Size Complexity of Rotating and Sweeping Automata[☆]

Christos Kapoutsis,, Richard Královič,, Tobias Mömke

Department of Computer Science, ETH Zurich

Abstract

We examine the succinctness of *one-way*, *rotating*, *sweeping*, and *two-way* deterministic finite automata (1DFAs, RDFAs, SDFAs, 2DFAs) and their nondeterministic and randomized counterparts. Here, a SDFA is a 2DFA whose head can change direction only on the end-markers and a RDFA is a SDFA whose head is reset to the left end of the input every time the right end-marker is read. We study the size complexity classes defined by these automata, i. e., the classes of problems solvable by small automata of certain type. For any pair of classes of one-way, rotating, and sweeping deterministic (1D, RD, SD), self-verifying (1Δ, RΔ, SΔ) and nondeterministic (1N, RN, SN) automata, as well as for their complements and reversals, we show that they are equal, incomparable, or one is strictly included in the other. The provided map of the complexity classes has interesting implications on the power of randomization for finite automata. Among other results, it implies that LasVegas sweeping automata can be exponentially more succinct than SDFAs. We introduce a list of language operators and study the corresponding closure properties of the size complexity classes defined by these automata as well. Our conclusions reveal also the logical structure of certain proofs of known separations among the complexity classes and allow us to systematically construct alternative witnesses of these separations.

1. Introduction

One of the major goals of the theory of computation is the comparative study of randomized computations, on one hand, and deterministic and nondeterministic computations, on the other. An important special case of this comparison concerns randomized computations of zero error (also known as "LasVegas computations"): how does ZPP compare to P and NP? Or, in informal terms: *Can every fast LasVegas algorithm be simulated by a fast deterministic one? Can every fast nondeterministic algorithm be simulated by a fast LasVegas one?* Similar questions can be asked also for other randomized models, such as one-sided error (Monte-Carlo) computations and bounded two-sided error computations.

Naturally, the computational model and resource for which we pose these questions are the Turing machine and time, respectively, as these give rise to the best available

[☆]This work was partially supported by the Swiss National Science Foundation grant 200021-107327/1.

theoretical model for the practical problems that we care about. The questions, however, have also been asked for other computational models and resources. Of particular interest to us is the case of restricted models, where the questions appear to be much more tractable. Conceivably, answering them there might also improve our understanding of the harder, more general settings.

In this direction, the case of finite automata has been usually studied using the size of the automata (number of states) as the efficiency measure. The comparison between determinism and nondeterminism in the two-way case was brought into attention in [20]: does every two-way nondeterministic finite automaton (2NFA) with n states have a deterministic equivalent (2DFA) with a number of states polynomial in n? Equivalently, if 2N is the class of families of languages that can be recognized by families of polynomially large 2NFAs and 2D is its deterministic counterpart, is $\text{2D} = \text{2N}$?

The relationship of 2D vs. 2N is one of the most prominent open problems in the area of finite automata. The answer is conjectured to be negative, even if all 2NFAs considered are actually one-way (1NFAs). That is, even $\text{2D} \not\supseteq \text{1N}$ is conjectured to be true, where 1N is the one-way counterpart of 2N.

Later, the case of randomized automata has been considered, too. Hromkovič and Schnitger [8] studied the case of one-way finite automata. They showed that, in this context, LasVegas computations are not more powerful than deterministic ones—intuitively, *every small one-way Las Vegas finite automaton* ($1\text{P}_0\text{FA}$) *can be simulated by a small deterministic one* (1DFA). Equivalently, if 1P_0 is the class of language families that can be recognized by families of polynomially large 1P_0FAs and 1D is its deterministic counterpart, it holds that $1\text{P}_0 = \text{1D}$. This immediately implies that, in contrast, nondeterministic computations are more powerful than LasVegas ones: *there exist small one-way nondeterministic finite automata* (1NFAs) *that cannot be simulated by any small* $1\text{P}_0\text{FA}$. Equivalently, $1\text{P}_0 \subsetneq \text{1N}$.

For the case of two-way finite automata (2DFAs, 2P_0FAs, and 2NFAs), though, the analogous questions remain open [9]: *Can every small* $2\text{P}_0\text{FA}$ *be simulated by a small* 2DFA*? Can every small* 2NFA *be simulated by a small* $2\text{P}_0\text{FA}$*?* Note that a negative answer to either question would solve the 2N vs. 2D problem. Since solving the 2N vs. 2D problem turned out to be hard, certain restricted special cases of the problem have been considered, too. Two of them, introduced in [20, 22], are the *rotating* and the *sweeping* 2DFAs (RDFAs and SDFAs, respectively).[1]

A SDFA is a 2DFA that changes the direction of its head only on the input end-markers. Thus, a computation is simply an alternating sequence of rightward and leftward one-way scans. A RDFA is a SDFA that performs no leftward scans: upon reading the right end-marker, its head jumps directly to the left end. The subsets of 2D that correspond to these restricted 2DFAs are called SD and RD.

Several facts about the size complexity of SDFAs have been known for quite a while (e. g., $\text{1D} \not\supseteq \text{SD}$ [22], $\text{SD} \not\supseteq \text{2D}$ [22, 1, 18], $\text{SD} \not\supseteq \text{1N}$ [22]) and, often, at the core of their proofs one can find proofs of the corresponding facts for RDFAs (e. g., $\text{1D} \not\supseteq \text{RD}$, $\text{RD} \not\supseteq \text{2D}$, etc.). Overall, though, the study of these automata has been fragmentary, exactly because they have always been examined only on the way to investigate the 2D vs. 2N question.

In this article we explore the complexity classes of sweeping and rotating automata. To be able to do so, we introduce a technique of *hardness propagation*, a general frame-

[1] In [20], rotating automata was defined in a slightly different way, and called *series finite automata.*

work for proving lower bounds on the size complexity. This technique allows us to use certain language operators to build hard language families for more powerful automata classes out of a simple, minimally hard, 'core' family. In this way, we show an exponential gap in size complexity between RDFAs and SDFAs. Moreover, we can use hardness propagation to find witnesses for previously known complexity class separations in a systematic way. We believe that this operator-based reconstruction of witnesses deepens our understanding of the relative power of the considered automata, since it uncovers the logical structure of the witness languages and explains how hardness propagates upwards in our map of complexity classes when appropriate operators are applied.

We address also the relationship between the complexity classes induced by randomized finite automata. We apply results of [9, 17] to relate randomized automata to their non-randomized counterparts, thus extending our results to randomized classes. The most interesting result here is that SP_0FAs can be exponentially more succinct than SDFAs.

The structure of the paper is as follows. In the next section, we introduce the basic notation, types of automata used, and the concept of the size complexity classes. The definition of randomized automata and their classes is, however, postponed until Section 5. In Section 3, we develop the technique of hardness propagation. Here, we introduce several language operators and prove the core lemmas that allow us to construct hard languages for complex classes out of simpler languages that are hard for simpler complexity classes. In Section 4, we prove closure properties and the relationships between the classes, heavily using the hardness propagation lemmas. Section 5 is devoted to the randomized automata. Section 6 lists our final conclusions.

2. Preliminaries

Let Σ be an alphabet, i. e., any finite set of symbols. By Σ^* we denote the set of all finite strings over Σ. If $z \in \Sigma^*$, then $|z|$, z_t, z^t, and z^{R} are its length, t-th symbol (if $1 \leq t \leq |z|$), t-fold concatenation with itself (if $t \geq 0$), and reverse. A *language* over Σ is any $L \subseteq \Sigma^*$, the *complement* of L is $\overline{L} := \Sigma^* \setminus L$, the *reverse* of L is $L^{\mathsf{R}} := \{w^{\mathsf{R}} \mid w \in L\}$. If $w \in L$, we say w is a *positive instance* of L, otherwise w is a *negative instance* of L. An automaton *recognizes* (or *solves* or *accepts*) a language if it accepts exactly the strings of that language.

2.1. Computational Models

We assume that the reader is familiar with the one-way and two-way automata. This subsection fixes some notation and introduces the rotating and sweeping models.

A *sweeping deterministic finite automaton* (SDFA) [22] over an alphabet Σ and a set of states Q is any triple $M = (q_s, \delta, q_a)$ of a *start* state $q_s \in Q$, an *accept* state $q_a \in Q$, and a *transition function* δ that partially maps $Q \times (\Sigma \cup \{\vdash, \dashv\})$ to Q, for some *end-markers* $\vdash, \dashv \notin \Sigma$. An input $z \in \Sigma^*$ is presented to M surrounded by the end-markers, as $\vdash z \dashv$. The computation starts at q_s and on $\vdash$. The next state is always derived from δ and the current state and symbol. The next position is always the adjacent one in the direction of motion; except when the current symbol is $\vdash$ or when the current symbol is $\dashv$ and the next state is not q_a, in which cases the next position is the adjacent one towards the other end-marker. Note that the computation can either loop, or hang, or fall off $\dashv$ into q_a. In the last case we call it *accepting* and say that M *accepts* z.

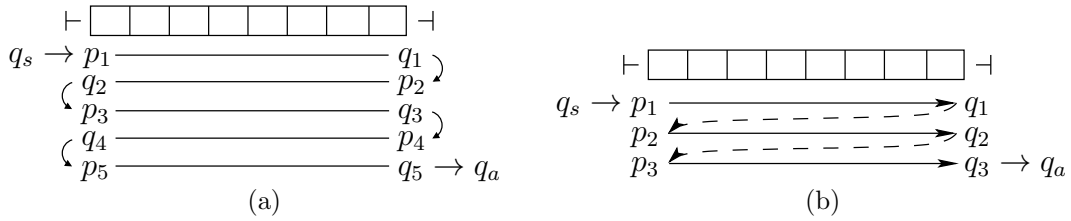

Figure 1: Computation of a (a) sweeping and (b) rotating automaton.

More generally, for any input string $z \in \Sigma^*$ and state p, the *left computation of M from p on z* is the unique sequence

$$\textsc{lcomp}_{M,p}(z) := (q_t)_{1 \le t \le m},$$

where $q_1 := p$; every next state is $q_{t+1} := \delta(q_t, z_t)$, provided that $t \le |z|$ and the value of δ is defined; and m is the first t for which this provision fails. If $m = |z| + 1$, we say that the computation *results in* q_m; otherwise, $1 \le m \le |z|$ and the computation *hangs at* q_m and *results in* $\perp$. The *right computation of M from p on z* is denoted by $\textsc{rcomp}_{M,p}(z)$ and defined symmetrically, i. e., $\textsc{rcomp}_{M,p}(z) = \textsc{lcomp}_{M,p}(z^{\mathsf{R}})$.

The *traversals of M on z* are the members of the unique sequence $(c_t)_{1 \le t < m}$ where $c_1 := \textsc{lcomp}_{M,p_1}(z)$ for $p_1 := \delta(q_s, \vdash)$; every next traversal c_{t+1} is either $\textsc{rcomp}_{M,p_{t+1}}(z)$, if t is odd and c_t results in a state q_t such that $\delta(q_t, \dashv) = p_{t+1} \neq q_a$, or $\textsc{lcomp}_{M,p_{t+1}}(z)$, if t is even and c_t results in a state q_t such that $\delta(q_t, \vdash) = p_{t+1}$; and m is either the first t for which c_t is not defined or ∞ if c_t exists for all t. Then, the *computation of M on z*, denoted by $\textsc{comp}_M(z)$, is the concatenation of $(q_s), c_1, c_2, \ldots$ and possibly also (q_a), if m is finite and even and c_{m-1} results in a state q_{m-1} such that $\delta(q_{m-1}, \dashv) = q_a$. An example of a computation of SDFA is depicted in Figure 1a.

If M is allowed more than one next move at each step, we say it is *nondeterministic* (a SNFA). Formally, this means that δ partially maps $Q \times (\Sigma \cup \{\vdash, \dashv\})$ to the set of *all non-empty subsets* of Q. Hence, on any $z \in \Sigma^*$, $\textsc{comp}_M(z)$ is a *set* of computations. If at least one of them is accepting, we say that M *accepts* z.

We say that M is a *rotating deterministic finite automaton* (RDFA) if its next position is decided differently: it is always the adjacent one to the right, except when the current symbol is $\dashv$ and the next state is not q_a, in which case it is the one to the right of $\vdash$.

The formal definition of the computation of a RDFA M on a string z is similar to the definition for a SDFA. The *traversals of M on z* are always defined in terms of LCOMP, i. e., as the members of the unique sequence $(c_t)_{1 \le t < m}$ where $c_1 := \textsc{lcomp}_{M,p_1}(z)$ for $p_1 := \delta(q_s, \vdash)$; every next traversal c_{t+1} is defined as $\textsc{lcomp}_{M,p_{t+1}}(z)$, if c_t results in a state q_t such that $\delta(q_t, \dashv) = p_{t+1} \neq q_a$; and m is either the first t for which c_t cannot be defined or ∞ if c_t exists for all t. The *computation of M on z*, denoted by $\textsc{comp}_M(z)$, is the concatenation of $(q_s), c_1, c_2, \ldots$ and possibly also (q_a) if m is finite and c_{m-1} results in a state q_{m-1} such that $\delta(q_{m-1}, \dashv) = q_a$. An example of a computation of a RDFA is depicted in Figure 1b. Similar to the definition of a SNFA, we define also the *rotating nondeterministic finite automaton*, which we denote as RNFA.

We say that M is a 1DFA if it halts immediately after reading $\dashv$: the value of δ on any state q and on $\dashv$ is always either q_a or undefined. If it is q_a, we say q is a *final* state; if

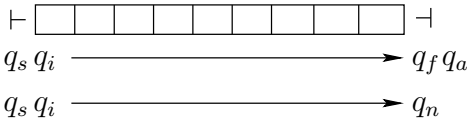

Figure 2: Computation of a one-way automaton. The computation starts in the start state q_s on $\vdash$ and the first symbol of the input word is read in the initial state q_i. After reading the last symbol of the input word, automaton either reaches a final state q_f and enters the accept state q_a afterwards, or it reaches a non-final state q_n and hangs.

it is undefined, we say q is *nonfinal*. The state $\delta(q_s, \vdash)$, if defined, is called *initial*. If M is allowed more than one next move at each step, we say it is *nondeterministic* (a 1NFA). An example of a one-way computation is shown in Figure 2.

2.2. Complexity Classes

Since 1DFAs and 2NFAs have equivalent computational power [16], all types of considered finite automata accept exactly the class of regular languages. Nevertheless, different types of automata may require different number of states to accept the same language. Hence, we focus on the *size complexity* of finite automata, which we measure by the number of states. We follow the approach of [20, 14, 15] and consider *families of languages* instead of individual languages.

Let $M_1, M_2, \ldots$ be finite automata and let $L_1, L_2, \ldots$ be languages over alphabets $\Sigma_1, \Sigma_2, \ldots$. A *family of automata* $\mathcal{M} = (M_n)_{n\geq 1}$ solves a *family of languages* $\mathcal{L} = (L_n)_{n\geq 1}$ if, for all n, M_n solves L_n, i. e., $L(M_n) = L_n$. The automata of $\mathcal{M}$ are "*small*" if, for some polynomial p and all n, M_n has at most $p(n)$ states.

The size complexity class 1D consists of every family of languages that can be solved by a family of small 1DFAs. The classes RD, SD, 2D, 1N, RN, SN, 2N are defined similarly, by replacing 1DFAs with RDFAs, SDFAs, 2DFAs, 1NFAs, RNFAs, SNFAs, 2NFAs. The naming convention is from [20]; in general, class $\mathfrak{C}$ consists of every family of languages that can be solved by a family of small $\mathfrak{C}$FAs.

Any language operator can be generalized to work on language families in a straightforward way: the operator is just separately applied to all languages in the family. In particular, we say that the *complement* of a family of languages $\mathcal{L} = (L_n)_{n\geq 1}$ is defined as $\overline{\mathcal{L}} := (\overline{L_n})_{n\geq 1}$, and the *reverse* of $\mathcal{L}$ is defined as $\mathcal{L}^{\mathrm{R}} := (L_n^{\mathrm{R}})_{n\geq 1}$.

If $\mathfrak{C}$ is a class, then co-$\mathfrak{C}$ consists of all families of languages whose complement is in $\mathfrak{C}$, and re-$\mathfrak{C}$ consists of all families of languages whose reverse is in $\mathfrak{C}$.

Due to the close connection with randomized automata (as discussed in Section 5), we also consider the "*self-verifying*" classes $1\Delta := 1\mathrm{N} \cap \text{co-}1\mathrm{N}$, $\mathrm{R}\Delta := \mathrm{RN} \cap \text{co-RN}$, $\mathrm{S}\Delta := \mathrm{SN} \cap \text{co-SN}$, and $2\Delta := 2\mathrm{N} \cap \text{co-}2\mathrm{N}$. In general, the naming convention is that $\mathrm{X}\Delta := \mathrm{XN} \cap \text{co-XN}$, for any X.

The notion of the *self-verifying nondeterminism* was introduced in [7, 2], where it was considered as a separate mode of computation: the *self-verifying automaton* is able to make nondeterministic choices, and is able to give three types of answers: "*yes*", "*no*", and "*I do not know*". Whenever the answer is "*yes*" or "*no*", it has to be correct, i. e., any possible computation is required to provide either correct answer or the "*I do not*

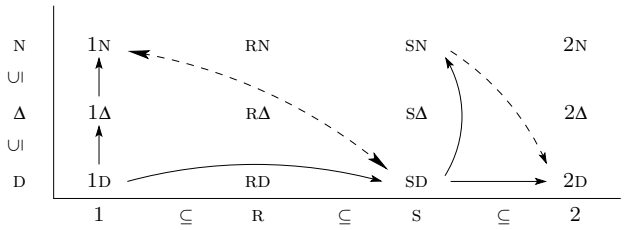

Figure 3: Complexity classes of finite automata. A solid arrow $\mathfrak{C} \to \mathfrak{C}'$ means that $\mathfrak{C} \subsetneq \mathfrak{C}'$, a dashed arrow $\mathfrak{C} \to \mathfrak{C}'$ means that $\mathfrak{C} \not\supseteq \mathfrak{C}'$.

know" answer. Furthermore, for each input there must exist at least one computation that gives the correct answer. This alternative definition is, however, equivalent to the definition given above.

Note that all basic classes introduced above can be described by two independent properties: the head motion (one-way, rotating, sweeping, two-way), and the mode of computation (deterministic, self-verifying, nondeterministic). Hence, we can arrange the corresponding classes on a map, as in Figure 3. Any class in this map is a superset of both its left and its lower neighbor: The fact that $\mathrm{X_D} \subseteq \mathrm{X}\Delta$ follows from $\mathrm{X_D} \subseteq \mathrm{X_N}$ and the fact that $\mathrm{X_D}$ is closed under complement for all presented classes $\mathrm{X_D}$ (discussed in Section 4.1).

In this paper, we completely describe the relationships between the classes of one-way, rotating, and sweeping deterministic, self-verifying, and nondeterministic automata, and their co- and re- classes: For any pair of such classes, we show that they are either equal, included in each other, or incomparable. Previously known results are presented in Figure 3: $1\mathrm{D} \subsetneq 1\Delta \subsetneq 1\mathrm{N}$ [20], $1\mathrm{D} \subsetneq \mathrm{SD}$ [22], $\mathrm{SD} \subsetneq 2\mathrm{D}$ [22, 1, 18], $\mathrm{SD} \not\supseteq 1\mathrm{N}$ [22], $1\mathrm{N} \not\supseteq \mathrm{SD}$ [20], $\mathrm{SN} \not\supseteq 2\mathrm{D}$ [11]. Several of these facts are also direct consequences of stronger results presented in this paper. The relationship between 2N and 2D is a long-standing open problem, raised by [20].

3. Hardness Propagation

Now we introduce a technique of *hardness propagation* for proving separations between the complexity classes. The high-level idea is the following. To separate classes $\mathfrak{C}_1$ and $\mathfrak{C}_2$, it is sufficient to provide a *witness*, i. e., a family of languages $\mathcal{L} \in \mathfrak{C}_2 \setminus \mathfrak{C}_1$. While proving that $\mathcal{L}$ is in $\mathfrak{C}_2$ can often be done by a straightforward construction of the corresponding family of small automata, proving that $\mathcal{L} \notin \mathfrak{C}_1$ is usually more difficult. It may be, however, feasible to prove that $\mathcal{L}$ is not in some (very restricted) class $\mathfrak{C}_0$. If we are able to find a language family operator $\mathcal{O}$ such that (1) for any language family it holds that $\mathcal{L} \notin \mathfrak{C}_0$ implies $\mathcal{O}(\mathcal{L}) \notin \mathfrak{C}_1$, and (2) $\mathfrak{C}_2$ is closed under $\mathcal{O}$, we directly obtain a witness $\mathcal{O}(\mathcal{L}) \in \mathfrak{C}_2 \setminus \mathfrak{C}_1$ of the desired separation. In this way, we can build a harder language $\mathcal{O}(\mathcal{L})$ out of an easier one $\mathcal{L}$; operator $\mathcal{O}$ *propagates* hardness from $\mathcal{L}$ vs. $\mathfrak{C}_0$ to $\mathcal{O}(\mathcal{L})$ vs. $\mathfrak{C}_1$.

Obviously, the hardness propagation can be done in multiple steps. We can start with a simple language family that is hard for some very restricted class, and, using several appropriate language operators, we propagate the hardness to more powerful classes.

We structure this section as follows. First, we introduce language operators that are later used in the propagation of hardness. Next, we introduce several types of finite automata used as intermediate steps in the propagation of hardness. Finally, we prove several hardness propagation lemmas, which are used in the next section to fill the map of class relationships.

3.1. Language Operators

Besides the complement and reverse, we define more language operators that are helpful in the hardness propagation.

Let L, L_1, L_2 be arbitrary languages over alphabet Σ. Fix a delimiter # that is not in Σ. The languages $L_1 \wedge L_2$, $L_1 \vee L_2$, $L_1 \oplus L_2$, $\bigwedge L$, $\bigvee L$, and $\bigoplus L$ over alphabet $\Sigma \cup \{\#\}$ are defined as follows.

$$\begin{aligned}
L_1 \wedge L_2 &:= \{\#x\#y\# \mid x, y \in \Sigma^*, x \in L_1 \wedge y \in L_2\} \\
L_1 \vee L_2 &:= \{\#x\#y\# \mid x, y \in \Sigma^*, x \in L_1 \vee y \in L_2\} \\
L_1 \oplus L_2 &:= \{\#x\#y\# \mid x, y \in \Sigma^*, x \in L_1 \Leftrightarrow y \notin L_2\} \\
\bigwedge L &:= \{\#x_1\# \dots \#x_l\# \mid l \geq 0, x_i \in \Sigma^*, (\forall i)(x_i \in L)\} \\
\bigvee L &:= \{\#x_1\# \dots \#x_l\# \mid l \geq 0, x_i \in \Sigma^*, (\exists i)(x_i \in L)\} \\
\bigoplus L &:= \{\#x_1\# \dots \#x_l\# \mid l \geq 0, x_i \in \Sigma^*, \text{the number of } i\text{'s such that} \\
&\qquad\qquad x_i \in L \text{ is odd } \}
\end{aligned} \tag{1}$$

We call these operators *conjunctive concatenation*, *disjunctive concatenation*, *parity concatenation*, *conjunctive iteration*, *disjunctive iteration*, and *parity iteration*, respectively. Informally, a language resulting from any of these operations consists of #-delimited *blocks* and a word is in the language if the blocks satisfy the boolean operation used to define the operator.

As noted in Section 2.2, all language operators can be generalized for language families in a straightforward way, by applying the language operator on every member of the language family separately. More formally, let $\mathcal{L}_1 = (L_{1,n})_{n\geq 1}$, $\mathcal{L}_2 = (L_{2,n})_{n\geq 1}$ be language families, then $\mathcal{L}_1 \wedge \mathcal{L}_2 := (L_{1,n} \wedge L_{2,n})_{n\geq 1}$, and $\bigwedge \mathcal{L}_1 := (\bigwedge L_{1,n})_{n\geq 1}$. The definitions for $\vee$, $\bigvee$, $\oplus$ and $\bigoplus$ are analogous.

All concatenation and iteration operators can be applied several times and every application uses a new delimiter symbol. For example, language $\bigwedge\bigvee L$ consists of #′ delimited blocks that belong to $\bigvee L$.

3.2. Parallel Automata

Now we introduce several additional models that are useful as intermediate steps of hardness propagation.

A (*two-sided*) *parallel automaton* (P$_2$DFA), introduced in [22], is any triple $M = (\mathfrak{L}, \mathfrak{R}, F)$ where $\mathfrak{L} = \{C_1, \dots, C_k\}$ and $\mathfrak{R} = \{D_1, \dots, D_l\}$ are disjoint families of 1DFAs, and $F \subseteq {C_1}^Q \times \cdots \times {C_k}^Q \times {D_1}^Q \times \cdots \times {D_l}^Q$, where A^Q is the state set of automaton A augmented by symbol $\perp$, i. e., the set of all possible results of runs of A. To run M on z means to run each $A \in \mathfrak{L} \cup \mathfrak{R}$ on z from its initial state and record the result, but with a twist: each $A \in \mathfrak{L}$ reads from left to right (i. e., reads z), while each $A \in \mathfrak{R}$ reads from right to left (i. e., reads z^R). We say that M accepts z if the tuple of the results of these computations is in F. More formally, let $C_i(z) \in {C_i}^Q$ be the result of $\textsc{lcomp}_{C_i,r_i}(z)$,

and let $D_i(z) \in {D_i}^Q$ be the result of $\text{LCOMP}_{D_i,s_i}(z^{\text{R}})$, where r_i is the initial state of C_i and s_i is the initial state of D_i. The parallel automaton M accepts z if

$$(C_1(z), \ldots, C_k(z), D_1(z), \ldots, D_l(z)) \in F.$$

When $\mathfrak{R} = \emptyset$ or $\mathfrak{L} = \emptyset$, we say M is *left-sided* (a P_{L}DFA) or *right-sided* (a P_{R}DFA), respectively.

A *parallel intersection automaton* ($\cap_2$DFA, $\cap_{\text{L}}$DFA, or $\cap_{\text{R}}$DFA) [20] is a parallel automaton whose F consists of all tuples where *all* results are final states. If F consists of all tuples where *some* result is a final state, the automaton is a *parallel union automaton* ($\cup_2$DFA, $\cup_{\text{L}}$DFA, or $\cup_{\text{R}}$DFA) [20]. Thus, a $\cap_2$DFA accepts its input if all components accept it; a $\cup_2$DFA accepts if at least one component does. Since, for parallel intersection and union automata, F is completely determined by the final states in their components, we simplify the notation for them, from $M = (\mathfrak{L}, \mathfrak{R}, F)$ to $M = (\mathfrak{L}, \mathfrak{R})$.

The number of states of a parallel automaton M is the total number of states over all components of $\mathfrak{L} \cup \mathfrak{R}$. Analogously to the previous definitions, we say that a family of parallel automata $\mathcal{M} = (M_n)_{n \geq 1}$ are 'small' if there exists some polynomial $p(n)$ such that, for all n, M_n has at most $p(n)$ states. Hence, automata of $\mathcal{M}$ have polynomially many components each with polynomially many states.

To use parallel intersection and union automata as intermediate steps in the propagation of hardness, we need to consider their complexity classes. Following our naming convention, class $\cap_2$D (respectively, $\cap_{\text{L}}$D, $\cap_{\text{R}}$D, $\cup_2$D, $\cup_{\text{L}}$D, $\cup_{\text{R}}$D, P_{L}D, P_2D) contains all language families recognizable by families of small $\cap_2$DFAs (respectively, $\cap_{\text{L}}$DFAs, $\cap_{\text{R}}$DFAs, $\cup_2$DFAs, $\cup_{\text{L}}$DFAs, $\cup_{\text{R}}$DFAs, P_{L}DFAs, P_2DFAs).

The following lemma explains basic relationships between parallel and rotating (sweeping) automata:

Lemma 3.1. *The following facts hold:*

1. $\text{1D} \subseteq \cap_{\text{L}}\text{D} \cap \cup_{\text{L}}\text{D}$, $\quad \cap_{\text{L}}\text{D} \cup \cap_{\text{R}}\text{D} \subseteq \cap_2\text{D}$,
2. $\cap_{\text{L}}\text{D} = \text{re-}\cap_{\text{R}}\text{D}$, $\quad \cup_{\text{L}}\text{D} = \text{re-}\cup_{\text{R}}\text{D}$, $\quad \cap_2\text{D} = \text{re-}\cap_2\text{D}$, $\quad \cup_2\text{D} = \text{re-}\cup_2\text{D}$
3. $\cap_{\text{L}}\text{D} = \text{co-}\cup_{\text{L}}\text{D}$, $\quad \cap_{\text{R}}\text{D} = \text{co-}\cup_{\text{R}}\text{D}$, $\quad \cap_2\text{D} = \text{co-}\cup_2\text{D}$
4. $\cap_{\text{L}}\text{D} \cup \cup_{\text{L}}\text{D} \subseteq \text{RD}$, $\quad \cap_2\text{D} \cup \cup_2\text{D} \subseteq \text{SD}$.
5. *Every* RDFA (SDFA) *with k states can be simulated by a k-component* P_{L}DFA *($2k$-component* P_2DFA*, respectively) whose components all have k states. Consequently,* $\text{RD} \subseteq \text{P}_{\text{L}}\text{D}$ *and* $\text{SD} \subseteq \text{P}_2\text{D}$.

Proof. 1. Since one-way automata are special cases of both parallel union and parallel intersection automata, and left-sided and right-sided parallel automata are special cases of two-sided parallel automata, the claim follows.

2. If L can be solved by $\cap_{\text{L}}$DFA or $\cap_2$DFA $M = (\mathfrak{L}, \mathfrak{R})$ with m states, then L^{R} can be solved by $\cap_{\text{R}}$DFA or $\cap_2$DFA $M' = (\mathfrak{R}, \mathfrak{L})$ with m states, and vice versa. The same holds for parallel union automata.
3. If L can be solved by a k-component $\cap_{\text{L}}$DFA M with m states, then $\overline{L}$ can be solved by a k-component $\cup_{\text{L}}$DFA M' with $m + k$ states: to construct M', we first make all components of M non-hanging by adding one new state to every component, then make all nonfinal states final and vice versa. Every word $w \notin L$ is rejected by some component of M, hence it is accepted by the corresponding component of M'. On

the other hand, every word $w \in L$ is accepted by all components of M, hence it is rejected by all components of M'. The arguments for $\cap_{\text{R}}$DFAs and $\cap_2$DFAs are similar.

4. A RDFA can simulate any $\cap_{\text{l}}$DFA or $\cup_{\text{l}}$DFA in a straightforward way, simulating one component per traversal. We assume that every component of the parallel automaton is non-hanging, what can be achieved by adding one new state to every component of the automaton. Then, the set of states of the RDFA consists of all states of the parallel automaton plus one new accept state, so a small family of parallel automata are simulated by a small family of rotating automata. Similar arguments hold for simulation of $\cap_2$DFA or $\cup_2$DFA by a SDFA.
5. Proven in [22] for SDFAs. Each component of the P_2DFA $M' = (\mathcal{L}, \mathcal{R}, F)$ simulates one traversal of the SDFA $M = (q_s, \delta, q_a)$. In this way, the i-th left (right) component of the P_2DFA ends in the same state as a left (right) traversal of the SDFA started in the i-th state. F contains all tuples such that there exists some states $q_1, \ldots, q_k$ such that the left component corresponding to q_a equals to q_1, the right component corresponding to q_1 equals to q_2, the left component corresponding to q_2 equals to q_3, etc., and the right component corresponding to q_n equals to q_a. Since both $\mathcal{L}$ and $\mathcal{R}$ consists of k components, M' consists of $2k$ components. The proof for RDFAs is analogous. □

3.3. The Core of the Hardness Propagation

We use two basic tools for the construction of inputs that are hard for parallel automata: the *confusing* and the *generic* strings. In this section, we present these tools and use them to prove the hardness propagation lemmas.

3.3.1. Confusing Strings

Let $M = (\mathfrak{L}, \mathfrak{R})$ be a $\cap_2$DFA and let L a language over alphabet Σ. We say a string $y \in \Sigma^*$ *confuses* M *on* L if $y \in L$ but some component hangs on it or if $y \notin L$ but every component treats it identically to some word from L:

$$\text{or} \quad \begin{array}{lll} y \in L & \text{and} & (\exists A \in \mathfrak{L} \cup \mathfrak{R})\big(A(y) = \bot\big) \\ y \notin L & \text{and} & (\forall A \in \mathfrak{L} \cup \mathfrak{R})(\exists \tilde{y} \in L)\big(A(y) = A(\tilde{y})\big) \end{array} \tag{2}$$

If some y confuses M on L, then M does not solve L. Note, though, that (2) is independent of the selection of final states in the components of M. Thus, if $\mathfrak{F}(M)$ is the class of $\cap_2$DFAs that may differ from M only in the selection of final states, then a y that confuses M on L confuses every $M' \in \mathfrak{F}(M)$, too, and thus no $M' \in \mathfrak{F}(M)$ solves L, either. The converse is also true.

Lemma 3.2. *Let $M = (\mathfrak{L}, \mathfrak{R})$ be a $\cap_2$DFA and L a language. Then, there exists a confusing string for M on L iff no member of $\mathfrak{F}(M)$ solves L.*

PROOF. [⇒] Suppose some y confuses M on L. Fix any $M' = (\mathfrak{L}', \mathfrak{R}') \in \mathfrak{F}(M)$. Since (2) is independent of the choice of final states, y confuses M' on L, too. *If $y \in L$:* By (2), some $A \in \mathfrak{L}' \cup \mathfrak{R}'$ hangs on y. So, M' rejects y, and thus fails. *If $y \notin L$:* If M' accepts y, it fails. If it rejects y, then some $A \in \mathfrak{L}' \cup \mathfrak{R}'$ does not accept y. Consider the $\tilde{y}$

guaranteed for this A by (2). Since $A(\tilde{y}) = A(y)$, we know $\tilde{y}$ is also not accepted by A. Hence, M' rejects $\tilde{y} \in L$, and fails again.

[$\Leftarrow$] Suppose no string confuses M on L. Then, no component hangs on a positive instance; and every negative instance is 'noticed' by some component, in the sense that the component treats it differently than all positive instances:

$$\text{and} \quad \begin{array}{l} (\forall y \in L)(\forall A \in \mathfrak{L} \cup \mathfrak{R})\big(A(y) \neq \bot\big) \\ (\forall y \notin L)(\exists A \in \mathfrak{L} \cup \mathfrak{R})(\forall \tilde{y} \in L)\big(A(y) \neq A(\tilde{y})\big). \end{array} \tag{3}$$

This allows us to find an $M' \in \mathfrak{F}(M)$ that solves L, as follows. We start with all states of all components of M unmarked. Then we iterate over all $y \notin L$. For each of them, we pick an A as guaranteed by (3) and, if the result $A(y)$ is a state, we mark it. When this (possibly infinite) iteration is over, we make all marked states nonfinal and all unmarked states final. The resulting $\cap_{\text{2DFA}}$ is our M'.

To see why M' solves L, consider any string y. *If $y \notin L$:* Then our method examined y, picked an A, and ensured $A(y)$ is either $\bot$ or a nonfinal state. So, this A does not accept y. Therefore, M' rejects y. *If $y \in L$:* Towards a contradiction, suppose M' rejects y. Then some component A^* does not accept y. By (3), $A^*(y) \neq \bot$. Hence, $A^*(y)$ is a state, call it q^*, and is nonfinal. Thus, at some point, our method marked q^*. Let $\hat{y} \notin L$ be the string examined at that point. Then, the selected A was A^* and $A(\hat{y})$ was q^*, and thus no $\tilde{y} \in L$ had $A^*(\tilde{y}) = q^*$ by (3). But this contradicts the fact that $y \in L$ and $A^*(y) = q^*$. □

Note that Lemma 3.2 is valid also for the empty $\cap_{\text{2DFA}}$ $M = (\emptyset, \emptyset)$. In this case, M solves Σ^*, since every word is accepted by all components of M. The class $\mathfrak{F}(M)$ contains only M. If $L \neq \Sigma^*$, any word $y \notin L$ confuses M on L, since every component of M (vacuously, since there is no such component) treats it identically to some word in L. Conversely, if some y confuses M on L, $y \notin L$ so $L \neq \Sigma^*$.

Confusing strings can be used to prove that a certain language is hard for $\cap_{\text{2DFA}}$s. At first we use this technique to propagate hardness from 1D to $\cap_{\text{1D}}$.

Lemma 3.3. *If no* 1DFA *with at most m states can solve L, then no $\cap_{\text{1DFA}}$ with at most m states per component can solve $\bigvee L$. Similarly, no such $\cap_{\text{1DFA}}$ can solve $\bigoplus L$.*

Proof. Suppose no m-state 1DFA can solve L. By induction on k, we prove the stronger claim that every $\cap_{\text{1DFA}}$ with k components, each having at most m states, is confused on $\bigvee L$ by some *well-formed* string y, i. e., by some $y \in \#(\Sigma^*\#)^*$. The proof for $\bigoplus L$ is identical.

If $k = 0$: fix any such $\cap_{\text{1DFA}}$ $M = (\mathfrak{L}, \emptyset) = (\emptyset, \emptyset)$. By definition, $\# \notin \bigvee L$. Furthermore, $\#$ confuses M on $\bigvee L$, because all components of M (vacuously, since $\mathfrak{L} = \emptyset$) treat it identically to some word in $\bigvee L$. Since $\#$ is well-formed, the claim holds.

If $k \geq 1$: fix any such $\cap_{\text{1DFA}}$ $M = (\mathfrak{L}, \emptyset)$. Pick any $D \in \mathfrak{L}$ and remove it from M to get $M_1 = (\mathfrak{L}_1, \emptyset) := (\mathfrak{L} - \{D\}, \emptyset)$. By the inductive hypothesis, some well-formed y confuses M_1 on $\bigvee L$.

Case 1: $y \in \bigvee L$. Then some $A \in \mathfrak{L}_1$ hangs on y. Since $A \in \mathfrak{L}$, too, y confuses M as well, so the inductive step is complete.

Case 2: $y \notin \bigvee L$. Then every $A \in \mathfrak{L}_1$ treats y identically to a positive instance:

$$(\forall A \in \mathfrak{L} - \{D\})(\exists \tilde{y} \in \bigvee L)\big(A(y) = A(\tilde{y})\big). \tag{4}$$

Now we define a single-component $\cap_{\text{l}}$DFA $M_2 = (\{D'\}, \emptyset)$. If D hangs on y (i. e., $D(y) = \bot$), we define D' to be a single-state automaton that hangs immediately. Otherwise, D' is derived from D by changing its initial state to $D(y)$. In any case, it holds that $D'(z) = D(yz)$ for any non-empty word z.

Since D' has at most m states, it does not solve L due to the assumption of the lemma and no member of $\mathfrak{F}(M_2)$ solves L neither. So, by Lemma 3.2, some x confuses M_2 on L. We claim that $yx\#$ confuses M on $\bigvee L$. Since $yx\#$ is well-formed, the induction is again complete. To prove the confusion, we examine two cases:

Case 2a: $x \in L$. Then $yx\# \in \bigvee L$, since y is well-formed and $x \in L$. And D' hangs on x (since x is confusing for M_2 and D' is the only component), thus $D(yx\#) = D'(x\#) = \bot$. So, component D of M hangs on $yx\# \in \bigvee L$. So, $yx\#$ confuses M on $\bigvee L$.

Case 2b: $x \notin L$. Then $yx\# \notin \bigvee L$, because y is well-formed and not in $\bigvee L$, and x does not contain #. And, since x is confusing for M_2, D' treats it identically to some $\tilde{x} \in L$: $D'(x) = D'(\tilde{x})$. Then, each component of M treats $yx\#$ identically to a positive instance of $\bigvee L$:

- D treats $yx\#$ as $y\tilde{x}\#$: $D(y\tilde{x}\#) = D'(\tilde{x}\#) = D'(x\#) = D(yx\#)$. We know that $y\tilde{x}\# \in \bigvee L$, because y is well-formed and $\tilde{x} \in L$.
- each $A \neq D$ treats $yx\#$ as $\tilde{y}_D x\#$, where $\tilde{y}_D$ is the string guaranteed for A by (4): $A(\tilde{y}_D x\#) = A(yx\#)$. And we know $\tilde{y}_D x\# \in \bigvee L$, since $\tilde{y}_D \in \bigvee L$ and x does not contain #.

Overall, $yx\#$ is again a well-formed confusing string for M on $\bigvee L$, as required. □

Corollary 3.4. *If $\mathcal{L} \notin$ 1D, then $\bigvee \mathcal{L} \notin \cap_{\text{l}}$D and $\bigoplus \mathcal{L} \notin \cap_{\text{l}}$D.*

In fact, Lemma 3.3 is stronger than Corollary 3.4, since it states that even an arbitrarily high number of components cannot help solve $\mathcal{L}$ if the components are small.

Next, we prove a hardness propagation from $\cap_{\text{l}}$D to $\cap_2$D.

Lemma 3.5. *Let $m, k \geq 1$. If L_1 has no $\cap_{\text{l}}$DFA with at most k components, each having at most m states, and L_2 has no $\cap_{\text{R}}$DFA with at most k components, each having at most m states, then $L_1 \vee L_2$ has no $\cap_2$DFA with at most k components, each having at most m states. Similarly, there is no such $\cap_2$DFA for $L_1 \oplus L_2$, neither.*

Proof. Let $M = (\mathfrak{L}, \mathfrak{R})$ be a $\cap_2$DFA with at most k components, each having at most m states; we show that M does not accept $L_1 \vee L_2$. By contradiction, assume that M accepts $L_1 \vee L_2$. Let $M_1 := (\mathfrak{L}', \emptyset)$ and $M_2 := (\emptyset, \mathfrak{R}')$ be the $\cap_2$DFAs derived from the two 'sides' of M after changing the initial state of each $A \in \mathfrak{L} \cup \mathfrak{R}$ to $A(\#)$. Note that we can assume that $A(\#) \neq \bot$: otherwise, A hangs on every well-formed word, hence M accepts the empty language. This implies that both L_1 and L_2 are empty, hence they can be accepted by a $\cap_{\text{l}}$DFA (a $\cap_{\text{R}}$DFA) with 1 component containing 1 state; a contradiction.

By the lemma's assumption, no member of $\mathfrak{F}(M_1)$ solves L_1 and no member of $\mathfrak{F}(M_2)$ solves L_2. So, by Lemma 3.2, some y_1 confuses M_1 on L_1 and some y_2 confuses M_2 on L_2. We claim that $\#y_1\#y_2\#$ confuses M on $L_1 \vee L_2$ and thus M fails.

Case 1: $y_1 \in L_1$ or $y_2 \in L_2$. Assume $y_1 \in L_1$ (if $y_2 \in L_2$, we work similarly). Then $\#y_1\#y_2\# \in L_1 \vee L_2$ and some $A' \in \mathfrak{L}'$ hangs on y_1. The corresponding $A \in \mathfrak{L}$ has $A(\#y_1\#y_2\#) = A'(y_1\#y_2\#) = \bot$. So, $\#y_1\#y_2\#$ confuses M on $L_1 \vee L_2$.

Case 2: $y_1 \notin L_1$ and $y_2 \notin L_2$. Then $\#y_1\#y_2\# \notin L_1 \vee L_2$, each component of M_1 treats y_1 identically to a positive instance of L_1, and each component of M_2 treats y_2

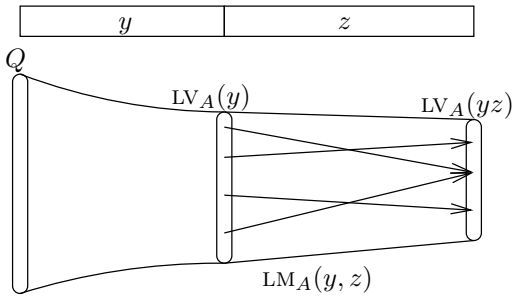

Figure 4: LV and LM.

identically to a positive instance of L_2:

$$(\forall A' \in \mathfrak{L}')(\exists \tilde{y}_1 \in L_1)\big(A'(y_1) = A'(\tilde{y}_1)\big), \tag{5}$$

$$(\forall A' \in \mathfrak{R}')(\exists \tilde{y}_2 \in L_2)\big(A'(y_2) = A'(\tilde{y}_2)\big). \tag{6}$$

Every $A \in \mathfrak{L}$ treats $\#y_1\#y_2\#$ as $\#\tilde{y}_1\#y_2\# \in L_1 \vee L_2$ ($\tilde{y}_1$ as guaranteed by (5)), and every $A \in \mathfrak{R}$ treats $\#y_1\#y_2\#$ as $\#y_1\#\tilde{y}_2\# \in L_1 \vee L_2$ ($\tilde{y}_2$ as guaranteed by (6)). Therefore, $\#y_1\#y_2\#$ confuses M on $L_1 \vee L_2$, again.

The proof for $\oplus$ is analogous. It is necessary, however, to split Case 1 into two sub-cases:

Case 1a: either $y_1 \in L_1$ but $y_2 \notin L_2$, or $y_2 \in L_2$ but $y_1 \notin L_1$. This case is completely analogous to Case 1 of the proof for $\vee$.

Case 1b: both $y_1 \in L_1$ and $y_2 \in L_2$. Since L_2 cannot be accepted by a $\cap_{\text{R}}$DFA with $m \geq 1$ states, it is nontrivial, i. e., there exists some $\tilde{y}_2 \notin L_2$ that does not contain #. Hence $\#y_1\#\tilde{y}_2\# \in L_1 \oplus L_2$ and, by similar arguments as in Case 1 for $\vee$, $\#y_1\#\tilde{y}_2\#$ confuses M. □

Corollary 3.6. *If $\mathcal{L} \notin \cap_{\text{L}}$D, then $\mathcal{L} \vee \mathcal{L}^{\text{R}} \notin \cap_2$D and $\mathcal{L} \oplus \mathcal{L}^{\text{R}} \notin \cap_2$D.*

3.3.2. Generic Strings

Generic strings, introduced in [22], are a powerful tool for proving lower bounds for the size complexity of rotating and sweeping automata. After describing the idea of generic strings, we use it to propagate hardness from $\cap_{\text{L}}$D to RD and from $\cap_2$D to SD.

Let A be a 1DFA over alphabet Σ and states Q, and $y, z \in \Sigma^*$. The *left views of A on y* is the set of states reached on the right boundary of y by left computations of A:

$$\text{LV}_A(y) := \{q \in Q \mid (\exists p \in Q)[\text{LCOMP}_{A,p}(y) \text{ results in } q]\}.$$

The (*left*) *mapping of A on y and z* is the partial function

$$\text{LM}_A(y, z) : \text{LV}_A(y) \to Q$$

which, for every $q \in \text{LV}_A(y)$, is defined only if $\text{LCOMP}_{A,q}(z)$ does not hang and, if so, returns the state that this computation results in. (See Figure 4.)

Fact 3.7. *Function* $\text{LM}_A(y, z)$ *is a* partial surjection *from the set* $\text{LV}_A(y)$ *to* $\text{LV}_A(yz)$.

Fact 3.8. *For all A, y, z as above*: $|\text{LV}_A(y)| \geq |\text{LV}_A(yz)|$.

Fact 3.9. *For all A, y, z as above*: $\text{LV}_A(yz) \subseteq \text{LV}_A(z)$.

In the following, we are going to prove that if there is no small $\cap_{\text{l}}$DFA solving L, then there is no small P_{l}DFA solving $\bigwedge L$. Using this result, we can easily obtain the hardness propagation from $\cap_{\text{l}}$DFAs to RDFAs, since, by Lemma 3.1, P_{l}DFAs are at least as strong as RDFAs.

Consider any P_{l}DFA $M = (\mathfrak{L}, \emptyset, F)$ and any language L. A string y is called L-*generic (for M) over L* if it is in L and, for any component $A \in \mathfrak{L}$, the size of $\text{LV}_A(y)$ cannot be decreased by replacing y by any right-extension $yz \in L$ of y:

$$y \in L \quad \text{and} \quad (\forall yz \in L)(\forall A \in \mathfrak{L})\big(|\text{LV}_A(y)| = |\text{LV}_A(yz)|\big) \tag{7}$$

Note that we can use equality in (7) by Fact 3.8.

It is easy to see that L-generic strings always exist: consider any $y \in L$ such that

$$\sum_{A \in \mathfrak{L}} |\text{LV}_A(y)|$$

is as small as possible. For any $yz \in L$, Fact 3.8 ensures that no term of the sum is increased. The definition of the string y implies that the sum cannot be decreased, thus y is L-generic.

The next lemma shows an important property of L-generic strings. Intuitively, it shows that the behavior of the parallel automaton is very limited after reading a generic string. Later we exploit this limitation to build a small $\cap_{\text{l}}$DFA for L from a small P_{l}DFA accepting $\bigwedge L$.

Lemma 3.10. *Suppose a P_{l}DFA $M = (\mathfrak{L}, \emptyset, F)$ solves $\bigwedge L$ and y is L-generic for M over $\bigwedge L$. Then, $x \in L$ iff $\text{LM}_A(y, xy)$ is total (i. e., defined for every $q \in \text{LV}_A(y)$) and injective for all $A \in \mathfrak{L}$.*

Proof. [$\Rightarrow$] Let $x \in L$. Then $yxy \in \bigwedge L$, because $y \in \bigwedge L$ and $x \in L$. So, yxy is a right-extension of y inside $\bigwedge L$. Since y is L-generic, $|\text{LV}_A(y)| = |\text{LV}_A(yxy)|$, for all $A \in \mathfrak{L}$. Hence, each partial surjection $\text{LM}_A(y, xy)$ has domain and codomain of the same size. This is possible only if the function is a bijection, i. e., it is both total and injective.

[$\Leftarrow$] Suppose that, for each $A \in \mathfrak{L}$, the partial surjection $\text{LM}_A(y, xy)$ is total and injective. Then it bijects the set $\text{LV}_A(y)$ into the set $\text{LV}_A(yxy)$, which is actually a subset of $\text{LV}_A(y)$ (Fact 3.9). This is possible only if this subset is the set itself. Hence, $\text{LM}_A(y, xy)$ is a permutation π_A of $\text{LV}_A(y)$.

Now pick $k \geq 1$ such that π_A^k is an identity for each A. It is always possible to find such k; for example, it is sufficient to choose $k = m!$, where m is the maximal number of states over all components of $\mathfrak{L}$. Let $z := y(xy)^k$. Since $\text{LM}_A\big(y, (xy)^k\big)$ equals $\text{LM}_A(y, xy)^k = \pi_A^k$, it is the identity on $\text{LV}_A(y)$. This means that, reading through z, the left computations of A do not notice the suffix $(xy)^k$ to the right of the prefix y. So, no A can distinguish between y and z: it either hangs on both or results in the same state.

Overall, M does not distinguish between y and z, neither: it either accepts both or rejects both. But M accepts y (because $y \in \bigwedge L$), so it accepts z. Hence, every #-delimited block of z is in L. In particular, $x \in L$. □

If $M = (\mathfrak{L}, \mathfrak{R}, F)$ is a P_2DFA, we can also work symmetrically with right computations and left-extensions: we can define $\text{RV}_A(y)$ and $\text{RM}_A(z, y)$ for $A \in \mathfrak{R}$, derive Facts 3.8, 3.9 for $\text{RV}_A(y)$ and $\text{RV}_A(zy)$, and define R-generic strings. In particular, $\text{RV}_A(y)$ is the set of states of A reached on the left boundary of y after reading it backwards and $\text{RM}_A(z, y)$ is the function that maps states of $\text{RV}_A(y)$ into the states reached after reading z backwards. We can then construct strings that are simultaneously L- and R-generic; we call such strings *generic*. Indeed, if $y_L\#$ is L-generic over $\bigwedge L$ and $\#y_R$ is R-generic over $\bigwedge L$, then $y_L\#y_R$ is a generic string over $\bigwedge L$.

Generic strings can be used to extend Lemma 3.10 for P_2DFAs. The following lemma can be proved by a straightforward extension of the proof of Lemma 3.10:

Lemma 3.11. *Suppose a* P_2DFA $M = (\mathfrak{L}, \mathfrak{R}, F)$ *solves* $\bigwedge L$ *and* y *is generic for* M *over* $\bigwedge L$. *Then,* $x \in L$ *iff* $\text{LM}_A(y, xy)$ *is total (i. e., defined for every* $q \in \text{LV}_A(y)$*) and injective for all* $A \in \mathfrak{L}$ *and* $\text{RM}_A(yx, y)$ *is total (i. e., defined for every* $q \in \text{RV}_A(y)$*) and injective for all* $A \in \mathfrak{R}$.

Now we can use the properties of generic strings to prove hardness propagation from $\cap_{\text{l}}\text{D}$ to RD. The following lemma proves a stronger result of hardness propagation from $\cap_{\text{l}}\text{DFA}$s to $\text{P}_{\text{l}}\text{DFA}$s. The actual hardness propagation to RD is stated as the following corollary; we can do this due to Lemma 3.1(5).

Lemma 3.12. *If* L *has no* $\cap_{\text{l}}\text{DFA}$ *with at most* $k \cdot \binom{m}{2}$ *components of at most* $\binom{m}{2}$ *states each, then* $\bigwedge L$ *has no* $\text{P}_{\text{l}}\text{DFA}$ *with at most* k *components of at most* m *states each.*

Proof. Let $M = (\mathfrak{L}, \emptyset, F)$ be a $\text{P}_{\text{l}}\text{DFA}$ solving $\bigwedge L$ with at most k components of at most m states each. Let y be L-generic for M over $\bigwedge L$. We build a $\cap_{\text{l}}\text{DFA}$ M' solving L.

By Lemma 3.10, an arbitrary x is in L iff $\text{LM}_A(y, xy)$ is total and injective for all $A \in \mathfrak{L}$; i. e., iff for all $A \in \mathfrak{L}$ and every two distinct $p, q \in \text{LV}_A(y)$,

$$\text{LCOMP}_{A,p}(xy) \text{ and } \text{LCOMP}_{A,q}(xy) \text{ do not hang and result in different states.} \quad (8)$$

So, checking $x \in L$ reduces to checking (8) for each A and two-set of states of $\text{LV}_A(y)$. The components of M' perform exactly these checks. To describe them, let us first define the following relation on the states of an $A \in \mathfrak{L}$:

$$r \asymp_A s \iff \text{LCOMP}_{A,r}(y) \text{ and } \text{LCOMP}_{A,s}(y) \text{ do not hang and result in different states,}$$

and restate our checks as follows: for all $A \in \mathfrak{L}$ and all distinct $p, q \in \text{LV}_A(y)$,

$$\text{LCOMP}_{A,p}(x) \text{ and } \text{LCOMP}_{A,q}(x) \text{ do not hang and result in states that relate under } \asymp_A. \quad (8')$$

Now, building 1DFAs to perform these checks is easy. For each $A \in \mathfrak{L}$ and $p, q \in \text{LV}_A(y)$, we use a separate component, i. e., a 1DFA having exactly one state for each two-set of states of A. The initial state is $\{p, q\}$. At each step, the automaton applies A's transition function on the current symbol and each state in the current two-set. If either application returns no value or both return the same value, it hangs; otherwise, it moves to the resulting two-set. A state $\{r, s\}$ is final iff $r \asymp_A s$.

Since for every $A \in \mathfrak{L}$ we constructed at most $\binom{m}{2}$ components of M', each with at most $\binom{m}{2}$ states, the proof is complete. □

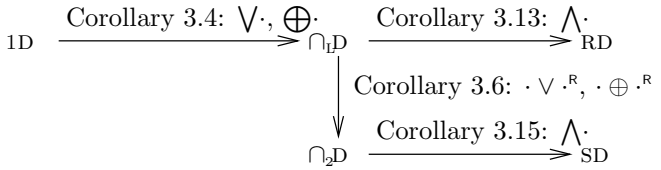

Figure 5: The structure of the hardness propagation lemmas.

	1D	1Δ	1N	$\cup_1$D	RD	$\cup_2$D	SD	SΔ	SN	2D	2Δ	2N	
$\overline{\cdot}$	+	+	−	−	+	−	+	+	−	+	+	?	1
$\cdot^R$	−	+	+	−	−	+	+	+	+	+	+	+	2
$\wedge$	+	+	+	+	+	−	+	+	+	+	+	+	3
$\vee$	+	+	+	+	+	+	+	+	+	+	+	+	4
$\oplus$	+	+	−	−	+	−	+	+	−	+	+	?	5
$\bigwedge$	+	+	+	−	−	−	−	?	?	+	+	+	6
$\bigvee$	+	+	+	+	−	+	−	?	?	+	+	+	7
$\bigoplus$	+	+	−	−	?	−	?	?	−	+	+	?	8
	A	B	C	D	E	F	G	H	I	J	K	L	

Figure 6: Closure properties: '+' means closure; '−' means non-closure; '?' means we do not know.

Corollary 3.13. *If $\mathcal{L} \notin \cap_1$D, then $\bigwedge\mathcal{L} \notin$ RD.*

The statements of the propagation of hardness from $\cap_2$D to SD are similar to Lemma 3.12 and Corollary 3.13. The proof is very similar to the case of rotating automata; RCOMP and RV are used similarly as LCOMP and LV.

Lemma 3.14. *If L has no $\cap_2$DFA with at most $k \cdot \binom{m}{2}$ components of at most $\binom{m}{2}$ states each, then $\bigwedge L$ has no P_2DFA with at most k components of at most m states each.*

Corollary 3.15. *If $\mathcal{L} \notin \cap_2$D, then $\bigwedge\mathcal{L} \notin$ SD.*

To conclude this section, we provide an overview of the presented hardness propagation lemmas in Figure 5.

4. Filling the Map

We now explore the relationships between complexity classes introduced so far, aiming to present a complete map of relationships between all introduced classes except those of two-way automata. To fill that map, it is essential to know the behavior of our classes with respect to several operators. So, we first prove the closures mentioned in Figure 6. They also imply that some classes are identical, simplifying the map of the classes significantly. Afterwards, we prove the separations between the classes and, in Section 4.4, we conclude with the non-closures of Figure 6.

The analysis of the relationship between various complexity classes can be simplified by the following observations:

Observation 4.1. *Let $\mathfrak{C}_1$, $\mathfrak{C}_2$ be any classes of language families. It holds that:*

1. re-(re-$\mathfrak{C}_1$) = $\mathfrak{C}_1$
2. co-(co-$\mathfrak{C}_1$) = $\mathfrak{C}_1$
3. *If* $\mathfrak{C}_1 \subseteq \mathfrak{C}_2$, *then* re-$\mathfrak{C}_1 \subseteq$ re-$\mathfrak{C}_2$.
4. *If* $\mathfrak{C}_1 \subseteq \mathfrak{C}_2$, *then* co-$\mathfrak{C}_1 \subseteq$ co-$\mathfrak{C}_2$.
5. *If* $\mathfrak{C}_1 \not\supseteq \mathfrak{C}_2$, *then* re-$\mathfrak{C}_1 \not\supseteq$ re-$\mathfrak{C}_2$.
6. *If* $\mathfrak{C}_1 \not\supseteq \mathfrak{C}_2$, *then* co-$\mathfrak{C}_1 \not\supseteq$ co-$\mathfrak{C}_2$.
7. *If* $\mathfrak{C}_1 \subseteq \mathfrak{C}_2 \not\supseteq \mathfrak{C}_3 \subseteq \mathfrak{C}_4$, *then* $\mathfrak{C}_1 \not\supseteq \mathfrak{C}_4$.

Observation 4.2. *The following equations hold both for languages and for language families L_1, L_2, L:*

$$\begin{array}{ll} (L_1 \wedge L_2)^{\mathsf{R}} & = L_2^{\mathsf{R}} \wedge L_1^{\mathsf{R}} \\ (L_1 \vee L_2)^{\mathsf{R}} & = L_2^{\mathsf{R}} \vee L_1^{\mathsf{R}} \\ (L_1 \oplus L_2)^{\mathsf{R}} & = L_2^{\mathsf{R}} \oplus L_1^{\mathsf{R}} \end{array} \qquad \begin{array}{ll} (\bigwedge L)^{\mathsf{R}} & = \bigwedge(L^{\mathsf{R}}) \\ (\bigvee L)^{\mathsf{R}} & = \bigvee(L^{\mathsf{R}}) \\ (\bigoplus L)^{\mathsf{R}} & = \bigoplus(L^{\mathsf{R}}) \\ (\overline{L})^{\mathsf{R}} & = \overline{(L^{\mathsf{R}})} \end{array}$$

The situation for complementation is similar but a little more complicated. For example, the language $\overline{(L_1 \wedge L_2)}$ contains all words from the language $\overline{L_1} \vee \overline{L_2}$, plus all words that do not consists of #-delimited blocks, i. e., not from the language $R := \#\Sigma^*\#\Sigma^*\#$. We call such words *not well-formed*. Hence, it holds that $\overline{(L_1 \wedge L_2)} = (\overline{L_1} \vee \overline{L_2}) \cup \overline{R}$. Similarly, $\overline{L_1} \vee \overline{L_2} = \overline{(L_1 \wedge L_2)} \cap R$. Nevertheless, both R and $\overline{R}$ are very simple regular languages, and union (intersection) with them usually does not have any significant impact on the size complexity:

Lemma 4.3. *Let $\mathcal{L}, \mathcal{L}_1$, and $\mathcal{L}_2$ be language families and let $\mathfrak{C}$ be a class of language families closed under union and intersection with any family from* 1D. *It holds that:*

$$\begin{array}{ll} \overline{(\mathcal{L}_1 \wedge \mathcal{L}_2)} \in \mathfrak{C} \Leftrightarrow \overline{\mathcal{L}_1} \vee \overline{\mathcal{L}_2} \in \mathfrak{C} & \qquad \overline{(\bigwedge \mathcal{L})} \in \mathfrak{C} \Leftrightarrow \bigvee \overline{\mathcal{L}} \in \mathfrak{C} \\ \overline{(\mathcal{L}_1 \vee \mathcal{L}_2)} \in \mathfrak{C} \Leftrightarrow \overline{\mathcal{L}_1} \wedge \overline{\mathcal{L}_2} \in \mathfrak{C} & \qquad \overline{(\bigvee \mathcal{L})} \in \mathfrak{C} \Leftrightarrow \bigwedge \overline{\mathcal{L}} \in \mathfrak{C} \\ \overline{(\mathcal{L}_1 \oplus \mathcal{L}_2)} \in \mathfrak{C} \Leftrightarrow \overline{\mathcal{L}_1} \oplus \mathcal{L}_2 \in \mathfrak{C} & \end{array}$$

Proof. To prove the first claim, let

$$\mathcal{L}_1 = (L_{1,n})_{n \geq 1}, \quad \mathcal{L}_2 = (L_{2,n})_{n \geq 1},$$

such that $L_{1,n}$, $L_{2,n}$ are languages over Σ_n. As explained above, it holds that

$$\overline{(\mathcal{L}_1 \wedge \mathcal{L}_2)} = (\overline{\mathcal{L}_1} \vee \overline{\mathcal{L}_2}) \cup \overline{\mathcal{R}}; \qquad \overline{\mathcal{L}_1} \vee \overline{\mathcal{L}_2} = \overline{(\mathcal{L}_1 \wedge \mathcal{L}_2)} \cap \mathcal{R}$$

where $\mathcal{R} := (\#\Sigma_n^*\#\Sigma_n^*\#)_{n \geq 1}$. Easily, $\mathcal{R}, \overline{\mathcal{R}} \in$ 1D, hence the first claim follows. The proofs of the remaining claims are analogous, with the exception of using the well-formed language family $\mathcal{R}$ defined as $\mathcal{R} := (\#(\Sigma_n^*\#)^*)_{n \geq 1}$ for the conjunctive iteration and disjunctive iteration.
□

All state complexity classes introduced so far are closed under the intersection with 1D. Indeed, given a 1DFA M_1 accepting language L_1 and any XFA M_2 accepting language L_2 such that both M_1 and M_2 have at most k states, we can use the well-known *Cartesian-product construction* to construct a XFA accepting $L_1 \cap L_2$ with $O(k^2)$ states.

Now we present lemmas and observations that we use in filling Figure 6. The following observation is a straightforward corollary of Observation 4.2:

Observation 4.4. *Let $\mathcal{O}$ be any operator used in Figure 6, and $\mathfrak{C}$ any class of language families. Then* re-$\mathfrak{C}$ *is closed under $\mathcal{O}$ iff $\mathfrak{C}$ is closed under $\mathcal{O}$.*

Lemma 4.5. *Let $\mathfrak{C}$ be any class used in Figure 6. If $\mathfrak{C}$ is closed under $\oplus$ or $\bigoplus$, then $\mathfrak{C}$ is closed under complement.*

PROOF. Fix a class $\mathfrak{C}$ of language families solvable by small families of XFAs, where X represents any automata type corresponding to the classes in Figure 6, and assume that $\mathfrak{C}$ is closed under $\bigoplus$. Let $\mathcal{L} = (L_n)_{n\geq 1} \in \mathfrak{C}$. The language family $\bigoplus\mathcal{L}$ is solvable by a family $\mathcal{M} = (M_n)_{n\geq 1}$ of small XFAs. We need to prove that $\overline{\mathcal{L}} \in \mathfrak{C}$, i. e., to find a small family $\mathcal{M}' = (M'_n)_{n\geq 1}$ of XFAs solving $(\overline{L_n})_{n\geq 1}$.

Now we discuss how to construct an automaton M'_n solving $\overline{L_n}$ from an automaton M_n solving $\bigoplus L_n$. The case of $L_n = \emptyset$ is trivial, so assume that L_n is not empty and fix some $w \in L_n$. For any word u it holds that $u \notin L_n$ iff $\#w\#u\# \in \bigoplus L_n$. Hence, it is sufficient that M'_n, given an input u, simulates M_n on input $\#w\#u\#$. I. e., M'_n simulates M_n on every symbol of the input word except for the end-markers. On $\vdash$, M'_n simulates the behavior of M on $\vdash\#w\#$ and on $\dashv$, M'_n simulates M on $\#\dashv$. Such a simulation is possible for all considered types of automata by adding only a constant number of new states (in case of parallel automata, a constant number of new states in each component). Hence, this construction transforms k-state automata into $O(k)$ state automata, and thus the resulting automata family $\mathcal{M}'$ is small.

The proof for $\oplus$ is analogous; it is sufficient to consider the family $\mathcal{L} \oplus \mathcal{L}$ instead of $\bigoplus\mathcal{L}$. □

Lemma 4.6. *Let $\mathfrak{C}$ be any class used in Figure 6. Class* co-$\mathfrak{C}$ *is closed under $\overline{\cdot}$ (respectively, $\cdot^{\mathrm{R}}$, $\wedge$, $\vee$, $\oplus$, $\bigwedge$, $\bigvee$, $\bigoplus$) iff $\mathfrak{C}$ is closed under $\overline{\cdot}$ (respectively, $\cdot^{\mathrm{R}}$, $\vee$, $\wedge$, $\oplus$, $\bigvee$, $\bigwedge$, $\bigoplus$).*

PROOF. The claim for the complement is trivial. The claim for $\cdot^{\mathrm{R}}$ follows directly from Observation 4.2. The claim for $\wedge$, $\vee$, $\bigwedge$, and $\bigvee$ follows from Lemma 4.3, and the fact that all classes in Figure 6 are closed under union and intersection with any language family from 1D, which can be verified by straightforward constructions.

If $\mathfrak{C}$ is closed under $\oplus$ (respectively, $\bigoplus$), Lemma 4.5 implies that $\mathfrak{C} =$ co-$\mathfrak{C}$. Hence co-$\mathfrak{C}$ is also closed under $\oplus$ (respectively, $\bigoplus$). The same argument can be used to show that if co-$\mathfrak{C}$ is closed under $\oplus$ ($\bigoplus$), so is $\mathfrak{C}$. □

We omit some of the introduced complexity classes from Figure 6, namely RΔ, RN, $\cup_{\mathrm{R}}$D, $\cap_{\mathrm{L}}$D, $\cap_{\mathrm{R}}$D, and $\cap_2$D. The closure properties of all these classes follow from the closure properties presented in Figure 6: as we show later, RΔ = SΔ and RN = SN. Furthermore, the closure properties of any class co-$\mathfrak{C}$ and re-$\mathfrak{C}$ are related to the closure properties of $\mathfrak{C}$, as described by Observation 4.4 and Lemma 4.6 By Lemma 3.1(2,3), the closure properties of the omitted classes for parallel automata follow.

4.1. Closures

All of the closures in Figure 6 either can be proven by straightforward constructions or were proven before. Since none of these constructions is hard, we describe only their main ideas.

Complement. Any self-verifying class is trivially closed under complement. The closure of 2D under complement was proven in [21] and (an improved construction) in [5]. To prove the closure of 1D under complement, it is sufficient to make all non-final states final and vice versa.

Any rotating or sweeping automaton can be modified so as to avoid infinite runs: any computation of a RDFA (SDFA) with k states consists of at most k (left-to-right) traversals. Hence, the modified automaton can count the number of left-to-right traversals and reject if the upper limit is exceeded. Since a k-state automaton is transformed into an $O(k^2)$-state one, a small family of automata is transformed into a small family. The closure of RD and SD under complement follows by straightforward negation of the answer of the corresponding automata that avoid infinite runs.

Reverse. Any k-state two-way or sweeping automaton accepting language L can be transformed into a $(k+1)$-state automaton of the same kind accepting the language L^{R}: at first the new automaton moves its head to the right end-marker, and then simulates the original automaton with swapped directions of moves. For a two-sided parallel automaton $M = (\mathfrak{L}, \mathfrak{R}, F)$, it is enough to swap $\mathfrak{L}$ with $\mathfrak{R}$.

The closure of 1N under reverse follows from the well-known argument of [19, Theorem 12] for the closure of the class of regular languages under reverse. Here, the core idea is that a 1NFA "guesses" the computation of the simulated automaton backwards and, in every step, verifies if the guess is correct. For the variant of 1NFAs without end-markers, it has been proven in [6, 10] that any 1NFA with k states accepting L can be converted into a 1NFA with $k+1$ states accepting L^{R} and that this bound is tight. In our definition of one-way automata (i. e., the variant with end-markers, which is equivalent to the definition used in [19]), any 1NFA can be reversed with no increase in the number of states.

The closure of 1Δ under reverse follows from the closure of 1N and Observation 4.2.

Parallel Automata. Parallel union automata are closed under both $\vee$ and $\bigvee$: if M_1 and M_2 are parallel union automata for languages L_1, L_2, we can modify all components of M_1 to consider only the left #-delimited block and all components of M_2 to consider only the right #-delimited block. Such transformation can be done by adding only a constant number of states to each component of M_1 and M_2. Afterwards, the union of all components of M_1 and M_2 forms the automaton for $L_1 \vee L_2$. The size of the newly-formed automaton is linear in the sizes of M_1 and M_2. For $\bigvee L_1$, it is sufficient to modify every component of M_1 to work on all #-delimited blocks separately and exit into the final state iff the original component exited into the final state on at least one block. Such a transformation adds only a constant number of states to each component of M_1, hence the size of the automaton for $\bigvee L_1$ is linear in the size of M_1.

Proving that $\cup_{\mathrm{l}}$D is closed under $\wedge$ is only slightly more involved. Let $M_1 = (\mathfrak{L}_1, \emptyset)$ and $M_2 = (\mathfrak{L}_2, \emptyset)$ be $\cup_{\mathrm{l}}$DFAs with at most k states accepting languages L_1, L_2. We construct a $\cup_{\mathrm{l}}$DFA M' accepting language $L_1 \wedge L_2$ with at most k^2 components, each

with $O(k)$ states: for every pair $A_1 \in \mathfrak{L}_1$, $A_2 \in \mathfrak{L}_2$, there is one component B of M' that simulates A_1 on the first #-delimited block of input word and A_2 on the second one. Component B exits into a final state iff both A_1 and A_2 do. It is easy to see that B can be constructed with $O(k)$ states. Furthermore, for every $x \in L_1$ and $y \in L_2$, some component of M' accepts #x#y#: the component simulating $A_1 \in \mathfrak{L}_1$ that accepts x and $A_2 \in \mathfrak{L}_2$ that accepts y does. Conversely, assume that some component B of M' accepts #x#y#. By definition of M', component B simulates some $A_1 \in \mathfrak{L}_1$ on x and some $A_2 \in \mathfrak{L}_2$ on y. Since B accepts only when both simulated components accept, we have that $x \in L_1$ and $y \in L_2$, thus, the construction is correct.

Remaining Closure Properties. It remains to show the closures of rows 3–8, columns A–C,E,G–L of Figure 6, i. e., the closure properties of one-way, rotating, sweeping and two-way automata under $\wedge$, $\vee$, $\oplus$, $\bigwedge$, $\bigvee$, $\bigoplus$. All these properties can be proved using straightforward constructions based on the same idea: the newly constructed automaton simulates the original automaton/automata on each #-delimited block of the input word separately and decides according to the results of this simulation. To do so, the new automaton needs enough head freedom.

Both one-way and two-way automata can simulate automata of the same type on each #-delimited block of the input word, regardless of the number of such blocks. Rotating and sweeping automata, however, are not able to do so for inputs consisting of arbitrarily many blocks, since they cannot "remember" the position of the processed block and return to it in another traversal. Nevertheless, if the input consists of two blocks only, they are able to simulate another automaton on the left (or right) block only. Hence, rotating and sweeping automata have sufficient head freedom for proving closures under $\wedge$, $\vee$, and $\oplus$ only, while one-way and two-way automata have sufficient freedom also for $\bigwedge$, $\bigvee$, and $\bigoplus$.

The ability to simulate an automaton of the same kind on each #-delimited block is enough to prove the closures in rows 3,6 and columns A–C,E,G–L in Figure 6, i. e., the closure properties related to $\wedge$ and $\bigwedge$. The same technique can be applied in a straightforward way for properties related to $\vee$ and $\bigvee$ if the simulated automaton never reaches an infinite loop. Infinite loops can be always avoided with polynomial blowup in size complexity for one-way automata (trivially), rotating and sweeping automata (as we have discussed in the paragraph about the complement, moreover, the argument can be extended for the case of nondeterministic automata as well, because a shortest accepting computation, if there exist some, consists of at most k left-to-right traversals), and for two-way deterministic automata [21]. In this way, it is possible to obtain a proof for the closures in rows 4,7 and columns A–C,E,G–J in Figure 6. Hence, only the case of 2N and 2Δ remains to be considered: a 2NFA accepting $\bigvee L$ can nondeterministically choose one block of the input word and simulate the corresponding 2NFA accepting L. Here, a possible infinite loop of the simulated machine does not pose a problem. A two-way self-verifying automaton accepting $\bigvee L$ can either verify that the input word is in the accepted language in the same way as 2NFA, or verify that the word is not in the language in the same way as a 2NFA for $\bigwedge \overline{L}$. An analogous construction works also for $L_1 \vee L_2$.

To use our technique for closure properties under $\oplus$ and $\bigoplus$, the new automaton needs the ability to negate the answer of the simulated automaton. This is trivial for deterministic classes and can be done easily for all classes closed under complement: the new automaton simulates both the original automaton and the complemented original

automaton on every block. Furthermore, the possibility of infinite loops does not pose a problem for two-way self-verifying automata, since the new automaton can, for every block of the input, nondeterministically guess if this block is in the corresponding language and verify this guess by running the simulated machine. Hence, the closure properties in rows 5,8 and columns A–C,E,G–L are correct, too.

4.2. Equalities

Now we are ready to augment the map in Figure 3 with classes co-X and re-X for each class X and draw a new map that takes into account equalities between these classes. To do so, we use the closures in Figure 6: 1D = co-1D, 1Δ = co-1Δ = re-1Δ, 1N = re-1N, RD = co-RD, SD = co-SD = re-SD, SΔ = co-SΔ = re-SΔ, SN = re-SN, 2D = co-2D = re-2D, 2Δ = co-2Δ = re-2Δ, and 2N = re-2N. Furthermore, we use the following lemma to show that RN = SN and hence also RΔ = SΔ.

Lemma 4.7. *Every k-state* SNFA *has an equivalent $O(k^3)$-state* RNFA*.*

PROOF. Let $M = (q_a, \delta, q_s)$ be a SNFA over a set of k states Q solving the language L over the alphabet Σ. We construct an $O(k^3)$-state RNFA M' solving L in the following way: every left-to-right traversal of M is simulated by M' in a straightforward way. To simulate a right-to-left traversal of M, the automaton M' guesses the computation of M backwards, in a similar way as in the proof that 1N is closed under reverse [19]. In doing so, M' needs to remember the starting state of the right-to-left traversal (which is checked at the end of the traversal simulation), and the guessed last state of the right-to-left traversal (from which the next left-to-right traversal is started).

More formally, $M' = (q_s, \delta', q_a)$ is an automaton over set of states

$$Q' = Q \cup (Q \times Q \times Q)$$

such that δ' is defined as follows:[2]

$$\begin{array}{rcll}
\delta'(q,a) & = & \delta(q,a) & \forall q \in Q, \forall a \in \{\vdash\} \cup \Sigma \\
\delta'(q,\dashv) & \ni & q_a & \forall q \in Q, \delta(q,\dashv) \ni q_a \\
\delta'(q,\dashv) & \ni & (q,q',q') & \forall q,q' \in Q \\
\delta'((q,p,q'),a) & \ni & (q,p',q') & \forall q,p,q',p' \in Q, \forall a \in \Sigma, \delta(p',a) \ni p \\
\delta'((q,p,q'),\dashv) & = & \delta(q',\vdash) & \forall q,p,q' \in Q, \delta(q,\dashv) \ni p \\
\delta'((q,p,q'),\dashv) & = & \emptyset & \text{otherwise}
\end{array}$$

States from Q are used for simulation of left-to-right traversals. At the end of the traversal, the current state is saved as the first component of the state (q,q',q'), and the state at the end of the right-to-left traversal q' is guessed nondeterministically. Afterwards, the computation of M is guessed nondeterministically backwards. Hence, $\text{LCOMP}_{M',(q,q',q')}(z)$ can exit into (q,p,q') iff $\text{RCOMP}_{M,p}(z)$ can exit into q'. So, M' can avoid hanging at $\dashv$ iff M could traverse the input word from right to left starting at state q, continuing with p, and exiting into q'. In that case, M' simulates the movement of M on $\vdash$ and continues with the simulation of a left-to-right traversal. □

[2]More precisely, we define $\delta'(q,a)$ as the set containing only those elements required by the definition.

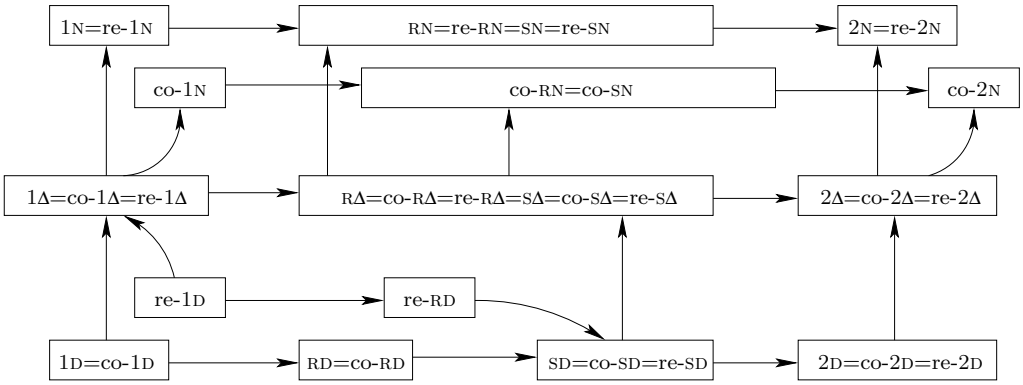

Figure 7: Size complexity classes. An arrow $\mathfrak{C} \to \mathfrak{C}'$ means that $\mathfrak{C} \subseteq \mathfrak{C}'$.

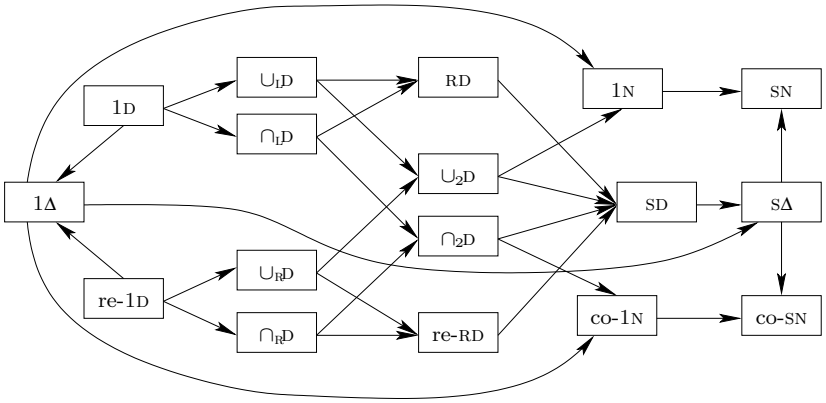

Figure 8: Zoom to the parallel automata classes. An arrow $\mathfrak{C} \to \mathfrak{C}'$ means that $\mathfrak{C} \subseteq \mathfrak{C}'$. Note that all the co- and re- classes corresponding to parallel automata are already included in the map by Lemma 3.1.

The combination of the results presented in Figures 6 and 3, together with Lemma 4.7, yields the map of the complexity classes depicted in Figure 7. After adding the classes of union and intersection parallel automata to the figure, we obtain Figure 8; there we have omitted the classes of two-way automata.

The correctness of the inclusions between one-way deterministic, parallel, and rotating automata in Figure 8 follows from Lemma 3.1 and from Observation 4.1. The inclusions RD $\cup$ re-RD $\subseteq$ SD $\subseteq$ 2D are very easy to verify. Since 1D is closed under complement and every 1DFA is a special case of a 1NFA, we have that 1D $\subseteq$ 1Δ. Furthermore, $\cup_2$D $\subseteq$ 1N: any $\cup_2$DFA can be simulated by a 1NFA that nondeterministically selects one of the components of the parallel automaton and simulates it. This gives also $\cap_2$D $\subseteq$ co-1N, where we verify if one of the components rejects. Because every 1DFA is also a 1NFA and every 1NFA can be reversed without any increase of the size complexity, this construction transforms a small family of $\cup_2$DFAs into a small family of 1NFAs. The remaining relationships follows from Observation 4.1, because 1Δ is closed under complement.

	re-1D	$\cup_L$D	$\cup_R$D	$\cap_L$D	$\cap_R$D	$\cup_2$D	RD	re-RD	$\cap_2$D	1Δ	1N	co-1N	SD	SΔ	SN	co-SN
1D	I	$\subsetneq$	I	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
re-1D		I	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
$\cup_L$D			I	I	I	$\subsetneq$	$\subsetneq$	I	I	I	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
$\cup_R$D				I	I	$\subsetneq$	I	$\subsetneq$	I	I	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
$\cap_L$D					I	I	$\subsetneq$	I	$\subsetneq$	I	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
$\cap_R$D						I	I	$\subsetneq$	$\subsetneq$	I	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
$\cup_2$D							I	I	I	I	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
RD								I	I	I	I	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
re-RD									I	I	I	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
$\cap_2$D										I	I	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$
1Δ											$\subsetneq$	$\subsetneq$	I	$\subsetneq$	$\subsetneq$	$\subsetneq$
1N												I	I	I	$\subsetneq$	I
co-1N													I	I	I	$\subsetneq$
SD														$\subsetneq$	$\subsetneq$	$\subsetneq$
SΔ															$\subsetneq$	$\subsetneq$
SN																I

Figure 9: Relationship between classes: '$\subsetneq$' means that the class represented by the row is strictly included in the class represented by the column and 'I' means that the classes are incomparable.

4.3. Separations

In this section, we separate some of our complexity classes and obtain a complete map of relationship between introduced classes, as shown in Figure 9. All of the facts in Figure 9 follow trivially from Figure 10. Hence, our focus is to show that all facts in Figure 10 are correct.

At first, we observe that all inclusions (marked by '+') in Figure 10 follow by taking a transitive closure of Figure 8:

Observation 4.8. *All symbols '+' in Figure 10 are correct.*

Next, we prove all non-inclusions in Figure 10 that are marked by thick black frame by using the technique of hardness propagation. These non-inclusions are depicted also in Figure 11.

We apply the hardness propagation on the core language family $\mathcal{J} = (J_n)_{n\geq 1}$ where

$$J_n := \{\alpha i \mid \alpha \subseteq [n] \text{ and } i \in \alpha\}, \tag{9}$$

where $[n] := \{1, \ldots, n\}$. Note that every language J_n consists of words of length two; the first symbol of every word is a subset of $[n]$ and the second symbol is an element of $[n]$. The basic membership properties of $\mathcal{J}$ are stated in the following lemma:

Lemma 4.9. $\mathcal{J} := (J_n)_{n\geq 1}$ *is not in* 1D *but is in* re-1D, 1N, co-1N, $\cap_L$D, $\cup_L$D.

Proof. Any 1DFA solving J_n needs at least 2^n states: let M be any 1DFA solving J_n. For each $\alpha \subseteq [n]$, consider the state q_α that is reached by J_n after reading $\vdash\alpha$. For any $\alpha \neq \alpha'$, it holds that $q_\alpha \neq q_{\alpha'}$: if this is not the case, then $q_\alpha = q_{\alpha'}$ for some $\alpha \neq \alpha'$.

	1D	re-1D	$\cup_L$D	$\cup_R$D	$\cap_L$D	$\cap_R$D	$\cup_2$D	RD	re-RD	$\cap_2$D	1Δ	1N	co-1N	SD	SΔ	SN	co-SN
1D	+	−	+	−	+	−	+	+	[−]	+	+	+	+	+	+	+	+
re-1D	−	+	−	+	−	+	+	**−**	+	+	+	+	+	+	+	+	+
$\cup_L$D	−	−	+	−	−	−	+	+	−	**−**	−	+	[−]	+	+	+	+
$\cup_R$D	−	−	−	+	−	−	+	−	+	[−]	−	+	[−]	+	+	+	+
$\cap_L$D	−	−	−	−	+	−	[−]	+	−	+	−	**−**	+	+	+	+	+
$\cap_R$D	−	−	−	−	−	+	[−]	−	+	+	−	[−]	+	+	+	+	+
$\cup_2$D	−	−	−	−	−	−	+	−	−	−	−	+	−	+	+	+	+
RD	−	−	−	−	−	−	−	+	−	−	−	−	−	+	+	+	+
re-RD	−	−	−	−	−	−	−	−	+	−	−	−	−	+	+	+	+
$\cap_2$D	−	−	−	−	−	−	−	−	−	+	−	−	+	+	+	+	+
1Δ	−	−	−	−	−	−	−	−	−	−	+	+	+	**−**	+	+	+
1N	−	−	−	−	−	−	−	−	−	−	−	+	−	−	−	+	**−**
co-1N	−	−	−	−	−	−	−	−	−	−	−	−	+	−	−	[−]	+
SD	−	−	−	−	−	−	−	−	−	−	−	−	−	+	+	+	+
SΔ	−	−	−	−	−	−	−	−	−	−	−	−	−	−	+	+	+
SN	−	−	−	−	−	−	−	−	−	−	−	−	−	−	**−**	+	−
co-SN	−	−	−	−	−	−	−	−	−	−	−	−	−	−	[−]	−	+

Figure 10: Inclusions between classes: '+' means that the class represented by the row is (maybe non-strictly) included in the class represented by the column and '−' means that it is not. Symbols '−' with a thick black frame are the core separations proven by hardness propagation, those with a thick white frame follow from Observation 4.1(5,6).

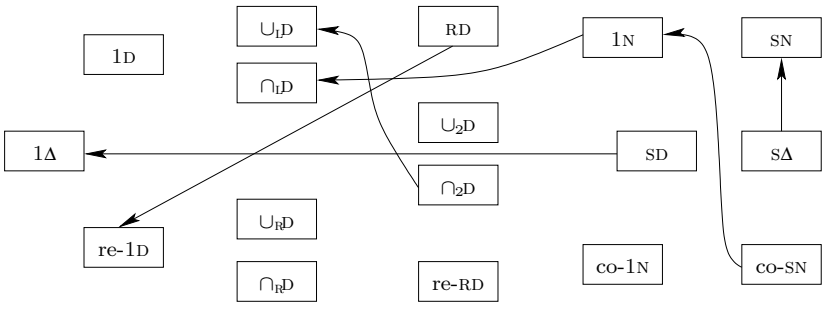

Figure 11: Separations of the complexity classes. An arrow $\mathfrak{C} \to \mathfrak{C}'$ means that $\mathfrak{C} \not\supseteq \mathfrak{C}'$.

Without loss of generality we can assume that $i \in \alpha$ but $i \notin \alpha'$ for some $i \in [n]$, thus exactly one of the words αi, $\alpha' i$ is in J_n. But M can not distinguish between these words; it either accepts both or rejects both, a contradiction to the correctness of M.

On the other hand, J_n can be solved by a 1NFA with $n+2$ states (by guessing i at first and verifying), by a $\cup_{\text{L}}$DFA with n components of 4 states each, and by a $\cap_{\text{L}}$DFA with n components of 4 states each (both the $\cup_{\text{L}}$DFA and the $\cap_{\text{L}}$DFA use one component for every possible guess of i). Also, $(J_n)^{\text{R}}$ can be solved by a 1DFA with $n+2$ states and $\overline{J_n}$ can be solved by a 1NFA with $n+2$ states. □

Now we are ready to prove all results in Figure 11:

Lemma 4.10. *All non-inclusions in Figure 10 marked by thick black frame (i. e., those in Figure 11) are correct.*

PROOF.
- RD $\not\supseteq$ re-1D: $\mathcal{L} := \bigwedge\bigvee\mathcal{J}$ is the witness. By Lemma 4.9, $\mathcal{J} \notin$ 1D, so Corollary 3.4 yields that $\bigvee\mathcal{J} \notin \cap_{\text{L}}$D and Corollary 3.13 ensures that $\mathcal{L} = \bigwedge\bigvee\mathcal{J} \notin$ RD. Since $\mathcal{J}^{\text{R}} \in$ 1D (Lemma 4.9), $\bigvee\mathcal{J}^{\text{R}} \in$ 1D (A7 of Figure 6), hence A6 of Figure 6 yields that $\bigwedge\bigvee\mathcal{J}^{\text{R}} \in$ 1D and, by Observation 4.2, $\mathcal{L} = \bigwedge\bigvee\mathcal{J} \in$ re-1D.
- $\cap_2$D $\not\supseteq$ $\cup_{\text{L}}$D: $\mathcal{L} := (\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J})^{\text{R}}$ is the witness. Since $J \notin$ 1D, we have, by Corollary 3.4, that $\bigvee\mathcal{J} \notin \cap_{\text{L}}$D, and hence, by Corollary 3.6, that $\mathcal{L} \notin \cap_2$D. Since $\mathcal{L} = (\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J}^{\text{R}})$ (Observation 4.2), $\mathcal{J}^{\text{R}} \in$ 1D $\subseteq \cup_{\text{L}}$D (Lemma 4.9, Figure 8), and $\mathcal{J} \in \cup_{\text{L}}$D (Lemma 4.9), D7 and D4 of Figure 6 yields that $\mathcal{L} \in \cup_{\text{L}}$D.
- SD $\not\supseteq$ 1Δ: $\mathcal{L} := \bigwedge\big((\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J})^{\text{R}}\big)$ is the witness. Since $(\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J})^{\text{R}} \notin \cap_2$D, applying Corollary 3.15 yields that $\mathcal{L} \notin$ SD. Since $(\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J})^{\text{R}} \in \cup_{\text{L}}$D $\subseteq$ 1N, C6 of Figure 6 yields that $\mathcal{L} \in$ 1N. It remains to show that $\mathcal{L} \in$ co-1N. Since 1N is closed under union and intersection with any family from 1D, we can use Lemma 4.9, Lemma 4.3, Observation 4.2, and C2, C3, C6, C7 of Figure 6 to derive the following: $\overline{\mathcal{J}} \in$ 1N, $\bigwedge\overline{\mathcal{J}} \in$ 1N, $\overline{\bigvee\mathcal{J}} \in$ 1N, $\overline{\bigvee\mathcal{J}^{\text{R}}} \in$ 1N, $\overline{\bigvee\mathcal{J}^{\text{R}}} \in$ 1N, $\overline{\bigvee\mathcal{J}} \wedge \overline{\bigvee\mathcal{J}^{\text{R}}} \in$ 1N, $\overline{(\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J}^{\text{R}})} \in$ 1N, $\bigvee\overline{(\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J}^{\text{R}})} \in$ 1N, $\overline{\mathcal{L}} = \overline{\bigwedge\big((\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J}^{\text{R}})\big)} \in$ 1N.
- 1N $\not\supseteq \cap_{\text{L}}$D: language family $\mathcal{D} = (D_n)_{n\geq 1}$, representing the *disjointness problem*, defined as
$$D_n := \{\alpha\beta \mid \alpha, \beta \subseteq [n], \alpha \cap \beta = \emptyset\}, \tag{10}$$
witnesses this separation. (Again, note that D_n consists of words of length 2 only.) To prove that any 1NFA M solving D_n needs at least 2^n states, consider accepting computations of M on words from the set $\{\alpha\overline{\alpha}\}$, where $\alpha \subseteq [n]$ and $\overline{\alpha} := [n] \setminus \alpha$. If M has less than 2^n states, there exist $\alpha_1 \neq \alpha_2$ such that M is in the same state when reading the second symbol of the word $\alpha_1\overline{\alpha_1}$ and of the word $\alpha_2\overline{\alpha_2}$. Without loss of generality, we may assume that α_1 and α_2 differ in element x, more precisely that $x \in \alpha_1$ and $x \notin \alpha_2$. Then M accepts the word $\alpha_1\overline{\alpha_2}$, which is not in D_n, since $x \in \alpha_1 \cap \overline{\alpha_2}$. Furthermore, it is not difficult to construct a $\cap_{\text{L}}$DFA with n components of 4 states each that solves D_n: for each $i \in [n]$, there is one component that verifies if $i \notin \alpha$ or $i \notin \beta$ for the input word $\alpha\beta$.
- co-SN $\not\supseteq$ 1N: proven in [12].
- S$\Delta \not\supseteq$ SN: since SΔ is closed under complement and SN is not [12], S$\Delta \neq$ SN. Since S$\Delta \subseteq$ SN, the claim follows. □

Applying Observation 4.1(5,6) on the results of Lemma 4.10 yields more non-inclusions:

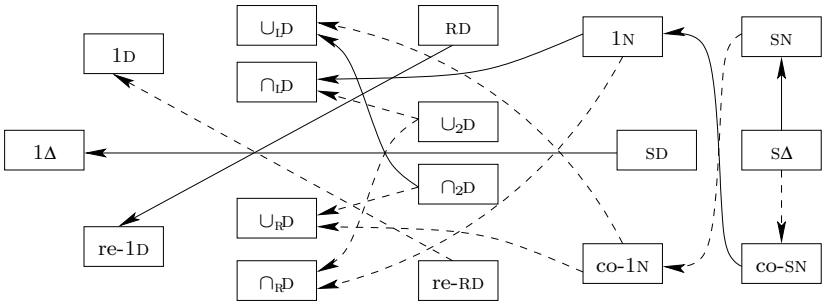

Figure 12: Separations of the complexity classes, including the relationships derived by Observation 4.1. An arrow $\mathfrak{C} \to \mathfrak{C}'$ means that $\mathfrak{C} \not\supseteq \mathfrak{C}'$. Full arrows are those presented in Figure 11, dashed arrows are derived by Observation 4.1 and Lemma 3.1.

Observation 4.11. *All non-inclusions in Figure 10 marked by thick white frame (i. e., those in Figure 12 marked by gray) are correct.*

We can now prove the correctness of the whole Figure 10.

Theorem 4.12. *All facts in Figure 10 are correct.*

Proof. The correctness of all '+' follows from Observation 4.8. We have proven the correctness of all '−' in thick black frame in Lemma 4.10, and the correctness of all '−' in thick white frame follows from Observation 4.11.

The correctness of all remaining '−' follow from these facts and Observation 4.1(7). In the context of Figure 10, it is possible to verify that $\mathfrak{C}_1 \not\subseteq \mathfrak{C}_2$ (i. e., that the '−' in row $\mathfrak{C}_1$ and column $\mathfrak{C}_2$ is correct) as follows: Consider *row* $\mathfrak{C}_2$, follow the row to find some column $\mathfrak{C}_2'$ with '+', follow the column to find some row $\mathfrak{C}_1'$ with '−' in a thick frame (black or white), and check if the *column* $\mathfrak{C}_1$ contains '+'. If such $\mathfrak{C}_2'$, $\mathfrak{C}_1'$ exist, Observation 4.1(7) ensures that $\mathfrak{C}_1 \not\subseteq \mathfrak{C}_2$. It is straightforward to verify that such $\mathfrak{C}_1'$, $\mathfrak{C}_2'$ exist for all '−' without thick frame. □

So far, we have proven a complete characterization of the relationship of all introduced complexity classes, except of those corresponding to the two-way automata (Figure 9). It is a long-standing open problem if 2D ≠ 2N, but we can at least separate two-way and sweeping automata: It was proven in [11] that SN ⊉ 2D. Combining this result with SD ⊆ SΔ ⊆ SN, 2D ⊆ 2Δ ⊆ 2N, and Observation 4.1(7) implies:

Observation 4.13. SD ⊊ 2D, SΔ ⊊ 2Δ, *and* SN ⊊ 2N.

4.4. Non-closures

Now we prove the correctness of all non-closures of Figure 6.

Reverse and Complement. All non-closures under reverse and complement follow directly from Theorem 4.12.

Parity Operators. All non-closures under $\oplus$ and $\bigoplus$ (i. e., in rows 5,8 of Figure 6) follow from the non-closures under complement. Indeed, Lemma 4.5 implies that any considered class of finite automata that is not closed under complement is not closed neither under $\oplus$ nor under $\bigoplus$.

Remaining Non-closures. The remaining properties follows from the previous results:

- **F6:** by Theorem 4.12, there exists $\mathcal{L} \in \cup_2\text{D} \setminus \cap_2\text{D}$. Corollary 3.15 yields that $\bigwedge\mathcal{L} \notin \text{SD}$, hence $\bigwedge\mathcal{L} \notin \cup_2\text{D}$.
- **D6:** by Lemma 4.6 and Lemma 3.1, this is equivalent to proving that $\cap_1\text{D}$ is not closed under $\bigvee$. By Theorem 4.12, there exists $\mathcal{L} \in \cap_1\text{D} \setminus \text{1D}$. Corollary 3.4 yields that $\bigvee\mathcal{L} \notin \cap_1\text{D}$.
- **F3:** as in D6, it is sufficient to prove that $\cap_2\text{D}$ is not closed under $\vee$. By Theorem 4.12, there exists $\mathcal{L} \in \cap_2\text{D} \setminus \cap_1\text{D}$. Since $\cap_2\text{D}$ is closed under reverse, $\mathcal{L}^\text{R} \in \cap_2\text{D}$. By Corollary 3.6, $\mathcal{L} \vee \mathcal{L}^\text{R} \notin \cap_2\text{D}$.
- **E6, G6:** by Theorem 4.12, there exists $\mathcal{L} \in \text{RD} \setminus \cap_1\text{D}$ (respectively, $\mathcal{L} \in \text{SD} \setminus \cap_2\text{D}$). By Corollary 3.13 (respectively, 3.15), it holds that $\bigwedge\mathcal{L} \notin \text{RD}$ (respectively, $\bigwedge\mathcal{L} \notin \text{SD}$).
- **E7, G7:** since RD and SD are closed under complement, the claim follows from E6, G6 and Lemma 4.6.

5. Randomized Models

So far, we have focused on deterministic and nondeterministic finite automata. It is possible to define also their randomized variants, in a similar way as for Turing machines. In this section, we relate the complexity classes of randomized finite automata to the non-randomized classes and provide some related results.

Essentially, a randomized automaton (sometimes called also probabilistic automaton) is just a nondeterministic automaton that, in each step, instead of applying nondeterminism, picks one of the possible choices according to some probability distribution: Consider a nondeterministic automaton over a set of states Q and alphabet Σ. The transition function δ of this automaton partially maps $Q \times (\Sigma \cup \{\vdash, \dashv\})$ to the set of all possible subsets of feasible actions A. The action of a one-way, rotating, or sweeping automaton is completely described by the new state, hence $A = Q$ in this case. For two-way automata, the action consists of the new state and the movement, hence $A = Q \times \{-1, 0, 1\}$ for two-way automata. The transition function δ of a randomized machine totally maps $Q \times (\Sigma \cup \{\vdash, \dashv\})$ to the set of all *probability distributions* over $A \cup \{\bot\}$, i. e., all total functions from $A \cup \{\bot\}$ to the real numbers that obey the axioms of probability. Hence, on any input word $z \in \Sigma^*$, the computation of M on z is a *probability distribution* over all possible computations. The expected length of a computation drawn from this distribution is called the *expected running time* of M on z. This way of defining a randomized automaton applies to one-way, rotating, sweeping, as well as two-way automata.

It remains to define which words are accepted by a randomized automaton. There are several ways of doing that, yielding several different types of randomized automata. We discuss these types and their respective complexity classes in the following.

5.1. Las Vegas Automata

The most restrictive setting, called *Las Vegas* randomization, is to require zero probability of error. A *Las Vegas* automaton M has a special *reject* state $q_r \in Q$ in addition to the accept state q_a. If M reaches q_r after reading $\dashv$, the computation of M halts, and we say that M *rejects* the input word. Besides accepting, M can also hang or run forever; in these cases, the M neither accepts nor rejects the input. An input word $z \in \Sigma^*$ is in the language $L(M)$ if M accepts z with probability at least $1/2$ and rejects z with probability 0. Furthermore, we require that every $z \in \Sigma^*$ not in $L(M)$ is accepted by M with probability 0 and rejected with probability at least $1/2$.

The definition of LasVegas automata applies to one-way ($1\mathrm{P}_0\mathrm{FA}$), rotating ($\mathrm{RP}_0\mathrm{FA}$), sweeping ($\mathrm{SP}_0\mathrm{FA}$), as well as two-way automata ($2\mathrm{P}_0\mathrm{FA}$). We denote the complexity classes induced by small families of these automata as $1\mathrm{P}_0$, $\mathrm{RP}_0\mathrm{X}$, $\mathrm{SP}_0\mathrm{X}$, and $2\mathrm{P}_0\mathrm{X}$, respectively. We use the symbol X to emphasize that there is no restriction on the expected running time of the automata except that we require it to be finite for all inputs. The expected running time can be even exponential in the length of the input word, but it can not be superexponential (see Lemma 5.1). To capture the concept of efficient computability, it is more natural to restrict the running time to polynomial. We reserve the notation RP_0, SP_0, and $2\mathrm{P}_0$ to such classes.

Lemma 5.1. *Let M be any two-way randomized automaton such that the expected running time of M is finite for all inputs. There exists some α such that expected running time of M on any input word of length n is $O(\alpha^n)$. The same claim holds for sweeping and rotating automata as well.*

Proof. Assume that M have k states and is given an input of length n. There are $(n+2)k$ different configurations of M. We say that a configuration C is *terminal* if M ends its computation when reaching C (i. e., it either accepts or rejects the input).

Consider any configuration C that is reachable with non-zero probability from the initial configuration of M. There must exist some terminal configuration of M that is reachable from C with non-zero probability: otherwise, all computation paths beginning in C would run in an infinite loop, what contradicts to the fact that the expected running time of M is finite. Furthermore, there must be some terminal configuration reachable from C within at most $(n+2)k$ steps of computation: if we consider the terminal configuration C_t reachable with the smallest number of steps, each configuration of M can occur at most once before reaching C_t. The probability of doing every particular step of the shortest computation from C to C_t is at least p, where p is the minimal non-zero probability in the transition function of M. Hence, the probability that M reaches C_t from C in at most $(n+2)k$ steps is at least $p^{(n+2)k}$.

Thus, at any step of the computation of M it holds that M reaches some terminal configuration with probability at least $p^{(n+2)k}$ within the next $(n+2)k$ steps. Hence, the expected running time of M is at most

$$\sum_{i=1}^{\infty} i(n+2)k \cdot p^{(n+2)k} \cdot \left(1 - p^{(n+2)k}\right)^{i-1} = (n+2)k \left(\frac{1}{p}\right)^{(n+2)k} = O\left(\alpha^n\right)$$

for any $\alpha > (1/p)^k$. The claim for the sweeping and the rotating automata follows directly from the claim for two-way automata. □

In the rest of this section, we locate the LasVegas classes in the map of Figure 7.

Theorem 5.2.

$$1\mathrm{P}_0 = 1\mathrm{D}, \quad 2\mathrm{P}_0\mathrm{X} = 2\Delta, \quad \mathrm{SP}_0\mathrm{X} = \mathrm{S}\Delta, \quad \mathrm{RP}_0\mathrm{X} = \mathrm{R}\Delta$$

PROOF. It was proven in [8] that for any k-state 1P$_0$FA there exists an equivalent $O(k^2)$-state 1DFA. Hence, $1\mathrm{P}_0 = 1\mathrm{D}$.

Theorem 1 in [9] implies that $2\mathrm{P}_0\mathrm{X} = 2\Delta$. The main idea behind this equivalence is as follows: any language L accepted by a k-state 2P$_0$FA can be accepted by a k-state 2NFA, since any LasVegas automaton can be converted into a nondeterministic one by simply replacing all transitions with non-zero probability by nondeterministic ones. Any k-state 2P$_0$FA accepting L can be transformed into a k-state 2P$_0$FA accepting $\overline{L}$ by simply swapping the accept and the reject state. Hence, $\overline{L}$ can be accepted by a k-state 2NFA, too, and $2\mathrm{P}_0\mathrm{X} \subseteq 2\Delta$. The other direction uses the same idea as [17]: consider a language L such that both L and $\overline{L}$ can be accepted by a small 2NFA. Denote these 2NFAs as M_1 and M_2. Then it is possible to construct a small 2P$_0$FA M accepting L as follows: M simulates M_1, but after every computation step, it tosses a coin. If the result is heads, then it continues the computation; if it is tails, it restarts the simulation. After a restart, it proceeds to the simulation of M_2. Analogously, during the simulation of M_2, another simulation of M_1 is started with probability $1/2$ after each simulated step, etc. If M finds an accepting computation of M_1 (M_2), it accepts (rejects). Obviously M never errs. Since every input word z is accepted by either M_1 or M_2, automaton M eventually finds the accepting computation in M_1 or M_2. After every restart of a simulation of the correct parity (i. e., a simulation of M_1 if the input is in L and a simulation of M_2 otherwise), there is a non-zero (albeit exponentially small) probability that the input word is accepted. Hence, the expected running time of the constructed LasVegas automaton is finite (although exponential).

The above-described idea works also for sweeping and rotating automata. Hence, we have that $\mathrm{SP}_0\mathrm{X} = \mathrm{S}\Delta$ and $\mathrm{RP}_0\mathrm{X} = \mathrm{R}\Delta$. □

Hence, our previous results imply an exponential gap in the number of states between determinism and LasVegas randomization for sweeping and rotating automata: by Figures 7 and 10, combined with the theorem above, we have $\mathrm{RD} \subsetneq \mathrm{S}\Delta = \mathrm{R}\Delta = \mathrm{RP}_0\mathrm{X}$ and $\mathrm{SD} \subsetneq \mathrm{S}\Delta = \mathrm{SP}_0\mathrm{X}$.

5.2. Monte-Carlo Automata

It is possible to obtain more powerful automata by relaxing the condition of the zero probability of error. A *Monte-Carlo* automaton M with one-sided error is required to obey the following constraint: any $z \in \Sigma^*$ is either accepted by M with probability at least $1/2$ or accepted with probability 0. The automaton M does not need any special rejecting state. The language recognized by M consists of all words accepted with nonzero probability.

We denote the one-way, rotating, sweeping, and two-way Monte-Carlo automata as 1P$_1$FA, RP$_1$FA, SP$_1$FA, and 2P$_1$FA, respectively. Furthermore, we denote the corresponding complexity classes as 1P$_1$, RP$_1$X, SP$_1$X, and 2P$_1$X. As in the case of LasVegas automata, the

symbol X denotes that there is no restriction on the expected running time of the automata except that it is always finite. Due to Lemma 5.1, this is equivalent to restricting the expected running time to be at most exponential.

Any Monte-Carlo automaton can be trivially simulated by a nondeterministic one. Hence, it holds that $1P_1 \subseteq 1N$, $RP_1X \subseteq RN$, $SP_1X \subseteq SN$, and $2P_1X \subseteq 2N$. The result of [17] directly proves that, for any k-state 2NFA, there exists an equivalent $O(k)$-state $2P_1FA$. Furthermore, the same idea can be used for sweeping and rotating automata, which implies the following observation ($SN = RN$ was proven in Lemma 4.7):

Observation 5.3.

$$2P_1X = 2N, \quad SP_1X = SN = RN = RP_1X$$

5.3. Bounded-Error Automata

As a next step, it is possible to relax the constraint of one-sided error and define *bounded-error* finite automata, analogously to the bounded-error randomized Turing machines. For any $z \in \Sigma^*$, a bounded-error automaton M is required to accept z either with probability at least $2/3$ or with probability at most $1/3$. The language recognized by M consists of all words accepted with probability at least $2/3$. Although bounded-error automata are not a trivial superclass of Monte-Carlo automata, any Monte-Carlo automaton can be transformed into an equivalent bounded-error one with no increase in the number of states.

We denote the one-way, rotating, sweeping, and two-way bounded-error automata as $1P_2FA$, RP_2FA, SP_2FA, and $2P_2FA$, respectively. These automata are very powerful. As implied by [4], even RP_2FA can accept non-regular languages. To obtain a more fair comparison of the power of bounded-error randomization with other modes, we define the corresponding complexity classes $1P_2$, RP_2X, SP_2X, and $2P_2X$ to contain regular languages only [13]. Again, the symbol X indicates that there is no restriction on the expected running time of the automata except that it is always finite, i. e., that the expected time is at most exponential.

Restricting the bounded-error complexity classes to regular languages only allows us to analyze the complexity aspects of the bounded-error randomization. Without this restriction, we could easily separate these classes from the Monte-Carlo complexity classes by the result of [4], i. e., by an argument based on the computational power of bounded-error randomization. By doing so, however, we would have missed all complexity phenomena involved in this comparison, because they were screened from us by the computability phenomenon.

The result of [3, Theorem 6.2] implies that $2P_2X \supsetneq 2N = 2P_1X$. This result relies on a possibility of two-way head motion. Nevertheless, analogous results for the rotating and the sweeping automata can be proven by exploiting closure properties of SN and SP_2X: while SN is not closed under complement, it is easy to observe that SP_2X is. Hence, $SN \neq SP_2X$, and because $SN = SP_1X \subseteq SP_2X$ holds, we have $SP_2X \supsetneq SN$. The same argumentation can be used to prove $RP_2X \supsetneq RN$ as well:

Observation 5.4.

$$2P_2X \supsetneq 2N = 2P_1X, \quad SP_2X \supsetneq SN = SP_1X, \quad RP_2X \supsetneq RN = RP_1X$$

The following theorem proves that $\text{RP}_2\text{X} \supseteq \text{SP}_2\text{X}$. Since $\text{RP}_2\text{X} \subseteq \text{SP}_2\text{X}$, we have $\text{RP}_2\text{X} = \text{SP}_2\text{X}$. The basic idea of the proof is the same as in Lemma 4.7. The RP_2FA M' simulates the left-to-right traversals of the SP_2FA M in a straightforward way. To simulate a right-to-left traversal, M' produces a right computation RCOMP of M uniformly at random. Automaton M' continues with the simulation of the next traversal of M with a probability equal to the probability that M performs RCOMP. Otherwise, the simulation of the right-to-left traversal is repeated with another randomly produced right computation.

Lemma 5.5. *Each* SP_2FA *with* k *states can be simulated by an* RP_2FA *with at most* $O(k^3)$ *states.*

PROOF. Given a k-state SP_2FA $M = (q_s, \delta, q_a)$ over an alphabet Σ and a set of states Q, we construct an equivalent RP_2FA $M' = (q'_s, \delta', q'_a)$ over the same alphabet with state set Q' such that $|Q'| = O(k^3)$.

The RP_2FA M' simulates each left computation of M in a straightforward manner. Thus each state q_i in M has a corresponding state q'_i in M' and the probability to reach a state q_j from q_i in M while reading the input string from left to right is exactly the same as the probability to reach q'_j from q'_i in M'.

Now we consider the simulation of a right computation. Let $w = w_1 w_2 \dots w_l$ be the input. Assume that M starts the right computation at w_l in state q_1, i. e., M reached q_1 after finishing the previous left computation and reading $\dashv$. We know that the computation $\text{RCOMP}_{M,q_1}(w)$ is a sequence of $l+1$ states. Consider any such sequence $\mathbf{s} = (s_1, s_2, \dots, s_l, s_{l+1})$. The probability that M performs $\mathbf{s}$ is

$$\pi_{\mathbf{s}} := [s_1 = q_1] \cdot \prod_{i=1}^{l} \delta(s_i, w_{l-i+1})(s_{i+1}),$$

where $[s_1 = q_1]$ is defined to be 1 if $s_1 = q_1$ and 0 otherwise. Now we consider all k^{l+1} sequences $\mathbf{s}$ and to each sequence we assign the probability of the corresponding computation. Obviously, a sequence that does not describe a valid right computation of M has probability zero and the sum of the assigned probabilities over all sequences is 1.

The basic idea is that M' chooses some sequence $\mathbf{s} = (s_1, \dots, s_{l+1})$ uniformly at random. With probability $\pi_{\mathbf{s}}$, the automaton M' proceeds to the simulation of the next left computation of M starting in state s_{l+1}, where $\pi_{\mathbf{s}}$ is the probability assigned to the chosen sequence. In this case, we say that M' *agreed with* the chosen sequence. With probability $1 - \pi_{\mathbf{s}}$, the automaton M' chooses another sequence and repeats the process.

The probability of a *repetition*, i. e., of not agreeing with one randomly chosen sequence is

$$1 - \sum_{(s_1=q_1, s_2, \dots, s_{l+1}) \in Q^{l+1}} \frac{1}{k^{l+1}} \cdot \prod_{i=1}^{l} \delta(s_i, w_{l-i+1})(s_{i+1}) = 1 - \frac{1}{k^{l+1}}.$$

The probability that M' makes exactly i repetitions, chooses the sequence $\mathbf{s}$ afterwards, and agrees with it, is

$$\frac{1}{k^{l+1}} \cdot \pi_{\mathbf{s}} \cdot \left(1 - \frac{1}{k^{l+1}}\right)^i$$

Hence, the probability that M' eventually agrees with $\mathbf{s}$ is

$$\frac{1}{k^{l+1}} \cdot \pi_{\mathbf{s}} \cdot \sum_{i \geq 0} \left(1 - \frac{1}{k^{l+1}}\right)^i = \frac{1}{k^{l+1}} \cdot \pi_{\mathbf{s}} \cdot \frac{1}{1 - \left(1 - \frac{1}{k^{l+1}}\right)} = \pi_{\mathbf{s}}.$$

We have just proven that the probability that M' agrees with sequence $\mathbf{s}$ is the same as the probability that M performs $\mathbf{s}$. Hence, the probability distribution of M' over its set of states is isomorphic to the probability distribution of M every time the simulation of left computation starts. Thus, M' correctly simulates M.

Now we discuss how to implement this idea. Instead of picking a sequence directly, M' can also pick a sequence uniformly at random state by state from left to right and agree with that sequence with the assigned probability. At first, M' selects state s_{l+1} uniformly at random and keeps q_1 and s_{l+1} stored in its states. Let $s_{l+1}, s_l, \ldots, s_{i+1}$ be the first $l - i + 1$ states of the sequence chosen by M'. Thus after $l - i + 1$ steps, M' knows q_1, s_{l+1}, s_{i+1} and, since it reads the input from left to right, symbol w_{l-i+1}. Now M' picks a state s_i uniformly at random. With probability $\delta(s_i, w_{l-i+1})(s_{i+1})$, the automaton M' proceeds to the next symbol. With probability $1-\delta(s_i, w_{l-i+1})(s_{i+1})$, the automaton M' moves the head to the first input symbol and starts a new simulation of the right computation. If $\dashv$ is reached, M' starts a new simulation if $s_1 \neq q_1$. Otherwise, the simulation of the right computation is finished, and the next left computation of M can be simulated. To do so, M' simulates the move of M from state s_{l+1} on $\vdash$ and proceeds to the first input symbol.

Now let us count the number of states of M'. The left computation can be simulated with k states. In the simulation of the right computation, picking a sequence and choosing whether to restart without storing the first and the last state of the computation only requires one extra copy of Q. Since we also store the first and the last state q_1 and s_{l+1}, we need k^2 extra copies of Q in total to simulate right computations of M. Thus M' has $O(k^3)$ states. □

Corollary 5.6.

$$\text{RP}_2\text{X} = \text{SP}_2\text{X}$$

The proof of the previous theorem can also be applied to Monte-Carlo and to LasVegas automata. Nevertheless, the results $\text{RP}_1\text{X} = \text{SP}_1\text{X}$ (respectively, $\text{RP}_0\text{X} = \text{SP}_0\text{X}$) follows directly from $\text{RP}_1\text{X} = \text{RN} = \text{SN} = \text{SP}_1\text{X}$ (respectively, $\text{RP}_0\text{X} = \text{R}\Delta = \text{S}\Delta = \text{SP}_0\text{X}$).

6. Conclusions

We explored the relationship between deterministic, nondeterministic, and randomized computations of finite automata with different capabilities of head motion. We focused on the size complexity classes of the considered automata, in a way proposed in [20] and used in [14, 15]. We presented an extensive map of these classes and showed that LasVegas sweeping automata can be exponentially more succinct than their deterministic counterparts. We, however, stress that the presented conclusions about the complexity classes induced by randomized finite automata concern automata with (finite) expected running time that is exponential in the length of the input, as our focus is on the size complexity only. Hence, our results could be interpreted as a first step towards the more

natural (and more faithful to the analogy with ZPP, P, and NP) case where size and time must be held small simultaneously.

Besides the number of states, there are several different ways how to measure the size of automata, such as the number of bits needed to describe the automaton (descriptional complexity) or the number of transitions in the transition function of the automaton. Although these measures are not equivalent, they are polynomially related if the size of the alphabet is polynomial. As any language family with polynomial descriptional complexity or transition complexity has a polynomial alphabet and problems with polynomially related complexity always fall into the same class, the definition of the complexity classes does not depend on the chosen measure if we consider only language families with alphabets of polynomial size. Even though we deal also with exponentially large alphabets in this paper, all presented results can be easily adapted for automata over binary alphabet as well. Hence, all relationships between different complexity classes presented in this paper hold also for the other measures of automata size.

To prove separations between different complexity classes, we have introduced the framework of hardness propagation. The core of this propagation was presented by Corollaries 3.4, 3.6, 3.13, and 3.15. This framework provides a systematic way of constructing language families witnessing the separations. In this way, we gain also more insight into the hardness structure of the witnesses than provided by the ad-hoc witnesses used in the previous proofs (e. g., the witness of separation between SD and SΔ used in [14]). In fact, using the introduced language operators, it is possible to obtain witnesses with similar structure as the ad-hoc ones (see concluding remarks of [15] for more information on this topic).

We have used *parallel automata* as an intermediate computational model in the hardness propagation. We have defined a family of parallel automata to be small if the automata contain only a polynomial number of components, each with a polynomial number of states. There is an alternative definition to this, used in [15], which places the constraint on the number of components only. In this way, small parallel automata are required only to have polynomially small components, the number of components being irrelevant. All hardness propagation results presented in this paper hold for this alternative definition as well. Nevertheless, we opted not to consider classes based on this alternative definition, as they are rather unnatural, since they correspond to automata with possibly large descriptional complexity. On the other hand, separating complexity classes of both variants of parallel automata is an interesting open problem. Since this apparently cannot be done by using the technique of generic words, a completely new technique for proving lower bounds on parallel automata seems to be necessary to achieve such a separation.

In this paper, we have not considered the complexity classes of general parallel automata, for similar reasons as explained above. The set of accepting tuples of a general parallel automaton can be exponentially large even if the automaton is polynomially small. Hence, such automata can have huge descriptional complexity as well.

Several problems mentioned in this paper are left open. For example, a few closure properties in Figure 6 are not known. For bounded-error automata, only the basic facts described in Section 5.3 are known. For other randomized classes except for one-way Monte-Carlo automata, we have provided a complete characterization. We have, however, not considered the time complexity of the randomized automata. In fact, the constructions used to prove the presented results [17, 9] yield randomized automata with

exponential expected running time. It is a natural open problem to ask if this is necessary, i. e., to analyze the size complexity classes of automata with polynomial expected running time. At last but not least, the relationship between determinism and nondeterminism in two-way automata (e. g., 2D vs. 2N) remains the most challenging open problem.

Acknowledgement

We would like to thank Juraj Hromkovič for initiating this research and for providing many helpful comments.

References

[1] P. Berman. A note on sweeping automata. In *Proc. of ICALP*, pages 91–97, 1980.

[2] P. Ďuriš, J. Hromkovič, J. D. P. Rolim, and G. Schnitger. Las Vegas versus determinism for one-way communication complexity, finite automata, and polynomial-time computations. In *Proceedings of the STACS*, pages 117–128, 1997.

[3] C. Dwork and L. Stockmeyer. A time complexity gap for two-way probabilistic finite-state automata. *SIAM Journal on Computing*, 19(6):1011–1123, 1990.

[4] R. Freivalds. Probabilistic two-way machines. In *Proceedings of the International Symposium on MFCS*, pages 33–45, 1981.

[5] V. Geffert, C. Mereghetti, and G. Pighizzini. Complementing two-way finite automata. *Information and Computation*, 205(8):1173–1187, 2007.

[6] M. Holzer and M. Kutrib. State complexity of basic operations on nondeterministic finite automata. In *Implementation and Application of Automata. Proc. of the 7th International Conference, CIAA 2002*, volume 2608 of *Lecture Notes in Computer Science*, pages 61–79, Berlin, 2003. Springer-Verlag.

[7] J. Hromkovič and G. Schnitger. Nondeterministic communication with a limited number of advice bits. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 551–560, New York, NY, USA, 1996. ACM.

[8] J. Hromkovič and G. Schnitger. On the power of Las Vegas for one-way communication complexity, OBDDs, and finite automata. *Information and Computation*, 169:284–296, 2001.

[9] J. Hromkovič and G. Schnitger. On the power of Las Vegas II: two-way finite automata. *Theoretical Computer Science*, 262(1–2):1–24, 2001.

[10] G. Jirásková. State complexity of some operations on binary regular languages. *Theoretical Computer Science*, 330(2):287–298, 2005. Descriptional Complexity of Formal Systems.

[11] C. Kapoutsis. *Algorithms and lower bounds in finite automata size complexity.* PhD thesis, Cambridge, MA, USA, 2006.

[12] C. A. Kapoutsis. Small sweeping 2NFAs are not closed under complement. In *Proceedings of the ICALP*, pages 144–156, 2006.

[13] C. A. Kapoutsis. Size complexity of two-way finite automata. In V. Diekert and D. Nowotka, editors, *Developments in Language Theory*, volume 5583 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 2009.

[14] C. A. Kapoutsis, R. Královič, and T. Mömke. An exponential gap between LasVegas and deterministic sweeping finite automata. In J. Hromkovič, R. Královič, M. Nunkesser, and P. Widmayer, editors, *SAGA*, volume 4665 of *Lecture Notes in Computer Science*, pages 130–141. Springer, 2007.

[15] C. A. Kapoutsis, R. Královič, and T. Mömke. On the size complexity of rotating and sweeping automata. In M. Ito and M. Toyama, editors, *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2008.

[16] A. N. Kolodin. Two-way nondeterministic automata. *Cybernetics and Systems Analysis*, 10(5):778–785, 1972.

[17] I. I. Macarie and J. I. Seiferas. Amplification of slight probabilistic advantage at absolutely no cost in space. *Inf. Process. Lett.*, 72(3-4):113–118, 1999.

[18] S. Micali. Two-way deterministic finite automata are exponentially more succinct than sweeping automata. *Information Processing Letters*, 12(2):103–105, 1981.

[19] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, (3), 1959.

[20] W. J. Sakoda and M. Sipser. Nondeterminism and the size of two way finite automata. In *Proceedings of the STOC*, pages 275–286, 1978.
[21] M. Sipser. Halting space-bounded computations. In *FOCS*, pages 73–74. IEEE, 1978.
[22] M. Sipser. Lower bounds on the size of sweeping automata. *Journal of Computer and System Sciences*, 21(2):195–202, 1980.