Divide-and-Conquer Determinization of Büchi Automata Based on SCC Decomposition

Yong Li[1], Andrea Turrini[1,2], Weizhi Feng[1,3], Moshe Y. Vardi[4], and Lijun Zhang[1,2,3](✉)

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

zhanglj@ios.ac.cn

[2] Institute of Intelligent Software, Guangzhou, China

[3] University of Chinese Academy of Sciences, Beijing, China

[4] Rice University, Houston, USA

Abstract. The determinization of a nondeterministic Büchi automaton (NBA) is a fundamental construction of automata theory, with applications to probabilistic verification and reactive synthesis. The standard determinization constructions, such as the ones based on the Safra-Piterman's approach, work on the whole NBA. In this work we propose a divide-and-conquer determinization approach. To this end, we first classify the strongly connected components (SCCs) of the given NBA as inherently weak, deterministic accepting, and nondeterministic accepting. We then present how to determinize each type of SCC *independently* from the others; this results in an easier handling of the determinization algorithm that takes advantage of the structure of that SCC. Once all SCCs have been determinized, we show how to compose them so to obtain the final equivalent deterministic Emerson-Lei automaton, which can be converted into a deterministic Rabin automaton without blow-up of states and transitions. We implement our algorithm in our tool COLA and empirically evaluate COLA with the state-of-the-art tools Spot and Owl on a large set of benchmarks from the literature. The experimental results show that our prototype COLA outperforms Spot and Owl regarding the number of states and transitions.

1 Introduction

Nondeterministic Büchi automata (NBAs) [6] are finite automata accepting infinite words; they are a simple and popular formalism used in model checking to represent reactive and non-terminating systems and their specifications, characterized by ω-regular languages [2]. Due to their nondeterminism, however, there are situations in which NBAs are not suitable, so deterministic automata are required, as it happens in probabilistic verification [2] and reactive synthesis from logical specifications [34]. Consequently, translating NBAs into equivalent deterministic ω-automata (that is, deterministic automata accepting the same

© The Author(s) 2022

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13372, pp. 152–173, 2022.
https://doi.org/10.1007/978-3-031-13188-2_8

ω-regular language) is a necessary operation for solving these problems. While there exists a direct translation from linear temporal logic (LTL) to deterministic ω-automata [15], not all problems of interests can be formalized by LTL formulas, since LTL cannot express the full class of ω-regular properties [42]. For instance, we have to use Linear Dynamic Logic (LDL) [11,41] instead of LTL to express the ω-regular property "the train will arrive in every odd minute". To the best of our knowledge, we still need to go through the determinization of NBAs for LDL to obtain deterministic ω-automata. Therefore, NBA determinization is very important in verifying the whole class of ω-regular properties.

The determinization of NBAs is a fundamental problem in automata theory that has been actively studied for decades. For the determinization of nondeterministic automata accepting finite words, it suffices to use a subset construction [20]. Determinization constructions for NBAs are, however, much more involved since the simple subset construction is not sufficient [36]. Safra [36] gave the first determinization construction for NBAs with the optimal complexity $2^{\mathsf{O}(n \log n)}$, here n is the number of states of the input NBA; Michel [30] then gave a lower bound $n!$ for determinizing NBAs. Safra's construction has been further optimized by Piterman [33] to $\mathsf{O}((n!)^2)$ [38], resulting in the widely known Safra-Piterman's construction. The Safra-Piterman's construction is rather challenging, while still being the most practical way for Büchi complementation [40]. Research on determinization since then either aims at developing alternative Safraless constructions [18,21,28] or further tightening the upper and lower bounds of the NBA determinization [9,26,39,43].

In this paper, we focus on the *practical* aspects of Büchi determinization. All works on determinization mentioned above focus on translating NBAs to either deterministic Rabin or deterministic parity automata. According to [37], the more relaxed an acceptance condition is, the more succinct a finite automaton can be, regarding the number of states. In view of this, we consider the translation of NBAs to deterministic *Emerson-Lei* automata (DELAs) [13,37] whose acceptance condition is an arbitrary Boolean combination of sets of transitions to be seen finitely or infinitely often, the most generic acceptance condition for a deterministic automaton. We consider here transition-based automata rather than the usual state-based automata since the former can be more succinct [12].

The Büchi determinization algorithms available in literature operate on the *whole* NBA structure at once, which does not scale well in practice due to the complex structure and the big size of the input NBA. In this work we apply a *divide-and-conquer* methodology to Büchi determinization. We propose a determinization algorithm for NBAs to DELAs based on their strongly connected components (SCCs) decomposition. We first classify the SCCs of the given NBA into three types: *inherently weak*, in which either all cycles do not visit accepting transitions or all must visit accepting transitions; *deterministic accepting* and *nondeterministic accepting*, which contain an accepting transition and are deterministic or nondeterministic, respectively. We show how to divide the whole Büchi determinization problem into the determinization for each type of SCCs *independently*, in which the determinization for an SCC takes advantage of the structure of that SCC. Then we show how to compose the results of the local

determinization for each type of SCCs, leading to the final equivalent DELA. An extensive experimental evaluation confirms that the divide-and-conquer approach pays off also for the determinization of the whole NBA.

Contributions. First, we propose a *divide-and-conquer* determinization algorithm for NBAs, which takes advantage of the structure of different types of SCCs and determinizes SCCs independently. Our construction builds an equivalent DELA that can be converted into a deterministic Rabin automaton without blowing up states and transitions (cf. Theorem 2). To the best of our knowledge, we propose the *first* determinization algorithm that constructs a DELA from an NBA. Second, we show that there exists a family of NBAs for which our algorithm gives a DELA of size 2^{n+2} while classical works construct a DPA of size at least $n!$ (cf. Theorem 3). Third, we implement our algorithm in our tool COLA and evaluate it with the state-of-the-art tools SPOT [12] and OWL [23] on a large set of benchmarks from the literature. The experiments show that COLA outperforms SPOT and OWL regarding the number of states and transitions. Finally, we remark that the determinization complexity for some classes of NBAs can be exponentially better than the known ones (cf. Corollary 1).

2 Preliminaries

Let Σ be a given alphabet, i.e., a finite set of letters. A transition-based Emerson-Lei automaton can be seen as a generalization of other types of ω-automata, like Büchi, Rabin or parity. Formally, it is defined in the HOA format [1] as follows:

Definition 1. *A* nondeterministic Emerson-Lei automaton *(NELA) is a tuple* $\mathcal{A} = (Q, \iota, \delta, \Gamma_k, \mathsf{p}, \mathsf{Acc})$, *where* Q *is a finite set of* states*;* $\iota \in Q$ *is the* initial state*;* $\delta \subseteq Q \times \Sigma \times Q$ *is a* transition relation*;* $\Gamma_k = \{0, 1, \cdots, k\}$, *where* $k \in \mathbb{N}$, *is a set of* colors*;* $\mathsf{p}\colon \delta \to 2^{\Gamma_k}$ *is a* coloring function *for transitions; and* Acc *is an* acceptance formula *over* Γ_k *given by the following grammar, where* $x \in \Gamma_k$*:*

$$\alpha := \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{Fin}(x) \mid \mathsf{Inf}(x) \mid \alpha \vee \alpha \mid \alpha \wedge \alpha.$$

We remark that the colors in Γ_k are not required to be all used in Acc. We call a NELA a *deterministic* Emerson-Lei automaton (DELA) if for each $q \in Q$ and $a \in \Sigma$, there is at most one $q' \in Q$ such that $(q, a, q') \in \delta$.

In the remainder of the paper, we consider δ also as a function $\delta\colon Q \times \Sigma \to 2^Q$ such that $q' \in \delta(q, a)$ whenever $(q, a, q') \in \delta$; we also write $q \xrightarrow{a} q'$ for $(q, a, q') \in \delta$ and we extend it to words $u = u_0 u_1 \cdots u_n \in \Sigma^*$ in the natural way, i.e., $q \xrightarrow{u} q' = q \xrightarrow{u[0]} q_1 \xrightarrow{u[1]} \cdots \xrightarrow{u[n]} q'$, where $\sigma[i]$ denotes the element s_i of the sequence of elements $\sigma = s_0 s_1 s_2 \cdots$ at position i. We assume without loss of generality that each automaton is *complete*, i.e., for each state $q \in Q$ and letter $a \in \Sigma$, we have $\delta(q, a) \neq \emptyset$. If it is not complete, we make it complete by adding a fresh state $q_\perp \notin Q$ and redirecting all missing transitions to it.

A *run* of $\mathcal{A}$ over an ω-word $w \in \Sigma^\omega$ is an infinite sequence of states ρ such that $\rho[0] = \iota$, and for each $i \in \mathbb{N}$, $(\rho[i], w[i], \rho[i+1]) \in \delta$.

The *language* $\mathsf{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$, i.e., the set of words $w \in \Sigma^\omega$ such that there exists a run ρ of $\mathcal{A}$ over w such that $\mathsf{p}(\mathit{inf}(\rho)) \models \mathsf{Acc}$, where $\mathit{inf}(\rho) = \{ (q, a, q') \in \delta \mid \forall i \in \mathbb{N}. \exists j > i. (\rho[j], w[j], \rho[j+1]) = (q, a, q') \}$ and the satisfaction relation $\models$ is defined recursively as follows: given $M \subseteq \Gamma_k$,

$$M \models \mathsf{tt}, \quad M \models \mathsf{Fin}(x) \text{ iff } x \notin M, \quad M \models \alpha_1 \vee \alpha_2 \text{ iff } M \models \alpha_1 \text{ or } M \models \alpha_2,$$
$$M \not\models \mathsf{ff}, \quad M \models \mathsf{Inf}(x) \text{ iff } x \in M, \quad M \models \alpha_1 \wedge \alpha_2 \text{ iff } M \models \alpha_1 \text{ and } M \models \alpha_2.$$

Intuitively, a run ρ over w is accepting if the set of colors (induced by p) that occur infinitely often in ρ satisfies the acceptance formula Acc. Here $\mathsf{Fin}(x)$ specifies that the color x only appears for finitely many times while $\mathsf{Inf}(x)$ requires the color x to be seen infinitely often.

The more common types of ω-automata, such as Büchi, parity and Rabin can be treated as Emerson-Lei automata with the following acceptance formulas.

Definition 2. *A NELA $\mathcal{A} = (Q, \iota, \delta, \Gamma_k, \mathsf{p}, \mathsf{Acc})$ is said to be*

- *a* Büchi automaton *(BA) if $k = 0$ and $\mathsf{Acc} = \mathsf{Inf}(0)$. Transition with color 0 are usually called* accepting *transitions. Thus, a run ρ is accepting if $\mathsf{p}(\mathit{inf}(\rho)) \cap \{0\} \neq \emptyset$, i.e., ρ takes accepting transitions infinitely often;*
- *a* parity automaton *(PA) if k is even and $\mathsf{Acc} = \bigvee_{c=0}^{k/2} (\bigwedge_{i=1}^{c} \mathsf{Fin}(2i-1) \wedge \mathsf{Inf}(2c))$. A run ρ is accepting if the minimum color in $\mathsf{p}(\mathit{inf}(\rho))$ is even;*
- *a* Rabin automaton *(RA) if k is an odd number and $\mathsf{Acc} = (\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee \cdots \vee (\mathsf{Fin}(k-1) \wedge \mathsf{Inf}(k))$. Intuitively, a run ρ is accepting if there exists an odd integer $0 < j \leq k$ such that $j - 1 \notin \mathsf{p}(\mathit{inf}(\rho))$ and $j \in \mathsf{p}(\mathit{inf}(\rho))$.*

When the NELA $\mathcal{A} = (Q, \iota, \delta, \Gamma_k, \mathsf{p}, \mathsf{Acc})$ is a nondeterministic BA (NBA), we just write $\mathcal{A}$ as (Q, ι, δ, F) where F is the set of accepting transitions. We call a set $C \subseteq Q$ a *strongly connected component* (SCC) of $\mathcal{A}$ if for every pair of states $q, q' \in C$, we have that $q \xrightarrow{u} q'$ for some $u \in \Sigma^*$ and $q' \xrightarrow{v} q$ for some $v \in \Sigma^*$, i.e., q and q' can be reached by each other; by default, each state $q \in Q$ reaches itself. C is a *maximal* SCC if it is *not* a proper subset of another SCC. *All* SCCs considered in the work are maximal. We call an SCC C *accepting* if there is a transition $(q, a, q') \in (C \times \Sigma \times C) \cap F$ and *nonaccepting* otherwise. We say that an SCC C' is *reachable* from an SCC C if there exist $q \in C$ and $q' \in C'$ such that $q \xrightarrow{u} q'$ for some $u \in \Sigma^*$. An SCC C is *inherently weak* if either every cycle going through the C-states visits at least one accepting transition or none of the cycles visits an accepting transition. We say that an SCC C is *deterministic* if for every state $q \in C$ and $a \in \Sigma$, we have $|\delta(q, a) \cap C| \leq 1$. Note that a state q in a deterministic SCC C can have multiple successors for a letter a, but at most one successor remains in C.

Figure 1 shows an example of NBA we will use for our examples in the remainder of the paper; we depict the accepting transitions with a double arrow. Clearly, inside each SCC, depicted as a box, each state can be reached by any other state, and the SCCs are maximal. The SCC $\{q_2, q_3\}$ is inherently weak and accepting, since every cycle takes an accepting transition; the SCC $\{q_6\}$ is also inherently weak, but nonaccepting, since every cycle never takes an accepting transition.

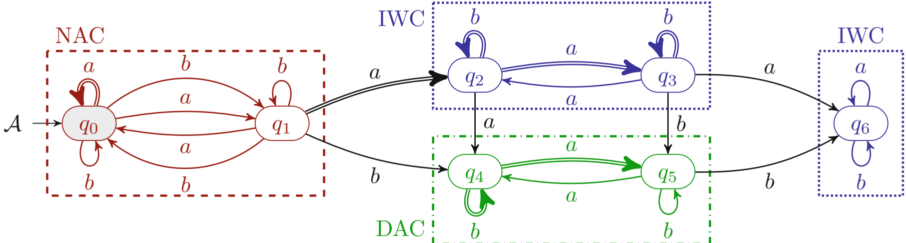


Fig. 1. An example of NBA.

The remaining two SCCs, i.e., $\{q_0, q_1\}$ and $\{q_4, q_5\}$, are not inherently weak, since some cycle takes accepting transitions (like the cycle $q_0 \xrightarrow{a} q_0$) while others do not (like the cycle $q_0 \xrightarrow{b} q_0$). Both SCCs contain an accepting transition, so they are accepting; the SCC $\{q_0, q_1\}$ is clearly nondeterministic, while the SCC $\{q_4, q_5\}$ is deterministic. Note that from q_5 we have two transitions labelled by b, but only the transition $q_5 \xrightarrow{b} q_4$ remains inside the SCC, while the other transition $q_5 \xrightarrow{b} q_6$ leaves the SCC, so the SCC is still deterministic.

The following proposition is well known and is often used in prior works.

Proposition 1. *Let $\mathcal{A}$ be an NBA and $w \in \Sigma^\omega$. A run of $\mathcal{A}$ over w will eventually stay in an SCC. Moreover, if $w \in \mathsf{L}(\mathcal{A})$, every accepting run of $\mathcal{A}$ over w will eventually stay in an accepting SCC.*

Proposition 1 is the key ingredient of our algorithm: it allows us to determinize the SCCs independently as $\mathsf{L}(\mathcal{A})$ is the union of the words whose runs stay in each accepting SCCs. In the remainder of the paper, we first present a translation from an NBA $\mathcal{A}$ to a DELA $\mathcal{A}^{\mathsf{E}}$ based on the SCC decomposition of $\mathcal{A}$. The obtained DELA $\mathcal{A}^{\mathsf{E}}$ in fact can be converted to a *deterministic* Rabin automaton (DRA) $\mathcal{A}^{\mathsf{R}}$ without blowing up states and transitions, i.e., we can just convert the coloring function and the acceptance formula of $\mathcal{A}^{\mathsf{E}}$ to DRAs.

3 Determinization Algorithms of SCCs

Determinizing each SCC of $\mathcal{A}$ independently is not straightforward since it may be reached from the initial state only after reading a nonempty finite word; moreover, there can be words of different length leading to the SCC, entering through different states. To keep track of the different arrivals in an SCC at different times, we make use of run DAGs [24], that are a means to organize the runs of $\mathcal{A}$ over a word w. In this section, we first recall the concept of run DAGs and then describe how to determinize SCCs with their help.

Definition 3. *Let $\mathcal{A} = (Q, \iota, \delta, F)$ be an NBA and $w \in \Sigma^\omega$ be a word. The run DAG $\mathcal{G}_{\mathcal{A},w} = \langle V, E \rangle$ of $\mathcal{A}$ over w is defined as follows: the set of vertices $V \subseteq Q \times \mathbb{N}$ is defined as $V = \bigcup_{l \geq 0} (V_l \times \{l\})$ where $V_0 = \{\iota\}$ and $V_{l+1} = \delta(V_l, w[l])$ for every $l \in \mathbb{N}$; there is an edge $(\langle q, l \rangle, \langle q', l' \rangle) \in E$ if $l' = l+1$ and $q' \in \delta(q, w[l])$.*

Intuitively, a state q at a level ℓ may occur in several runs and only one vertex is needed to represent it, i.e., the vertex $\langle q, \ell \rangle$ who is said to be on level ℓ. Note that by definition, there are at most $|Q|$ vertices on each level. An edge $(\langle q, l \rangle, \langle q', l+1 \rangle)$ is an *F-edge* if $(q, w[l], q') \in F$. An infinite sequence of vertices $\gamma = \langle q_0, 0 \rangle \langle q_1, 1 \rangle \cdots$ is called an *ω-branch* of $\mathcal{G}_{\mathcal{A},w}$ if $q_0 = \iota$ and for each $\ell \in \mathbb{N}$, we have $(\langle q_\ell, \ell \rangle, \langle q_{\ell+1}, \ell+1 \rangle) \in E$. We can observe that there is a bijection between the set of runs of $\mathcal{A}$ on w and the set of ω-branches in $\mathcal{G}_{\mathcal{A},w}$. In fact, to a run $\rho = q_0 q_1 \cdots$ of $\mathcal{A}$ over w corresponds the ω-branch $\hat{\rho} = \langle q_0, 0 \rangle \langle q_1, 1 \rangle \cdots$ and, symmetrically, to an ω-branch $\gamma = \langle q_0, 0 \rangle \langle q_1, 1 \rangle \cdots$ corresponds the run $\hat{\gamma} = q_0 q_1 \cdots$. Thus w is accepted by $\mathcal{A}$ if and only if there exists an ω-branch in $\mathcal{G}_{\mathcal{A},w}$ that takes F-edges infinitely often.

In the remainder of this section, we will introduce the algorithms for computing the successors of the current states inside different types of SCCs, with the help of run DAGs. We fix an NBA $\mathcal{A} = (Q, \iota, \delta, F)$ and a word $w \in \Sigma^\omega$. We let $Q = \{q_1, \ldots, q_n\}$ and apply a total order $\preccurlyeq$ on Q such that $q_i \preccurlyeq q_j$ if $i < j$. Let $S_\ell \subseteq Q$, $\ell \in \mathbb{N}$, be the set of states reached at the level ℓ in the run DAG $\mathcal{G}_{\mathcal{A},w}$; we assume that this sequence $S_0, \cdots, S_\ell, \cdots$ is available as a *global* variable during the computations of every SCC where $S_0 = \{\iota\}$ and $S_{\ell+1} = \delta(S_\ell, w[\ell])$.

When determinizing the given NBA $\mathcal{A}$, we classify its SCCs into three types, namely inherently weak SCCs (IWCs), deterministic-accepting SCCs (DACs) and nondeterministic-accepting SCCs (NACs). We assume that all DACs and NACs are *not* inherently weak, otherwise they will be classified as IWCs.

In our determinization construction, every level in $\mathcal{G}_{\mathcal{A},w}$ corresponds to a state in our constructed DELA $\mathcal{A}^\mathsf{E}$ while reading the ω-word w. Let m_ℓ be the state of $\mathcal{A}^\mathsf{E}$ at level ℓ. The computation of the successor $m_{\ell+1}$ of m_ℓ for the letter $w[\ell]$ will be divided into the successor computation for states in IWCs, DACs and NACs independently. Then the successor $m_{\ell+1}$ is just the Cartesian product of these successors. In the remainder of this section, we present how to compute the successors for the states in each type of SCCs.

3.1 Successor Computation Inside IWCs

As we have seen, $\mathcal{G}_{\mathcal{A},w}$ contains all runs of $\mathcal{A}$ over w, including those within DACs and NACs. Since we want to compute the successor only for IWCs, we focus on the states inside the IWCs and ignore other states in DACs and NACs. Let W be the set of states in all IWCs and $\mathsf{WA} \subseteq \mathsf{W}$ be the set of states in all accepting IWCs.

For the run DAG $\mathcal{G}_{\mathcal{A},w}$, we use a pair of sets of states $(P_\ell, O_\ell) \in 2^\mathsf{W} \times 2^\mathsf{WA}$ to represent the set of IWC states reached in $\mathcal{G}_{\mathcal{A},w}$ at level ℓ. The set P_ℓ is used to keep track of the states in W reached at level ℓ, while O_ℓ, inspired by the breakpoint construction used in [31], keeps only the states reached in WA, that is, it is used to track the runs that stay in accepting IWCs. Since by definition each cycle inside an accepting IWC must visit an accepting transition, for each run tracked by O_ℓ we do not need to remember whether we have taken an accepting transition: it suffices to know whether the run is still inside some accepting IWC or whether the run has left them.

We now show how to compute the sets (P_ℓ, O_ℓ) along w. For level 0, we simply set $P_0 = \{\iota\} \cap \mathsf{W}$ and $O_0 = \emptyset$. For the other levels, given (P_ℓ, O_ℓ) at level $\ell \in \mathbb{N}$, the encoding $(P_{\ell+1}, O_{\ell+1})$ for the next level $\ell + 1$ is defined as follows:

– $P_{\ell+1} = S_{\ell+1} \cap \mathsf{W}$, i.e., $P_{\ell+1}$ keeps track of the W-states reached at level $\ell + 1$;
– if $O_\ell \neq \emptyset$, then $O_{\ell+1} = \delta(O_\ell, w[\ell]) \cap \mathsf{WA}$, otherwise $O_{\ell+1} = P_{\ell+1} \cap \mathsf{WA}$.

Intuitively, the O-set keeps track of the runs that stay in the accepting IWCs. So if $O_\ell \neq \emptyset$, then $O_{\ell+1}$ maintains the runs remaining in some accepting IWC; otherwise, $O_\ell = \emptyset$ means that at level ℓ all runs seen so far in the accepting IWCs have left them, so we can just start to track the new runs that entered the accepting IWCs but were not tracked yet.

On the right we show the fragment of the run DAG $\mathcal{G}_{\mathcal{A},a^\omega}$ for the NBA $\mathcal{A}$ shown in Fig. 1 and its IWCs; we have $\mathsf{W} = \{q_2, q_3, q_6\}$ and $\mathsf{WA} = \{q_2, q_3\}$. The set P_ℓ contains all states q at level ℓ; the set O_ℓ contains the underlined ones. As a concrete application of the construction given above, from $P_3 = \{q_2, q_3\}$ and $O_3 = \delta(O_2, a) \cap \mathsf{WA} = \{q_3\}$, at level 4 we get $P_4 = \{q_2, q_3, q_6\}$ and $O_4 = \delta(O_3, a) \cap \mathsf{WA} = \{q_2\}$.

ℓ	$\downarrow$	$\underline{q} \in O$	
2	$\underline{q_2}$		
3	$\underline{q_3}$		q_2
4	$\underline{q_2}$	q_6	q_3
5	$\underline{q_3}$	q_6	q_2

It is not difficult to see that checking whether w is accepted reduces to check whether the number of empty O-sets is finite. We assign color 1 to the transition from (P_ℓ, O_ℓ) to $(P_{\ell+1}, O_{\ell+1})$ via $w[\ell]$ if $O_\ell = \emptyset$, otherwise we assign color 2. Lemma 1 formalizes the relation between accepting runs staying in accepting IWCs and the colors we get from our construction.

Lemma 1. *(1) There exists an accepting run of $\mathcal{A}$ over w eventually staying in an accepting IWC if and only if we receive color 1 finitely many times when constructing the sequence $(P_0, O_0) \cdots (P_\ell, O_\ell) \cdots$ while reading w. (2) The number of possible (P, O) pairs is at most $3^{|\mathsf{W}|}$.*

The proof idea is trivial: an accepting run ρ that stays in an accepting IWC will make the O-set contain ρ forever and we always get color 2 from some point on. A possible pair (P, O) can be seen as choosing a state from W, which can be from $\mathsf{W} \setminus P$, $P \cap O$ and $P \setminus O$, respectively. It thus gives at most $3^{|\mathsf{W}|}$ possibilities.

To ease the construction for the whole NBA $\mathcal{A}$, we make the above computation of successors available as a function $\mathsf{weakSucc}$, which takes as input a pair of sets (P, O) and a letter a, and returns the successor (P', O') and the corresponding color $c \in \{1, 2\}$ for the transition $((P, O), a, (P', O'))$.

The construction we gave above works on all IWCs at the same time; considering IWCs separately does not improve the resulting complexity. If there are two accepting IWCs with n_1 and n_2 states, respectively, then the number of possible (P, O) pairs for the two IWCs is 3^{n_1} and 3^{n_2}, respectively. When combining the pairs for each IWC together, the resulting number of pairs in the Cartesian product is $3^{n_1} \times 3^{n_2} = 3^{n_1+n_2}$, which is the same as considering them together. On the other hand, for each accepting IWC, we need to use two colors, so we need $2 \cdot i$ colors in total for i accepting IWCs, instead of just two colors by operating on all IWCs together. Hence, we prefer to work on all IWCs at once.

3.2 Successor Computation Inside DACs

In contrast to IWCs, we do not work on all DACs at once but we process each DAC separately. This is because there may be nondeterminism between DACs: a run in a DAC may branch into multiple runs that jump to different DACs, which requires us to resort to a Safra-Piterman's construction [33,36] when considering all DACs at once. Working on each DAC separately, instead, allows us to take advantage of the internal determinism: for a given DAC D, the transition relation δ inside D, denoted as $\delta_D = (\mathsf{D} \times \Sigma \times \mathsf{D}) \cap \delta$, is now deterministic.

Although every run ρ entering D can have only one successor in D, ρ may just leave D while new runs can enter D, which makes it difficult to check whether there exists an accepting run that remains trapped into D. In order to identify accepting runs staying in D, we identify the following two rules for distinguishing runs that come to D by means of *unique* labelling numbers: **(1)** the runs already in D have precedence over newly entering runs, thus the latter get assigned a higher number. In practice, the labelling keeps track of the relative order of entering D, thus the lower the labelling value is, the earlier the run came to D; **(2)** when two runs in D merge, we only keep the run that came to D earlier, i.e., the run with lower number. If two runs enter D at the same time, we let them enter according to the total state order $\preccurlyeq$ for their respective entry states.

We use a level-labelling function $\mathfrak{g}_\ell \colon \mathsf{D} \to \{1, \cdots, 2 \cdot |\mathsf{D}|\} \cup \{\infty\}$ to encode the set of D-states reached at level ℓ of the run DAG $\mathcal{G}_{\mathcal{A},w}$. Here we use $\mathfrak{g}_\ell(q) = \infty$ to indicate that the state $q \in \mathsf{D}$ is not reached by $\mathcal{A}$ at level ℓ.

At level 0, we set $\mathfrak{g}_0(q) = \infty$ for every state $q \in \mathsf{D} \setminus \{\iota\}$, and $\mathfrak{g}_0(\iota) = 1$ if $\iota \in \mathsf{D}$. Note that the SCC that ι resides in can be an IWC, a DAC or a NAC.

For a given level-labelling function $\mathfrak{g}_\ell$, we will make $\{\, q \in \mathsf{D} \mid \mathfrak{g}_\ell(q) \neq \infty \,\} = S_\ell \cap \mathsf{D}$ hold, i.e., tracing correctly the set of D-states reached by $\mathcal{A}$ at level ℓ; we denote the set $\mathfrak{g}_\ell(\mathsf{D}) \setminus \{\infty\}$ by $\beta(\mathfrak{g}_\ell)$, so $\beta(\mathfrak{g}_\ell)$ is the set of unique labelling numbers at level ℓ. By the construction given below about how to generate $\mathfrak{g}_{\ell+1}$ from $\mathfrak{g}_\ell$ on reading $w[\ell]$, we ensure that $\beta(\mathfrak{g}_\ell) \subseteq \{1, \cdots, 2 \cdot |\mathsf{D}|\}$ for all $\ell \in \mathbb{N}$.

We now present how to compute the successor level-labelling function $\mathfrak{g}_{\ell+1}$ of $\mathfrak{g}_\ell$ on letter $w[\ell]$. The states reached by $\mathcal{A}$ at level $\ell+1$, i.e., $S_{\ell+1} \cap \mathsf{D}$, may come from two sources: some state may come from states not in D via transitions in $\delta \setminus \delta_D$; some other via δ_D from states in $S_\ell \cap \mathsf{D}$. In order to generate $\mathfrak{g}_{\ell+1}$, we first compute an intermediate level-labelling function $\mathfrak{g}'_{\ell+1}$ as follows.

1. To obey Rule **(2)**, for every state $q' \in \delta_D(S_\ell \cap \mathsf{D}, w[\ell])$, we set

$$\mathfrak{g}'_{\ell+1}(q') = \min\{\, \mathfrak{g}_\ell(q) \mid q \in S_\ell \cap \mathsf{D} \wedge \delta_D(q, w[\ell]) = q' \,\}.$$

   That is, when two runs merge, we only keep the run with the lower labelling number, i.e., the run entered in D earlier.
2. To respect Rule **(1)**, we set $\mathfrak{g}'_{\ell+1}(q') = |\mathsf{D}| + i$ for the i-th newly entered state $q' \in (S_{\ell+1} \cap \mathsf{D}) \setminus \delta_D(S_\ell \cap \mathsf{D}, w[\ell])$ and the states q' are ordered by the total order $\preccurlyeq$ of the states. Since every state in $\delta_D(S_\ell \cap \mathsf{D}, w[\ell])$ is on a run that already entered D, its labelling has already been determined by the case 1.

It is easy to observe that in order to compute the transition relation between two consecutive levels, we only need to know the labelling at the previous level. More precisely, we do not have to know the exact labelling numbers, since it suffices to know their relative order. Therefore, we can compress the level-labelling $\mathfrak{g}'_{\ell+1}$ to $\mathfrak{g}_{\ell+1}$ as follows. Let $\mathsf{ord}\colon \beta(\mathfrak{g}'_{\ell+1}) \to \{1, \cdots, |\beta(\mathfrak{g}'_{\ell+1})|\}$ be the function that maps each labelling value in $\beta(\mathfrak{g}'_{\ell+1})$ to its relative position once the values in $\beta(\mathfrak{g}'_{\ell+1})$ have been sorted in ascending order. For instance, if $\beta(\mathfrak{g}'_{\ell+1}) = \{2, 4, 7\}$, then $\mathsf{ord} = \{2 \mapsto 1, 4 \mapsto 2, 7 \mapsto 3\}$. Then we set $\mathfrak{g}_{\ell+1}(q) = \mathsf{ord}(\mathfrak{g}'_{\ell+1}(q))$ for each $q \in S_{\ell+1} \cap \mathsf{D}$, and $\mathfrak{g}_{\ell+1}(q') = \infty$ for each $q' \in \mathsf{D} \setminus S_{\ell+1}$. In this way, all level-labelling functions $\mathfrak{g}_\ell$ we use are such that $\beta(\mathfrak{g}_\ell) \subseteq \{1, \cdots, |\mathsf{D}|\}$.

The intuition behind the use of these level-labelling functions is that, if we always see a labelling number h in the intermediate level-labelling $\mathfrak{g}'_\ell$ for all $\ell \geq k$ after some level k, we know that there is a run that eventually stays in D and is eventually always labelled with h. To check whether this run also visits infinitely many accepting transitions, we will color every transition $e = (\mathfrak{g}_\ell, w[\ell], \mathfrak{g}_{\ell+1})$. To decide what color to assign to e, we first identify which runs have merged with others or got out of D (corresponding to *bad* events and *odd* colors) and which runs still continue to stay in D and take an accepting transition (corresponding to *good* events and *even* colors).

The bad events correspond to the discontinuation of labelling values between $\mathfrak{g}_\ell$ and $\mathfrak{g}'_{\ell+1}$, defined as $\mathsf{B}(e) = \beta(\mathfrak{g}_\ell) \setminus \beta(\mathfrak{g}'_{\ell+1})$. Intuitively, if a labelling value k exists in the set $\mathsf{B}(e)$, then the run ρ associated with labelling k merged with a run with lower labelling value $k' < k$, or ρ left the DAC D. The good events correspond to the occurrence of accepting transitions in some runs, whose labelling we collect into $\mathsf{G}(e) = \{\, k \in \beta(\mathfrak{g}_\ell) \mid \exists (q, w[\ell], q') \in F. \mathfrak{g}_\ell(q) = \mathfrak{g}'_{\ell+1}(q') = k \neq \infty \,\}$. In practice, a labelling value k in $\mathsf{G}(e)$ indicates that we have seen a run with labelling k that visits an accepting transition. We then let $\mathsf{B}(e) = \mathsf{B}(e) \cup \{|\mathsf{D}| + 1\}$ and $\mathsf{G}(e) = \mathsf{G}(e) \cup \{|\mathsf{D}| + 1\}$ where the value $|\mathsf{D}| + 1$ is used to indicate that no bad (i.e., no run merged or left the DAC) or no good (i.e., no run took an accepting transition) events happened, respectively.

In order to declare a sequence of labelling functions as accepting, we want the good events to happen infinitely often and bad events to happen only finitely often, when the runs with bad events have a labelling number lower than that of the runs with good events. So we assign the color $c = \min\{2 \cdot \min \mathsf{B}(e) - 1, 2 \cdot \min \mathsf{G}(e)\}$ to the transition e. Since the labelling numbers are in $\{1, \cdots, |\mathsf{D}|\}$, we have that $c \in \{1, \cdots, 2 \cdot |\mathsf{D}| + 1\}$. The intuition why we assign colors in this way is given as the proof idea of the following lemma.

Lemma 2. *(1) An accepting run of $\mathcal{A}$ over w eventually stays in the DAC D if and only if the minimal color c we receive infinitely often is even. (2) The number of possible labelling functions $\mathfrak{g}$ is at most $3 \cdot |\mathsf{D}|!$.*

The proof idea is as follows: an accepting run ρ on the word w that stays in D will have stable labelling number, say $k \geq 1$, after some level since the labelling value cannot increase by construction and is finite. So all runs on w that have labelling values lower than k will not leave D: if they would leave or just merge with other runs, their labelling value vanishes, so ord would decrease the value

for ρ. This implies that the color we receive afterwards infinitely often is either 1) an odd color larger than $2k$, due to vanishing runs with value at least $k+1$ or simply because no bad or good events occur, or 2) an even color at most $2k$, depending on whether there is some run with value smaller than ρ also taking accepting transitions. Thus the minimum color occurring infinitely often is even. The number of labelling functions $\mathfrak{g}$ is bounded by $\sum_{i=0}^{|\mathsf{D}|} \binom{|\mathsf{D}|}{i} \cdot i! \leq 3 \cdot |\mathsf{D}|!$.

The fragment of the DAG $\mathcal{G}_{\mathcal{A},a^\omega}$ shown on the right is relative to the only DAC $\mathsf{D} = \{q_4, q_5\}$. The value of $\mathfrak{g}'_\ell(q)$, $\mathfrak{g}_\ell(q)$ and the corresponding ord is given by the mapping near each state q; as a concrete application of the construction given above, consider how to get $\mathfrak{g}_4$ from $\mathfrak{g}_3$, defined as $\mathfrak{g}_3(q_4) = 1$ and $\mathfrak{g}_3(q_5) = \infty$: since $q_5 \in \delta_D(S_3 \cap \mathsf{D}, a)$, according to case 1 we define $\mathfrak{g}'_4(q_5) = 1$ because $q_5 = \delta_D(q_4, a)$ and $\mathfrak{g}_3(q_4) = 1$; since $q_4 \in (S_4 \cap \mathsf{D}) \setminus \delta_D(S_3 \cap \mathsf{D}, a)$, then case 2 applies, so $\mathfrak{g}'_4(q_4) = 3$. The function ord is $\mathsf{ord} = [1 \mapsto 1, 3 \mapsto 2]$, thus we get $\mathfrak{g}_4(q_4) = 2$ and $\mathfrak{g}_4(q_5) = 1$. As bad/good sets for the transition $e = \mathfrak{g}_3 \xrightarrow{a} \mathfrak{g}_4$, we have $\mathsf{B}(e) = \emptyset \cup \{3\}$ while $\mathsf{G}(e) = \{1\} \cup \{3\}$, so the resulting color is 2.

ℓ		
3	$q_4, 3 \mapsto 1$	
4	$q_5, 1 \mapsto 1$	$q_4, 3 \mapsto 2$
5	$q_4, 1 \mapsto 1$	$q_5, 2 \mapsto 2$

Again, we make the above computation of successors available as a function $\mathsf{detSucc}$, which takes as input the DAC D, a labelling $\mathfrak{g}$ and a letter a, and returns the successor labelling $\mathfrak{g}'$ and the color $c \in \{1, \cdots, 2 \cdot |\mathsf{D}| + 1\}$.

3.3 Successor Computation Inside NACs

The computation of the successor inside a NAC is more involved since runs can branch, so it is more difficult to check whether there exists an accepting run. To identify accepting runs, researchers usually follow the Safra-Piterman's idea [33,36] to give the runs that take more accepting transitions the precedence over other runs that join them. We now present how to compute labelling functions encoding this idea for NACs, instead of the whole NBA. Differently to the previous case about DACs, the labelling functions we use here use lists of numbers, instead of single numbers, to keep track of the branching, merging and new incoming runs. This can be seen as a generalization of the numbered brackets used in [35] to represent ordinary Safra-Piterman's trees. Differently from this construction, in our setting the main challenge we have to consider is how to manage correctly the newly entering runs, which are simply not occurring in [35] since there the whole NBA is considered. The fact that runs can merge, instead, is a common aspect, while the fact that a run ρ leaves the current NAC can be treated similarly to dying out runs in [35]. Below we assume that N is a given NAC; we denote by $\delta_N = (\mathsf{N} \times \Sigma \times \mathsf{N}) \cap \delta$ the transition function δ inside N.

To manage the branching and merging of runs of $\mathcal{A}$ over w inside a NAC, and to keep track of the accepting transitions taken so far, we use level-labelling functions as for the DAC case. For a given NAC N, the functions we use have lists of natural numbers as codomain; more precisely, let $\mathcal{L}_{\mathsf{N}}$ be the set of lists taking value in the set $\{1, \cdots, 2 \cdot |\mathsf{N}|\}$, where a list is a finite sequence of values in ascending order. Given two lists $[v_1, \cdots, v_k]$ and $[v'_1, \cdots, v'_{k'}]$, we say that

$[v_1, \cdots, v_k]$ is a prefix of $[v'_1, \cdots, v'_{k'}]$ if $1 \leq k \leq k'$ and for each $1 \leq j \leq k$, we have $v_j = v'_j$. Note that the empty list *is not* a prefix of any list. Given two lists $[v_1, \cdots, v_k]$ and $[v'_1, \cdots, v'_{k'}]$, we denote by $[v_1, \cdots, v_k] \frown [v'_1, \cdots, v'_{k'}]$ their concatenation, that is the list $[v_1, \cdots, v_k, v'_1, \cdots, v'_{k'}]$. Moreover, we define a total order on lists as follows: given two lists $[v_1, \cdots, v_k]$ and $[v'_1, \cdots, v'_{k'}]$, we order them by padding the shorter of the two with ∞ in the rear, so to make them of the same length, and then by comparing them by the usual lexicographic order. This means, for instance, that the empty list $[]$ is the largest list and that $[1, 3, 5]$ is smaller than $[1, 3]$ but larger than $[1, 2]$. The lists help to keep track of the branching history from their prefixes, such as $[1, 2]$ is branched from $[1]$.

As done for DACs, we use a level-labelling function $\mathfrak{t}_\ell \colon \mathsf{N} \to \mathcal{L}_\mathsf{N}$ to encode the set of N-states reached in the run DAG $\mathcal{G}_{\mathcal{A},w}$ at level ℓ. We denote by $\beta(\mathfrak{t}_\ell)$ the set of non-empty lists in the image of $\mathfrak{t}_\ell$, that is, $\beta(\mathfrak{t}_\ell) = \{ \mathfrak{t}_\ell(q) \mid q \in \mathsf{N} \land \mathfrak{t}_\ell(q) \neq [] \}$. We use the empty list $[]$ for the states in N that do not occur in the vertexes of $\mathcal{G}_{\mathcal{A},w}$ at level ℓ, so $\beta(\mathfrak{t}_\ell)$ contains only lists associated with states that $\mathcal{A}$ is currently located at. Similarly to the other types of SCCs, at level 0, we set $\mathfrak{t}_0(\iota) = [1]$ if $\iota \in \mathsf{N}$, and $\mathfrak{t}_0(q) = []$ for each state $q \in \mathsf{N} \setminus \{\iota\}$.

To define the transition from $\mathfrak{t}_\ell$ to $\mathfrak{t}_{\ell+1}$ through the letter $w[\ell]$, we use again an intermediate level-labelling function $\mathfrak{t}'_{\ell+1}$ that we construct step by step as follows. We start with $\mathfrak{t}'_{\ell+1}(q) = []$ for each $q \in \mathsf{N}$ and with the set of unused numbers $U = \{ u \geq 1 \mid u \notin \beta(\mathfrak{t}_\ell) \}$, i.e., the numbers not used in $\beta(\mathfrak{t}_\ell)$.

1. For every state $q' \in \delta_N(S_\ell \cap \mathsf{N}, w[\ell])$, let $P_{q'} = \{ q \in S_\ell \cap \mathsf{N} \mid (q, w[\ell], q') \in \delta_N \}$ be the set of currently reached predecessors of q', and $C_{q'} = \emptyset$. For each $q \in P_{q'}$, if $(q, w[\ell], q') \in F$, then we add $\mathfrak{t}_\ell(q) \frown [u]$ to $C_{q'}$, where $u = \min U$, and we remove u from U, so that each number in U is used only once; otherwise, for $(q, w[\ell], q') \in \delta_N \setminus F$, we add $\mathfrak{t}_\ell(q)$ to $C_{q'}$. Lastly, we set $\mathfrak{t}'_{\ell+1}(q') = \min C_{q'}$, where the minimum is taken according to the list order.
   Intuitively, if a run ρ can branch into two kinds of runs, some via accepting transitions and some others via nonaccepting transitions at level $\ell + 1$, then we let those from nonaccepting transitions inherit the labelling from ρ, i.e., $\mathfrak{t}_\ell(\rho[\ell])$; for the runs taking accepting transitions we create a new labelling $\mathfrak{t}_\ell(\rho[\ell]) \frown [u]$. In this way, the latter get precedence over the former. Moreover, if a run ρ has received multiple labelling values, collected in $C_{\rho[\ell+1]}$, then it will keep the smallest one, by $\mathfrak{t}'_{\ell+1}(\rho[\ell + 1]) = \min C_{\rho[\ell+1]}$.
2. For each state $q' \in (S_{\ell+1} \cap \mathsf{N}) \setminus \delta_N(S_\ell \cap \mathsf{N}, w[\ell])$ taken according to the state order $\preccurlyeq$, we first set $\mathfrak{t}'_{\ell+1}(q') = [u]$, where $u = \min U$, and then we remove u from U, so we do not reuse the same values. That is, we give the newly entered runs lower precedence than those already in N, by means of the larger list $[u]$.

We now need to prune the lists in $\beta(\mathfrak{t}'_{\ell+1})$ and recognize good and bad events. Similarly to DACs, a bad event means that a run has left N or has been merged with runs with smaller labelling, which is indicated by a discontinuation of a labelling between $\beta(\mathfrak{t}_\ell)$ and $\beta(\mathfrak{t}'_{\ell+1})$. For the transition $e = (\mathfrak{t}_\ell, w[\ell], \mathfrak{t}_{\ell+1})$ we are constructing, to recognize bad events, we put into the set $\mathsf{B}(e)$ the num-

ber $|\mathsf{N}| + 1$ and all numbers in $\beta(\mathfrak{t}_\ell)$ that have disappeared in $\beta(\mathfrak{t}'_{\ell+1})$, that is, $\mathsf{B}(e) = \{|\mathsf{N}| + 1\} \cup \{\, v \in \mathbb{N} \mid v \,\text{occurs in}\, \beta(\mathfrak{t}_\ell) \,\text{but not in}\, \beta(\mathfrak{t}'_{\ell+1}) \,\}$.

Differently from the good events for DACs, which require to visit an accepting transition, we need all runs branched from a run to visit an accepting transition, which is indicated by the fact that there are no states labelled by $\mathfrak{t}'_{\ell+1}$ with some list $l \in \beta(\mathfrak{t}_\ell)$ but there are extensions of l associated with some state. To recognize good events, let $\mathsf{G}(e) = \{|\mathsf{N}| + 1\}$ and $\mathfrak{t}''_{\ell+1}$ be another intermediate labelling function. For each $q' \in S_{\ell+1} \cap \mathsf{N}$, consider the list $\mathfrak{t}'_{\ell+1}(q')$: if for each prefix $[v_1, \cdots v_k]$ of $\mathfrak{t}'_{\ell+1}(q')$ we have $[v_1, \cdots v_k] \in \beta(\mathfrak{t}'_{\ell+1})$, then we set $\mathfrak{t}''_{\ell+1}(q') = \mathfrak{t}'_{\ell+1}(q')$. Otherwise, let $[v_1, \cdots v_{\bar{k}}] \notin \beta(\mathfrak{t}'_{\ell+1})$ be the shortest prefix of $\mathfrak{t}'_{\ell+1}(q')$ not in $\beta(\mathfrak{t}'_{\ell+1})$; we set $\mathfrak{t}''_{\ell+1}(q') = [v_1, \cdots v_{\bar{k}}]$ and add $v_{\bar{k}}$ to $\mathsf{G}(e)$. Setting $\mathfrak{t}''_{\ell+1}(q') = [v_1, \cdots v_{\bar{k}}]$ in fact corresponds, in the Safra's construction [36], to the removal of all children of a node $\mathfrak{N}$ for which the union of the states in the children is equal to the states in $\mathfrak{N}$. Lastly, similarly to the DAC case, we set $\mathfrak{t}_{\ell+1}(q) = \mathsf{ord}(\mathfrak{t}''_{\ell+1}(q))$ for each $q \in S_{\ell+1} \cap \mathsf{N}$ and $\mathfrak{t}_{\ell+1}(q') = []$ for each $q' \in \mathsf{N} \setminus S_{\ell+1}$, where $\mathsf{ord}([v_1, \cdots, v_k]) = [\mathsf{ord}(v_1), \cdots, \mathsf{ord}(v_k)]$. Regarding the color to assign to the transition e, we just assign the color $c = \min\{2 \cdot \min \mathsf{G}(e), 2 \cdot \min \mathsf{B}(e) - 1\}$.

Lemma 3. *(1) An accepting run of $\mathcal{A}$ over w eventually stays in the NAC N if and only if the minimal color c we receive infinitely often is even. (2) The number of possible labelling functions $\mathfrak{t}$ is at most $2 \cdot (|\mathsf{N}|!)^2$.*

Similarly to DACs, also for NACs we have handled each NAC independently. The reason for this is that this potentially reduces the complexity of the single cases: assume that we have two NACs N_1 and N_2. If we apply the Safra-Piterman's construction directly to $\mathsf{N}_1 \cup \mathsf{N}_2$, we might incur in the worst-case complexity $2 \cdot ((|\mathsf{N}_1| + |\mathsf{N}_2|)!)^2$, as mentioned in the introduction. However, if we determinize them separately, then the worst complexity for each NAC N_i is $2 \cdot (\mathsf{N}_i!)^2$, for an overall $4 \cdot (|\mathsf{N}_1|! \cdot |\mathsf{N}_2|!)^2$, much smaller than $2 \cdot ((|\mathsf{N}_1| + |\mathsf{N}_2|)!)^2$.

As usual, we make the above construction available as a function $\mathsf{nondetSucc}$, which takes as input the NAC N, a labelling $\mathfrak{t}$ and a letter a, and returns the successor labelling $\mathfrak{t}'$ and the corresponding color $c \in \{1, \cdots, 2 \cdot |\mathsf{N}| + 1\}$.

Similarly to the constructions for other SCCs, we show on the right the fragment of run DAG $\mathcal{G}_{\mathcal{A},a^\omega}$ for the NAC $\mathsf{N} = \{q_0, q_1\}$, with $q_0 \preccurlyeq q_1$. The construction of $\mathfrak{t}_1$ is easy, so consider its a-successor $\mathfrak{t}_2$: we start with $U = \{3, 4, \cdots\}$; for q_0, we have $P_{q_0} = \{q_0, q_1\}$ and $C_{q_0} = \{[1, 2, 3], [1]\}$, hence $\mathfrak{t}'_2(q_0) = [1, 2, 3]$. For q_1, we get $P_{q_1} = \{q_0\}$ and $C_{q_1} = \{[1, 2]\}$, so $\mathfrak{t}'_2(q_1) = [1, 2]$. Thus, for $e = (\mathfrak{t}_1, w[1], \mathfrak{t}_2)$, we have $\mathsf{B}(e) = \{3\}$ while $\mathsf{G}(e) = \{1, 3\}$, since both lists in $\beta(\mathfrak{t}'_2) = \{[1, 2], [1, 2, 3]\}$ are missing the prefix $[1]$, so we get $\mathfrak{t}_2(q_0) = \mathfrak{t}_2(q_1) = [1]$ and color $c = 2$.

ℓ
0 $q_0, [1]$
1 $q_0, [1, 2]$ $q_1, [1]$
2 $q_0, [1]$ $q_1, [1]$

4 Determinization of NBAs to DELAs

In this section, we fix an NBA $\mathcal{A} = (Q, \iota, \delta, F)$ with $n = |Q|$ states and we show how to construct an equivalent DELA $\mathcal{A}^{\mathsf{E}} = (Q^{\mathsf{E}}, \iota^{\mathsf{E}}, \delta^{\mathsf{E}}, \Gamma^{\mathsf{E}}, \mathsf{p}^{\mathsf{E}}, \mathsf{Acc}^{\mathsf{E}})$,

by using the algorithms developed in the previous section. We assume that $\mathcal{A}$ has $\{\mathsf{D}^1, \cdots, \mathsf{D}^d\}$ as set of DACs and $\{\mathsf{N}^1, \cdots, \mathsf{N}^k\}$ as set of NACs.

When computing the successor for each type of SCCs while reading a word w, we just need to know the set S_ℓ of states reached at the current level ℓ and the letter $a \in \Sigma$ to read. We can ignore the actual level ℓ, since if $S_\ell = S_{\ell'}$, then their successors under the same letter will be the same. As mentioned before, every state of $\mathcal{A}^{\mathsf{E}}$ corresponds to a level of $\mathcal{G}_{\mathcal{A},w}$. We call a state of $\mathcal{A}^{\mathsf{E}}$ a *macrostate* and a run of $\mathcal{A}^{\mathsf{E}}$ a *macrorun*, to distinguish them from those of $\mathcal{A}$.

Macrostates Q^{E}. Each macrostate consists of the pair (P, O) for encoding the states in IWCs, a labelling function $\mathfrak{g}^i \colon \mathsf{D}^i \to \{1, \cdots, |\mathsf{D}^i|\} \cup \{\infty\}$ for the states of each DAC D^i and a labelling function $\mathfrak{t}^j \colon \mathsf{N}^j \to \mathcal{L}_{\mathsf{N}^j}$ for each NAC N^j, without the explicit level number. The initial macrostate ι^{E} of $\mathcal{A}^{\mathsf{E}}$ is the encoding of level 0, defined as the set $\{(P_0, O_0)\} \cup \{\, \mathfrak{g}_0^i \mid \mathsf{D}^i \text{is a DAC}\,\} \cup \{\, \mathfrak{t}_0^j \mid \mathsf{N}^j \text{is a NAC}\,\}$, where each encoding for the different types of SCCs is the one for level 0.

We note that ι must be present in one type of SCCs. In particular, if ι is a transient state, then $\{\iota\}$ is classified as an IWC.

Transition Function δ^{E}. Let m be the current macrostate in Q^{E} and $a \in \Sigma$ be the letter to read. Then we define $m' = \delta^{\mathsf{E}}(m, a)$ as follows.

(i) For $(P_m, O_m) \in m$, we set $(P_{m'}, O_{m'}) = \mathsf{weakSucc}((P_m, O_m), a)$ in m'.
(ii) For $\mathfrak{g}_m^i \in m$ relative to the DAC D^i, we set $\mathfrak{g}_{m'}^i = \mathsf{detSucc}(\mathsf{D}^i, \mathfrak{g}_m^i, a)$ in m'.
(iii) For $\mathfrak{t}_m^j \in m$ from the NAC N^j, we set $\mathfrak{t}_{m'}^j = \mathsf{nondetSucc}(\mathsf{N}^j, \mathfrak{t}_m^j, a)$ in m'.

Note that the set S of the current states of $\mathcal{A}$ used by the different successor functions is implicitly given by the sets P, $\{\, q \in \mathsf{D}^i \mid \mathfrak{g}^i(q) \neq \infty \,\}$ for each DAC D^i and $\{\, q \in \mathsf{N}^j \mid \mathfrak{t}^j(q) \neq [] \,\}$ for each NAC N^j in the current macrostate m.

Color Set Γ^{E} and Coloring Function p^{E}. From the constructions given in Sect. 3, we have two colors from the IWCs, $2 \cdot |\mathsf{D}^i| + 1$ colors for each DAC D^i, and $2 \cdot |\mathsf{N}^j| + 1$ colors for each NAC N^j, yielding a total of at most $3 \cdot |Q|$ colors. Thus we set $\Gamma^{\mathsf{E}} = \{0, 1, \cdots, 3 \cdot |Q|\}$ with color 0 not being actually used.

Regarding the color to assign to each transition, we need to ensure that the colors returned by the single SCCs are treated separately, so we transpose them. For a transition $e = (m, a, m') \in \delta^{\mathsf{E}}$, we define the coloring function p^{E} as follows.

- If we receive color 1 for the transition $((P_m, O_m), a, (P_{m'}, O_{m'}))$, then we put $1 \in \mathsf{p}^{\mathsf{E}}(e)$. Intuitively, every time we see an empty O-set along reading an ω-word w in the IWCs, we put the color 1 on the transition (m, a, m').
- For each DAC D^i, we transpose its colors after the colors for the IWCs and the other DACs with smaller index. So we set the base number for the colors of the DAC D^i to be $\mathsf{b}_i = 2 + \sum_{1 \le h < i} (2 \cdot |\mathsf{D}^h| + 1)$, i.e., the number of colors already being used. Then, if we receive the color c for the transition $(\mathfrak{g}_m^i, a, \mathfrak{g}_{m'}^i)$ from $\mathsf{detSucc}$, we put $c + \mathsf{b}_i \in \mathsf{p}^{\mathsf{E}}(e)$.
- We follow the same approach for the NAC N^j: we set its base number to be $\mathsf{b}_j = 2 + \sum_{1 \le h \le d} (2 \cdot |\mathsf{D}^h| + 1) + \sum_{1 \le h < j} (2 \cdot |\mathsf{N}^h| + 1)$. Then, if we receive the color c for the transition $(\mathfrak{t}_m^j, a, \mathfrak{t}_{m'}^j)$ from $\mathsf{nondetSucc}$, we put $c + \mathsf{b}_j \in \mathsf{p}^{\mathsf{E}}(e)$.

Intuitively, we make the colors returned for each SCC not overlap with those of other SCCs without changing their relative order. In this way, we can still independently check whether there exists an accepting run staying in an SCC.

Acceptance Formula $\mathsf{Acc}^{\mathsf{E}}$. We now define the acceptance $\mathsf{Acc}^{\mathsf{E}}$, which is basically the *disjunction* of the acceptance formula for each different types of SCCs, after transposing them. Regarding the IWCs, we trivially define $\mathsf{Acc}^{\mathsf{E}}_{\mathsf{W}} = \mathsf{Fin}(1)$, since this is the acceptance formula for IWCs; as said before, color 0 is not used.

For DACs and NACs, the definition is more involved. For instance, regarding the DAC D^i, we know that all returned colors are inside $\{1, \cdots, 2 \cdot |\mathsf{D}^i| + 1\}$. According to Lemma 2, an accepting run eventually stays in D^i if and only if the minimum color that we receive infinitely often is even. Thus, the acceptance formula for the above lemma is $\mathsf{parity}(|\mathsf{D}^i|) = \bigvee_{c=1}^{|\mathsf{D}^i|}(\bigwedge_{j=1}^{c} \mathsf{Fin}(2j-1) \wedge \mathsf{Inf}(2c))$. Let $\mathsf{b}_i = 2 + \sum_{h<i}(2 \cdot |\mathsf{D}_h| + 1)$ be the base number for the colors of D^i, which is also the number of colors already used by IWCs and the DACs D^h with $h < i$. Since we have added the base number b^i to every color of D^i, we then have the acceptance formula $\mathsf{Acc}^{\mathsf{E}}_{\mathsf{D}^i} = \bigvee_{c=1}^{|\mathsf{D}^i|}(\bigwedge_{j=1}^{c} \mathsf{Fin}(2j-1+\mathsf{b}_i) \wedge \mathsf{Inf}(2c+\mathsf{b}_i))$.

For each NAC N^j, the colors we receive are in $\{1, \cdots, 2 \cdot |\mathsf{N}^j| + 1\}$. Let $\mathsf{b}_j = 2 + \sum_{1 \le h \le d}(2 \cdot |\mathsf{D}^h| + 1) + \sum_{h<j}(2 \cdot |\mathsf{N}^j| + 1)$ be the base number for N^j. Similarly to the DAC case, for each NAC N^j, we let $\mathsf{Acc}^{\mathsf{E}}_{\mathsf{N}^j} = \bigvee_{c=1}^{|\mathsf{N}^j|}(\bigwedge_{i=1}^{c} \mathsf{Fin}(2i - 1 + \mathsf{b}_j) \wedge \mathsf{Inf}(2c + \mathsf{b}_j))$.

The acceptance formula for $\mathcal{A}^{\mathsf{E}}$ is $\mathsf{Acc}^{\mathsf{E}} = \mathsf{Acc}^{\mathsf{E}}_{\mathsf{W}} \vee \bigvee_{i=1}^{d} \mathsf{Acc}^{\mathsf{E}}_{\mathsf{D}^i} \vee \bigvee_{j=1}^{k} \mathsf{Acc}^{\mathsf{E}}_{\mathsf{N}^j}$.

Consider again the NBA $\mathcal{A}$ given in Fig. 1 and its various SCCs. As acceptance formula for the constructed DELA, it is the disjunction of the formulas $\mathsf{Acc}^{\mathsf{E}}_{\mathsf{W}} = \mathsf{Fin}(1)$; $\mathsf{Acc}^{\mathsf{E}}_{\mathsf{D}} = \bigvee_{c=1}^{2}(\bigwedge_{j=1}^{c} \mathsf{Fin}(2j-1+2) \wedge \mathsf{Inf}(2c+2))$, since the base number for D is 2; and $\mathsf{Acc}^{\mathsf{E}}_{\mathsf{N}} = \bigvee_{c=1}^{2}(\bigwedge_{i=1}^{c} \mathsf{Fin}(2i-1+7) \wedge \mathsf{Inf}(2c+7))$, since 7 is the base number for N.

The construction given in this section is correct, as stated by Theorem 1.

Theorem 1. *Given an NBA $\mathcal{A}$ with $n = |Q|$ states, let $\mathcal{A}^{\mathsf{E}}$ be the DELA constructed by our method. Then (1) $\mathsf{L}(\mathcal{A}^{\mathsf{E}}) = \mathsf{L}(\mathcal{A})$ and (2) $\mathcal{A}^{\mathsf{E}}$ has at most $3^{|\mathsf{W}|} \cdot \left(\prod_{i=1}^{d} 3 \cdot |\mathsf{D}^i|!\right) \cdot \left(\prod_{j=1}^{k} 2 \cdot (|\mathsf{N}^i|!)^2\right)$ macrostates and $3n+1$ colors.*

Obviously, if $d = k = 0$, $\mathcal{A}$ is a weak BA [32]. If $k = 0$, $\mathcal{A}$ is an elevator BA, a new class of BAs recently introduced in [19] which have only IWCs and DACs, a strict superset of semi-deterministic BAs (SDBAs) [10]. SDBAs will behave *deterministically* after seeing acceptance transitions. An elevator BA that is not an SDBA can be obtained from the NBA $\mathcal{A}$ shown in Fig. 1 by setting q_2 as initial state and by removing all states and transitions relative to the NAC.

It is known that the lower bound for determinizing SDBAs is $n!$ [14,27]. Then the determinization complexity of weak BAs and elevator BAs can be easily improved exponentially as follows.

Corollary 1. *(1) Given a weak Büchi automaton $\mathcal{A}$ with $n = |Q|$ states, the DELA constructed by our algorithm has at most 3^n macrostates. (2) Given an*

elevator Büchi automaton $\mathcal{A}$ with $n = |Q|$ states, our algorithm constructs a DELA with $\Theta(n!)$ macrostates; it is asymptotically optimal.

The upper bound for determinizing weak BAs is already known [5]. Elevator BAs are, to the best of our knowledge, the *largest* subclass of NBAs known so far to have determinization complexity $\Theta(n!)$.

The acceptance formula for an SCC can be seen as a parity acceptance formula with colors being shifted to different ranges. A parity automaton can be converted into a Rabin one without blow-up of states and transitions [16]. Since $\mathsf{Acc}^{\mathsf{E}}$ is a disjunction of parity acceptance formulas, Theorem 2 then follows.

Theorem 2. *Let $\mathcal{A}^{\mathsf{E}}$ be the constructed DELA for the given NBA $\mathcal{A}$. Then $\mathcal{A}^{\mathsf{E}}$ can be converted into a DRA $\mathcal{A}^{\mathsf{R}}$ without blow-up of states and transitions.*

Translation to Deterministic Parity Automata (DPAs). We note that there is an *optimal* translation from a DRA to a DPA described in [7], implemented in SPOT via the function acd_transform [8].

5 Empirical Evaluation

To analyze the effectiveness of our Divide-and-Conquer determinization construction proposed in Sect. 3, we implemented it in our tool COLA, which is built on top of SPOT [12]. The source code of COLA is publicly available from https://github.com/liyong31/COLA. We compared COLA with the official versions of SPOT [12] (2.10.2) and OWL [23] (21.0). SPOT implements the algorithm described in [35], a variant of [33] for transition-based NBAs, while OWL implements the algorithms described in [28,29], both constructing DPAs as result. To make the comparison fair, we let all tools generate DPAs, so we used the command `autfilt --deterministic --parity=min\ even -F file.hoa` to call SPOT and `owl nbadet -i file.hoa` to call OWL. Recall that we use the function acd_transform [8] from SPOT for obtaining DPAs from our DRAs. The tools above also implement optimizations for reducing the size of the output DPA, like simulation and state merging [29], or stutter invariance [22] (except for OWL); we use the default settings for all tools. We performed our experiments on a desktop machine equipped with 16GB of RAM and a 3.6 GHz Intel Core i7-4790 CPU. We used BENCHEXEC[1] [3] to trace and constrain the tools' executions: we allowed each execution to use a single core and 12 GB of memory, and imposed a timeout of 10 min. We used SPOT to verify the results generated by three tools and found only outputs equivalent to the inputs.

As benchmarks, we considered all NBAs in the HOA format [1] available in the AUTOMATA-BENCHMARKS repository.[2] We have pre-filtered them with `autfilt` to exclude all deterministic cases and to have nondeterministic BAs, obtaining in total 15,913 automata coming from different sources in literature.

The artifact with tools, benchmarks, and scripts to run the experiments and generate the plots is available at [25].

[1] https://github.com/sosy-lab/benchexec/.
[2] https://github.com/ondrik/automata-benchmarks/.

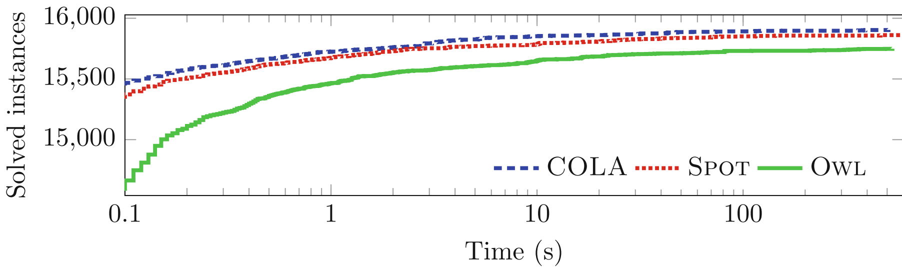


Fig. 2. The cactus plot for the determinization of NBAs from AUTOMATA-BENCHMARKS.

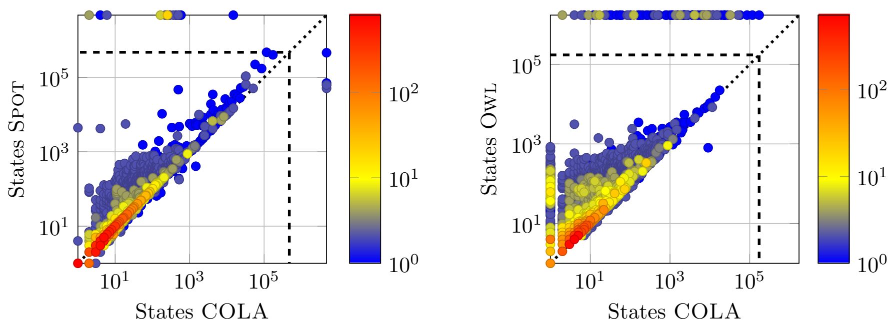
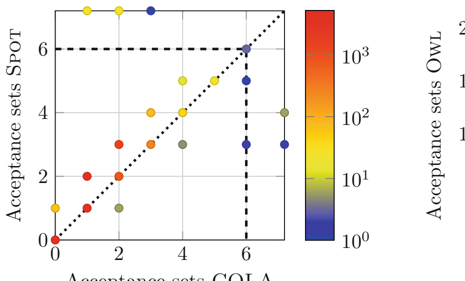


Fig. 3. States comparison for the determinization of NBAs from AUTOMATA-BENCHMARKS. (Color figure online)

In Fig. 2 we show a cactus plot reporting how many input automata have been determinized by each tool, over time. As we can see, COLA works better than Spot, with COLA solving in total 15,903 cases and Spot 15,862 cases, with Owl solving in total 15,749 cases and taking more time to solve as many instances as COLA and Spot. From the plot given in Fig. 2 we see that COLA is already very competitive with respect to its performance.

In Fig. 3 we show the number of states of the generated DPAs. In the plot we indicate with the bold dashed line the maximum number of states of the automata produced by either of the two tools, and we place a mark on the upper or right border of the plot to indicate that one tool has generated an automaton with that size while the other tool just failed. The color of each mark represents how many instances have been mapped to the corresponding point. As the plots show, Spot and COLA generate automata with similar size, with COLA being more likely to generate smaller automata, in particular for larger outputs. Owl, instead, very frequently generates automata larger than COLA. In fact, on the 15,710 cases solved by all tools, on average COLA generated 44 states, Spot 65, and Owl 87. If we compare COLA with just one tool at a time, on the 15,854 cases solved by both COLA and Spot, we have 125 states for COLA and 246 for Spot; on the 15,749 cases solved by both COLA and



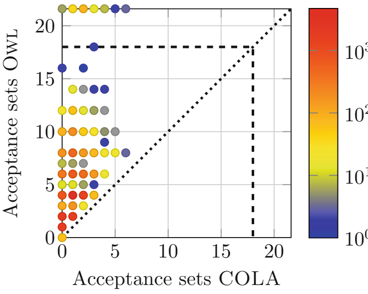


Fig. 4. Acceptance sets comparison for the determinization of NBAs from AUTOMATA-BENCHMARKS. (Color figure online)

Table 1. Pearson correlation coefficients for the AUTOMATA-BENCHMARKS experiments.

	# input states	# input SCCs	average SCC size
runtime	0.77	0.62	−0.01
output states	0.41	0.17	0.05

OWL, we have 45 states for COLA and 88 for OWL. A similar situation occurs for the number of transitions, so we omit it.

Lastly, in Fig. 4 we compare the number of acceptance sets (i.e., the colors in Definition 1) of the generated DPAs; more precisely, we consider the integer value occurring in the mandatory `Acceptance: INT acceptance-cond` header item of the HOA format [1], which can be 0 for the automata with all or none accepting transitions. From the plots we can see that COLA generates more frequently DPAs with a number of colors that is no more than the number used by SPOT, as indicated by the yellow/red marks on (10,394 cases) or above (5,495 cases) the diagonal. Only in very few cases COLA generates DPAs with more colors than SPOT (22 cases), as indicated by the few blue/greenish marks below the diagonal. Regarding OWL, however, from the plot we can clearly see that COLA uses almost always (15,840 cases) fewer colors than OWL; the only exception is for the mark at $(0, 0)$ representing 63 cases.

The number and sizes of SCCs influence the performance of COLA, so we provide some statistics about the correlation between these and the runtime and size of the generated DPA. By combining the execution statistics with the input SCCs and states, we get the Pearson correlation coefficients shown in Table 1. Here the larger the number in a cell is, the stronger the positive correlation between the element that the row and the column represent. From these coefficients we can say that there is a quite strong positive correlation between the number of states and of SCCs and the running time, but not for the average SCC size; regarding the output states, the situation is similar but much weaker.

We also considered a second set of benchmarks – 644 NBAs generated by SPOT's `ltl2tgba` on the LTL formulas considered in [23], as available in the

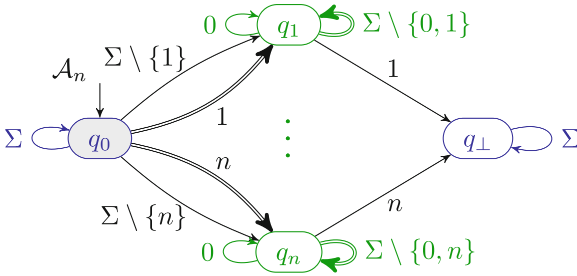


Fig. 5. The family of NBAs $\mathcal{A}_n$ with $\Sigma = \{0, 1, \cdots, n\}$.

OWL's repository at https://gitlab.lrz.de/i7/owl. The outcomes for these benchmarks are similar, but a bit better for COLA, to the ones for AUTOMATA-BENCHMARKS, so we do not present them in detail.

6 Related Work

To the best of our knowledge, our determinization construction is the *first* algorithm that determinizes SCCs *independently* while taking advantage of different structures of SCCs, which is the main difference between our algorithm and existing works. We illustrate other minor differences below.

Different types of SCCs, like DACs and IWCs, are also taken with special care in [29] as in our work, modulo the handling details. However, the work [29] does not treat them independently as the labelling numbers in those SCCs still have relative order with those in other SCCs. Thus their algorithm can be exponentially worse than ours (cf. Theorem 3) and performs not as well as ours in practice; see the comparison with OWL in Sect. 5. The determinization algorithm given in [14] for SDBAs is a special case of the one presented in [35] for NBAs, which gives precedence to the deterministic runs seeing accepting transitions earlier, while we give precedence to runs that enter DACs earlier. More importantly, the algorithm from [14] does not work when there is nondeterminism between DACs, while our algorithm overcomes this by considering DACs separately and by ignoring runs going to other SCCs.

Current works for determinization of general NBAs, such as [18,21,28,35,36,38] can all be interpreted as different flavours of the Safra-Piterman based algorithm. Our determinization of NACs is also based on Safra-trees and inspired by SPOT, except that we may have newly arriving states from other SCCs while other works only need to consider the successors from the current states in the Safra-tree. The modular approach for determinizing Büchi automata given in [17] builds on reduced split trees [21] and can construct the deterministic automaton with a given tree-width. The algorithm constructs the final deterministic automaton by running in parallel the NBA for all possible tree-widths, rather than working on SCCs independently as we do in this work.

Compared to the algorithms operating on the whole NBA, our algorithm can be exponentially better on the family of NBAs shown in Fig. 5, as formalized in Theorem 3; we can encounter some variation of this family of NBAs when working with fairness properties. The intuition is that we take care of the DACs $\{q_i\}_{i=1}^{n}$ independently, so for each of them we have only two choices: either the run is in the DAC, or it is not in the DAC; resulting in a single exponential number of combinations. Existing works [14,21,28,33,35,36] order the runs entering the DACs based on when they visit accepting transitions, in which every order corresponds to a permutation of $\{q_1, \cdots, q_n\}$.

Theorem 3. *There exists a family of NBAs $\mathcal{A}_n$ with $n+2$ states for which the algorithms in [14, 21, 28, 33, 35, 36] give a DPA with at least $n!$ macrostates while ours gives a DELA with at most 2^{n+2} macrostates.*

In practice, for each NBA $\mathcal{A}_n$, $n \geq 3$, COLA produces a DELA/DPA with n macrostates, while both Spot and Owl give a DPA with $n! + 1$ macrostates.

7 Conclusion and Future Work

We proposed a divide-and-conquer determinization construction for NBAs that takes advantage of the structure of different types of SCCs and determinizes them independently. In particular, our construction can be exponentially better than classical works on a family of NBAs. Experiments showed that our algorithm outperforms the state-of-the-art implementations regarding the number of states and transitions on a large set of benchmarks. To summarize, our divide-and-conquer determinization construction is very *practical*, being a good complement to existing theoretical approaches.

Our divide-and-conquer approach for NBAs can also be applied to the complementation problems of NBAs. By Proposition 1, w is not accepted by $\mathcal{A}$ if and only if there are no accepting runs staying in an SCC. Thus we can construct a *generalized* Büchi automaton with a conjunction of $\mathsf{Inf}(i)$ as the acceptance formula to accept the complement language $\Sigma^\omega \setminus \mathsf{L}(\mathcal{A})$ of $\mathcal{A}$; the generalized Büchi automaton in fact takes the *intersection* of the complement language of each type of SCCs. For complementing IWCs, we use the same construction as determinization except that the acceptance formula will be $\mathsf{Inf}(1)$. For complementing DACs, we can borrow the idea of NCSB complementation construction [4] which complements SDBAs in time 4^n. For complementing NACs, we just adapt the *slice-based* complementation [21] of general NBAs. We leave the details of this divide-and-conquer complementation construction for NBAs as future work.

Acknowledgement. We thank the anonymous reviewers for their valuable suggestions to this paper. This work is supported in part by the National Natural Science Foundation of China (Grant Nos. 62102407 and 61836005), NSF grants IIS-1527668, CCF-1704883, IIS-1830549, CNS-2016656, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 101008233.

References

1. Babiak, T., et al.: The Hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
4. Blahoudek, F., Heizmann, M., Schewe, S., Strejček, J., Tsai, M.-H.: Complementing semi-deterministic Büchi automata. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 770–787. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_49
5. Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 611–625. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45744-5_50
6. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: The Collected Works of J. Richard Büchi, pp. 425–435. Springer, Cham (1990). https://doi.org/10.1007/978-1-4613-8928-6_23
7. Casares, A., Colcombet, T., Fijalkow, N.: Optimal transformations of games and automata using Muller conditions. In: ICALP. LIPIcs, vol. 198, pp. 123:1–123:14 (2021)
8. Casares, A., Duret-Lutz, A., Meyer, K.J., Renkin, F., Sickert, S.: Practical applications of the alternating cycle decomposition. In: TACAS. LNCS, vol. 13244, pp. 99–117. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_6
9. Colcombet, T., Zdanowski, K.: A tight lower bound for determinization of transition labeled Büchi automata. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 151–162. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02930-1_13
10. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
11. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI, pp. 854–860 (2013)
12. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
13. Emerson, E.A., Lei, C.: Modalities for model checking: branching time logic strikes back. Sci. Comput. Program. **8**(3), 275–306 (1987)
14. Esparza, J., Křetínský, J., Raskin, J.-F., Sickert, S.: From LTL and limit-deterministic Büchi automata to deterministic parity automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 426–442. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_25
15. Esparza, J., Křetínský, J., Sickert, S.: A unified translation of linear temporal logic to ω-automata. J. ACM **67**(6), 1–61 (2020)
16. Farwer, B.: Omega-automata. In: Automata, Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500, pp. 3–20 (2001)

17. Fisman, D., Lustig, Y.: A modular approach for Büchi determinization. In: CONCUR. LIPIcs, vol. 42, pp. 368–382 (2015)
18. Fogarty, S., Kupferman, O., Vardi, M.Y., Wilke, T.: Profile trees for Büchi word automata, with application to determinization. Inf. Comput. **245**, 136–151 (2015)
19. Havlena, V., Lengál, O., Smahlíková, B.: Sky is not the limit. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13244, pp. 118–136. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_7
20. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)
21. 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.) ICALP 2008. LNCS, vol. 5125, pp. 724–735. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70575-8_59
22. Klein, J., Baier, C.: On-the-fly stuttering in the construction of deterministic ω-automata. In: Holub, J., Žd́árek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 51–61. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76336-9_7
23. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: a library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 543–550. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_34
24. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001)
25. Li, Y., Turrini, A., Feng, W., Vardi, M.V., Zhang, L.: Artifact for "Divide-and-conquer determinization of Büchi automata based on SCC decomposition" (2022). https://doi.org/10.5281/zenodo.6558928
26. Liu, W., Wang, J.: A tighter analysis of Piterman's Büchi determinization. Inf. Process. Lett. **109**(16), 941–945 (2009)
27. Löding, C.: Optimal bounds for transformations of ω-automata. In: Rangan, C.P., Raman, V., Ramanujam, R. (eds.) FSTTCS 1999. LNCS, vol. 1738, pp. 97–109. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-46691-6_8
28. Löding, C., Pirogov, A.: Determinization of Büchi automata: unifying the approaches of Safra and Muller-Schupp. In: ICALP. LIPIcs, vol. 132, pp. 120:1–120:13 (2019)
29. Löding, C., Pirogov, A.: New optimizations and heuristics for determinization of Büchi automata. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 317–333. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_18
30. Michel, M.: Complementation is more difficult with automata on infinite words. Technical report, CNET, Paris (Manuscript) (1988)
31. Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theor. Comput. Sci. **32**(3), 321–330 (1984)
32. Muller, D.E., Saoudi, A., Schupp, P.E.: Alternating automata, the weak monadic theory of trees and its complexity. Theor. Comput. Sci. **97**(2), 233–244 (1992)
33. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Log. Methods Comput. Sci. **3**(3), 1–21 (2007)
34. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
35. Redziejowski, R.R.: An improved construction of deterministic omega-automaton using derivatives. Fundam. Informaticae **119**(3–4), 393–406 (2012)

36. Safra, S.: On the complexity of ω-automata. In: FOCS, pp. 319–327 (1988)
37. Safra, S., Vardi, M.Y.: On omega-automata and temporal logic (preliminary report). In: STOC, pp. 127–137 (1989)
38. Schewe, S.: Büchi complementation made tight. In: STACS. LIPIcs, vol. 3, pp. 661–672 (2009)
39. Schewe, S.: Tighter bounds for the determinisation of Büchi automata. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1_13
40. Tsai, M., Fogarty, S., Vardi, M., Tsay, Y.: State of Büchi complementation. Log. Methods Comput. Sci. **10**(4), 1–27 (2014)
41. Vardi, M.Y.: The rise and fall of linear temporal logic. In: GandALF (2011). Invited talk
42. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
43. Yan, Q.: Lower bounds for complementation of ω-automata via the full automata technique. Log. Methods Comput. Sci. **4**(1:5), 1–20 (2008)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.