denote the outgoing transitions of a state q, and lifts the colouring function p to sets in the usual manner. The algorithm is an augmentation of the on-the-fly emptiness check for GBAs by Couvreur [18], which may also be seen as a variation of Tarjan's algorithm [49] for exploring SCCs. To avoid technicalities, Algorithm 1 is based on its high-level presentation. We emphasize that although this algorithm does not immediately resemble Tarjan's algorithm, the merged sets of transitions might efficiently be implemented similarly as Tarjan's SCCs, using a technique resembling Tarjan's stack with its low-links.

The algorithm explores the automaton in a depth-first manner from I. The stack S stores the currently explored path. When a back-edge transition t (a transition to a state already on the stack) is detected, the top part of the stack from

Algorithm 1: On-the-fly emptiness test for SGRAs

Input: an SGRA $\mathcal{A} = (Q, \delta, I, \Gamma = \{0, 1, \ldots, k\text{-}1\}, \mathtt{p})$
Output: *true* iff $\mathcal{L}(\mathcal{A}) = \emptyset$, *false* otherwise
Data: sets of transitions *Entry*, *Explored*, a stack S of sets of transitions

```
1  Entry ← outgoing(I); Explored ← ∅; S ← ();
2  while ∃t ∈ Entry \ Explored do Explore(t) ;
3  return true;
4  Procedure Explore(t):
5      Explored ← Explored ∪ {t};
6      if 0 ∈ p(t) then Entry ← Entry ∪ {outgoing(tgt(t))} ;
7      else
8          S.push({t});
9          Merge the stack S from the lowest occurrence of tgt(t) up to top;
10         if {1, ..., k-1} ⊆ p(S.top) then exit program and return false ;
11         while ∃t' ∈ outgoing(tgt(S.top)) \ Explored do Explore(t') ;
12         S.pop();
```

the element including the target of t is merged and the united set of transitions is placed on the top of the stack. The set represents a (non-maximal) SCC. The stack thus becomes a sequence of sets of transitions. Line 9 specifically transforms the stack S, a sequence of sets of transitions $T_1, \ldots, T_n$ (the top being the rightmost element), into $T_1, \ldots, T_{\ell-1}, \bigcup_{i=\ell}^{n} T_i$ where ℓ is the lowest index such that T_ℓ contains a transition t' with $\mathtt{src}(t') = \mathtt{tgt}(t)$, or it can be also $\mathtt{tgt}(t') = \mathtt{tgt}(t)$ if T_ℓ has a cycle. Next, Couvreur's algorithm concludes non-emptiness when it detects a GBA cycle, i.e., a merged set of transitions on top of the stack that contains all colours. Algorithm 1, on the other hand, needs to find a *simple generalized Rabin accepting cycle*, i.e., one that satisfies the GBA acceptance and does not contain 0. In other words, it is a GBA accepting cycle with respect to the set $\{1, \ldots, k\text{-}1\}$ in the automaton $\mathcal{A} \setminus 0$ (the automaton $\mathcal{A}$ with 0-transitions removed) that is still reachable from I in $\mathcal{A}$. To detect such cycles, we run a variant of Couvreur's algorithm as a sub-procedure, each time in a part of $\mathcal{A} \setminus 0$ discovered to be reachable in $\mathcal{A}$. Each run of Couvreur's algorithm is used (i) to detect SGRA accepting cycles in a part of $\mathcal{A} \setminus 0$ rooted at some transition t; and (ii) to discover new parts of $\mathcal{A} \setminus 0$ reached from the explored part by 0-transitions. This is iterated until a GBA accepting cycle is found or there are no more unexplored reachable parts of $\mathcal{A} \setminus 0$.

Algorithm 1 uses the set *Entry* of *entry transitions* as a data structure initialized by the outgoing transitions of I (Line 1), and applies Couvreur's algorithm to explore parts of $\mathcal{A} \setminus 0$ from all transitions of *Entry*, while the iterations of the Couvreur's algorithm also keep adding transitions to *Entry* (due to the **while** loop on Line 2). We modify the internals of Couvreur's algorithm, presented essentially as the procedure `Explore` on Line 4, in one point: When exploring a 0-transition, the normal depth-first exploration (the **else** branch on Line 7) is bypassed, the transition is only marked as explored (Line 5), and the transitions continuing from its target are put to *Entry* as the entry transitions of newly discovered parts of $\mathcal{A} \setminus 0$ (Line 6).

Theorem 2. *Algorithm 1 terminates and returns false iff* $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

Proof (Sketch). Algorithm 1 terminates since each visited transition is stored in *Explored* and never revisited, and $\mathcal{A}$ has finitely many transitions. The proof follows the proof of Couvreur's algorithm. We show that $\mathcal{A}$ has an accepting run iff Algorithm 1 returns *false*. First, $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff $\mathcal{A} \setminus$ 0 contains a reachable cycle visiting all colours 1, ..., k-1. Whenever a transition t has colour 0, the algorithm stops the current DFS branch and schedules $\texttt{tgt}(t)$ as a new entry point (Line 6). Since all explored transitions are reachable from an initial state, every state in *Entry* is reachable; so the search effectively explores $\mathcal{A} \setminus$ 0. Each newly visited transition is pushed onto S (Line 8). If it closes a cycle, namely if $\texttt{tgt}(t)$ equals $\texttt{src}(t')$ or $\texttt{tgt}(t')$ for some explored t' on S, the colours on that cycle are collected and stored in $S.\texttt{top}$ (Line 9). If the algorithm returns *false*, then $S.\texttt{top}$ holds all colours 1, ..., k-1 (Line 10). So a reachable cycle in $\mathcal{A} \setminus$ 0 visiting all non-0 colours has been found, satisfying the accepting condition; therefore $\mathcal{A}$ has an accepting run. Conversely, if $\mathcal{A}$ has an accepting run, then $\mathcal{A} \setminus$ 0 contains a reachable cycle visiting all non-0 colours. Once the search reaches the cycle, it coincides with Couvreur's algorithm on $\mathcal{A} \setminus$ 0 and must discover it. So Line 10 is eventually triggered, and Algorithm 1 returns *false*. □

6 Experimental Evaluation

We have experimentally evaluated the complementation and inclusion checking capabilities of KOFOLA and compared it with state-of-the-art tools. All experiments were run on an Ubuntu GNU/Linux 24.04 virtual machine with 64 GiB RAM running on an AMD EPYC 9124 CPU. The timeout was set to 120 s. Correctness of answers was evaluated by cross-comparison between the tools.

6.1 Complementation

Used Tools. For complementation, the default setting of KOFOLA uses the determinization-based approach (see Section 4.2) for complementing NACs. We use KOFOLASLICE to denote the version that complements NACs using the slice-based construction. Before outputting a result, we reduce it using SPOT's post-processor with level set to `Low` (this performs removal of useless states and one run of the simulation-based reduction [14]). We compared KOFOLA against state-of-the-art tools for Büchi complementation: SPOT (version 2.14.2) [22] (using the command line tool `autfilt --complement`), RANKER [29], OWL [34] (we ran its determinization into a parity automaton, where complementation is trivial), and the original version of KOFOLA from [26], named KOFOLAOLD hereafter. KOFOLAOLD uses the same post-processing as KOFOLA and RANKER uses a custom post-processing, which is not exactly the same, but similar to the one in SPOT.

Benchmarks. We collected as many BAs as we could, obtaining, in total, 31,273 automata [1]. These come from various sources: (i) 915 BAs from checking program termination by ULTIMATE AUTOMIZER [16] (AUTOMIZER$_C$), (ii) 6,301 automata from solving the first-order logic of Sturmian words by PECAN [42,31]

Table 1: Statistics for complementation. The column ✗ has numbers of automata (out of 31,273) for which the tool ran out of resources (time or memory), **avg** contains the average number of states of the output, **time** the total time in seconds, and **unsup** the number of input automata with unsupported acceptance condition.

tool	✗	avg	time	unsup
Kofola	0	79.24	1,069	0
Spot	2	115.38	199	0
KofolaSlice	4	166.03	1,376	0
KofolaOld	44	42.74	2,471	0
Owl	104	37.39	1,240	507
Ranker	443	45.80	22,817	1,336

Table 2: Statistics for inclusion checking. Column **unsolved** has numbers of inclusion problems (out of 1,014) for which the tool ran out of resources (time/memory), **time** contains the total time (in seconds) for the solved problems, **wins**/**losses** contains the number of problems where Kofola strictly won/lost over the tool, and **missing** contains for how many problems we could not convert the inputs into `.ba` files.

tool	unsolved	time	wins	losses	missing
Kofola	2	141			0
Spot(Det)	28	122	263	56	0
Forklift	35	1,273	997	9	8
Spot(Forq)	54	1,134	307	89	0
Rabit	57	4,300	996	8	8
Bait	64	1,981	1,000	5	8

(Pecan$_\mathcal{C}$; these are not only BAs, but also, e.g,. co-Büchi and parity automata), (iii) 72 BAs from model checking of HyperLTL properties using AutoHyper [10] (AutoHyper$_\mathcal{C}$), (iv) 370 BAs from deciding S1S formulae [27] (S1S), (v) 18 BAs from LTL to limit-deterministic BA translation [47] (Ldba4Ltl), (vi) 1,721 BAs from translation from real-world and random LTL formulae [11] (Seminator), and (vii) 21,876 randomly generated BAs [50] (Sobc).

Results. We compared the sizes of the complement automata that the tools generated. We let the tools output automata with any acceptance condition (Kofola and Spot could exploit this to obtain smaller automata, whereas Ranker always generates automata with the Büchi condition). Statistics of the experiment are in Table 1 and interesting scatter plots are in Fig. 3 (the scatter plot for Owl looks similar as the one for Spot with more unsolved cases). We do not include the median, which was 2 for all tools, in the table. We can see that Kofola was the only tool that could complement all input automata.

The second tool, Spot, could complement all but two automata (it ran out of memory for those), however the average size of the output is much bigger. We can see this also in the plot comparing the two tools, where Kofola can get much smaller complements for many Sobc automata. For complementation, KofolaSlice gives worse average results than Kofola. There are, however, cases where KofolaSlice outputs a smaller result. The improvement of Kofola over KofolaOld can be seen mainly in the number of solved cases. While the average size is better for KofolaOld, this should be considered in the context that Kofola can solve the 44 benchmarks that KofolaOld could not, often with a large output, which skews the average. From the plot, we can see that KofolaOld could in many cases output a slightly smaller automaton than Kofola. This is due to the settings of the algorithms, which we optimized in Kofola to maximize the number of solved cases; as a consequence, sometimes the algorithms perform slightly worse. The dominance of Kofola over

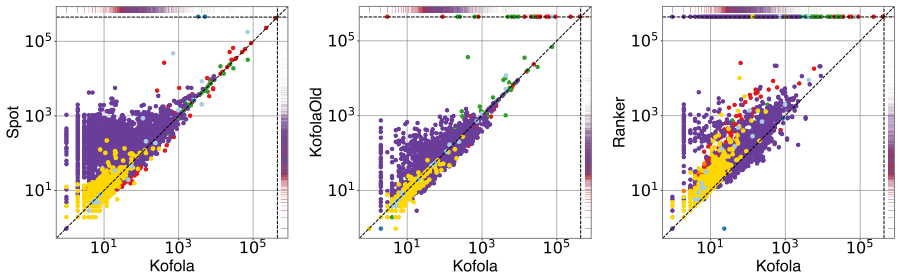

Fig. 3: Comparison of state counts of complement automata generated by KOFOLA and other tools; axes are logarithmic. Dashed lines are out-of-resources limit and rug plots on margins show distribution of data points. Colour indicates benchmark source: SOBC ●, SEMINATOR ●, AUTOMIZER$_\mathcal{C}$ ●, LDBA4LTL ●, AUTOHYPER$_\mathcal{C}$ ●, PECAN$_\mathcal{C}$ ●, S1S ●.

RANKER and OWL is clear from the number of solved cases and the scatter plot (RANKER has a lower average since it cannot solve harder benchmarks). 1,336 (for RANKER) and 507 (for OWL) input automata (all from PECAN) used an unsupported acceptance condition.

Our procedure for complementing IADACs helped in 225 cases, reducing the number of states of the output by an average of 5 (among the cases where it had effect). The most significant improvement was observed on one automaton from the SEMINATOR benchmark, where the number of states decreased by 236.

6.2 Inclusion Checking

Used Tools. For inclusion, KOFOLA uses the slice-based approach (Section 4.2) for complementing NACs, which is more suitable for on-the-fly exploration. We evaluated KOFOLA against state-of-the-art inclusion checkers BAIT [21], FORKLIFT [20], RABIT [17] v2.5.1 with argument `-fast`, and SPOT 2.14.2 [22] (using `autfilt --included-in`). For SPOT, we tried two versions: determinization-based [45,43,44] (SPOT(DET)) and well-quasiorder-based [20] (SPOT(FORQ)).

Benchmarks. We used inclusion problems involving BAs given in the `HOA` format [8] from all available sources we are aware of that are practically motivated (the number of automata pairs is given in parentheses): (i) inclusion problems emerging from model checking of hyperproperties using the tool AUTOHYPER [10], denoted as AUTOHYPER$_\mathcal{I}$ (36; max. 66,051, avg. 2,059 states), (ii) instances occurring in program termination checking using ULTIMATE AUTOMIZER [16], denoted as AUTOMIZER$_\mathcal{I}$ (404; max. 88,304, avg. 904 states), (iii) inclusion problems coming from verification of concurrent systems [4] (13 instances denoted as CONCUR, omitting one benchmark from the source due to a missing counterpart automaton; max. 1,532, avg. 589 states), and (iv) logical implication tasks in word combinatorics from PECAN [42,31], denoted as PECAN$_\mathcal{I}$ (54 instances where it was clear how to pair them; max. 712,184, avg. 7,729 states). Without performing any pre-selection on the existing benchmarks

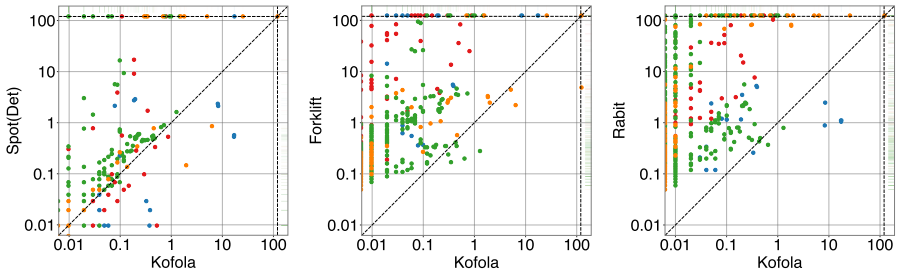

Fig. 4: Comparison of runtime of inclusion checking for KOFOLA and other tools. Times are in seconds, axes are logarithmic. Dashed lines represent out-of-resources limit and rug plots on margins show distribution of data points. The colour of a point indicates the source of the benchmark: AUTOHYPER$_{\mathcal{I}}$ ●, CONCUR ●, PECAN$_{\mathcal{I}}$ ●, AUTOMIZER$_{\mathcal{I}}$ ●.

of the literature, 98 % of the 507 BAs appearing on the right-hand side of the inclusion (i.e., those that are being complemented) are elevator automata. This emphasizes the importance of exploiting automata structure in inclusion checking. From the automata on the left-hand side, 76 % were elevator automata (in total, 87 % of the involved BAs were elevator). To get more interesting examples for inclusion checking, we also swapped the sides of the benchmarks for a total of 1,014 inclusion problems (the vast majority of these benchmarks were non-trivial, often harder than the original problems). For tools not supporting the `HOA` format (BAIT, FORKLIFT, and RABIT), we employed SPOT to convert the input automata into the `.ba` format. For four of the original inclusion problems (eight benchmarks in total), this was not possible (the translation took too many resources). Overall, inclusion holds in 12 % of the benchmarks.

Results. We give statistics for inclusion checking in Table 2 and interesting scatter plots in Fig. 4. KOFOLA won by far on the number of solved instances and the total time. The second tool was SPOT(DET), whose time is slightly lower, but solved 26 less instances. The scatter plot shows that it can sometimes beat KOFOLA. Ramsey-based tools follow with run times being an order of magnitude worse. The best of them is FORKLIFT (surprisingly to us, it solves more problems than SPOT(FORQ)) with SPOT(FORQ), RABIT, and BAIT trailing. From the scatter plots (we omit BAIT, which mostly behaves worse than FORKLIFT and SPOT(FORQ)), the tool most complementary to KOFOLA is SPOT.

Acknowledgments. We thank the anonymous reviewers for their constructive feedback. This work was supported in part by the National Natural Science Foundation of China (Grant No. 62102407), the CAS Project for Young Scientists in Basic Research (Grant No. YSBR-040), the Czech Science Foundation projects 26-22640S and 25-17934S, and the FIT BUT internal project FIT-S-26-9011.

Disclosure of Interests. The authors have no competing interests.

Data Availability Statement. An environment with the tools and data used for the experimental evaluation in the current study is available at [5].

References

1. Automata-benchmarks. https://github.com/ondrik/automata-benchmarks/tree/master/omega/ (2025)
2. KOFOLA. https://github.com/VeriFIT/kofola (2025)
3. Abdulla, P.A., Chen, Y., Clemente, L., Holík, L., Hong, C., Mayr, R., Vojnar, T.: Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 132–147. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_14, https://doi.org/10.1007/978-3-642-14295-6_14
4. Abdulla, P.A., Chen, Y., Clemente, L., Holík, L., Hong, C., Mayr, R., Vojnar, T.: Advanced Ramsey-based Büchi automata inclusion testing. In: Katoen, J., König, B. (eds.) CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6901, pp. 187–202. Springer (2011). https://doi.org/10.1007/978-3-642-23217-6_13, https://doi.org/10.1007/978-3-642-23217-6_13
5. Alexaj, O., Havlena, V., Holík, L., Lengál, O., Li, Y., Mazzocchi, N.: Artifact: KOFOLA 1.0: A modular approach to ω-regular complementation and inclusion checking (october 2025). https://doi.org/10.5281/zenodo.17478623, https://doi.org/10.5281/zenodo.17478623
6. Alexaj, O., Havlena, V., Holík, L., Lengál, O., Li, Y., Mazzocchi, N.: Kofola 1.0: A modular approach to ω-regular complementation and inclusion checking (technical report). CoRR **abs/2605.15390** (2026), http://arxiv.org/abs/2605.15390
7. Allred, J.D., Ultes-Nitsche, U.: A simple and optimal complementation algorithm for Büchi automata. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 46–55. ACM (2018). https://doi.org/10.1145/3209108.3209138, https://doi.org/10.1145/3209108.3209138
8. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Kretínský, J., Müller, D., Parker, D., Strejcek, J.: The Hanoi omega-automata format. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 479–486. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_31, https://doi.org/10.1007/978-3-319-21690-4_31
9. Baier, C., Blahoudek, F., Duret-Lutz, A., Klein, J., Müller, D., Strejcek, J.: Generic emptiness check for fun and profit. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 445–461. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_26, https://doi.org/10.1007/978-3-030-31784-3_26

10. Beutner, R., Finkbeiner, B.: AutoHyper: Explicit-state model checking for HyperLTL. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13993, pp. 145–163. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_8, https://doi.org/10.1007/978-3-031-30823-9_8
11. Blahoudek, F., Duret-Lutz, A., Strejček, J.: Seminator 2 can complement generalized Büchi automata via improved semi-determinization. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 15–27. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_2, https://doi.org/10.1007/978-3-030-53291-8_2
12. Breuers, S., Löding, C., Olschewski, J.: Improved Ramsey-based Büchi complementation. In: Birkedal, L. (ed.) Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7213, pp. 150–164. Springer (2012). https://doi.org/10.1007/978-3-642-28729-9_10, https://doi.org/10.1007/978-3-642-28729-9_10
13. Büchi, J.R.: On a decision method in restricted second order arithmetic, logic, methodology and philosophy of science (proc. 1960 internat. congr.) (1962)
14. Bustan, D., Grumberg, O.: Simulation based minimization. In: McAllester, D.A. (ed.) Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1831, pp. 255–270. Springer (2000). https://doi.org/10.1007/10721959_20, https://doi.org/10.1007/10721959_20
15. Chen, Y., Havlena, V., Lengál, O.: Simulations in rank-based Büchi automata complementation. In: Lin, A.W. (ed.) Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11893, pp. 447–467. Springer (2019). https://doi.org/10.1007/978-3-030-34175-6_23, https://doi.org/10.1007/978-3-030-34175-6_23
16. Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 135–150. ACM (2018). https://doi.org/10.1145/3192366.3192405, https://doi.org/10.1145/3192366.3192405
17. Clemente, L., Mayr, R.: Efficient reduction of nondeterministic automata with application to language inclusion testing. Log. Methods Comput. Sci. **15**(1) (2019). https://doi.org/10.23638/LMCS-15(1:12)2019, https://doi.org/10.23638/LMCS-15(1:12)2019
18. Couvreur, J.: On-the-fly verification of linear temporal logic. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I. Lecture Notes in Computer Science, vol. 1708, pp. 253–271. Springer (1999). https://doi.org/10.1007/3-540-48119-2_16, https://doi.org/10.1007/3-540-48119-2_16

19. Diekert, V., Muscholl, A., Walukiewicz, I.: A note on monitors and Büchi automata. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9399, pp. 39–57. Springer (2015). https://doi.org/10.1007/978-3-319-25150-9_3, https://doi.org/10.1007/978-3-319-25150-9_3
20. Doveri, K., Ganty, P., Mazzocchi, N.: FORQ-based language inclusion formal testing. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 109–129. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_6, https://doi.org/10.1007/978-3-031-13188-2_6
21. Doveri, K., Ganty, P., Ranzato, F.: Inclusion testing of Büchi automata based on well-quasiorders. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 3:1–3:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPICS.CONCUR.2021.3, https://doi.org/10.4230/LIPIcs.CONCUR.2021.3
22. Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H.: From Spot 2.0 to Spot 2.10: What's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_9, https://doi.org/10.1007/978-3-031-13188-2_9
23. Emerson, E.A., Lei, C.: Modalities for model checking: Branching time logic strikes back. Sci. Comput. Program. **8**(3), 275–306 (1987). https://doi.org/10.1016/0167-6423(87)90036-0, https://doi.org/10.1016/0167-6423(87)90036-0
24. Esparza, J., Blondin, M.: Automata Theory: An Algorithmic Approach. MIT Press (2023)
25. Friedgut, E., Kupferman, O., Vardi, M.Y.: Büchi complementation made tighter. Int. J. Found. Comput. Sci. **17**(4), 851–868 (2006). https://doi.org/10.1142/S0129054106004145, https://doi.org/10.1142/S0129054106004145
26. Havlena, V., Lengál, O., Li, Y., Šmahlíková, B., Turrini, A.: Modular mix-and-match complementation of Büchi automata. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 249–270. Springer Nature Switzerland, Cham (2023), https://doi.org/10.1007/978-3-031-30823-9_13
27. Havlena, V., Lengál, O., Smahlíková, B.: Deciding S1S: down the rabbit hole and through the looking glass. In: Echihabi, K., Meyer, R. (eds.) Networked Systems - 9th International Conference, NETYS 2021, Virtual Event, May 19-21, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12754, pp. 215–222. Springer (2021). https://doi.org/10.1007/978-3-030-91014-3_15, https://doi.org/10.1007/978-3-030-91014-3_15
28. Havlena, V., Lengál, O.: Reducing (to) the ranks: Efficient rank-based Büchi automata complementation. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 2:1–2:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.2, https://doi.org/10.4230/LIPIcs.CONCUR.2021.2

29. Havlena, V., Lengál, O., Šmahlíková, B.: Complementing Büchi automata with Ranker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 188–201. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_10, https://doi.org/10.1007/978-3-031-13188-2_10
30. Havlena, V., Lengál, O., Šmahlíková, B.: Sky is not the limit: Tighter rank bounds for elevator automata in Büchi automata complementation. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 118–136. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_7, https://doi.org/10.1007/978-3-030-99527-0_7
31. Hieronymi, P., Ma, D., Oei, R., Schaeffer, L., Schulz, C., Shallit, J.O.: Decidability for Sturmian words. Log. Methods Comput. Sci. **20**(3) (2024). https://doi.org/10.46298/LMCS-20(3:12)2024, https://doi.org/10.46298/lmcs-20(3:12)2024
32. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Automata, Languages and Programming. pp. 724–735. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70575-8_59, https://doi.org/10.1007/978-3-540-70575-8_59
33. Karmarkar, H., Chakraborty, S.: On minimal odd rankings for Büchi complementation. In: Liu, Z., Ravn, A.P. (eds.) Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5799, pp. 228–243. Springer (2009). https://doi.org/10.1007/978-3-642-04761-9_18, https://doi.org/10.1007/978-3-642-04761-9_18
34. Kretínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings. pp. 543–550. Lecture Notes in Computer Science, Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_34, https://doi.org/10.1007/978-3-030-01090-4_34
35. Kupferman, O., Vardi, M.Y.: Verification of fair transition systems. In: Alur, R., Henzinger, T.A. (eds.) Computer Aided Verification. pp. 372–382. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
36. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001). https://doi.org/10.1145/377978.377993, https://doi.org/10.1145/377978.377993
37. Li, Y., Tsay, Y., Turrini, A., Vardi, M.Y., Zhang, L.: Congruence relations for Büchi automata. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13047, pp. 465–482. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_25, https://doi.org/10.1007/978-3-030-90870-6_25
38. Li, Y., Turrini, A., Feng, W., Vardi, M.Y., Zhang, L.: Divide-and-conquer determinization of Büchi automata based on SCC decomposition. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022,

Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 152–173. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_8, https://doi.org/10.1007/978-3-031-13188-2_8

39. Li, Y., Turrini, A., Zhang, L., Schewe, S.: Learning to complement Büchi automata. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10747, pp. 313–335. Springer (2018). https://doi.org/10.1007/978-3-319-73721-8_15, https://doi.org/10.1007/978-3-319-73721-8_15
40. Löding, C., Pirogov, A.: New optimizations and heuristics for determinization of Büchi automata. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 317–333. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_18, https://doi.org/10.1007/978-3-030-31784-3_18
41. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoretical Computer Science **32**(3), 321–330 (1984). https://doi.org/https://doi.org/10.1016/0304-3975(84)90049-5, https://www.sciencedirect.com/science/article/pii/0304397584900495
42. Oei, R., Ma, D., Schulz, C., Hieronymi, P.: Pecan: An automated theorem prover for automatic sequences using Büchi automata. CoRR **abs/2102.01727** (2021), https://arxiv.org/abs/2102.01727
43. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3) (2007). https://doi.org/10.2168/LMCS-3(3:5)2007, https://doi.org/10.2168/LMCS-3(3:5)2007
44. Redziejowski, R.R.: An improved construction of deterministic omega-automaton using derivatives. Fundam. Informaticae **119**(3-4), 393–406 (2012). https://doi.org/10.3233/FI-2012-744, https://doi.org/10.3233/FI-2012-744
45. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988. pp. 319–327. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21948, https://doi.org/10.1109/SFCS.1988.21948
46. Schewe, S.: Büchi complementation made tight. In: Albers, S., Marion, J. (eds.) 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2009). https://doi.org/10.4230/LIPIcs.STACS.2009.1854, https://doi.org/10.4230/LIPIcs.STACS.2009.1854
47. Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 312–332. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_17, https://doi.org/10.1007/978-3-319-41540-6_17
48. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987). https://doi.org/10.1016/0304-3975(87)90008-9, https://doi.org/10.1016/0304-3975(87)90008-9
49. Tarjan, R.: Depth-first search and linear graph algorithms. In: 12th Annual Symposium on Switching and Automata Theory (swat 1971). pp. 114–121 (1971). https://doi.org/10.1109/SWAT.1971.10

50. Tsai, M., Fogarty, S., Vardi, M.Y., Tsay, Y.: State of Büchi complementation. Log. Methods Comput. Sci. **10**(4) (2014). https://doi.org/10.2168/LMCS-10(4:13)2014, https://doi.org/10.2168/LMCS-10(4:13)2014
51. Vardi, M.Y.: Automata-theoretic model checking revisited. In: Chockler, H., Hu, A.J. (eds.) Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5394, p. 2. Springer (2008). https://doi.org/10.1007/978-3-642-01702-5_2, https://doi.org/10.1007/978-3-642-01702-5_2