Determinization of Büchi Automata: Unifying the Approaches of Safra and Muller-Schupp

Christof Löding
RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
loeding@cs.rwth-aachen.de

Anton Pirogov
RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
pirogov@cs.rwth-aachen.de

Abstract
Determinization of Büchi automata is a long-known difficult problem, and after the seminal result of Safra, who developed the first asymptotically optimal construction from Büchi into Rabin automata, much work went into improving, simplifying, or avoiding Safra’s construction. A different, less known determinization construction was proposed by Muller and Schupp. The two types of constructions share some similarities but their precise relationship was still unclear. In this paper, we shed some light on this relationship by proposing a construction from nondeterministic Büchi to deterministic parity automata that subsumes both constructions: Our construction leaves some freedom in the choice of the successor states of the deterministic automaton, and by instantiating these choices in different ways, one obtains as particular cases the construction of Safra and the construction of Muller and Schupp. The basis is a correspondence between structures that are encoded in the macrostates of the determinization procedures—Safra trees on one hand, and levels of the split-tree, which underlies the Muller and Schupp construction, on the other hand. Our construction also allows for mixing the mentioned constructions, and opens up new directions for the development of heuristics.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects

Keywords and phrases Büchi automata, determinization, parity automata

Funding Anton Pirogov: This author is supported by the German research council (DFG) Research Training Group 2236 UnRAVeL

1 Introduction

Büchi automata are finite automata for infinite words, and were initially introduced to show decidability of the logic S1S [2]. Infinite words can be used to model infinite execution traces of reactive, non-terminating systems, and serve as a translation target from logics like LTL (see, e.g., [15, 5]), which is a popular and well-understood specification formalism. For this reason, Büchi automata nowadays play a central role in formal methods like model-checking [1] and runtime-verification [6], because they can represent all ω-regular languages and are suitable for efficient algorithmic treatment. Unfortunately, the simplicity of the Büchi acceptance condition makes it crucially dependent on nondeterminism, i.e., not every ω-regular language (or LTL formula) can be accepted by a deterministic Büchi automaton (see, e.g., [16]). In some settings, this nondeterminism causes difficulties, such that algorithms require a representation of the property by a deterministic automaton, like in probabilistic model-checking (see, e.g., [11 Section 10.3]), or in synthesis (see [17] for an overview of the theory, and [9] for recent developments in practice).
A first determinization procedure that translates nondeterministic Büchi automata into deterministic automata was presented in [8]. The first asymptotically optimal and most well-known determinization construction for Büchi automata is the construction of Safra [13]. It translates a nondeterministic Büchi automaton with \( n \) states into a deterministic Rabin automaton with at most \( 2^{O(n \log n)} \) states and \( O(n) \) sets in the acceptance condition. In applications like synthesis, the deterministic automaton is used to build a game that inherits as winning condition the acceptance condition of the automaton. In the theory of infinite duration games, the parity condition plays a central role (see, e.g., the survey [18]). For this reason, Piterman modified Safra’s construction in order to directly obtain a parity automaton [11]. This construction was reformulated in [14], where also a tighter analysis of its state complexity is given with an upper bound of \( O(n^2) \) for the number of states. A similar construction is presented in [12], adapted to the translation of \( \omega \)-regular expressions directly into parity automata.

It is known that the Safra construction is essentially optimal [3], so there is no hope of significantly improving the worst-case upper bounds of the known constructions. However, the data structure of Safra trees (or history trees) that is used for the states of the deterministic automata, is challenging to deal with in implementations. Therefore, alternative approaches for determinization have been studied, leading to a family of constructions that are based on a construction by Muller and Schupp, which appeared in [10] as a by-product of a translation from alternating to non-deterministic tree automata. An explicit description of the construction specifically for determinization of Büchi automata is presented in [7]. A refinement of that construction is presented in [4], in which the states of the deterministic automaton are no longer represented as trees but as ordered and labelled tuples of sets.

The two approaches of Safra and Muller-Schupp show some similarities, as pointed out in the conclusion of [4], but from the existing formulations of the constructions, their precise relationship is not clear.

In this paper, we provide a construction for transforming nondeterministic Büchi automata into deterministic parity automata that cleanly explains the connections between the approaches of Safra and Muller-Schupp. It turns out that both types of constructions can be formulated on the same data structure, which can either be understood as ordered tuples of sets in which each set has an additional rank (a natural number), or as Safra trees in which each node has an additional rank (the same structure is essentially used in the constructions from [11] and [14]). The transitions are defined in terms of a sequence of simple operations, and it turns out that the two constructions only differ in one of these operations. In summary, our contributions are the following:

- We provide a new and relatively simple formulation of a Muller-Schupp style determinization construction that yields deterministic parity automata. Compared to previous constructions from [7] and [4], we encode less information in the states, and obtain a construction that has the same worst-case upper bound as the Safra style constructions.
- We extend our Muller-Schupp style construction by introducing a degree of freedom in the choice of the successor states. This freedom can be used to make the construction correspond to Safra’s construction as presented in [11] and [14]. We therefore obtain a construction that unifies the approaches of Safra and Muller-Schupp in one general construction. Furthermore, the freedom in the choice of the successors of transitions also yields new ways of obtaining deterministic parity automata, and can be used in implementations as a heuristic to reduce the state space of the resulting automaton.

This work is organized as follows. After introducing the basic notations in Section 2, we present the new variant of the Muller-Schupp construction in Section 3 and then briefly
review Safra’s construction in Section 4. We explain the structural relationship between those two constructions in Section 5 and finally introduce our generalized construction as a simple extension of the presented Muller-Schupp construction in Section 6. In Section 7 we discuss and conclude.

2 Preliminaries

First we briefly review basic definitions concerning $\omega$-automata and $\omega$-languages. If $\Sigma$ is a finite alphabet, then $\Sigma^\omega$ is the set of all infinite words $w = w_0w_1\ldots$ with $w_i \in \Sigma$. For $w \in \Sigma^\omega$ we denote by $w(i)$ the $i$-th symbol $w_i$. For convenience, we write $[n]$ for the set of natural numbers $\{1, \ldots, n\}$. A Büchi automaton $A$ is a tuple $(Q, \Sigma, \Delta, Q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $Q_0, F \subseteq Q$ are the sets of initial and accepting states, respectively. When $Q$ is understood and $X \subseteq Q$, then $\overline{X} := Q \setminus X$. We write $\Delta(p, x) := \{q \mid (p, x, q) \in \Delta\}$ to denote the set of successors of $p$ on symbol $x$ and $\Delta(F, x)$ for $\bigcup_{p \in F} \Delta(p, x)$. A run of an automaton on a word $w \in \Sigma^\omega$ is an infinite sequence of states $q_0, q_1, \ldots$ starting in some $q_0 \in Q_0$ such that $(q_i, w(i), q_{i+1}) \in \Delta$ for all $i \geq 0$. An automaton is deterministic if $|Q_0| = 1$ and $|\Delta(p, x)| \leq 1$ for all $p \in Q, x \in \Sigma$, and non-deterministic otherwise. In this work, we assume Büchi automata to be non-deterministic and refer to them as NBA. A transition-based deterministic parity automaton (TDPA) is a deterministic automaton $(Q, \Sigma, \Delta, Q_0, c)$ where instead of $F \subseteq Q$ there is a priority function $c: \Delta \to \mathbb{N}$ assigning a natural number to each transition.

A run of an NBA is accepting if it contains infinitely many accepting states. A run of a TDPA is accepting if the smallest priority that appears infinitely often on transitions along the run is even. An automaton $A$ accepts $w \in \Sigma^\omega$ if there exists an accepting run on $w$, and the language $L(A) \subseteq \Sigma^\omega$ recognized by $A$ is the set of all accepted words. To avoid confusion, we sometimes refer to states of TDPA that we construct as macrostates to distinguish them from the states of the underlying Büchi automaton.

3 A Simplified Muller-Schupp Construction

The essential idea for determinization using the Muller-Schupp construction is the following: given some Büchi automaton $A$ and input word $w$, the resulting deterministic automaton conceptually traverses a specific run-tree of $A$ on $w$, called reduced split-tree in [7], and tracks enough information to decide whether an infinite path with a specific shape exists in this tree. Such a path is known to exist if and only if $w$ is accepted by $A$. The construction presented in [4] uses a structure called contraction trees in order to track the relevant information. This has been simplified in [3] to macrostates that consist of an ordered tuple of disjoint sets of Büchi states, and two preorders over the states appearing in the tuple.

In this section, we further simplify the structure of the macrostates for the deterministic automaton to ordered tuples of disjoint sets of Büchi states, and a single additional linear order on these sets (formally expressed as a ranking function that assigns to each set a natural number). This also results in a relatively simple transition function on the macrostates.

The reduced split-tree $t^*(A, w)$ for NBA $A$ and word $w \in \Sigma^\omega$ is an ordered infinite tree in which the nodes are labelled by state-sets, and each node has at most two successors. Formally, it is constructed as follows. The first level of the tree consists of the root node labelled by the initial states $Q_0$. To construct level $i + 1$ from level $i$, for each node at level $i$ labelled by set $S$ of states, let the left child of $S$ be labelled by $\Delta(S, w(i)) \cap F$ and the right child by $\Delta(S, w(i)) \cap \overline{F}$, i.e., accepting and non-accepting successor states are separated.
Determination of Büchi Automata

Then keep only the leftmost (wrt. the natural ordering of neighbors) occurrence of each state in the level and finally remove nodes labelled by \( \emptyset \). Clearly, because of the normalization, the number of nodes on each level can be at most \(|Q|\). An example of a reduced split-tree is shown in Figure 1. We call an infinite path in the tree that takes the left branch infinitely often a left-path. Reduced split-trees have the following useful property:

\[ A := \begin{array}{c}
q_0 \xrightarrow{a} q_2 \\
q_1 \xrightarrow{a} \text{q1} \xrightarrow{a} \text{q1}
\end{array} \]

\[ t^{rs}(A, a^\omega) := \begin{array}{c}
\{q_0\} \\
\{q_1\} \{q_2\} \{q_0\} \\
\{q_1\} \{q_2\} \cdots \{q_0\}
\end{array} \]

\[ \begin{array}{c}
\text{Figure 1} \end{array} \]

Example of a reduced split-tree

Below, we formally define these operations (\( \text{step} \), \( \text{prune} \), and \( \text{normalize} \)), which are used to infer a left-path.

\[ \begin{array}{c}
\text{step} \end{array} \]

interprets \( t \) as nodes on a reduced split-tree level and calculates the next level sets,

\[ \begin{array}{c}
\text{prune} \end{array} \]

removes the empty sets produced by \( \text{step} \), reassigning ranks in a specific way, and

\[ \begin{array}{c}
\text{normalize} \end{array} \]

just turns the ranking function obtained after \( \text{prune} \) into a bijection again.

Below, we formally define these operations (\( \text{step} \) and \( \text{prune} \) are illustrated in Figure 2).

First we describe \( \text{step} \), which constructs the next level of the reduced split-tree and passes each existing rank on to the respective right child. Let

\[ \Delta_i(q, x) := \Delta(q, x) \setminus \Delta(\bigcup_{i=1}^{\text{idx}(q)-1} S_i, x), \]

restricting for each state \( q \in Q_t \) the successors to only those which are not reached by some other state located in a set to the left of \( q \). Then, for each node \( S_i \), let \( \hat{S}_{2i-1} := \Delta_{\bar{\alpha}}(S_i, x) \cap F \) be the left child and \( \hat{S}_{2i} := \Delta_{\bar{\alpha}}(S_i, x) \cap \bar{F} \) the right child, containing the accepting and non-accepting normalized successors, respectively. Let \( \bar{\alpha}(2i) := \alpha(i) \) and \( \bar{\alpha}(2i - 1) := n + 1 \), i.e., the right children inherit the rank of the parent and the left children all get the same new maximal rank \( n + 1 \), resulting in a pre-slice \((\bar{\alpha}, \hat{t})\).
\[ (\alpha, t) = (\hat{S}_1^{\alpha(1)}, \hat{S}_2^{\alpha(2)}, \ldots, \hat{S}_n^{\alpha(n)}) \]

\[ \text{step} \]
\[ (\hat{\alpha}, \hat{t}) = (\hat{S}_1^{n+1}, \hat{S}_2^{n+1}, \hat{S}_3^{n+1}, \ldots) \]

\[ \text{prune} \]
\[ (\hat{\alpha}, \hat{t}) = (\hat{S}_1^{\hat{\alpha}(1)}, \hat{S}_2^{\hat{\alpha}(2)}, \hat{S}_3^{\hat{\alpha}(3)}, \ldots) \]

**Figure 2** Abstract illustration of step and prune in a Muller-Schupp transition on some \( x \in \Sigma \). The superscripts represent the assigned ranks. First, step calculates the normalized successors, separating accepting from non-accepting states and passing the parent rank on to the right child. In the illustration, we assume that the sets \( \hat{S}_i \neq \emptyset \) for \( i \in \{2, 5, 2n-1\} \), i.e., \( x_1 = 2, x_2 = 5, x_3 = 2n-1 \). Then prune keeps sets at these positions for the resulting tuple \( \hat{t} \), and \( \hat{\alpha} \) is obtained by taking the minimum of the ranks given by \( \hat{\alpha} \) in the ranges spanning from one \( x_i \) up to the position before \( x_{i+1} \). Finally \( t' := \hat{t} \) and \( \hat{\alpha} \) is normalized to \( \alpha' \), while preserving strict ordering between positions wrt. \( \hat{\alpha} \).

The dotted edges connect parent sets (in the top row) and resulting left/right children sets (bottom row) in the conceptual reduced split-tree, the solid edges show the evolutions of the rank values assigned to the sets.

Intuitively, in the prune operation, all ranks that mark empty sets after step are relocated onto the closest non-empty set to the left (or removed, if no such set exists). When multiple ranks occupy the same set, then the smallest one is preserved. Ranks that moved to the left in this way and are not removed, indicate a good (green) event, whereas ranks which were removed indicate a bad (red) event.

Formally, let \( x_1 < x_2 < \ldots < x_n \) be the increasing sequence of all indices such that \( \hat{S}_{x_i} \neq \emptyset \). Then prune returns \( (\hat{\alpha}, \hat{t}) \) with the tuple \( \hat{t} := (\hat{S}_{x_1}, \ldots, \hat{S}_{x_n}) \) without empty sets, where \( \hat{\alpha} \) is defined as \( \hat{\alpha}(i) := \min \{ \hat{\alpha}(j) \mid x_i \leq j < x_{i+1} \} \) with \( x_{n+1} := |\hat{t}| + 1 \).

The set of green ranks is given by \( G := \text{img}(\hat{\alpha}) \cap \{ \hat{\alpha}(j) \mid \hat{S}_j = \emptyset \} \), where \( \text{img}(\hat{\alpha}) \) denotes the image of \( \hat{\alpha} \). These are the ranks that mark empty sets after step and are not removed by prune. The set of red ranks given by \( R := \text{img}(\hat{\alpha}) \setminus \text{img}(\hat{\alpha}) \) contains the ranks that were not preserved during prune. The set of active ranks is \( A := G \cup R \). Let \( k := \min A \) (or \( k := |Q| + 1 \) if \( A = \emptyset \)) denote the dominating rank of the transition, i.e., the smallest active rank. We define the priority \( p \) of the transition as \( 2k \) if \( k \in G \) and \( 2k - 1 \) otherwise.

The function \( \hat{\alpha} \) might assign the same rank to several sets, and it might have gaps (unused rank values between used ones). So finally, normalize returns \( (\alpha', t') \) with \( t' := \hat{t} \) and a final bijective ranking function \( \alpha' : [t'] \to [t'] \) such that \( \hat{\alpha}(j) < \hat{\alpha}(k) \Rightarrow \alpha'(j) < \alpha'(k) \) for all \( j, k \in \{1, \ldots, |t'|\} \), i.e., a total order which is compatible with the preorder induced by \( \hat{\alpha} \). If there are several such ranking functions \( \alpha' \), then any of these works.

A TDPA \( B \) is obtained by taking the initial state \( (\alpha_0, t_0) \) with \( t_0 := (Q_0), \alpha_0(1) := 1 \) and a transition function that picks for each state a valid successor that satisfies the description above, and assigns the corresponding priority \( p \) to the edge. Observe that by construction, the sequence of states visited along some word \( w \in \Sigma^* \) from the initial state represents exactly the levels of \( t'^*(A, w) \), marked with ranks.

**Theorem 2.** For a given NBA with \( n \) states, the TDPA obtained by the Muller-Schupp construction accepts the same language as the NBA, and its number of states is in \( O(n^2) \).
Figure 3 Example of a Safra-tree transition on letter $a$, based on NBA $\mathcal{A}$. The LIR position of nodes is depicted as superscript of the sets. The “redundant” states that are implicit in our definition are depicted in gray in the initial and resulting tree. In the intermediate step, the tree is depicted after calculating and pruning successor state sets. In the final tree the remaining actions are performed and LIR positions are updated. The transition has a red event for LIR position 2 and a green event for position 3. Because of the removal of the node at position 2 in the LIR, the node that originally was at position 3 moved up, whereas the fresh node labelled by $\{q_3\}$ comes last.

The correctness follows from the correctness of the generalized construction presented in Section 6. The claim on the state complexity directly follows from the upper bound given in [14, Proposition 2], and the bijection between the set of ranked slices and the set of ranked Safra trees presented in Section 5.

4 Sketch of the Safra Construction

In this section, we roughly illustrate the used structures and operations of the Safra construction along the lines of [11, 14], so that we can demonstrate its relationship with the Muller-Schupp construction in the next section. As before, $\mathcal{A}$ is an NBA with the usual components.

A Safra tree is a finite ordered tree with non-empty state-sets as labels. Usually, it is required that a parent is labelled by a strict superset of all states in its subtree and siblings are labelled by pairwise disjoint sets. We use the equivalent requirement that all labels in the tree are pairwise disjoint, i.e., refrain from listing states in the parent label which are already present in some descendant. One can easily reconstruct the “full” label set of a node wrt. the classical definition by taking the union of all the labels in its subtree. To obtain parity automata, each node of the Safra tree is associated with a number from $\{1, \ldots, n\}$, where $n$ is the number of nodes in the Safra tree [11]. These numbers satisfy the property that parent nodes have smaller numbers than their children, and a node has a smaller number than its right sibling. The numbers correspond to the ranks that we use in Section 3, and we therefore refer to Safra trees in combination with these numbers as ranked Safra trees. Two ranked Safra trees are shown in Figure 3 (and an intermediate tree in the middle).

In [14], a slightly different representation is used based on a later introduction record (LIR), which just lists the tree nodes in their introduction order, i.e., nodes appear in this list after parents and older siblings (in this representation, nodes have canonical names depending only on their position in the tree). Safra trees with LIR directly correspond to ranked Safra trees by annotating each tree node with its position in the LIR.

A transition on symbol $x \in \Sigma$ is constructed as follows (see Figure 3 for an example). First, for each label set $S$, the set $S' := \Delta(S, x)$ of successor states is calculated. After this, each node gets a fresh right-most child, and the accepting states in $S'$, that is $S' \cap F$, are moved into the label of this child. Then, disjointness is ensured by keeping of each state only the copy which is located at the deepest node along the leftmost branch where that state occurs (this stage is represented by the middle tree in Figure 3). If now some internal node has an empty label, but a non-empty subtree (a good event for the node), its subtree
is collapsed into a single node by removing all descendants and moving the states in their labels into the parent label. Finally, all remaining sets that are labelled by \( \emptyset \) are removed (being removed is a bad event for a node). In the following, we refer to good and bad events as green and red, respectively. The priority for the transition is derived from the green and red events, which are associated with the relative position of the corresponding nodes in the LIR. The LIR for the new tree is obtained by deleting removed nodes from the LIR and appending fresh nodes that remain in the resulting tree in arbitrary order.

### 5 From Safra-trees to ranked slices and back

In this section we state the key observation that was the starting point of this work: there is a bijection between the set of ranked slices and the set of ranked Safra trees. From a ranked Safra tree, one obtains the ranked slice by simply listing the nodes of the Safra tree by a depth-first post-order traversal (i.e., a parent processed after all its children). We formalize this relationship below, and then explain that the transitions defined in the Muller-Schupp construction and in the Safra construction are very similar, which then leads to the unified construction.

Let \((\alpha, t)\) be a ranked slice with \(t = (S_1, \ldots, S_n)\). The tuple index of the parent of \(S_i\) is the closest index to the right of \(i\) that has a smaller rank and is formally defined as \(\uparrow(i) := \min_{1 \leq k \leq n} \{k \mid \alpha(k) < \alpha(i)\}\). As we require by definition of ranked slices that the rightmost position in the tuple always has rank 1, this is the only position in the tuple for which the parent is undefined. The ordered tree induced by \(\uparrow\), with siblings in tuple index order, is called the rank tree of \((\alpha, t)\). The tuple index of the left subtree boundary of \(S_i\) is the closest index to the left with a smaller rank, and is denoted by \(\leftarrow(i) := \max_{1 \leq k < i} \{k \mid \alpha(k) < \alpha(i)\}\) or 0 if no such index exists. It points to either the direct left sibling of \(i\), or the left sibling of the closest ancestor, if one exists. Effectively, \(\leftarrow(i)\) is the closest neighbor to the left which is not a descendant of \(i\). As children by definition are always to the left of their parents, every node at indices \(\leftarrow(i) + 1, \ldots, i\) is in the subtree of \(i\).

For an example, consider the tuple \(\{(q_3)^4, (q_1)^2, (q_2)^3, (q_0)^1\}\), where the superscripts denote the assigned rank (e.g., \(\alpha(1) = 4\)). The rightmost position 4 of the tuple is the root of the tree. For the positions 2 and 3, which have rank 2 and 3 respectively, the next position to the right with a smaller rank is in both cases position 4, i.e., \(\uparrow(2) = \uparrow(3) = 4\). Finally, position 1 in the tuple has position 2 as parent, i.e., \(\uparrow(1) = 2\). The discussed tuple is depicted with the parent edges at the bottom right of Figure 4. There is also one non-trivial left subtree boundary in this tuple, assigned by \(\leftarrow(3) = 2\), i.e. index 2 is not in the subtree of index 3, and in this case is an actual left sibling of index 3.

We use the notation \(\uparrow_\alpha := \alpha \circ \uparrow \circ \alpha^{-1}\) to denote the parent rank of another rank directly, without mentioning the indices in the tuple. In the previous example, we have \(\uparrow_\alpha(4) = 2\), and \(\uparrow_\alpha(2) = \uparrow_\alpha(3) = 1\). We identify the age-ranks \(\alpha(i)\) as nodes of the tree, while each set \(S_i\) determines the label of the node \(\alpha(i)\), called hosted set. We write \(S_i^t := \bigcup_{k = \leftarrow(i) + 1} S_k\) for the subtree set of node \(\alpha(i)\).

**Definition 3.** Let safra2slice be the mapping which takes a ranked Safra tree and returns \((\alpha, t)\), with \(t := (S_1, \ldots, S_n)\) being the label sets of the nodes in depth-first post-order traversal (i.e., a parent processed after all its children) traversal order and ranking \(\alpha\) defined by the ranks of the corresponding Safra tree nodes.

Let slice2safra be the mapping which takes a ranked slice \((\alpha, t)\) and returns the ranked Safra tree given by the rank tree of \((\alpha, t)\), i.e. the tree structure defined by \(\uparrow\) and the ordering of siblings given by the order of the corresponding sets in \(t\).
Determination of Büchi Automata

\[ \mathcal{A} : \] ranked Safra tree sequence:
\[
\begin{array}{cccccccc}
\{q_0\} & \{q_1\} & \{q_2\} & \{q_3\} & \{q_4\} & \{q_5\} & \{q_6\} & \{q_7\} \\
\{a\} & \{b\} & \{c\} & \{a\} & \{c\} & \{a\} & \{c\} & \{a\} \\
b, c \rightarrow & a, b & & & & & & \\
\end{array}
\]

Muller-Schupp sequence of ranked slices:
\[
\begin{array}{cccccccc}
\{q_1\} & \{q_2\} & \{q_3\} & \{q_4\} & \{q_5\} & \{q_6\} & \{q_7\} \\\n\{a\} & \{b\} & \{c\} & \{a\} & \{c\} & \{a\} & \{c\} \\
\end{array}
\]

During last transition, after step:

\[ \text{Safra} \]
\[ \text{Muller-Schupp} \]
\[ (\{q_1, q_2\}^2 \{q_3\}^3 \{q_4\}^1, \{q_5\}^2 \{q_6\}^2 \{q_7\}^2 \{q_8\}^2 \{q_9\}^1) \]

\[ \text{Safra2slice} \]
\[ \text{Slice2safra} \]

\[ (\{q_1\}^3 \{q_2\}^3 \{q_3\}^2 \{q_4\}^1, \{q_5\}^2 \{q_6\}^2 \{q_7\}^2 \{q_8\}^2 \{q_9\}^1) \]

**Figure 4** Transitions based on NBA \( \mathcal{A} \) using both constructions. The superscripts denote the ranks of tree nodes / sets in the slice tuple. The subscripts are added for illustration purposes and conceptually track nodes throughout time, i.e., the same symbol marks the “same” node at different times. The algorithms agree on all but the last transition, where they differ due to different handling of green nodes/ranks, in this case rank \( 2 \) that marks an empty set after calculating and splitting the successors (illustrated on the bottom right). In the Muller-Schupp case, the rank is moved left during prune, resulting in a child being pulled into the parent in the rank tree, whereas in the Safra construction the whole subtree is collapsed. The solid edges between sets depict the rank tree of the resulting new slice. The normalization steps that make the successor sets pairwise disjoint also yield the same results. The ranks of nodes with green events in the Safra construction coincide with ranks of sets that signal green in the ranked slices, and ranks of Safra nodes with red events with ranks of sets that signal red. The removal of empty sets by prune and renumbering the ranks with normalize is the same as the removal of the corresponding nodes in the Safra tree and updating the LIR, i.e., the ranks of Safra nodes.

It is easy to see from the definitions that safra2slice and slice2safra are injective and return a valid ranked slice and ranked Safra tree, respectively. This implies that there exists a bijection between the sets of ranked Safra trees and ranked slices. It is not also very hard to see that the following holds (a proof can be found in Appendix A):

**Lemma 4.** safra2slice and slice2safra are inverses of each other and provide a bijection between ranked Safra trees and ranked slices.

As we have established that both constructions, Muller-Schupp and Safra, operate on essentially the same structures, from now on we talk about ranked slices and trees interchangeably. Using this relationship, one can take the same tree/slice and apply both the successor calculation of the Safra construction and of the Muller-Schupp construction to it. What one first notices, is that the resulting tree/slice is very similar or equal in many cases. This is owed to the fact that most operations in one construction have an equivalent operation in the other, just formulated for the other representation.

For example, moving accepting successor states into a fresh child node in Safra’s construction corresponds to splitting accepting successors from non-accepting ones during step in the Muller-Schupp construction, as in the successor tuple the new child (in the conceptual split-tree) gets a fresh, larger rank and by definition becomes the rightmost child in the rank tree of the resulting new slice. The normalization steps that make the successor sets pairwise disjoint also yield the same results. The ranks of nodes with green events in the Safra construction coincide with ranks of sets that signal green in the ranked slices, and ranks of Safra nodes with red events with ranks of sets that signal red. The removal of empty sets by prune and renumbering the ranks with normalize is the same as the removal of the corresponding nodes in the Safra tree and updating the LIR, i.e., the ranks of Safra nodes.
In fact, the only difference between the constructions is what happens with a tree node in case of a green event. Recall that in Safra’s construction, the whole subtree of a green node is collapsed to a single node. In the Muller-Schupp construction, the green ranks are those that end up on an empty set after step, and that survive the prune operation, in which the ranks are moved to the next non-empty set to the left, and only the minimal ones are kept on each non-empty set. In the view of ranked trees, this corresponds to a green node absorbing its rightmost, uppermost child node into it, while keeping the rest of the subtree unchanged. See Figure 4 for an illustration.

After observing that both constructions differ in only a minor step and noticing that both yield correct (but possibly different) automata, it becomes apparent that the exact step performed for green events is not essential and there must be a more general mechanism to uncover. The construction we present in Section 6 results from this line of thought.

On the practical side, it is worth mentioning that the cost of switching between the representations using the presented bijection is negligible—the traversal of a ranked Safra tree to obtain a ranked slice is obviously possible in linear time. For the other direction there also exists a simple linear time algorithm (presented in Appendix B) that calculates the parent and left subtree boundary relation from the ranking $\alpha$.

## 6 The unified construction

In this section, we present a construction that builds on the Muller-Schupp construction from Section 3, and unifies it with Safra’s construction by adding another operation, called merge, between prune and normalize: $((\alpha, t)) \xrightarrow{\text{step}} ((\tilde{\alpha}, \tilde{t})) \xrightarrow{\text{prune}} ((\bar{\alpha}, \bar{t})) \xrightarrow{\text{merge}} ((\tilde{\alpha}, \tilde{t}) \xrightarrow{\text{normalize}} (\alpha', t'))$. This new operation is nondeterministic, and can be instantiated in different ways. In particular, it can be instantiated trivially and thus corresponds to the Muller-Schupp construction, and it can be used to emulate the Safra construction.

We first describe the idea of merge, and then give a formal definition. Assume that, after step and prune have been applied to some ranked slice $(\alpha, t)$, we have the pre-slice $(\tilde{\alpha}, \tilde{t})$, and the dominating (minimal active) rank $k$ (determined by prune, see Section 3). Then merge can collapse groups of neighbouring sets in the tuple, and preserves the minimum rank from each collapsed range, similar to prune. In contrast to prune, which “merges” one non-empty set with multiple empty sets in a deterministic manner, merge may actually take the union of multiple adjacent non-empty sets, depending on the ranks currently assigned to them.

The non-overlapping intervals of sets that are collapsed together are not uniquely determined in general. They only have to satisfy the constraints that no sets with rank smaller than the dominating rank $k$ are merged with anything else, and that the set with rank $k$ is not merged with anything to the right of it. These constraints are important for the correctness, and ensure that in the ranked Safra tree perspective, the nodes with rank smaller than $k$ do not change, and that the node with the dominating rank $k$ is not merged with sets outside of its subtree.

Formally, merge returns a pre-slice $(\tilde{\alpha}, \tilde{t})$ obtained in the following way (see Figure 5 for an illustration). Let $I_1, I_2, \ldots, I_{n'}$ be a sequence of sets partitioning the set of indices $\{1, \ldots, n\}$ in $\tilde{t}$ into adjacent groups, i.e., $\min I_1 = 1$, $\max I_{n'} = n$ and for all $j > 1$ we have $\min I_j = \max I_{j-1} + 1$. This grouping should satisfy the following property for all $1 \leq j \leq n'$ and $l \in I_j$: if $\tilde{\alpha}(l) < k$, then $|I_j| = 1$, and if $\tilde{\alpha}(l) = k$, then $\max I_j = l$. Then the pre-slice $(\tilde{\alpha}, \tilde{t})$ is defined by the sets $\tilde{S}_i := \bigcup_{j \in I_i} \tilde{S}_j$ and the ranking function $\tilde{\alpha}(i) := \min \{\tilde{\alpha}(j) \mid j \in I_i\}$ for all $i \in \{1, \ldots, n'\}$, i.e., for each interval, the union of the sets and the smallest rank is taken.
Determinization of Büchi Automata

\[(\hat{\alpha}, \hat{t}) = (S_1 > k S_2 > k S_3 > k S_4 > k S_5 > k S_6 > k S_7 > k S_8 > k)\]

\[(\hat{\alpha}, \hat{t}) = (\bigcup_{\min} \bigcup_{\min} \hat{S}_1 \hat{S}_2 \hat{S}_3 \hat{S}_4 \hat{S}_5 \hat{S}_6 \hat{S}_7 \hat{S}_8)\]

\[I_1 = \{1, 2\} \quad I_2 = \{3\} \quad I_3 = \{4, 5, 6\} \quad I_4 = \{7\} \quad I_5 = \{8\}\]

\[\forall i \in [n'], l \in I_i : \hat{\alpha}(l) \in G \implies \hat{\alpha}(l) + 1 \in I_i \quad (\text{complete subtrees collapsed})\]

\[\text{Figure 5: Illustration of the general merge operation that comes after prune and before normalize, with the minimal active rank } k \text{ and ranks depicted as set superscripts. The illustrated intervals are the coarsest partitioning of indices in } \hat{t} \text{ satisfying the constraints.}\]

As in the Muller-Schupp construction, normalize is applied to \((\hat{\alpha}, \hat{t})\) to obtain the successor macrostate \((\alpha', t')\). This extended transition relation is used to obtain the transition-based deterministic parity automaton, as before.

An example showing how the choice of different merge strategies leads to different successor states is illustrated in Figure 5. Observe that we can recover the Muller-Schupp construction by using the identity function for merge, or in other words, putting each index into its own interval, which is the finest partitioning of indices that satisfies the requirements on merge. On the other hand, we can also take the coarsest compatible partitioning, i.e., minimize the number of intervals. We call this kind of update maximal collapse.

We can emulate a Safra-update by imposing some additional constraints on the intervals, ensuring that only the complete subtrees of nodes with green ranks are merged. More concretely, we require that intervals that are not singletons span exactly the nodes of the complete subtree that is rooted in a green rank in the view of the slice as ranked Safra tree.

Note that for an index \(l\) in the tuple, the subtree of the corresponding node in the ranked Safra tree corresponds to the interval that starts one step right of the left subtree boundary of \(l\), and ends in \(l\), that is, the interval \(l' + 1, \ldots, l\). Thus, for imitating the Safra merge rule, the intervals \(I_1, I_2, \ldots, I_{n'}\) from merge are the unique smallest intervals satisfying

\[\forall i \in [n'], l \in I_i : \hat{\alpha}(l) \in G \implies \hat{\alpha}(l') + 1 \in I_i \quad (\text{complete subtrees collapsed})\]

\[\text{Proposition 5. The operation merge can be instantiated such that the transitions of the constructed TDPA correspond to the transitions of the Muller-Schupp construction or to the transitions of the Safra construction.}\]

Notice that for all merge rules except for the Muller-Schupp update, the relationship of ranked slices and consecutive levels of the reduced split-tree (see Section 3) breaks down. One can, however, reflect the merges also in the reduced split-tree by doing the merges of the corresponding sets on each level, which leads to an acyclic graph instead of a tree. This view is helpful in the correctness proof of the construction.

\[\text{Theorem 6. Let } A \text{ be an NBA. Then a deterministic parity automaton } B, \text{ obtained by the described determinization construction, has at most } O(n^2) \text{ states and recognizes the same language as } A.\]

The upper bound holds because the same macrostates are used as in the presented Muller-Schupp construction in Section 3. The correctness can be shown by a refinement of the original correctness proof of the Safra construction [13], and is given in Appendix C.
Figure 6. \( \mathcal{A} \) illustrates the relevant part of an NBA during a transition on some symbol \( x \in \Sigma \), that is, the arrows correspond to the \( x \)-transitions of \( \mathcal{A} \). The gray edges are the ones pruned in the reduced transition relation \( \Delta_r \). The current macrostate \((\alpha, t)\) is represented as the rank tree to the right of \( \mathcal{A} \), and as ranked slice below \( \mathcal{A} \). The step and prune operations (see Fig. 2 for details) result in ranks 1, 3 and 4 being passed down along the right child. Ranks 2 and 6 are moved to the left and hence are green. Rank 5 is overwritten by 2 and hence is red. Rank 7 is a fresh rank which is larger than the others. The dominating rank \( k \) is 2. The choice of different merge intervals (as shown in Fig. 3) results in different successors. The successors for the three discussed variants, Muller-Schupp, Safra, and maximal collapse, are shown as rank trees on the right. The 5 other permitted successors are depicted at the bottom.
We have presented a new variant of the Muller-Schupp construction for determinization of Büchi automata into parity automata, reducing the information stored in the macrostates to ordered tuples of disjoint sets annotated with ranks. These ranked slices are in bijection with ranked Safra trees, which leads to a general construction that can emulate the Muller-Schupp construction and the Safra-construction. This answers, in some sense, the question from [4] on the relation between the two types of constructions.

In general, one can obtain many different valid deterministic automata by choosing different deterministic transition functions that are compatible with the described successor relation. One can also imagine this as constructing a non-deterministic automaton with all permitted successors, and then pruning the edges arbitrarily, while preserving for each state only one outgoing transition for each symbol, to “carve” out a valid deterministic automaton.

This non-determinism comes from two sources. One degree of freedom in our construction comes from the different ways of assigning ranks (to new nodes, and when closing gaps resulting from deleted ranks). This freedom is already mentioned in [14]. But here the flexibility is just in the choice of the specific permutation, which still describes structurally the same tree in any case. The novel and in our opinion powerful degree of freedom in our construction is the possibility for different valid merge operations, which allows for a vastly larger pool of possible successors, as the results may describe structurally different trees. Furthermore, the smaller the smallest active rank, the more different a permitted successor may look like.

We have explicitly mentioned the merge strategies that lead to the Muller-Schupp and Safra constructions, and also have mentioned a third strategy, the maximal collapse rule that merges as many sets as possible (as shown in e.g. Figure 6). We also want to point out that, while fixing one such merge-rule for the whole construction is the simplest implementation, the construction permits using any valid successor without the need to disambiguate the merge operation beforehand, i.e., picking the successor of a state from the set of permitted ones is a local choice. One may think of schemes where the successor is chosen dynamically, depending on the input or already computed information. For example, one can check whether a valid successor has already been constructed, and only add a new state according to a fixed policy if this is not the case. We have already implemented a prototype making use of such an optimization (among others) with encouraging results.

We also want to point out that the presented construction works equally well with transition-based Büchi automata as input, in which case the step operation separates states which are reached by at least one accepting transition from those that are not. One can easily verify that this does not impact the reasoning in the proofs.

It is also possible to adapt the construction to yield Rabin automata, such that the corresponding Safra construction as presented in [14] is subsumed. In this setting, however, the presentation of macrostates as ordered tuples of sets is less natural. Furthermore, in this setting the merges of sets needs to be restricted to subtrees of green nodes, because there is no total order of importance of nodes as provided by the ranks.

References

1. Christel Baier and Joost-Pieter Katoen. Principles of model checking. MIT Press, 2008.
2. J Richard Büchi. On a decision method in restricted second order arithmetic. In Studies in Logic and the Foundations of Mathematics, volume 44, pages 1–11. Elsevier, 1966.
3 Thomas Colcombet and Konrad Zdanowski. A tight lower bound for determination of transition labeled Büchi automata. In ICALP 2009, volume 5556 of Lecture Notes in Computer Science, pages 151–162. Springer, 2009.

4 Seth Fogarty, Orna Kupferman, Moshe Y. Vardi, and Thomas Wilke. Profile trees for Büchi word automata, with application to determinization. Information and Computation, 245:136–151, 2015.

5 Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings, pages 53–65, 2001. doi:10.1007/3-540-44585-4_6

6 Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA, pages 412–416, 2001. doi:10.1109/ASE.2001.969841.

7 Detlef Kähler and Thomas Wilke. Complementation, disambiguation, and determinization of Büchi automata unified. In ICALP 2008, pages 724–735. Springer, 2008.

8 Robert McNaughton. Testing and generating infinite sequences by a finite automaton. Information and control, 9(5):521–530, 1966.

9 Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. Strix: Explicit reactive synthesis strikes back! In Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I, pages 578–586, 2018. doi:10.1007/978-3-319-96145-3_31

10 David E. Muller and Paul E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. Theoretical Computer Science, 141(1-2):69–107, 1995.

11 Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In Logic in Computer Science, 2006 21st Annual IEEE Symposium on, pages 255–264. IEEE, 2006.

12 Roman R. Redziejowski. An improved construction of deterministic omega-automaton using derivatives. Fundamenta Informaticae, 119(3-4):393–406, 2012.

13 Shmuel Safra. On the complexity of omega-automata. In 29th Annual Symposium on Foundations of Computer Science, 1988., pages 319–327. IEEE, 1988.

14 Sven Schewe. Tighter bounds for the determinisation of Büchi automata. In FOSSACS, pages 167–181. Springer, 2009.

15 Fabio Somensi and Roderick Bloem. Efficient Büchi automata from LTL formulae. In Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings, pages 248–263, 2000. doi:10.1007/10722167_21

16 Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, Handbook of Formal Languages, Vol. 3, pages 389–455. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

17 Wolfgang Thomas. Church’s problem and a tour through automata theory. In Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, pages 635–655. Springer, 2008. doi:10.1007/978-3-540-78127-1

18 Moshe Y. Vardi and Thomas Wilke. Automata: from logics to algorithms. In Logic and automata - history and perspectives, volume 2 of Texts in Logic and Games, pages 629–724. Amsterdam University Press, 2007.
Proof of the bijection between ranked Safra trees and ranked slices

In this section we provide the missing proof for Lemma 4. We make use of the following technical lemma to omit one direction of the proof, using the property that both functions are already known to be injective:

Lemma 7. Let $f : A \rightarrow B, g : B \rightarrow A$ be two injective functions and $g \circ f = \text{id}_A$. Then it follows that $f$ and $g$ are inverse mappings of a bijection.

Proof. From injectivity of $g$ follows that $g$ has a left inverse $g_{li}$. Hence:

$$g^{li} \circ g = \text{id}_B \Rightarrow g^{li} \circ g \circ f \circ g \circ f = \text{id}_B \circ f \circ g \circ f \Rightarrow g^{li} = f$$

As both are left inverses of each other, they are also right inverses of each other by definition and therefore $f^{-1} = g$ and $g^{-1} = f$. ▶

Now we can show the actual statement:

Lemma 4. $\text{safra2slice}$ and $\text{slice2safra}$ are inverses of each other and provide a bijection between ranked Safra trees and ranked slices.

Proof. As both mappings are injective, by Lemma 7 it suffices to argue that $\text{slice2safra} \circ \text{safra2slice}$ returns the original ranked Safra tree to prove the statement.

Take a ranked Safra tree with $n$ nodes. Recall that a parent node always has a smaller rank than its children, and siblings to the right of a node have larger ranks than that node.

Construct the ranked slice $(\alpha, t)$ using $\text{safra2slice}$ and then consider its rank tree. Clearly, both trees have the same number of nodes and the same set of labels. Also, nodes with the same label also have the same rank. What remains to be shown is the equality of the tree structure. Let $\text{node}(S_i)$ denote the tree node in the original ranked Safra tree that is labelled by set $S_i$ from $t$.

Notice that by the visiting order of the post-order traversal it is ensured that parent nodes, which have smaller assigned ranks, appear to the right of all their descendants in the tuple and thus closer ancestors of a node are listed earlier than further ancestors. Thus, the set $S_j$ of the parent $\text{node}(S_j)$ of some node $\text{node}(S_i)$ will be the closest set to the right of the set $S_i$ in the tuple $t$ which has a smaller rank. Hence, by definition of $\uparrow$, the original tree is reconstructed. For the ordering of siblings, observe that if $\text{node}(S_i)$ is the left sibling of $\text{node}(S_j)$, then $\text{node}(S_j)$ is visited only after completing the subtree of $\text{node}(S_i)$ and hence $S_j$ appears later in the tuple. Hence, the rank tree obtained via $\uparrow$ from the ranked slice is exactly the original ranked Safra tree. ▶
B  From ranked slices to rank trees in linear time

It is easy to see that going from a ranked Safra tree to a ranked slice is possible in linear time, as \texttt{safra2slice} is just a depth-first traversal. In this section we show how to efficiently compute \texttt{slice2safra}, i.e., how the rank tree can be obtained in linear time from a ranked slice, which can be useful in implementations of the presented determinization construction.

The parent and left subtree boundary relationships, which capture the tree structure of the ranked slice, can be computed using the following algorithm:

\begin{algorithm}
\SetKwFunction{unflatten}{\texttt{unflatten}}
\SetKwProg{function}{function}{}{\end{function}}
\SetKwFunction{new}{\texttt{new}}
\SetKwFunction{array}{\texttt{array}}
\SetKwFunction{stack}{\texttt{stack}}
\SetKwFunction{push}{\texttt{push}}
\SetKwFunction{top}{\texttt{top}}
\SetKwFunction{pop}{\texttt{pop}}
\SetKwFunction{new}{\texttt{new}}
\SetKwFunction{array}{\texttt{array}}
\SetKwFunction{stack}{\texttt{stack}}
\SetKwFunction{push}{\texttt{push}}
\SetKwFunction{top}{\texttt{top}}
\SetKwFunction{pop}{\texttt{pop}}
\Function{unflatten($\alpha$, $n$)}{\textbf{ranking} $\alpha$ : \texttt{n} $\rightarrow$ \texttt{n}, \texttt{n} sets in tuple}
\SetKwData{unflattened}{P}
\SetKwData{leftSubtree}{L}
\SetKwData{parentBound}{S}
\SetKwFunction{mainLoop}{main loop}
\SetKwFunction{innerLoop}{inner loop}
\SetKwFunction{cleanUpLoop}{clean-up loop}
\SetKwData{loopIndex}{i}
\SetKwData{mainLoopIndex}{i}
\SetKwData{sliceIndex}{i}
\SetKwData{topIndex}{i}
\SetKwData{parentIndex}{i}
\SetKwData{leftSubtreeIndex}{i}
\SetKwData{currentSet}{S}
\SetKwData{currentRank}{rank($\alpha$, $n$)}
\SetKwData{currentIndex}{i}
\SetKwData{leftSubtreeBoundary}{L($\alpha$, $n$)}
\SetKwData{parentBoundaries}{P($\alpha$, $n$)}
\SetKwData{sliceStart}{1}
\SetKwData{sliceEnd}{n}
\SetKwData{currentSetIndex}{1}
\SetKwData{currentIndex}{1}
\For{$i := n$ down to $1$}{\mainLoop}
\While{$S$ not empty and $\alpha_i < \alpha_{\top(S)}$}{\innerLoop}
\LoopIndex := $i$, \pop{(S)}
\parentBoundaries($i$) := \top{(S)}, \pop{(S)}
\For{$S$ not empty}{\cleanUpLoop}
L($\top{(S)}$) := 0, \pop{(S)}
\Return{($P$, $L$)}
\end{algorithm}

\begin{lemma}
\texttt{unflatten} calculates the rank tree from a ranked slice $(\alpha, t)$ in linear time.
\end{lemma}

\begin{proof}
Let $(\alpha, t)$ be a ranked slice with $t = (S_1, \ldots, S_n)$. The main loop is repeated exactly $n$ times. In each iteration one index is pushed onto the stack. All other loops are iterated at most $n$ times in total, as they require a non-empty stack and pop an element in each iteration. Hence, the algorithm completes in linear time. It remains to be shown that the resulting arrays $P$ and $L$ correspond to $\uparrow$ and $\leftarrow$.

In the first iteration of the main loop the stack is empty and hence only the index of the last set $S_n$ with rank 1 (i.e., the root) is pushed. This index is never popped from the stack during the main loop, as it has the smallest rank, and in the last iteration of the clean-up loop is assigned 0 as left subtree boundary index and has an undefined parent. Therefore, the root is treated correctly.

Next we analyze under which conditions the assignments to $P$ and $L$ can be incorrect and show that they lead to a contradiction. Let $j$ and $k$ with $1 \leq j < k \leq n$ be two indices. Notice that index $k$ is pushed onto the stack before $j$ by the main loop.

First consider the case that $\uparrow(j) = k$.

For contradiction, assume that $k$ is popped from the stack before the loop iteration with $i = j$ (i referring to the variable in the algorithm). This implies that there is an index $l$ with $j < l < k$, $\alpha(l) < \alpha(k)$ and $k$ being on top of the stack in the loop iteration with $i = l$. But as $\alpha(j) > \alpha(l) > \alpha(k)$ violates the assumption that $\uparrow(j) = k$ by definition of $\uparrow$, because then $l$ should be the parent of $j$.

Next, assume that $k$ is not on top of the stack in the iteration with $i = j$. Then there is an $l$ with $j < l < k$ and $\alpha(l) > \alpha(k)$ which is pushed onto the stack and stays there until $P(j)$ is assigned. As $\alpha(l) > \alpha(j)$, $j$ must be in a different subtree to the left of $l$. But then index $l$ must be removed in the inner loop during the iteration $i = j$, before $P(j)$ is assigned, which is a contradiction. Hence, we conclude that the parent array is assigned correctly.

Now, consider the case that $j = \leftarrow(k)$.
First assume that \( j = 0 \). Then there is no index with a smaller rank to the left of \( k \) and hence the inner loop was not entered whenever \( k \) was on top of the stack, so that \( k \) remained on the stack during the main loop. But then \( k \) will be assigned 0 as left subtree boundary in the clean-up loop.

Now assume that \( j > 0 \), i.e., there exists a non-trivial left subtree boundary. Again, assume for contradiction that \( k \) is popped before loop iteration with \( i = j \). Then there was an index \( l \) with \( j < l < k \) such that \( k \) was on top of the stack and \( \alpha(l) < \alpha(k) \), so that \( k \) was popped from the stack in the inner loop. By definition of \( \alpha \) this implies that \( l \) is the left subtree boundary of \( k \), violating the assumption.

Finally, assume that \( k \) is not on top of the stack when \( i = j \). Then some index \( l \) with \( j < l < k \) was pushed onto the stack with \( \alpha(l) > \alpha(k) \) and in the main loop iteration with \( i = j \) the last such index \( l \) was assigned as parent of \( j \), because \( \alpha(j) > \alpha(l) \) (implied by the fact that \( l \) remained on the stack). But then \( \alpha(j) > \alpha(k) \), contradicting the assumption that \( j \) is the left subtree boundary index of \( k \), which must have a smaller rank. Hence we conclude that the left subtree boundary must also be assigned correctly, completing the proof.
C. Löding and A. Pirogov

C Correctness proof for the determinization construction

In this section, we provide the proof for the main result:

▶ Theorem 6. Let \( A \) be an NBA. Then a deterministic parity automaton \( B \), obtained by the described determinization construction, has at most \( O(n!^2) \) states and recognizes the same language as \( A \).

As discussed, the claim on the state complexity directly follows from the upper bound given in [14, Proposition 2], and the bijection between the set of ranked slices and the set of ranked Safra trees presented in Section 5.

Hence, it remains to be shown that an automaton \( B \) with transitions that satisfy the described successor relation of the determinization construction presented in Section 6 indeed recognizes the same language as \( A \).

For the rest of this section, fix some arbitrary word \( w \in \Sigma^* \), NBA \( A \) and let \( B \) be the obtained TDPA.

C.1 From an accepting NBA run to an accepting TDPA run

To show that \( B \) accepts \( w \) if there exists an accepting run in \( A \), we introduce the concept of the rank-profile of a state in a pre-slice \( (\alpha, t) \), which is just the sequence of nodes in the corresponding ranked Safra tree starting from the root, down to the node labelled by the set containing the state. The sequence is strictly ascending, because children have larger ranks than their parents. Formally, the rank-profile \( \text{rp} : Q_t \rightarrow [n]^* \) maps each state \( q \) to a sequence \( a_1a_2 \ldots a_m \) such that \( a_1 = \alpha(n) = 1 \), \( a_m = \alpha(q) \), and \( a_{i-1} = \uparrow_{\alpha}(a_i) \) is the rank of the parent of the node with rank \( a_i \), for all \( i > 1 \). Consider, for example, the ranked slice \((\{q_1\}^1,\{q_1\}^2,\{q_2\}^3,\{q_0\}^1)\). The rank profile of \( q_3 \) is 1, 2, 4, and the rank profile of \( q_2 \) is 1, 3. Notice that rank-profiles have at most the length \( |Q| \) and are fully determined by the ascending set of ranks, so we can interpret the profile also as a set of ranks.

Let \( a = a_1 \ldots a_m, b = b_1 \ldots b_n \) be two rank profiles with \( m := \min(n_a, n_b) \) being the length of the shorter one. We say that \( a \) is better than \( b \) and write \( a < b \), if \( a_1 \ldots a_m < b_1 \ldots b_m \) wrt. lexicographic order, or if \( a_1 \ldots a_m = b_1 \ldots b_m \) and \( n_a > n_b \). So the “better than” relation almost corresponds to “being smaller in the lexicographic ordering” but with the difference that in our ordering a strict prefix of another rank profile is not better but worse. This definition is useful because of the following property:

▶ Lemma 9. Let \( (\alpha, t) \) be a ranked slice, and \( p, q \in Q_t \).

Then \( \text{idx}(p) < \text{idx}(q) \iff \text{rp}(p) < \text{rp}(q) \).

Proof. If \( \text{idx}(p) < \text{idx}(q) \), then, by definition of the rank tree, either \( p \) is in the subtree of \( \alpha(q) \) or in the subtree of some left sibling of \( q \) or one of its ancestors. In the first case, the rank profile of \( p \) must agree with the rank profile of \( q \) on the whole length of the latter and then, as \( p \) is a descendant, there must be at least one additional node along the branch leading to \( p \) from \( q \), i.e., the rank profile is longer. Hence, \( \text{rp}(p) < \text{rp}(q) \). In the second case, the rank profiles of \( p \) and \( q \) must agree on some prefix up to their latest common ancestor, whereas afterwards the rank profile of \( p \) continues with a node at the root of some different subtree, which is located more to the left in the rank tree and has a smaller rank. But then the rank profile is lexicographically smaller and again we have \( \text{rp}(p) < \text{rp}(q) \). Clearly, reverting this case analysis shows the other direction.

\[\square\]
Determination of Büchi Automata

The $k$-cut of a rank-profile $a$ is a prefix of the rank-profile including all ranks less than $k$ and, if possible, the first rank which is larger than or equal to $k$. Formally, $k$-cut$(a_1 \ldots a_n) := a_1 \ldots a_i$ with $i := \min\{n \cup \{i \mid a_i \geq k\}\}$. Consider, for example, the rank profile $1, 2, 4$. For $k \geq 3$, the $k$-cut is the complete profile $1, 2, 4$. The 2-cut is 1, 2, and the 1-cut is 1.

The ordering $\preceq$ compares the $k$-cuts of two rank profiles, i.e., for rank profiles $a$ and $b$, $a \preceq b$ if $k$-cut$(a) \preceq k$-cut$(b)$ and consequently, $a = b$ if $a \preceq b$ and $b \preceq a$. For example, $1, 2, 4 = 1, 2, 5$, but $1, 2, 4 \preceq 1, 2, 5$.

We write $k$-cut$(q)$ for $k$-cut$(\rho(q))$. This notion is useful when we analyze how the rank profile can change over time in case that the smallest rank that is active infinitely often is $k$.

In the presentation of the construction in Sections 3 and 6 it was specified that the rank assigned to a set during the prune operation is the same fresh rank, and only after normalize all sets have different ranks. To simplify the reasoning in the proofs, in the following we assume that the left child sets all get fresh and unused, but already different ranks. In this case, normalize only closes “gaps” due to removed ranks. It is easy to see, that both formulations are equivalent, but this variant has the advantage that the rank-tree (and also rank-profiles) is also defined on all intermediate pre-slices and that normalize never makes the rank assigned to a set larger than before.

We start by showing that the constructed DPA $B$ accepts all words that are accepted by the NBA $A$.

Lemma 10. $w \in L(A) \implies w \in L(B)$

Proof. Let $\rho = q_0, q_1 \ldots$ be an accepting run of $A$ on $w$. Take the run of $B$ on $w$ and consider its sequence of ranked slices $s_0, s_1, \ldots$ with $s_i = (a_i, t_i)$. Let $p_0, p_1, \ldots$ be the sequence of rank-profiles such that $p_i = \rho(p_i)$ are the rank profiles of states along the run, let $k_0, k_1, \ldots$ be the sequence of dominating ranks, where $k_i$ is the dominating rank in the transition from $s_i$ to $s_{i+1}$, let $k$ be the smallest $k_i$ appearing infinitely often in this sequence, and finally, let $c_i := k$-cut$(p_i)$ and $m_i := \max c_i$ be the $k_i$-cuts of the rank profiles and the last rank along those prefixes, respectively. The proof is structured as follows:

1. First, we show that $p_i \geq k$, $p_{i+1}$ holds in every transition.
2. This implies that for some time $i_0$, $p_i = k$, $p_{i+1}$ must hold for all $i > i_0$.
3. We show that the last ranks $m_i$ of the $k_i$-cuts $c_i$ must be $\geq k$ for all $i > i_0$.
4. Finally, we show that rank $k$ cannot be red after $i_0$ and thus must be inf. often green.

1: Pick some time $i$ and consider the transition from $i$ to $i + 1$.

Notice that regardless of the current dominating rank $k_i$, the rank profile $p_i$ cannot get worse (i.e., increase) during normalize or step. For normalize this is easy to see, because by definition it must preserve the relative ordering and only closes the unused rank “gaps” after removal of empty sets and eventual merges. Hence it can make a rank profile only lexicographically smaller in each position by reassigning ranks, without changing the length. For step, observe that if $q_{i+1} \in \Delta_{i}, (q_i, w(i+1))$, then step either puts $q_{i+1}$ into a set with the same rank as before (right branch) or, if $q_{i+1}$ is accepting (left branch), moves it into a child wrt. the rank tree, thereby increasing the length of the rank profile. If $q_{i+1} \notin \Delta_{i}, (q_i, w(i+1))$, then, by definition of the restricted transition relation $\Delta_{i}, q_{i+1}$ must have a predecessor $q_i'$ in a set with lower tuple index. By Lemma 9 this implies that $\rho_{i+1}(q_{i+1}) \preceq \rho_{i+1}(q_i') \prec \rho_{i+1}(q_i)$. Therefore, the step operation also can only make the rank profile better.

As we have the dominating rank $k_i$, it means that no rank $< k_i$ was red or green, which means for the rank profile $p_i$ that no rank $< k_i$ marked an empty set after step or was overwritten during prune. Also, merge may not modify sets with ranks $< k_i$. Hence, if $m_i < k_i$, then clearly the $k_i$-cut $c_i$ is not influenced in the transition.
If $m_i \geq k_i$, notice that regardless whether $k_i$ was red or green, at least one rank was removed during prune. If $k_i$ itself was red, then all sets with ranks $> k_i$ will be reassigned a smaller rank to close the gap. If $k_i$ was green, it means that it has overwitten some larger rank during prune. If the overwritten rank was $m_i$, then the $k_i$-cut decreased directly. Otherwise, some other rank was overwritten and the set with rank $m_i$ either keeps the same rank during normalize (if $m_i$ was smaller than the overwritten rank), or it will be reassigned to smaller rank (if it was larger). Neither prune nor merge can make the $k_i$-cut shorter than before, as for prune this would imply that a rank $< k_i$ was green and for merge, that it collapses sets with ranks $< k_i$, which is forbidden.

2: There is some time $i'$ after which no rank $< k$ is dominating again. After that, we have $p_i \geq_k p_{i+1}$ for all $i > i'$, as $p_i \geq_k p_{i+1}$ implies that $p_i \geq_k p_{i+1}$ for all $k' > k$ (if one sequence is not lexicographically larger than the other, clearly this also applies to their prefixes). Hence, eventually, after some $i_0 > i'$ we have that $p_i = k$ for all $i > i_0$, i.e., the $k$-cut prefix eventually stabilizes and the rank profile then can only change at positions after this prefix.

3: Next, we show that $m_i \geq k$ for all $i > i_0$. For contradiction, assume that $m_i < k$ at some time $i > i_0$. Observe that the accepting run $\rho$ visits an accepting state $q_F$ infinitely often. This means, that eventually $q_F$ goes into the left child set during step, which would either make $m_i$ (which is $< k$ by assumption) green, if the right child set is empty, or otherwise it would make the $k$-cut longer, violating either the fact that no rank $< k$ is active after $i_0$, or that the $k$-cut does not change anymore after $i_0$.

4: Finally, observe that $k$ cannot be red after $i_0$. If $k$ was red after $i_0$, it means that it was either overwritten by some smaller green rank, which we excluded by choice of $k$ and $i_0$, or it would mark an empty set after step and be removed during prune, which is also not possible, because if a rank is removed not due to being overwritten, it means that all sets to the left were empty, whereas some set must contain the current state of the run $\rho$, and it cannot be to the right of $k$ because $m_i \geq k$.

As $k$ is active infinitely often and cannot be red after $i_0$, it can only be red finitely often and must be green infinitely often, which implies that the smallest assigned priority is even and $B$ accepts $w$.

### C.2 From an accepting TDPA run to an accepting NBA run

To show the other direction, it is convenient to define the notion of the run-DAG. Intuitively, the run-DAG can be obtained from the reduced split-tree (see Section 3) by applying the corresponding merge operation on each level of the tree (i.e., merging the corresponding sets and redirecting the edges to the new union set), before constructing the next level.

**Definition 11.** Let $w \in \Sigma^\omega$ and $s_0, s_1, \ldots$ with $s_i = (\alpha_i, t_i)$ the sequence of macrostates on $w$. The run-DAG of $B$ on $w$ is defined as follows. Level $i$ of the DAG has the sets $S_{i,1}, \ldots, S_{i,n}$ of the tuple $t_i$ as nodes. An edge $S_{i,j} \to S'_{i+1,j'}$ exists between sets on two adjacent levels $i$ and $i+1$ of the DAG, if the target set contains a non-trivial subset of normalized successors of the source set, or formally, if $\emptyset \neq \Delta_i(S_{i,j}, w(i)) \subseteq S'_{i+1,j'}$. The edge is marked (written $S_{i,j} \xrightarrow{\Delta} S'_{i+1,j'}$), if $\Delta_i(S_{i,j}) \cap S'_{i+1,j'} \subseteq F$, i.e., if it contains no non-accepting normalized successors from the source set. Otherwise, it is unmarked. An infinite path in the run-DAG is a bad path, if it contains only unmarked edges.

See Figure 7 for an illustration of the relationship of the determinization transitions and the run-DAG. We also need the following related definitions, collected here for reference:
\textbf{Definition 12.} Let \( w \in L(B) \) and let \( k \) be the smallest rank that is infinitely often green and finitely often red during the run of \( B \) on \( w \), i.e. the rank witnessing the acceptance.

Let \( i_0 \) be some time after which no rank \(<k \) is ever active again and \( k \) is never red again.

Let \( r_i \) denote the tuple index of the set with rank \( k \) at time \( i \), and let \( l_i < r_i \) be the index of the bad left neighbor, defined as the maximal index less than \( r_i \) such that there is a bad path of the run-DAG starting in the set \( S_{i,l_i} \) (if no such left bad neighbor exists, let \( l_i := 0 \)).

Finally, let \( \text{good}(i) := \bigcup_{j=l_i+1}^{r_i} S_{i,j} \).

Intuitively, if the DPA accepts a word, this means that the smallest rank \( k \) which is infinitely often active is only finitely often red, and therefore will not be reassigned to a completely unrelated set of states in the ranked slices. We show that \( k \) must eventually mark sets containing states of at least one accepting run infinitely often. Unfortunately, considering only the sets with rank \( k \) in isolation is not sufficient to verify that this is the case, but the following Lemma \[13\] gives us a sequence of “checkpoints” and sets such that we can obtain an infinite sequence of suitable pieces to construct an accepting NBA run using König’s Lemma. An illustration of this technical result is found in Figure \[8\].

\textbf{Lemma 13.} Let \( w, i_0 \) and \( \text{good}(i) \) be defined as in Definition \[12\]. Then for every time \( i \geq i_0 \) it holds that \( \text{good}(i) \not= \emptyset \), and there is some \( i' > i \) such that for every \( q \in \text{good}(i') \) there is some \( p \in \text{good}(i) \) such that there is a path from \( p \) to \( q \) that is labelled by the substring \( w(i) \ldots w(i' - 1) \) of \( w \) and visits at least one accepting state.

\textbf{Proof.} Fix a time \( i \geq i_0 \) and let \( k, r_i \) and \( l_i \) also be defined as in Definition \[12\].

Clearly, \( \text{good}(i) \) is not empty, as it at least contains the set \( S_{i,r_i} \). Notice that by choice of time \( i_0 \) and rank \( k \), the set with rank \( k \) will never be merged together with some set to the right, i.e. a set with index \( > r_i \), as this would violate the constraints of \text{merge} (because after \( i_0 \) no rank \(< k \ can be active). For the same reason, it will also be never merged with its left subtree boundary set (if any), which by definition has a smaller rank and therefore must be preserved unchanged.

Next, observe that the set with rank \( k \) can also never be merged with a set that lies on a bad path (and also cannot be a bad path itself), because lying on a bad path means that during \text{step} the set of non-accepting normalized successors (the right child set) would never become empty (leading to the unmarked edges in the run-DAG), hence it would always
inherit the rank $k$ and forever prevent $k$ from becoming green (which requires the right successor set $\Delta_{l_i}(S_{i,r_i}, w(i)) \cap F$ to be empty), contradicting the choice of $k$.

As after time $i_0$ no rank $< k$ will be active again, it means that the left subtree boundary set of $k$ (if it exists) will be neither red nor green ever again. But this implies, that in step it always has a non-empty set of non-accepting successor states, which inherits its rank and therefore this set, by definition, must lie on a bad path in the run-DAG. Hence, $l_i \geq \epsilon_{l_i}(r_i)$.

This motivates the definition of $\text{good}(i) := \bigcup_{j=l_i+1}^{r_i} S_{i,j}$ as the union of all sets whose successors could (in principle) be merged with the successors of $S_{i,r_i}$ without violating the assumptions.

As, by definition, no set contained in $\text{good}(i)$ can lie on a bad path, it means that eventually every path of the run-DAG starting in some set inside $\text{good}(i)$ must either terminate or eventually go through a marked edge. Pick some $i' > i$ such that this is the case.

Observe that tracing the left and right boundary indices $l_i$ and $r_i$ over time gives us an interval $(l_i, r_i)$ of indices for each level of the run-DAG such that no edge in the run-DAG from outside this interval can ever go inside this interval (if an edge enters from the left, then it contains a bad path and if an edge enters from the right, then the set with rank $k$ is merged with sets to the right, both cases we excluded).

For this reason, every state $q \in \text{good}(i')$ must have been reached from some state $p \in \text{good}(i)$. As every path of the run-DAG starting in a set contained in $\text{good}(i)$ must go through at least one marked edge to reach a set in $\text{good}(i')$, it means that every path from a state $p \in \text{good}(i)$ to some state $q \in \text{good}(i')$ agreeing with the run-DAG must contain at least one accepting state, because a run-DAG edge is only marked, if only accepting successors of the source set end up in the target set of the edge.

Figure 8 An abstract sketch of the run-DAG, the different entities in Definition [12] and the statement of Lemma [13]. On the levels $i, i' \geq i_0$, the sets with the smallest active rank $k$ ($S_{i,r_i}$ and $S_{i',r_i}$) are illustrated with their bad left neighbors ($S_{i,l_i}$ and $S_{i',l_i}$) from which bad paths in the run-DAG start. Since $S_{i,r_i}$ has rank $k$, its left subtree boundary set (in the tree view) has rank $< k$. Since no rank $< k$ is active after $i_0$, there must be a bad path starting from the left subtree boundary set of $S_{i,r_i}$, and hence the bad left neighbor of $S_{i,r_i}$ is between (including) $S_{i,r_i}$ and (excluding) $S_{i,r_i}$. The unions of the highlighted intervals form the sets $\text{good}(i)$ and $\text{good}(i')$, respectively. The time $i'$ is chosen such that all paths starting in the interval that defines $\text{good}(i)$ have used at least one marked edge of the run-DAG. If the bad left neighbor does not exist, then the interval defining the set $\text{good}$ goes up to the left border. Paths are marked with $M$ when they contain a marked edge, terminated paths in the run-DAG are marked with $\times$. Notice that paths can leave the interval boundary, but cannot enter.
Using this result, we can show how an accepting run of the NBA can be constructed from an accepting run of the DPA:

**Lemma 14.** \( w \in L(B) \implies w \in L(A) \)

*Proof.* Let \( w \in L(B) \), rank \( k \), time \( i_0 \) and sets \( \text{good}(i) \) as defined in Definition 12.

Then, by Lemma 13, there is a set \( \text{good}(i_0) \) that contains the current set with rank \( k \) and thus is non-empty, and furthermore has the property that there exists some \( i_1 > i_0 \) such that \( \text{good}(i_1) \) also is non-empty and all states in \( \text{good}(i_1) \) are reached from at least one state in \( \text{good}(i_0) \) by some path that visits some accepting state. By iteratively using Lemma 13, we can construct an infinite sequence of times \( i_0, i_1, i_2, \ldots \) such that every pair of times \( i_j \) and \( i_{j+1} \) satisfies these properties.

We now can construct a finitely branching infinite DAG in which the edges are labelled by finite run segments of the NBA. Level 0 of the DAG has one node for each initial state of the NBA. For \( j \geq 1 \), level \( j \) has one node for each state in \( \text{good}(i_{j-1}) \). From level 0 to level 1, there is an edge from \( q_0 \) to \( q \) if there is a run of \( A \) from \( q_0 \) to \( q \) on the input \( w(0) \cdots w(i_0 - 1) \). For \( j \geq 1 \), there is an edge from \( p \in \text{good}(i_{j-1}) \) on level \( j \) to \( q \in \text{good}(i_j) \) on level \( j + 1 \), if there is a run from \( p \) to \( q \) on the input \( w(i_{j-1}) \cdots w(i_j - 1) \) that visits an accepting state. Label the edge by this run. By Lemma 13, for each \( q \) on level \( j + 1 \) there is some \( p \) on level \( j \) such that there is an edge from \( p \) to \( q \).

By König’s lemma, there is an infinite path through this DAG starting on level 0. The concatenations of edge labels of this path yields a run of \( A \) on \( w \) by construction. Starting from level 1, all edge labels contain at least one accepting state, and therefore this run must be accepting.

By Lemmas 10 and 14, we have shown that \( L(A) = L(B) \), which completes the proof of Theorem 6.