Synchronizing Data Words for Register Automata

KARIN QUAAS, Universität Leipzig
MAHSA SHIRMOHAMMADI, Oxford University

Register automata (RAs) are finite automata extended with a finite set of registers to store and compare data from an infinite domain. We study the concept of synchronizing data words in RAs: Does there exist a data word that sends all states of the RA to a single state?

For deterministic RAs with k registers (k-DRAs), we prove that inputting data words with $2k + 1$ distinct data, from the infinite data domain, is sufficient to synchronize. We show that the synchronizing problem for DRAs is in general PSPACE-complete, and is NLOGSPACE-complete for 1-DRAs. For nondeterministic RAs (NRAs), we show that Ackermann(n) distinct data (where n is the size of the RA) might be necessary to synchronize. The synchronizing problem for NRAs is in general undecidable, however, we establish Ackermann-completeness of the problem for 1-NRAs. Our most substantial achievement is proving NEXPTIME-completeness of the length-bounded synchronizing problem for NRAs (length encoded in binary). A variant of this last construction allows to prove that the bounded universality problem for NRAs is co-NEXPTIME-complete.

CCS Concepts: •**Theory of computation** →*Automata over infinite objects;*

General Terms: Register automata, Synchronization

Additional Key Words and Phrases: Register automata, Data words, Synchronization Problem, Bounded Universality

ACM Reference format:
Karin Quaas and Mahsa Shirmohammadi. 2017. Synchronizing Data Words for Register Automata. *ACM Trans. Comput. Logic* -, -, Article – (September 2017), 45 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Synchronizing words for finite automata have been studied since the 70fis, see (8; 23; 25; 31); such a word w drives the automaton from an unknown or unobservable state to a specific state q_w that only depends on w. The famous Černý conjecture on synchronizing words is a long-standing open problem in automata theory. The conjecture claims that the length of a shortest synchronizing data word for a deterministic finite automaton (DFA) with n states is at most $(n-1)^2$. There exists a family of DFAs, where the length of the shortest synchronizing word is exactly $(n-1)^2$, which attains the exact claimed bound in the conjecture. Despite all received attention in the last decades, this conjecture has not been proved or disproved.

This work was partially supported by Deutsche Forschungsgemeinschaft (DFG), project QU 316/1-2.
Author's addresses: Karin Quaas, Institut für Informatik, Abt. Automaten und Sprachen, PF 100920, D-04009 Leipzig; Mahsa Shirmohammadi, Wolfson Building, Parks Road, Oxford OX1 3QD.

Synchronizing words have applications in planning, control of discrete event systems, biocomputing, and robotics (3; 15; 31). Over the past few years, this classical notion has sparked renewed interest thanks to its generalization to games on transition systems (20; 21; 28), and to infinite-state systems (9; 14), which are relevant for modelling complex systems such as distributed data networks or real-time embedded systems.

In this paper, we are interested in *synchronizing data words* for *register automata. Data words* are sequences of pairs where the first element is taken from a finite alphabet and the second element is taken from *an infinite data domain* such as the natural numbers or ASCII strings. In recent years, this structure has become an active subject of research thanks to applications in querying and reasoning about data models with complex structural properties, in XML, and lately also in graph databases (1; 2; 5; 16). For reasoning about data words, various formalisms have been considered, ranging over first-order logic for data words (4; 6), extensions of linear temporal logic (11–13; 22), data automata (4; 7), register automata (11; 19; 24; 26) and extensions thereof, *e.g.* (10; 17; 30).

Register automata (RAs) are a natural generalization of finite automata for processing data words. RAs are equipped with a finite set of registers. While processing a data word, the automaton may store the data value of the current position in one or more registers. It may also test the data value of the current position for equality with the values stored in the registers, where the result of this test determines how the behaviour of the RA evolves. This allows for handling parameters like user names, passwords, identifiers of connections, sessions, etc., in a fashion similar to, and more expressive than, the class of data-independent systems. RAs come in different variants, e.g., one-way vs. two-way, deterministic vs. non-deterministic, alternating vs. non-alternating. For alternating RAs, classical decision problems like emptiness, universality and language inclusion are undecidable. We focus on the class of one-way RAs without alternation: They have a decidable emptiness problem (19), and the subclass of nondeterministic RAs with a single register has a decidable universality problem (11).

Semantically, an RA defines an infinite-state system, due to the unbounded domain for data stored in registers. Synchronizing words were introduced for infinite-state systems with infinite branching in (14; 28); in particular, the notion of synchronizing words is motivated and studied for weighted automata and timed automata. In some infinite-state settings such as nested word automata (or, equivalently, visibly pushdown automata), finding the right definition of synchronizing words is however more challenging (9). We define the synchronizing problem for RAs along the suggested framework in (14; 28): Given an RA $\mathcal{R}$, does there exist a data word w that sends each of the infinitely many states of $\mathcal{R}$ to some specific state (depending only on w)? Such a data word is called a *synchronizing data word.*

Contribution. The problem of finding synchronizing data words for RAs imposes new challenges in the area of synchronization. It is natural to ask how many distinct data are necessary and sufficient to synchronize an RA, which we refer to by the notion of *data efficiency* of synchronizing data words. We show that the data efficiency is polynomial in the number of registers for deterministic RAs (DRAs). For nondeterministic RAs (NRAs), we provide an example that shows that the data efficiency may be Ackermann(n), where n is the number of states of the NRA. Remarkably, data efficiency is tightly related to the complexity of deciding the existence of a synchronizing data word: for DRAs, we prove that for all automata $\mathcal{R}$ with k registers, if $\mathcal{R}$ has a synchronizing data word, then it also has one with data efficiency *at most* $2k + 1$. We provide a family $(\mathcal{R}_k)_{k\in\mathbb{N}}$ with k registers, for which indeed a polynomial data efficiency (in the size of k) is necessary to synchronize. This bound is the base of an (N)PSPACE-algorithm for DRAs; we prove a matching PSPACE lower bound by ideas carried over from timed settings (14). We show that the synchronizing problems for DRAs with a single register (1-DRAs) and for DFAs are NLOGSPACE-interreducible, implying that the problem is NLOGSPACE-complete for 1-DRAs.

For NRAs, a reduction from the non-universality problem yields the undecidability of the synchronization problem. For single-register NRAs (1-NRAs), we prove Ackermann-completeness of the problem by a novel construction proving that the synchronizing problem and the non-universality problem for 1-NRAs are polynomial-time interreducible. We believe that this technique is useful in studying synchronization in all nondeterministic settings, requiring careful analysis of the size of the construction.

Our most substantial achievement is proving NEXPTIME-completeness of the *length-bounded synchronizing problem* for NRAs: Does there exist a synchronizing data word with at most a given length (encoded in binary)? For the lower bound, we present a non-trivial reduction from the bounded non-universality problem for *regular-like expressions with squaring*, which is NEXPTIME-complete (29). The crucial ingredient in this reduction is a family of RAs implementing binary counters. A variant of our construction yields a proof for co-NEXPTIME-completeness of the *bounded universality problem* for NRAs; the bounded universality problem asks whether all data words with at most a given length (encoded in binary) are in the language of the automaton.

An extended abstract of this article has appeared in the Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science, (MFCS) 2016. In comparison with the extended abstract, here we simplify two main constructions in addition to presenting detailed proofs of all results. The noticeable improvement is a huge simplification in the NEXPTIME-hardness reduction for the length-bounded synchronizing problem for NRAs.

2 PRELIMINARIES

A deterministic finite-state automaton (DFA) is a tuple $\mathcal{A} = \langle Q, \Sigma, \Delta \rangle$, where Q is a finite set of states, Σ is a finite alphabet, and $\Delta : Q \times \Sigma \to Q$ is a transition function that is totally defined. The function Δ extends to finite words in a natural way: $\Delta(q, wa) = \Delta(\Delta(q, w), a)$ for all words $w \in \Sigma^*$ and letters $a \in \Sigma$; it extends to all sets $S \subseteq Q$ by $\Delta(S, w) = \bigcup_{q \in S} \Delta(q, w)$.

Data Words and Register automata. For the rest of this paper, fix an infinite data domain D. Given a finite alphabet Σ, a *data word over* Σ is a finite words over $\Sigma \times D$. For a data word $w = (a_1, d_1)(a_2, d_2) \cdots (a_n, d_n)$, the length of w is $|w| = n$. We use $\text{data}(w) = \{d_1, \ldots, d_n\} \subseteq D$ to refer to the set of data values occurring in w, and we define the *data efficiency of* w to be $|\text{data}(w)|$.

Let R be a finite set of *register variables.* We define *register constraints* ϕ *over* R by the grammar

$$\phi \ ::= \mathsf{true} \mid \ =r \mid \phi \wedge \phi \mid \neg\phi,$$

where $r \in R$. We denote by $\Phi(R)$ the set of all register constraints over R. We may use $\neq r$ for the inequality constraint $\neg(=r)$. A *register valuation* is a mapping $v : R \to D$ that assigns a data value to each register; we sometimes write $v = (v(r_1), \cdots, v(r_k)) \in D^k$, where $R = \{r_1, \cdots, r_k\}$. The satisfaction relation of register constraints is defined on $D^k \times D$ as follows: (v, d) satisfies the constraint $=r$ if $v(r) = d$; the other cases follow. For example, $((d_1, d_2, d_1), d_2)$ satisfies $((= r_1) \wedge (= r_2)) \vee (\neq r_3))$ if $d_1 \neq d_2$. For the set $\mathsf{up} \subseteq R$, we define the *update* $v[\mathsf{up} := d]$ *of valuation* v by $(v[\mathsf{up} := d])(r) = d$ if $r \in \mathsf{up}$, and $(v[\mathsf{up} := d])(r) = v(r)$ otherwise.

A *register automaton* (RA) is a tuple $\mathcal{R} = \langle L, R, \Sigma, T \rangle$, where L is a finite set of locations, R is a finite set of registers, Σ is a finite alphabet and $T \subseteq L \times \Sigma \times \Phi(R) \times 2^R \times L$ is a transition relation. We may use $\ell \xrightarrow{\phi \ a \ \mathsf{up}\downarrow} \ell'$ to show transitions $(\ell, a, \phi, \mathsf{up}, \ell') \in T$. We call $\ell \xrightarrow{\phi \ a \ \mathsf{up}\downarrow} \ell'$ an a-transition and ϕ the *guard* of this transition. A guard true is vacuously true and may be omitted. Likewise we may omit up if $\mathsf{up} = \emptyset$. We may write $r\downarrow$ when $\mathsf{up} = \{r\}$ is singleton. For NRAs with only one register, we may shortly write $=$ and $\neq$ for the guards $=r$ and $\neq r$, respectively, and $\downarrow$ for the update $\downarrow r$.

A *configuration* of $\mathcal{R}$ is a pair $(\ell, v) \in L \times D^{|R|}$ of a location ℓ and a register valuation v. We describe the behaviour of $\mathcal{R}$ as follows: Given that $\mathcal{R}$ is in configuration $q = (\ell, v)$, on inputting $(a, d) \in \Sigma \times D$ an a-transition $\ell \xrightarrow{\phi \ a \ \mathsf{up}\downarrow} \ell'$ may be fired if (v, d) satisfies the constraint ϕ; then $\mathcal{R}$ moves to *successor configuration* $q' = (\ell', v')$, where $v' = v[\mathsf{up} := d]$ is the update of v. By $\mathsf{post}(q, (a, d))$, we denote the set of all successor configuration q' of q on inputting (a, d). We extend post to sets $S \subseteq L \times D^{|R|}$ of configurations by $\mathsf{post}(S, (a, d)) = \bigcup_{q \in S} \mathsf{post}(q, (a, d))$; and we extend post to words by $\mathsf{post}(S, w \cdot (a, d)) = \mathsf{post}(\mathsf{post}(S, w), (a, d))$ for all words $w \in (\Sigma \times D)^*$, and all inputs $(a, d) \in \Sigma \times D$.

A run of $\mathcal{R}$ over the data word $w = (a_1, d_1)(a_2, d_2) \cdots (a_n, d_n)$ is a sequence of configurations $q_0 q_1 \ldots q_n$, where $q_i \in \mathsf{post}(q_{i-1}, (a_i, d_i))$ for all $1 \le i \le n$. During processing a word w, if $\mathcal{R}$ reaches a configuration $q = (\ell, x)$, we may say that *the x-token is in ℓ*, or, more generally, *a token is in ℓ*.

In the rest of the paper, we consider *complete* RAs, meaning that for all configurations $q \in L \times D^{|R|}$ and all inputs $(a, d) \in \Sigma \times D$, there is at least one successor: $|\mathsf{post}(q, (a, d))| \ge 1$. We also classify the RAs into *deterministic* RAs (DRAs) and *nondeterministic* (NRAs), where an RA is deterministic if $|\mathsf{post}(q, (a, d))| \le 1$ for all configurations q and all inputs (a, d). A *k-NRA* (*k-DRA*, respectively) is an NRA (DRA, respectively) with $|R| = k$.

Synchronizing words and synchronizing data words. *Synchronizing words* are a well-studied concept for DFAs, see, *e.g.*, (31). Informally, a synchronizing word leads the automaton from every state to the same state. Formally, the word $w \in \Sigma^+$ is synchronizing for a DFA $\mathcal{A} = \langle Q, \Sigma, \Delta \rangle$ if there exists some state $q \in Q$ such that $\Delta(Q, w) = \{q\}$. The *synchronizing problem* for DFAs asks, given a DFA $\mathcal{A}$, whether there exists some synchronizing word for $\mathcal{A}$.

The synchronizing problem for DFAs is in NLOGSPACE by using the *pairwise synchronization* technique: Given a DFA $\mathcal{A} = \langle Q, \Sigma, \Delta \rangle$, it is known that it has a synchronizing word if and only if for all pairs of states $q, q' \in Q$, there exists a word v such that $\Delta(q, v) = \Delta(q', v)$ (see (31) for more details). The pairwise synchronization algorithm initially sets $S_{|Q|} = Q$. Then, starting with $i = |Q| - 1$ and while there exist two distinct states $q, q' \in S_{i+1}$ and $i \ge 1$: pick two distinct states $q, q' \in S_{i+1}$, find a word v_i such that $\Delta(q, v_i) = \Delta(q', v_i)$, set $i = i - 1$ and $S_i = \Delta(S_{i+1}, v_i)$. The word $w = v_{|Q|-1} \cdots v_j$, for some $j \ge 1$, is synchronizing for $\mathcal{A}$.

We introduce synchronizing data words for RAs: a data word $w \in (\Sigma \times D)^+$ is *synchronizing* for an RA $\mathcal{R} = \langle L, R, \Sigma, T \rangle$ if there exists some configuration (ℓ, v) such that $\mathsf{post}(L \times D^{|R|}, w) = \{(\ell, v)\}$. The *synchronizing problem* for RAs asks, given an RA $\mathcal{R}$ over a data domain D, whether there exists some synchronizing data word for $\mathcal{R}$. The *bounded synchronizing problem* for RAs decides, given an RA $\mathcal{R}$ and $N \in \mathbb{N}$ encoded in binary, whether there exists some synchronizing data word w for $\mathcal{R}$ satisfying $|w| \le N$.

3 SYNCHRONIZING DATA WORDS FOR DRAs

In this section, we first show that the synchronizing problems for 1-DRAs and DFAs are NLOGSPACE-interreducible, implying that the problem is NLOGSPACE-complete for 1-DRAs. Next, we prove that the problem for k-DRAs, in general, can be decided in PSPACE; a reduction similar to a timed setting, as in (14), provides the matching lower bound. To obtain the complexity upper bounds, we prove that inputting words with data efficiency $2|R| + 1$ is sufficient to synchronize a DRA.

The concept of synchronization requires that all runs of an RA, whatever the initial configuration (initial location and register valuations), end in the same configuration $(\ell_{\mathsf{synch}}, v_{\mathsf{synch}})$, only depending on the synchronizing data word w_{synch}: $\mathsf{post}(L \times D^{|R|}, w_{\mathsf{synch}}) = \{(\ell_{\mathsf{synch}}, v_{\mathsf{synch}})\}$. While processing a synchronizing data word, the infinite set of configurations of RAs must necessarily shrink to a finite set of configurations. The RA $\mathcal{R}$ with 3 registers depicted in Figure 1 illustrates this phenomenon. Consider the set $\{x_1, x_2, x_3\} \subseteq D$ of distinct data values: starting from any of the

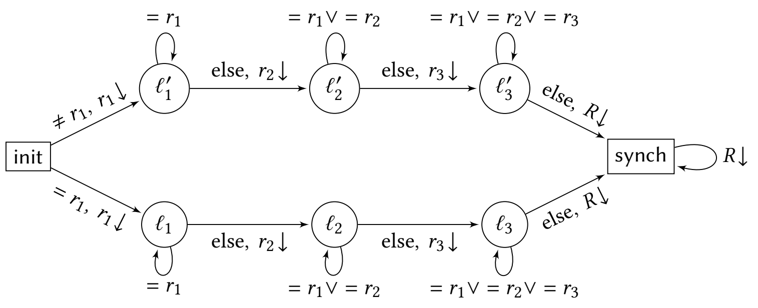


Fig. 1. A DRA with registers r_1, r_2, r_3 and single letter a (omitted from transitions) that can be synchronized in the configuration (synch, x_4) by the data word $w_{\text{synch}} = (a, x_1)(a, x_2)(a, x_3)(a, x_4)$ if $\{x_1, x_2, x_3, x_4\} \subseteq D$ is a set of 4 distinct data.

infinite configurations in $\{\text{init}\} \times D^3$, when processing the data word $(a, x_1)(a, x_2)(a, x_3)$, $\mathcal{R}$ will be in a configuration in the finite set $\{(\ell_3, (x_1, x_2, x_3)), (\ell'_3, (x_1, x_2, x_3))\}$. We use this observation to provide a linear bound on the number of distinct data values that is sufficient for synchronizing RAs.

In Lemma 3.1 below, we prove that data words over only $|R|$ distinct data values are sufficient to shrink the infinite set of all configurations of RAs to a finite set. We establish this result based on the following two key facts:

(1) When processing a synchronizing data word w_{synch} from a configuration (ℓ, ν) with $\nu(r) \notin \text{data}(w_{\text{synch}})$ for some $r \in R$, the register r must be updated. Observe that such updates must happen at inequality-guarded transitions, which themselves must be accessible by inequality-guarded transitions (possibly with no update). As an example, consider the RA $\mathcal{R}$ in Figure 1, and assume that $d_1, d_2 \notin \text{data}(w_{\text{synch}})$. The two runs of $\mathcal{R}$ starting from $(\text{init}, d_1, d_1, d_1)$ and $(\text{init}, d_2, d_2, d_2)$ first take the transition init $\xrightarrow{\neq r_1 \ a \ r_1\downarrow} \ell'_1$ updating register r_1. Next, the two runs must take $\ell'_1 \xrightarrow{\text{else} \ a \ r_2\downarrow} \ell'_2$ to update r_2 and $\ell'_2 \xrightarrow{\text{else} \ a \ r_3\downarrow} \ell'_3$ to update r_3; otherwise these two runs would never synchronize in a single configuration.

(2) Moreover, to shrink the set $L \times D^{|R|}$, for every $\ell \in L$, one can find a word w_ℓ that leads the RA from $\{\ell\} \times D^{|R|}$ to some finite set. Since $\mathcal{R}$ is deterministic, appending some prefix or suffix to w_ℓ achieves the same objective. This allows us to use a variant of pairwise synchronization to shrink the infinite set $L \times D^{|R|}$ to a finite set, by successively inputting w_ℓ for a location ℓ that appears with infinitely many data in the current successor set of $L \times D^{|R|}$.

Lemma 3.1. *For all DRAs for which there exist synchronizing data words, there exists some data word w such that* $\text{data}(w) \leq |R|$ *and* $\text{post}(L \times D^{|R|}, w) \subseteq L \times (\text{data}(w))^{|R|}$.

Proof. Let $\mathcal{R} = \langle L, R, \Sigma, T \rangle$ be a DRA on the data domain D with $k \geq 1$ registers. Let v be a synchronizing data word for $\mathcal{R}$ with $N = |\text{data}(v)|$ distinct data. Suppose that $k < N$; otherwise the statement of the lemma trivially holds.

For all $1 \leq i \leq k$, we say that x_i *is the i-th datum in the synchronizing data word* $v = (a_1, d_1)(a_2, d_2) \cdots (a_n, d_n)$ if there exists $j \leq k$ such that $x_i = d_j$, $x_i \notin \{d_1, \cdots, d_{j-1}\}$ and $|\{d_1, \cdots, d_j\}| = i$. For every $i \leq k$, denote by $\langle L, i \rangle$ the following set

$$\langle L, i \rangle = L \times \{\nu \in D^k \mid \exists R' \in R \cdot |R'| \geq i \cdot \forall r \in R' \cdot \nu(r) \in \{x_1, \cdots, x_i\}\}.$$

We **Claim** that for all locations $\ell \in L$ and all $1 \leq i \leq k$, there exists some data word u_i such that

- $\text{data}(u_i) \subseteq \{x_1, x_2, \cdots, x_i\}$, and

- $\mathsf{post}(\{\ell\} \times D^k, u_i) \subseteq \langle L, i \rangle$, meaning that after reading u_i all reached configurations have at least i registers with values from $\{x_1, x_2, \cdots, x_i\}$.

For $\ell \in L$, let $w_\ell = u_k$ satisfy the above condition. Set $S_0 = L \times D^k$ and $w_0 = \varepsilon$. Then, for all $i = 1, \cdots, |L|$, repeat the following: if there exists some $\ell \in L$ such that $\{\ell\} \times (D \setminus \{x_1, \cdots, x_k\})^k \cap S_{i-1} \neq \emptyset$, then set $w_i = w_\ell$ and $S_i = \mathsf{post}(S_{i-1}, w_i)$. Otherwise set $w_i = w_{i-1}$ and $S_i = S_{i-1}$. Observe that $w = (w_i)_{1 \leq i \leq |L|}$ proves the statement of Lemma. It remains to prove the **Claim**.

Proof of Claim. Let $\hat{\ell}$ be some location in the DRA $\mathcal{R}$. The proof is by an induction on i.
Base of induction. Let $\mathsf{wait} = \{\hat{\ell}\} \times (D \setminus \mathsf{data}(v))^k$ be the set of configurations with location $\hat{\ell}$ such that the data stored in all k registers is not in $\mathsf{data}(v)$. Note that for all configurations $(\hat{\ell}, \nu) \in \mathsf{wait}$, the unique run of $\mathcal{R}$ starting in $(\hat{\ell}, \nu)$ on (a prefix of) v consists of the same sequence of the following transitions:

- a prefix of transitions $\xrightarrow{\bigwedge_{r \in R} \neq r \quad \emptyset\downarrow}$, with inequality guards on all registers and with no register update,
- followed by a transition $\xrightarrow{\bigwedge_{r \in R} \neq r \quad \mathsf{up}\downarrow}$, with inequality guard on all registers and with an update for some non-empty set $\mathsf{up} \subseteq R$.

Otherwise, the two runs starting from any pair of configurations $(\hat{\ell}, \nu_1), (\hat{\ell}, \nu_2) \in \mathsf{wait}$ with unequal valuations $\nu_1 \neq \nu_2$ would end up in distinct configurations, say $(\ell, \nu_1'), (\ell, \nu_2')$ with $\nu_1' \neq \nu_2'$. This is a contradiction to the fact that the data word v is synchronizing.

Now let the inequality-guarded transition $\xrightarrow{\bigwedge_{r \in R} \neq r \quad \mathsf{up}\downarrow}$, updating the registers in up, be fired at the j-th input (a_j, d_j) while reading v; see Figure 2. We prove that the data word $u_1 = (a_1, x_1)(a_2, x_1) \cdots (a_j, x_1)$ with $\mathsf{data}(u_1) = \{x_1\}$ brings $\{\hat{\ell}\} \times D^k$ to a subset in which each configuration has some register with value x_1: $\mathsf{post}(\{\hat{\ell}\} \times D^k, u_1) \subseteq \langle L, 1 \rangle$. This phenomenon is depicted in Figure 3 and can be argued as follows. Observe that $x_1 = d_1$ is the first input datum; thus after inputting (a_1, x_1) the set of successors is a disjoint union of two branches:

- either at least one register r has datum x_1 after the transition $\xrightarrow{\bigvee_{r \in R} = r \quad a_1}$. All the following successors in this branch, on inputting $(a_2, x_1)(a_3, x_1) \cdots (a_j, x_1)$ preserve the datum x_1 in the register r;
- or none of the registers is assigned to x_1 after the transitions $\xrightarrow{\bigwedge_{r \in R} \neq r \quad a_1}$. By inputting $(a_2, x_1)(a_3, x_1) \cdots (a_j, x_1)$, all the following successors in this branch, thus, take inequality-guarded transitions, and would not update any registers, except for the last transition $\xrightarrow{\bigwedge_{r \in R} \neq r \quad \mathsf{up}\downarrow}$ fired by (a_j, x_1).

The above argument proves that u_1 with $\mathsf{data}(u_1) \subseteq \{x_1\}$ is such that $\mathsf{post}(\{\hat{\ell}\} \times D^k, u_1) \subseteq \langle L, 1 \rangle$. The base of induction holds.
Step of induction. Assume that the induction hypothesis holds for $i - 1$, namely, there exists some word u_{i-1} with $\mathsf{data}(u_{i-1}) \subseteq \{x_1, \cdots, x_{i-1}\}$ such that $\mathsf{post}(\{\hat{\ell}\} \times D^k, u_{i-1}) \subseteq \langle L, i-1 \rangle$. To construct u_i, we define the concept of a symbolic state: we say $(\ell, \mathsf{up}, \nu, j)$ is a symbolic state if $\ell \in L$, the set $\mathsf{up} \subseteq R$ of registers is such that $|\mathsf{up}| \geq \min(j, k)$ and $\nu \in \{x_1, \cdots, x_j\}^k$ and $j \leq N$. The semantics of $(\ell, \mathsf{up}, \nu, j)$ is the following set:

$$[\![(\ell, \mathsf{up}, \nu, j)]\!] = \{\ell\} \times \{\nu' \in D^k \mid \nu'(r) = \nu(r) \text{ if } r \in \mathsf{up}\}.$$

Denote by Γ the set of all such symbolic states $(\ell, \mathsf{up}, \nu, i-1)$. By definition, the set Γ is finite. Now we can construct u_i as follow. Let $S_0 = \mathsf{post}(\{\hat{\ell}\} \times D^k, u_{i-1})$ and $w_0 = u_{i-1}$. Recall that $S_0 \subseteq \langle L, i-1 \rangle$ and observe that $S_0 \subseteq \bigcup_{q \in \Gamma} [\![q]\!]$. Start with $j = 0$ and while $S_j \neq \emptyset$: pick a symbolic state $q = (\ell, \mathsf{up}, \nu, i-1)$ such that $[\![q]\!] \cap S_j \neq \emptyset$ and construct a word u_q (explained the details below) such that

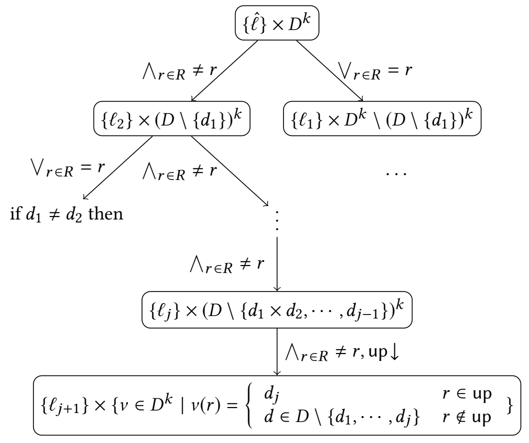

$\{\hat{\ell}\} \times D^k$
$\bigwedge_{r\in R} \neq r$
$\bigvee_{r\in R} = r$
$\{\ell_2\} \times (D \setminus \{d_1\})^k$
$\{\ell_1\} \times D^k \setminus (D \setminus \{d_1\})^k$
$\bigvee_{r\in R} = r$
$\bigwedge_{r\in R} \neq r$
if $d_1 \neq d_2$ then
$\bigwedge_{r\in R} \neq r$
$\{\ell_j\} \times (D \setminus \{d_1 \times d_2, \cdots, d_{j-1}\})^k$
$\bigwedge_{r\in R} \neq r$, up$\downarrow$
$\{\ell_{j+1}\} \times \{v \in D^k \mid v(r) = \begin{cases} d_j & r \in \text{up} \\ d \in D \setminus \{d_1, \cdots, d_j\} & r \notin \text{up} \end{cases}\}$

Fig. 2. Runs of $\mathcal{R}$ over $(a_1, d_1)(a_2, d_2) \cdots (a_j, d_j)$.

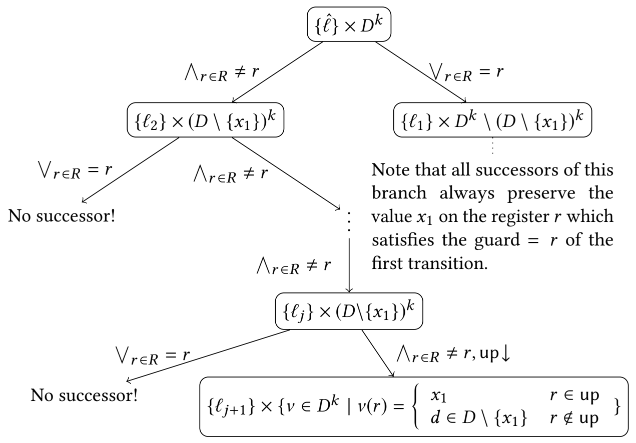

$\{\hat{\ell}\} \times D^k$
$\bigwedge_{r\in R} \neq r$
$\bigvee_{r\in R} = r$
$\{\ell_2\} \times (D \setminus \{x_1\})^k$
$\{\ell_1\} \times D^k \setminus (D \setminus \{x_1\})^k$
$\bigvee_{r\in R} = r$
$\bigwedge_{r\in R} \neq r$
No successor!
Note that all successors of this branch always preserve the value x_1 on the register r which satisfies the guard $= r$ of the first transition.
$\bigwedge_{r\in R} \neq r$
$\{\ell_j\} \times (D \setminus \{x_1\})^k$
$\bigvee_{r\in R} = r$
$\bigwedge_{r\in R} \neq r$, up$\downarrow$
No successor!
$\{\ell_{j+1}\} \times \{v \in D^k \mid v(r) = \begin{cases} x_1 & r \in \text{up} \\ d \in D \setminus \{x_1\} & r \notin \text{up} \end{cases}\}$

Fig. 3. Runs of $\mathcal{R}$ over $u_1 = (a_1, x_1)(a_2, x_1) \cdots (a_j, x_1)$

- $\text{data}(u_q) = \{x_1, x_2, \cdots, x_i\}$, and
- $\text{post}([\![q]\!], u_q) \subseteq \langle L, i \rangle$.

Let $S_{j+1} = \text{post}(S_j \setminus [\![q]\!], u_q)$ and $w_{j+1} = w_j \cdot u_q$. Repeat the loop for $j + 1$. Observe that $u_i = w_{j^*}$, where $j^* \leq |S_0|$ is such that $S_{j^*} = \emptyset$, satisfies the the induction statement.

Below, given a symbolic state $q = (\ell, \text{up}, v, i-1)$, the aim is to construct the data word u_q. Without loss of generality, we assume that $|\text{up}| = i - 1$; otherwise $u_q = u_{i-1}$. Let

$$\text{wait} = [\![(\ell, \text{up}, v, i-1)]\!] \cap \{\ell\} \times \{v' \mid v'(r) \in D \setminus \text{data}(v) \text{ if } r \notin \text{up}\}$$

be the set of all configurations in the symbolic state q where stored data in all registers $r \notin$ up are not in data(v). Similarly to the induction base, no matter what the register valuation in a configuration in wait looks like, the unique run of $\mathcal{R}$ on the synchronizing word $v = (a_1, d_1)(a_2, d_2) \cdots (a_n, d_n)$ starting in that configuration takes the same sequence of transitions. Since $v \in \{x_0, \cdots, x_{i-1}\}^k$, after inputting successive data from data(v), all successors of configurations in wait are elements of a symbolic state. For all $0 \leq j \leq n$, let the symbolic state $q^j = (\ell^j, \text{up}^j, v^j, N)$ be such that $[\![q^0]\!] = [\![q]\!] \cap \text{wait}$, and $\text{post}([\![q^{j-1}]\!], (a_j, d_j)) \subseteq [\![q^j]\!]$ if $j \geq 1$.
In the sequel, we argue that there exists some $1 \leq m \leq n$ such that, in the sequence of transitions from one symbolic state to another symbolic state over the prefix $(a_1, d_1)(a_2, d_2) \cdots (a_m, d_m)$ of v (the first m inputs), the following holds:

- on inputting (a_j, d_j) for all $1 \leq j < m$, the transition $\xrightarrow{(\bigwedge_{r\in\Lambda_j} = r) \wedge (\bigwedge_{r\notin\Lambda_j} \neq r) \ a_j \ \Gamma_j\downarrow}$ with $\Lambda_j, \Gamma_j \subseteq$ up is taken from q^{j-1} to q^j. It implies that $v^{j-1}(r) = d_j$ for all $r \in \Lambda_j$, and $v^j(r) = d_j$ for all $r \in \Gamma_j$.
- and on inputting (a_m, d_m), the transition $\xrightarrow{(\bigwedge_{r\in\Lambda_m} = r) \wedge (\bigwedge_{r\notin\Lambda_m} \neq r) \ a_m \ \Gamma_m\downarrow}$, that is taken from q^{m-1} to q^m, is such that $\Lambda_m \subseteq \text{up}^m$ whereas $\Gamma_m \not\subseteq \text{up}^m$.

Now from the prefix $(a_1, d_1)(a_2, d_2) \cdots (a_m, d_m)$ of v, *i.e.*, the first m inputs, and from the set of data $\{x_1, x_2, \cdots, x_i\}$, we construct the word $u_q = (a_1, y_1)(a_2, y_2) \cdots (a_m, y_m)$ for $q = (\ell, \text{up}, v, i-1)$ as follows: For all $1 \leq j \leq m$,

- if $\Lambda_j \neq \emptyset$, *i.e.*, some register $r \in$ up already stores the datum d_j, then $y_j = d_j$.

- if $\Lambda_j = \emptyset$, *i.e.*, none of the registers $r \in \mathsf{up}$ stores the datum d_j, then $y_j = d$ where $d \in \{x_1, x_2, \cdots, x_i\} \setminus \{\nu^{j-1}(r) \mid r \in \mathsf{up}\}$. The existence of such d is guaranteed since $|\mathsf{up}| = i - 1$ and $|\{x_1, x_2, \cdots, x_i\}| = i$. Moreover, since the transitions $\xrightarrow{(\bigwedge_{r \in \mathsf{up}} \neq r)\ a_j\ \Gamma_j \downarrow}$ have inequality guards for all registers, then changing the datum from d_j to y_j would result only in taking the same transition.

Observe that the data of the word u_q ranges over $\{x_1, \cdots, x_i\}$. As a result, all registers updated along the runs of $\mathcal{R}$ over u_q store some datum from $\{x_1, \cdots, x_i\}$. This argument shows that $\mathsf{post}([\![q]\!], u_q) \subseteq \langle L, i \rangle$. This concludes the step of induction, and completes the proof.

□

After reading some word that shrinks the infinite set of configurations of RAs to a finite set S of configurations, we generalize the *pairwise synchronization* technique (31) to synchronize configurations in S. By this generalization, we achieve the following Lemma 3.2, for which the detailed proof can be found in Appendix 6.

Lemma 3.2. *For all DRAs for which there exist synchronizing data words, there exists a synchronizing data word w such that $|w| \leq 2|R| + 1$.*

Given a 1-DRA $\mathcal{R}$, the synchronizing problem can be solved by (1) ensuring that from each location ℓ an update on the single register is achieved by going through inequality-guarded transitions, which can be done in NLOGSPACE. Lemma 3.1 suggests that feeding $\mathcal{R}$ consecutively with a single datum $x \in D$ is sufficient for this phase and the set of successors of $L \times D$ would be a subset of $L \times \{x\}$. Next (2) picking an arbitrary set $\{x, y, z\}$ of data including x, by Lemma 3.2 and the pairwise synchronization technique, the problem reduces to the synchronizing problem for DFAs where data in registers and input data extend locations and the alphabet: $Q = L \times \{x, y, z\}$ and $\Sigma \times \{x, y, z\}$. Since a 1-DRA, where all transitions update the register and are guarded with true, models a DFA, we obtain the next theorem.

Theorem 3.3. *The synchronization problem for 1-DRAs is* NLOGSPACE*-complete.*

We provide a family of DRAs, for which a linear bound on the data efficiency of synchronizing data words, depending on the number of registers, is necessary. This necessary and sufficient bound is crucial to establish membership of synchronizing DRAs in PSPACE.

Lemma 3.4. *There is a family of single-letter DRAs $(\mathcal{R}_n)_{n \in \mathbb{N}}$, with $n = |R|$ registers and $O(n)$ locations, such that all synchronizing data words have data efficiency $\Omega(n)$.*

Proof. The family of RAs $\mathcal{R}_n (n \in \mathbb{N})$ are defined over an infinite data domain D. The RA $\mathcal{R}_n$ has n registers and a single letter a. The structure of $\mathcal{R}_n$ is composed of two distinguished locations init and synch and two chains, where each chain has n locations: $\ell_1, \ell_2, \cdots, \ell_n$ and $\ell'_1, \ell'_2, \cdots, \ell'_n$. The RA $\mathcal{R}_3$ is shown in Figure 1. The only transition in synch is a self-loop with update on all n registers, thus $\mathcal{R}_n$ can only be synchronized in synch. There are two transitions in init, each going to one of the chains:

$$\mathsf{init} \xrightarrow{=r_1\ a\ r_1\downarrow} \ell_1 \text{ and } \mathsf{init} \xrightarrow{\neq r_1\ a\ r_1\downarrow} \ell'_1.$$

Then, $\mathsf{post}(\{\mathsf{init}\} \times D^n, (a, x)) = \{\ell_1, \ell'_1\} \times (\{x\} \times D^{n-1})$ for all $x \in D$.

From $\{\ell_1, \ell'_1\} \times (\{x\} \times D^{n-1})$, informally speaking, in both chains the respective i-th locations are simultaneously reached after inputting i distinct data: for all $1 \leq i < n$, in each ℓ_i and ℓ'_i there are two transitions. One transition is a self-loop, with a satisfied equality guard on at least one of the updated registers $r_1, \ldots, r_i$ so far. The other transition goes to the next location ℓ_{i+1} in the chain, with an inequality guard on all updated registers $r_1, r_2, \cdots, r_i$ so far, and an update on the next register r_{i+1}.

$$\ell_i \xrightarrow{\bigvee_{r\in\{r_1,\cdots,r_i\}}(=r_i)\ \ a} \ell_i \text{ and } \ell_i \xrightarrow{\bigwedge_{r\in\{r_1,\cdots,r_i\}}(\neq r_i)\ \ a\ \ r_{i+1}\downarrow} \ell_{i+1},$$
$$\ell'_i \xrightarrow{\bigvee_{r\in\{r_1,\cdots,r_i\}}(=r_i)\ \ a} \ell'_i \text{ and } \ell'_i \xrightarrow{\bigwedge_{r\in\{r_1,\cdots,r_i\}}(\neq r_i)\ \ a\ \ r_{i+1}\downarrow} \ell'_{i+1}.$$

At the last locations ℓ_n and ℓ'_n of the two chains, there is one transition with inequality guards on all registers leaving the chain to synch, and there is one transition which is, again, a self-loop with an equality constraint for at least one of the registers.

$$\ell_n \xrightarrow{\bigwedge_{r\in R}(\neq r_i)\ \ a\ \ R\downarrow} \text{synch and } \ell_n \xrightarrow{\text{else}\ \ a} \ell_n \qquad\qquad \ell'_n \xrightarrow{\bigwedge_{r\in R}(\neq r_i)\ \ a\ \ R\downarrow} \text{synch and } \ell'_n \xrightarrow{\text{else}\ \ a} \ell'_n.$$

By construction, we see that $n+1$ distinct data values must be read for reaching synch from the infinite set $\{\text{init}\}\times D^n$. Since $\mathcal{R}_n$ can only be synchronized in synch, all synchronizing data words must have data efficiency at least $n+1 \in O(n)$.

It remains to prove that $\mathcal{R}_n$ has indeed some synchronizing word. Let $\{x_1, x_2, \cdots, x_{n+1}\}$ be a set of $n+1$ distinct data values and $w_{\text{synch}} = (a, x_1)(a, x_2)\cdots(a, x_n)(a, x_{n+1})$. For the configuration space $L = \{\text{init}, \text{synch}, \ell_1, \cdots, \ell_n, \ell'_1, \cdots, \ell'_n\}$, observe that $\text{post}(L \times D^n, w_{\text{synch}}) = \{(\text{synch}, x_{n+1})\}$ and $|\text{data}(w_{\text{synch}})| = n+1$. The proof is complete. □

Theorem 3.5. *The synchronizing problem for k-DRAs is* PSPACE*-complete.*

Proof. (Sketch) The synchronization problem for k-DRA is in PSPACE using the following co-(N)PSPACE algorithm: (1) pick a set $X = \{x_1, x_2, \cdots, x_{2k+1}\}$ of distinct data values. (2) guess some location $\ell \in L$ and check if there is no word $w \in (\Sigma \times \{x_1, x_2, \cdots, x_k\})^*$ with length $|w| \leq 2^{k|L||\Sigma|}$ such that along firing inequality-guarded (on all k registers) transitions, some registers are not updated. If (2) is satisfied, then return "no" (meaning that there is no synchronizing data word for the input k-DRA). Otherwise, (3) guess two configurations $q_1, q_2 \in L \times X^k$ such that there is no word $w \in (\Sigma \times X)^*$ with length $|w| \leq 2^{(2k+1)|L||\Sigma|}$ such that $|\text{post}(\{q_1, q_2\}, w)| = 1$. If (3) is satisfied, then the algorithm returns "no"; Otherwise return "yes".

For PSPACE-hardness, we adapt an established reduction (see, e.g., (14)) from the non-emptiness problem for k-DRA, see Appendix 6. The result then follows by PSPACE-completeness of the non-emptiness problem for k-DRA (11).

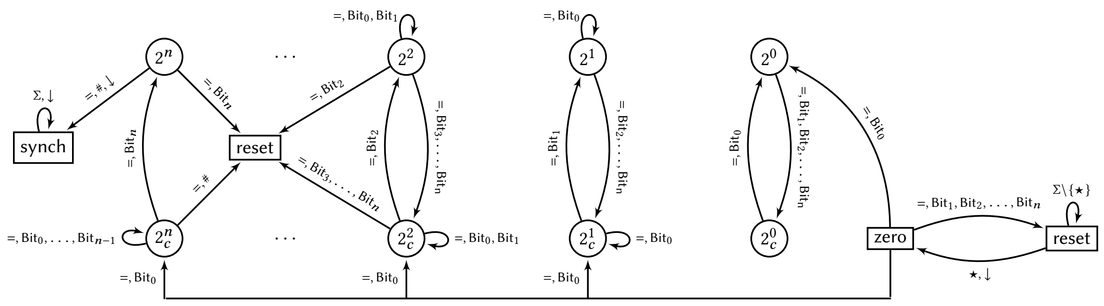
□

4 SYNCHRONIZING DATA WORDS FOR NRAs

In this section, we study the synchronizing problems for NRAs. We slightly update a result in (14) to present a general reduction from the *non-universality* problem to the synchronizing problem for NRAs. This reduction proves the *undecidability* result for the synchronizing problem for k-NRAs, and Ackermann-hardness in 1-NRAs. We then prove that for 1-NRAs, the synchronizing and non-universality problems are indeed interreducible, which completes the picture by Ackermann-completeness of the synchronizing problem for 1-NRAs.

In the nondeterministic synchronization setting, we present two kinds of *counting features*, which are useful for later constructions. For the first one, we define a family $(\mathcal{R}_{\text{counter}(n)})_{n\in\mathbb{N}}$ of 1-NRAs with size only linear in n, where an input datum $x \in D$ must be read 2^n times to achieve synchronization.

Lemma 4.1. *There is a family of* 1*-NRAs $(\mathcal{R}_{\text{counter}(n)})_{n\in\mathbb{N}}$ with $O(n)$ locations, such that for all synchronizing data words w, some datum $d \in \text{data}(w)$ appears in w at least 2^n times.*



2^n
2^2
2^1
2^0
2^n_c
2^2_c
2^1_c
2^0_c
synch
reset
zero
reset
$\Sigma, \downarrow$
$=, \#, \downarrow$
$=, \mathrm{Bit}_n$
$=, \mathrm{Bit}_2$
$=, \mathrm{Bit}_0, \mathrm{Bit}_1$
$=, \mathrm{Bit}_0$
$=, \#$
$=, \mathrm{Bit}_3, \dots, \mathrm{Bit}_n$
$=, \mathrm{Bit}_2, \dots, \mathrm{Bit}_n$
$=, \mathrm{Bit}_1, \mathrm{Bit}_2, \dots, \mathrm{Bit}_n$
$=, \mathrm{Bit}_1$
$=, \mathrm{Bit}_0, \dots, \mathrm{Bit}_{n-1}$
$\Sigma \backslash \{\star\}$
$\star, \downarrow$

Fig. 4. A partial picture of the 1-NRA $\mathcal{R}_{\mathrm{counter}(n)}$ implementing a binary counter, for $n \geq 3$. We use two copies of reset in order to avoid crossing edges. All locations have inequality-guarded self-loops for all letters in $\Sigma \backslash \{\star\}$. All missing equality-guarded $\star$-transitions are directed to zero. For all $0 \leq i < n$, missing equality-guarded #-transitions from 2^i_c are guided to synch with an update on the register. All other non-depicted equality-guarded transitions are directed to reset, and inequality-guarded transitions are self-loops.

Proof. (Sketch) The 1-NRA $\mathcal{R}_{\mathrm{counter}(n)}$ shown in Figure 4 encodes a binary counter: in every synchronizing data word w, some datum $x \in \mathrm{data}(w)$ must appear at least 2^n times. The location synch has self-loops on all letters, thus, $\mathcal{R}_{\mathrm{counter}(n)}$ can only be synchronized in synch. Generally speaking, the counting involves an *initializing process* and several *incrementing processes*. The *initializing process* is started by firing an $\star$-transition, which places a token, let us say: an x-token, into location zero. This sets the counter to `0`. Note that firing $\star$-transitions is the only way to guide tokens out of reset; hence, whenever there is some token in reset, a new initializing process must be started. We use this to enforce a new initializing process whenever some transition is fired that is incorrect with respect to the incrementing process.

An *incrementing process* can be set off by inputting the datum x via equality guards. The numbers $1 \leq m \leq 2^n$ are represented by placing a copy of the x-token in the locations corresponding to the binary representation of m. An x-token in location 2^i (in 2^i_c, respectively) means that the i-th least significant bit in the binary representation is set to 1 (to `0`, respectively). First, a Bit_0-transition places a copy of the x-token in each of $\{2^n_c, \dots, 2^2_c, 2^1_c, 2^0\}$ to represent `0. . . 001`. In each increment step the x-tokens are re-placed by firing specific Bit_i-transitions ($0 \leq i \leq n$), following the standard procedure of binary addition. At the end, when finally a copy of the x-token resides in each of $\{2^n, 2^{n-1}_c, \dots, 2^0_c\}$ (representing `10. . . 0`), then #-transitions guide all tokens to location synch and finally synchronize $\mathcal{R}_{\mathrm{counter}}$. We present a detailed explanation of the structure of $\mathcal{R}_{\mathrm{counter}(n)}$ in Appendix 7. □

With the second kind of counting feature, we aim to give the intuition why synchronization of 1-NRAs is hard. In Lemma 4.2, we define a family of 1-NRAs (again with only $O(n)$ locations), where tower(n) distinct data must be read to gain synchronization. Recall, *e.g.* from (27), that the function tower resides at level three of the infinite *Ackermann hierarchy* $(A_k)_{k \in \mathbb{N}}$ of fast-growing functions $A_i : \mathbb{N} \to \mathbb{N}$, inductively defined by $A_1(n) = 2n$ and $A_{k+1}(n) = A^n_k(1) = \underbrace{A_k(\dots(A_k}_{n \text{ times}}(n))\dots)$. Hence, applying doub $\overset{\mathrm{def}}{=} A_1$, exp $\overset{\mathrm{def}}{=} A_2$, and tower $\overset{\mathrm{def}}{=} A_3$, respectively, on some natural number n results in some number that is *double*, *exponential*, and *tower*, respectively, in n. The function $A_\omega(n) = A_n(n)$ is a non-primitive recursive Ackermann-like function, defined by diagonalization.

Lemma 4.2. *There is a family of* 1*-NRAs* $(\mathcal{R}_{\mathrm{tower}(n)})_{n \in \mathbb{N}}$ *with* $O(n)$ *locations, such that* $|\mathrm{data}(w)| \geq \mathrm{tower}(n)$ *for all synchronizing data words* w.

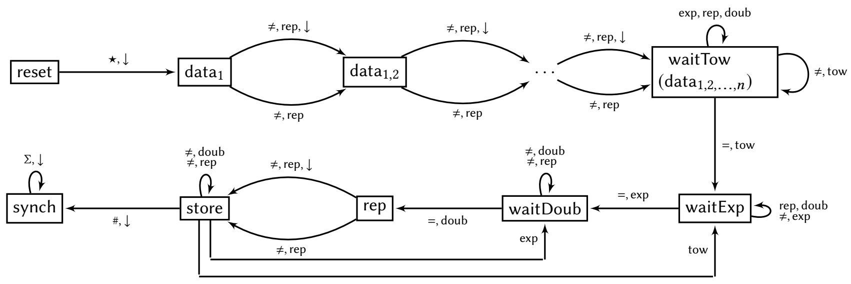


Fig. 5. A partial illustration of the 1-NRA $\mathcal{R}_{\text{tower}(n)}$ for $n \geq 3$. All $\star$-transitions are guided to data$_1$ with an update on the register. All other missing non-depicted transitions are directed to reset.

Proof. The domain of the family of 1-NRAs $(\mathcal{R}_{\text{tower}(n)})_{n\in\mathbb{N}}$ is the natural numbers $\mathbb{N}$. The alphabet of $\mathcal{R}_{\text{tower}(n)}$ is $\Sigma = \{\#, \star, \text{rep}, \text{doub}, \text{exp}, \text{tow}\}$. The structure of $\mathcal{R}_{\text{tower}(n)}$ is composed of n locations data$_1$, data$_{1,2}$, $\cdots$, data$_{1,2,\cdots,n}$ and 6 more locations reset, synch, store, rep, waitDoub, waitExp. The general structure of $\mathcal{R}_{\text{tower}(n)}$ is partially depicted in Figure 5. The RA $\mathcal{R}_{\text{tower}(n)}$ is such that $|\text{data}(w)| \geq \text{tower}(n)$ for all synchronizing data words w.

All transitions in synch are self-loops with an update on the register synch $\xrightarrow{\Sigma\ r\downarrow}$ synch; thus, $\mathcal{R}_{\text{tower}(n)}$ can only be synchronized in synch. Moreover, synch is only accessible from store by a #-transition. Assuming w is one of the shortest synchronizing words, we see that $\text{post}(L \times D, w) = \{(\text{synch}, x)\}$, where w ends with $(\#, x)$.

From all locations $\ell \in L \setminus \{\text{synch}\}$, we have $\ell \xrightarrow{\star\ r\downarrow}$ data$_1$; we say that $\star$-transitions *reset* $\mathcal{R}_{\text{tower}(n)}$. Moreover, the only outgoing transition in location reset is the $\star$-transition. Thus, a *reset* must occur in order to synchronize $\mathcal{R}_{\text{tower}(n)}$. After this forced reset, say on reading $(\star, 1)$, the set of reached configurations is $\{(\text{data}_1, 1), (\text{synch}, 1)\}$. Since resetting is inefficient, we try to avoid it; we call all transitions leading to reset *inefficient.*
For all locations data$_{1,\cdots,i}$ with $1 \leq i < n$, we define the two transitions

$$\text{data}_{1,\cdots,i} \xrightarrow{\neq r\ \text{rep}} \text{data}_{1,\cdots,i+1} \quad \text{and} \quad \text{data}_{1,\cdots,i} \xrightarrow{\neq r\ \text{rep}\ r\downarrow} \text{data}_{1,\cdots,i+1}.$$

All other transitions in data$_{1,\cdots,i}$ are inefficient and directed to reset. Below, we rename data$_{1,2,\cdots,n}$ to waitTow. We partially depict the transitions from waitTow, waitExp, waitDoub, rep and store in Figure 5. All transitions are inefficient, except

- waitTow $\xrightarrow{=r\ \text{tow}}$ waitExp, waitTow $\xrightarrow{\neq r\ \text{tow}}$ waitTow, and waitTow $\xrightarrow{\sigma}$ waitTow for all $\sigma \in \{\text{doub}, \text{exp}, \text{rep}\}$.
- waitExp $\xrightarrow{=r\ \text{exp}}$ waitDoub, waitExp $\xrightarrow{\text{doub}}$ waitExp and waitExp $\xrightarrow{\text{rep}}$ waitExp.
- waitDoub $\xrightarrow{=r\ \text{doub}}$ rep, waitDoub $\xrightarrow{\neq r\ \text{doub}}$ waitDoub and waitDoub $\xrightarrow{\neq r\ \text{rep}}$ waitDoub,
- rep $\xrightarrow{\neq r\ \text{rep}}$ store and rep $\xrightarrow{\neq r\ \text{rep}\ r\downarrow}$ store,
- store $\xrightarrow{\text{tow}}$ waitExp, store $\xrightarrow{\text{exp}}$ waitDoub, store $\xrightarrow{\neq r\ \text{doub}}$ store and store $\xrightarrow{\neq r\ \text{rep}}$ store, and
- store $\xrightarrow{\#\ r\downarrow}$ synch.

We remark that store $\xrightarrow{\#\ r\downarrow}$ synch is the only #-transition that is not inefficient. This implies that for efficiently synchronizing $\mathcal{R}_{\text{tower}(n)}$, one needs to re-move all produced tokens to store before firing a #-transition. The main issue in re-moving produced tokens, however, is that some inequality-guarded transitions are unavoidable, and these transitions may *replicate* the tokens. For example, if one token is in data$_1$, firing two transitions data$_1$ $\xrightarrow{\neq r\ \text{rep}}$ data$_{1,2}$ and

$\mathsf{data}_1 \xrightarrow{\neq r\ \ \mathsf{rep}\ \ r\downarrow} \mathsf{data}_{1,2}$ replicates it to two tokens in $\mathsf{data}_{1,2}$. Using this, one can implement *doubling*, *exponentialization*, and *towering* of distinct tokens, as explained in the following.

Doubling: Assume that there are n distinct tokens $\{1, 2, \ldots, n\}$ in $\mathsf{waitDoub}$. The only efficient transition is $\mathsf{waitDoub} \xrightarrow{=r\ \ \mathsf{doub}} \mathsf{waitRep}$. In particular, all $\{\#, \mathsf{exp}, \mathsf{tow}\}$-transitions activate a reset. As a result, as long as some token is in $\mathsf{waitDoub}$, $\{\#, \mathsf{exp}, \mathsf{tow}\}$-transitions should be avoided for the sake of efficiency. This implies that for all $1 \leq i \leq n$, the i-token in $\mathsf{waitDoub}$ can leave the location only individually on the input (doub, i). Now, inputting (doub, i) moves the i-token to $\mathsf{waitRep}$. Here the i-token must immediately move on to store via the inequality-guarded rep-transitions, which will replicate the i-token into two tokens. Note that we must fire rep-transitions with some "fresh" datum j such that $j \notin \{1, \ldots, n\}$, otherwise a reset is evoked. (For simplicity, we use $j = i + n$ by convention.) It can now be easily seen that the only efficient way to guide all n tokens out of $\mathsf{waitDoub}$ is by inputting the data word

$$w_{\mathsf{doub}(n)} = (\mathsf{doub}, 1)(\mathsf{rep}, n + 1)(\mathsf{doub}, 2)(\mathsf{rep}, n + 2) \ldots (\mathsf{doub}, n)(\mathsf{rep}, 2n),$$

which puts $2n$ distinct tokens into store.

Exponentialization: Assume there are n distinct tokens $\{1, 2, \ldots, n\}$ in $\mathsf{waitExp}$. The only efficient transition is $\mathsf{waitExp} \xrightarrow{=r\ \ \mathsf{exp}} \mathsf{waitDoub}$. In particular, all $\{\#, \mathsf{tow}\}$-transitions activate a reset, and should be avoided as long as some token is in $\mathsf{waitExp}$. This implies that for all $1 \leq i \leq n$, the i-token in $\mathsf{waitExp}$ can leave the location only individually on the input (exp, i). Now, inputting $(\mathsf{exp}, 1)$ moves the 1-token to $\mathsf{waitDoub}$. From above we know that the only efficient way for guiding a single token in $\mathsf{waitDoub}$ towards synchronization is by inputting the data word $w_{\mathsf{doub}(1)}$, resulting in two distinct tokens in store: 1 and 2. We can now proceed to remove the 2-token from $\mathsf{waitExp}$ by inputting $(\mathsf{exp}, 2)$. Note that this also guides the $\{1, 2\}$-tokens residing in store to $\mathsf{waitDoub}$. Again, for efficient synchronization, we must input the data word $w_{\mathsf{doub}(2)}$, which results in four distinct tokens $\{1, 2, 3, 4\}$ in store. It is now easy to see that the only efficient way to guide all n tokens out of $\mathsf{waitExp}$ is by inputting the data word

$$w_{\mathsf{exp}(n)} = (\mathsf{exp}, 1) \cdot w_{\mathsf{doub}(1)} \cdot (\mathsf{exp}, 2) \cdot w_{\mathsf{doub}(2)} \cdot (\mathsf{exp}, 3) \cdot w_{\mathsf{doub}(4)} \cdot \ldots \cdot (\mathsf{exp}, n) \cdot w_{\mathsf{doub}(2^{n-1})},$$

which puts 2^n distinct tokens into store.

Towering: Assume there are n distinct tokens $\{1, 2, \ldots, n\}$ in $\mathsf{waitTow}$. The only efficient transition is $\mathsf{waitExp} \xrightarrow{=r\ \ \mathsf{tow}} \mathsf{waitExp}$. In particular, firing $\#$-transitions activates a reset, and should be avoided as long as some token is in $\mathsf{waitTow}$. This implies that for all $1 \leq i \leq n$, the i-token in $\mathsf{waitTow}$ can leave the location only individually on the input (tow, i). Now, inputting $(\mathsf{exp}, 1)$ moves the 1-token to $\mathsf{waitExp}$. From above we know that the only efficient way for guiding a single token in $\mathsf{waitTow}$ towards synchronization is by inputting the data word $w_{\mathsf{exp}(1)}$, resulting in two distinct tokens in store: 1 and 2. We can now proceed to remove the 2-token from $\mathsf{waitTow}$ by inputting $(\mathsf{tow}, 2)$. Note that this also guides the $\{1, 2\}$-tokens residing in store to $\mathsf{waitExp}$. Again, for efficient synchronization, we must input the data word $w_{\mathsf{exp}(2)}$, which results in four distinct tokens $\{1, 2, 3, 4\}$ in store. It is now easy to see that the only efficient way to guide all n tokens out of $\mathsf{waitTow}$ is by inputting the data word

$$w_{\mathsf{tow}(n)} = (\mathsf{tow}, 1) \cdot w_{\mathsf{exp}(1)} \cdot (\mathsf{tow}, 2) \cdot w_{\mathsf{exp}(2)} \cdot (\mathsf{tow}, 3) \cdot w_{\mathsf{exp}(4)} \cdot \ldots \cdot (\mathsf{tow}, n) \cdot w_{\mathsf{exp}(\mathsf{tower}(n-1))},$$

which puts $\mathsf{tower}(n)$ distinct tokens into store.

Now, after the (forced) initial reset by firing $\star$-transitions, it is easy to see that the only data word that advances in synchronizing is $(\mathsf{rep}, 2)(\mathsf{rep}, 3) \cdots (\mathsf{rep}, n)$. It replicates the 1-token to n distinct tokens $1, 2, \cdots, n$, which are placed into $\mathsf{waitTow}$. From above we know that the only efficient way to guide all n tokens out of $\mathsf{waitTow}$ is by inputting

$w_{\mathrm{tow}(n)}$, which places tower(n) distinct tokens into store. We can now fire #-transitions to synchronize $\mathcal{R}_{\mathrm{tower}(n)}$ without evoking a reset, but note that due to the equality guard at the #-transition from store to synch, each of the tower(n) distinct tokens in store can move to synch only individually. This implies $|\mathrm{data}(w)| \geq \mathrm{tower}(n)$ for all synchronizing words w. □

We can now use similar ideas as in Lemma 4.2 for defining a family of 1-NRAs $\mathcal{R}_{A_n(m)}$ $(n, m \in \mathbb{N})$ such that all synchronizing data words of $\mathcal{R}_{A_n(m)}$ have data efficiency at least $A_n(m)$, where A_n is at level n of the Ackermann hierarchy. This gives already a good intuition that the synchronizing problem for NRAs must be Ackermann-hard, even if only a single register is used. In the following, we prove that the synchronizing problem and the non-universality problem for NRAs are interreducible

To define the language of a given NRA $\mathcal{R}$, we equip it with an initial location ℓ_{in} and a set L_{f} of accepting locations, where, without loss of generality, we assume that all outgoing transitions from ℓ_{in} update all registers. The language $L(\mathcal{R})$ is the set of all data words $w \in (\Sigma \times D)^*$, for which there is a run from $(\ell_{\mathrm{in}}, v_{\mathrm{in}})$ to $(\ell_{\mathrm{f}}, v_{\mathrm{f}})$ such that $\ell_{\mathrm{f}} \in L_{\mathrm{f}}$ and $v_{\mathrm{in}}, v_f \in D^{|R|}$. The non-universality problem asks, given an RA, whether there exists some data word w over Σ such that $w \notin L(\mathcal{R})$. We adopt an established reduction in (14) to provide the following Lemma.

Lemma 4.3. *The non-universality problem is reducible to the synchronizing problem for NRAs.*

The detailed proof can be found in Appendix 7. As an immediate result of Lemma 4.3 and the undecidability of the non-universality problem for NRAs (Theorems 2.7 and 5.4 in (11)), we obtain the following theorem.

Theorem 4.4. *The synchronizing problem for NRAs is undecidable.*

Next, we present a reduction showing that, for 1-NRAs, the synchronizing problem is reducible to the non-universality problem, providing the tight complexity bounds for the synchronizing problem.

Lemma 4.5. *The synchronizing problem is reducible to the non-universality problem for* 1*-NRAs.*

Proof. We establish a reduction from the synchronizing problem to the non-universality problem for 1-NRAs as follows. Given a 1-NRA $\mathcal{R} = \langle L, R, \Sigma, T \rangle$, we construct a 1-NRA $\mathcal{R}_{\mathrm{comp}}$ equipped with an initial location and a set of accepting locations such that $\mathcal{R}$ has some synchronizing words if, and only if, there exists some data word that is not in $L(\mathcal{R}_{\mathrm{comp}})$.

First, we see that an analogue of Lemma 3.1 holds for 1-NRAs: for all 1-NRAs with some synchronizing data word, there exists some word w with data efficiency 1 such that $\mathrm{post}(L \times D, w) \subseteq L \times \mathrm{data}(w)$. For all locations $\ell \in L$, such a data word must update the register by firing an inequality-guarded transition that is reached only via inequality-guarded transitions; this can be checked in NLOGSPACE. Given $\mathcal{R}$, we assume that such a data word w always exists; otherwise, we define $\mathcal{R}_{\mathrm{comp}}$ to be a 1-NRA with a single (initial and accepting) location equipped with self-loops for all letters, so that $L(\mathcal{R}_{\mathrm{comp}}) = (\Sigma \times D)^*$. Given $\mathrm{data}(w) = \{x\}$, we say that $\mathcal{R}$ has some synchronizing word v if $\mathrm{post}(L \times \{x\}, v)$ is a singleton.

Second, we define a data language lang such that data words in this language are encodings of the synchronizing process. Let $L = \{\ell_1, \ell_2, \cdots, \ell_n\}$ be the set of locations and x, y two distinct data. Informally, each data word in lang, if there exists any, starts with the

- *initial block*: a delimiter $(\star, y)$, the sequence $(\ell_1, x), (\ell_2, x), \cdots, (\ell_n, x)$ and an input $(a, d) \in \Sigma \times D$ as the beginning of a synchronizing word. The initial block is followed by several

- *normal blocks*: the delimiter $(\star, y)$, the set of successor configurations reached from the configurations and the input of the previous block, and the next input (a', d') of the synchronizing data word. The data word finally ends with the
- *final block*: the delimiter $(\star, y)$, a single successor configuration reached from the configurations and the input of the previous block, and the delimiter $(\star, y)$.

Formally, the language lang is defined over the alphabet $\Sigma_{\mathsf{lang}} = \Sigma \cup L \cup \{\star\}$ where $\star \notin \Sigma \cup L$. It contains all data words u that satisfy the following *membership conditions*:

(1) The data words u starts with $(\star, y)(\ell_1, x), (\ell_2, x), \cdots, (\ell_n, x)$ for some $x, y \in D$ with $y \neq x$; this condition guarantees the correctness of the encoding for the initial block.
(2) Let $\mathsf{proj}(u)$ be the projection of u into Σ_{lang} (*i.e.*, omitting the data values). Then there exists some $\ell_{\mathsf{synch}} \in L$ where $\mathsf{proj}(u) \in (\star L^+ \Sigma)^+ \star \ \ell_{\mathsf{synch}} \star$. This condition guarantees the right form of data words as the encodings of synchronizing processes.

The next two conditions guarantee the uniqueness of the delimiter:

(3) The letter $\star$ in u occurs only with datum y.
(4) No other letter in u occurs with datum y.

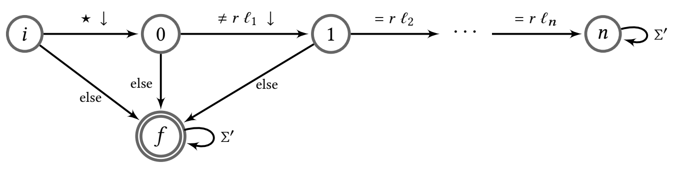
The next three conditions guarantee that all the successors that can be reached from configurations and inputs in each block are correctly inserted in the next block. For all $(\ell, x) \in L \times D$ and $(a, d) \in \Sigma \times D$ in the same block,

(5) if $x = d$ and there exists a transition $\ell \xrightarrow{=r \ \ a} \ell'$ (with or without update), then (ℓ', x) must be in the next block.
(6) if $x \neq d$ and there exists a transition $\ell \xrightarrow{\neq r \ \ a} \ell'$, then (ℓ', x) must be in the next block.
(7) if $x \neq d$ and there exists a transition $\ell \xrightarrow{\neq r \ \ a \ \ r\downarrow} \ell'$ then (ℓ', d) must be in the next block.

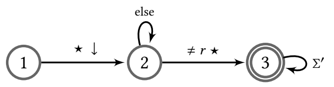
By construction, we see that the RA $\mathcal{R}$ has some synchronizing data word if, and only if, $\mathsf{lang} \neq \emptyset$. Below, we construct a 1-NRA $\mathcal{R}_{\mathsf{comp}}$ that accepts the complement of lang. Then, the RA $\mathcal{R}$ has some synchronizing data word if, and only if, there exists some data word that is not in $L(\mathcal{R}_{\mathsf{comp}})$.

The 1-NRA $\mathcal{R}_{\mathsf{comp}}$ is the union of several 1-NRAs that are in the family of 1-NRAs $\mathbf{R}_1, \mathbf{R}_2, \cdots, \mathbf{R}_7$, where an 1-NRA is in the family $\mathbf{R}_i$ if it violates the i-th condition among the membership conditions in lang.

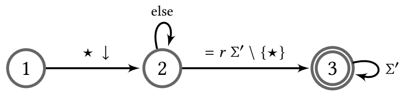
(1) Family $\mathbf{R}_1$: We add a 1-NRA that accepts data words not starting with $(\star, y)(\ell_1, x), \cdots, (\ell_n, x)$.



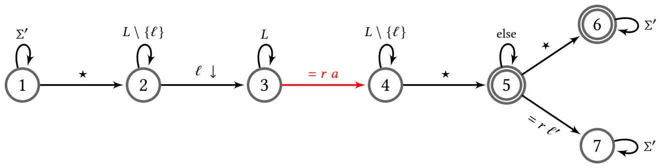
(2) Family $\mathbf{R}_2$: we add a DFA that accepts data words u such that $\mathsf{proj}(u)$ is not in the regular language $(\star L^+ \Sigma)^+ \star \ \ell_{\mathsf{synch}} \star$.
(3) Family $\mathbf{R}_3$: we add a 1-NRA that accepts data words in which the two delimiters $\star$ have different data.



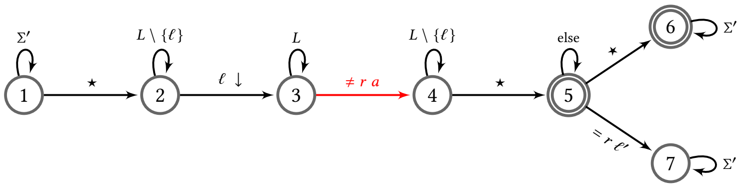
(4) Family $\mathbf{R}_4$: we add a 1-NRA that accepts data words in which the datum of first $\star$ is not used only by occurrences of $\star$.



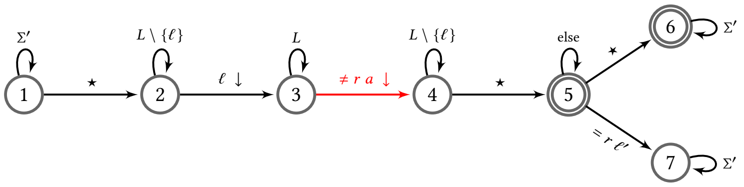
(5) Family $\mathbf{R}_5$: for all transitions $\ell \xrightarrow{=r\ \ a} \ell'$, we add a 1-NRA that only accepts data words such that one block contains some (ℓ, x) and (a, d) with $x = d$ where the next block does not have (ℓ', x).



(6) Family $\mathbf{R}_6$: for all transitions $\ell \xrightarrow{\neq r\ \ a} \ell'$, we add a 1-NRA that only accepts data words such that one block contains some (ℓ, x) and (a, d) with $x \neq d$ where the next block does not have (ℓ', x).



(7) Family $\mathbf{R}_7$: for all transitions $\ell \xrightarrow{\neq r\ \ a\ \ r\downarrow} \ell'$, we add a 1-NRA that only accepts data words such that one block contains some (ℓ, x) and (a, d) with $x \neq d$ where the next block does not have (ℓ', d).



The proof is complete. □

By Lemmas 4.3 and 4.5 and Ackermann-completeness of the non-universality problem for 1-NRA, which follows from Theorem 2.7 and the proof of Theorem 5.2 in (11), and the result for counter automata with incrementing errors in (18), we obtain the following theorem.

Theorem 4.6. *The synchronizing problem for* 1*-NRAs is* Ackermann*-complete.*

5 BOUNDED SYNCHRONIZING DATA WORDS FOR NRAs

The synchronizing problem for NRAs is undecidable in general. In this section, we study, for NRAs, the bounded synchronizing problem, that requires the synchronizing data words to have at most a given length.

To decide the synchronizing problem in 1-RAs, in both deterministic and nondeterministic settings, we chiefly rely on Lemma 3.1. We thus assume that the RA inputs the same datum x (chosen arbitrary) as many times as necessary to have the successor set included in $L \times \{x\}$; next, we synchronize this successor set in a singleton. The RA $\mathcal{R}$ shown in Figure 6 shows that this approach is not useful when the length of synchronizing words is asked to not exceed a given bound. Observe that the data word $(a, x)(b, y)(b, z)$ is synchronizing with length 3 (not exceeding the bound 3). However, all synchronizing data words that repeat a datum such as x, to first bring the RA to a finite set, have length

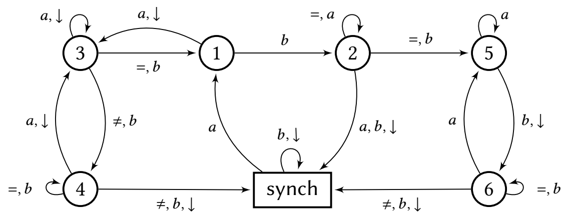


Fig. 6. An RA with synchronizing data word $(a, x)(b, y)(b, z)$ with three distinct data values x, y, z. The approach of using a unique data value to shrink the infinite set of configurations to a finite subset only yields synchronizing data words of length greater than 3.

at least 4. The example shows that one cannot rely on the techniques developed in Section 4 to decide the bounded synchronization problem for NRA.

In this section, we prove

Theorem 5.1. *The bounded synchronization problem for NRAs is* NEXPTIME*-complete.*

The NEXPTIME-membership of the bounded synchronization problem can be easily seen: guess a data word w with $|w| \leq N'$ and check in EXPTIME whether w is synchronizing. Our main contribution is to prove the NEXPTIME-hardness of this problem, which is based on the binary counting feature in NRAs.

The proof is by a reduction from the bounded non-universality problem for *regular-like expressions with squaring*. A regular-like expression with squaring over a finite alphabet A is a well-parenthesized expression built by constants $a \in A$ and the empty word ε, two binary operations $\cdot$ (concatenation) and $+$ (union), and a unary operation ^{2} (squaring). The language $L(E)$ of such expressions E is defined inductively as for regular expressions, where $L(E^2) = L(E) \cdot L(E)$. In the following, given an expression E and $k \in \mathbb{N}$, we write E^{2^k} short for

$$\underbrace{(\dots((E^2)^2)\dots)^2}_{\text{k times}}.$$

Given $B = \{a_1, \dots, a_\ell\} \subseteq A$, we write B^{2^k} short for $(a_1 + \dots + a_\ell)^{2^k}$, and similarly $(B + \varepsilon)^{2^k}$ for $(a_1 + \dots + a_\ell + \varepsilon)^{2^k}$. We say that an expression E is *simple* if all squaring expressions occurring in E are of the form B^{2^k} or $(B + \varepsilon)^{2^k}$, where $B \subseteq A$ and $k \in \mathbb{N}$.

The *bounded non-universality problem for regular-like expressions with squaring* asks, given a regular-like expression with squaring E and length $N \in \mathbb{N}$ written in binary, whether there exists some string with length at most N that is not in $L(E)$; in other words whether $A^{\leq N} \not\subseteq L(E)$, where $A^{\leq N}$ denotes the set of all strings over A of length at most N.

Lemma 5.2 (an immediate conclusion from the hardness proof in (29)). *The bounded non-universality problem for regular-like expressions with squaring is* NEXPTIME*-complete, even if the regular-like expressions are simple.*

Given a simple regular-like expression with squaring E and length $N \in \mathbb{N}$, we construct a 1-NRA $\mathcal{R}$ and $N' \in \mathbb{N}$, such that $L(E)$ is bounded non-universal if, and only if, $\mathcal{R}$ has some synchronizing data word with length at most N'. The reduction is such that the RA $\mathcal{R}$ checks whether a string $u \in A^*$ is a witness for bounded non-universality of E, *i.e.*, whether u is not generated by E and whether the length of u is not greater than N. In this case (and only in this case) $\mathcal{R}$ synchronizes.

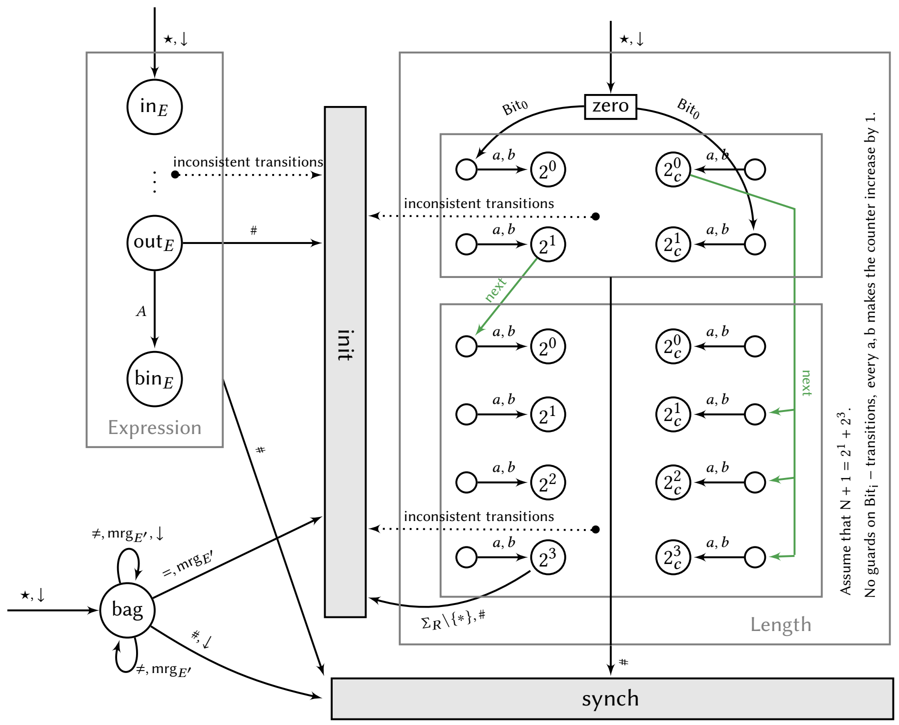


Fig. 7. Sketch of the reduction of the bounded non-universality problem for regular-like expressions to the bounded synchronization problem in NRAs.

To accomplish this, the RA $\mathcal{R}$ consists of two main gadgets called *expression gadget* and *length gadget*, as well as three distinguished locations synch, init and bag. The location synch is the location in which $\mathcal{R}$ synchronizes iff there exists some string u that is a witness for bounded non-universality. For checking whether a string u is a witness for bounded non-universality, $\mathcal{R}$ *simulates* the generation of u by the expression E. By construction, whenever a token is in the location init, the simulation must restart. At the beginning of every simulation, by inputting the distinguished letter ⋆, a single token is placed in each of the initial locations of the expression gadget and the length gadget as well as into the location bag. At the same time, all tokens residing in other locations of $\mathcal{R}$ are removed. The simulation then proceeds by guessing a string $u \in A^*$, letter-by-letter, and the simulation may be finalized by inputting the distinguished letter # to finally check whether u is a witness for bounded non-universality. When # is input to $\mathcal{R}$, the length gadget checks whether $|u| \leq N$, and the expression gadget checks whether $u \notin L(E)$. If this is the case, all tokens in $\mathcal{R}$ move to synch; otherwise, all tokens move to init, where a new simulation, with a potentially new guessed string, may be restarted. The distinguished location bag is used to guarantee that, during a simulation, a *globally fresh* datum is generated whenever the expression gadget fires a mrg-transition (see below for details). For this, all tokens (*i.e.*, all generated data) are residing in bag during a simulation. Whenever a mrg-transition is fired together with some datum that is already residing in bag, the corresponding token moves to init, hence requiring a restart of a simulation.

In the following, we give the details about the construction of the expression gadget, length gadget, and the distinguished locations synch, init, and bag. We define $\mathcal{R}$ over the input alphabet $\Sigma_{\mathcal{R}}$, which depends on E and is defined in detail below.

The location synch**:** all transitions in synch are unguarded self-loops with update $\text{synch} \xrightarrow{\Sigma_{\mathcal{R}} \ r\downarrow} \text{synch}$; thus, $\mathcal{R}$ can only be synchronized in synch. Moreover, synch is only accessible by #-transitions, so that #-transitions must be fired to synchronize $\mathcal{R}$.

The location init**:** all transitions in init, except for $\star$-transitions, are self-loops; thus, the only way for a token to leave init is to fire a $\star$-transition. Since the location synch is the only location where $\mathcal{R}$ can synchronize, the role of init is to *enforce a new simulation.* We use this by sending tokens to init whenever the simulation is not faithful. A simulation is not faithful, if, for instance, the number of letters in A in the guessed string has already exceeded N. We call *inconsistent* all those transitions that send some token to init, and likewise, we call *consistent* all those transitions that do not send a token to init. A short witness for bounded non-universality of E should not fire inconsistent transitions, because it necessarily restarts a new simulation, implying that the prefix of computation until this new simulation could have been avoided.

By inputting $\star$, we start a new simulation. The $\star$-transitions in all locations in $\mathcal{R}$ (except for those in synch) place a single token into each of bag, synch, the initial locations of the expression gadget, and the initial location of the length gadget, which is the location called zero: we have, for all $\ell \in L\backslash\{\text{synch}\}$ and each initial location ℓ_{in} of the expression gadget,

$$\ell \xrightarrow{\star \ r\downarrow} \text{bag} \quad \ell \xrightarrow{\star \ r\downarrow} \text{synch}, \quad \ell \xrightarrow{\star \ r\downarrow} \ell_{\text{in}}, \text{ and } \ell \xrightarrow{\star \ r\downarrow} \text{zero}.$$

The length gadget: This gadget checks whether the length $|u|$ of the guessed string u during the current simulation is not greater than N. The length gadget counts the letters in A that are read during the current simulation. If the counter is at most N, then firing #-transitions will move successfully all tokens residing in $\mathcal{R}$ to synch. Otherwise, *i.e.*, if the counter is exceeding N, firing #-transitions will move at least one token to init, making a restart of a simulation necessary.

The gadget is constructed based on the family of counting RAs described in Lemma 4.1. We refer to members of the counting family using $\mathcal{R}_{\text{counter}(i)}$, where $\mathcal{R}_{\text{counter}(i)}$ counts until 2^i (see Figure 4). To construct the length gadget, we assume, without loss of generality, that $2^n \leq N + 1 < 2^{n+1}$, so that $n + 1$ bits are sufficient to encode $N + 1$ in binary. Consider the binary representation of $N + 1$ (the least significant bit first). The length gadget is a chain of (modified) counting RAs, where $\mathcal{R}_{\text{counter}(i)}$ is the j-th member in the chain, if the bit with 2^i-significance in the binary representation of N is the j-th bit set to 1. For example, if $N = 9$ and $N + 1 = 10 = 2^1 + 2^3 = (\texttt{1010})$, then the length gadget is a chain composed of RAs $\mathcal{R}_{\text{counter}(1)}$ and $\mathcal{R}_{\text{counter}(3)}$. Since $2^n \leq N + 1 < 2^{n+1}$, the chain always ends with the counter RA $\mathcal{R}_{\text{counter}(n)}$. We use a new letter next to move along the chain; see Figure 7. The detailed construction can be found in Appendix 8.

The expression gadget: recall that the role of the expression gadget is to check whether the guessed string u for the current simulation is not generated by E. To do so, starting with the single token initially placed into the initial locations of the expression gadget, the gadget simulates all possible ways in E to generate a (prefix of) u. During a simulation, whenever a prefix u' of u is generated by E, a single token will reach the distinguished location out_E of the expression gadget. Otherwise, if no prefix of u is generated by E, then there will be no token in out_E. The #-transition in out_E is directed to init, while the #-transitions in all other locations $\ell_E \neq \text{out}_E$ in the expression gadget are directed to synch, with an update on the register:

$$\text{out}_E \xrightarrow{\#} \text{init} \quad \text{and} \quad \ell_E \xrightarrow{\# \ \downarrow r} \text{synch}.$$

Intuitively speaking, #-transitions are fired to check whether the current string is a witness for bounded non-universality of E. Whenever such a check happens, all tokens in the expression gadget move to synch except for the potential token in out_E. A token in out_E moves to init to restart a new simulation. As described, in case there is no token in out_E, all other tokens move to synch, showing the existence of a string not generated by E, a witness for non-universality of E. Additionally, if the length of this witness is less than N, it is a witness for bounded non-universality.

The expression gadget is constructed inductively. It has a subexpression gadget for each subexpression E' of E. To have a faithful simulation, the subexpression gadgets follow the intuitive structure of putting a token in $\mathsf{out}_{E'}$ if, and only if, E' generates the prefix string processed so far. For example, if after reading the prefix u_1 a token enters the initial locations of the gadget for subexpression E', there will be a token in $\mathsf{out}_{E'}$ after processing the string $u_1 \cdot u_2$ if, and only if, $u_2 \in L(E')$.

Due to the union operator, several tokens may be moving around in several subexpression gadgets, where each of these tokens represents one way of generating a prefix. For example, the gadget for the union expression $E_1 = (a + ab)$ consists of two subexpression gadgets: one for a and one for ab. The distinguished location out_{E_1} is identified with the distinguished locations out_a and out_{ab} of its two subexpression gadgets. After inputting a, one token arrives in out_{E_1}, as a is indeed generated by the subexpression a. However, there is another token inside the subexpression gadget for ab, as the letter a may contribute to generating a string in $L(ab)$. This token waits inside the gadget for the next letter from A to come.

If the next input letter from A is a, then both the token in out_{E_1} and the token residing in the subexpression gadget for ab are moving to bin_{E_1}, as the word aa does not contribute to any string generated by E_1. In bin_E, we collect all tokens representing prefixes of u that do not correspond to prefixes of strings that can be generated by E. Recall that these tokens will be guided to synch when a #-transition is fired.

A core property of our construction is that, even though a string u may be generated in more than a single way by a (sub)expression E', there will be only one distinct token arriving in $\mathsf{out}_{E'}$. For example, for the subexpression $E_2 = (a + ab) \cdot (b + \epsilon)$, the word ab is produced in two different ways, namely $a \cdot b$ and $ab \cdot \epsilon$. Consequently, there will be two tokens in the E_2-gadget, each of them following one way of producing ab, but right before arriving in out_{E_2} we *merge* those tokens and let only one globally fresh distinct token move to out_{E_2}.

In the following we present the inductive definitions for each kind of subexpressions of a simple regular-like expression E. We start with the four basic building blocks: the expression gadgets for a letter a, for the empty word ε, for B^{2^k} and $(B + \varepsilon)^{2^k}$, where $a \in A$ and $B \subseteq A$.

- ***a*-gadget:** Figure 8 depicts the gadget for a. The gadget is defined over the alphabet $A \cup \{\star, \#\}$. The gadget has a single initial location in_a. There are two further distinguished locations out_a and bin_a. The only letter that can be generated by the expression a is a; hence, an a-transition guides a token from in_a to out_a, while all b-transitions, for $b \in A \backslash \{a\}$, guide a token from in_a to bin_a. Further, firing a b-transition, for all $b \in A$, guides a token from out_a to bin_a, respectively keeps a token in bin_a. All #-transitions are directed to synch, except for those in out_a, which are directed to init.

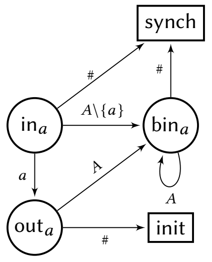


Fig. 8. Gadget for a

- ε**-gadget:** Figure 9 depicts the gadget for ε. The gadget is defined over the alphabet $A \cup \{\star, \#\}$. The gadget has a single initial location in_ε which is identified with the distinguished location out_ε, indicated in the picture by dotted lines surrounding them. There is one further distinguished location bin_ε. From in_ε, all a-transitions, for $a \in A$, are directed to bin_ε, as no letter can be generated by the expression ε. Firing some further a-transition, for $a \in A$, keeps a token in bin_ε. A #-transition guides a token from bin_ε to synch, and it guides a token from in_ε to init.

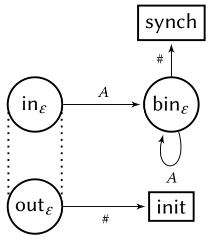

synch
#
in_ε
A
bin_ε
A
out_ε
#
init

Fig. 9. Gadget for ε

- B^{2^k}**-gadget:** Figure 10 depicts the gadget for subexpression $E' = B^{2^k}$ with $B \subseteq A$. The gadget is defined over the alphabet $A \cup \{\star, \#\} \cup \Sigma_{E'}$, where $\Sigma_{E'}$ contains distinguished letters $\mathsf{run}_{E'}, \mathsf{leave}_{E'}$ and $\mathsf{Bit}_0^{E'}, \mathsf{Bit}_1^{E'}, \ldots, \mathsf{Bit}_k^{E'}$. The gadget consists of four distinguished locations $\mathsf{in}_{E'}$, $\mathsf{out}_{E'}$, $\mathsf{bin}_{E'}$, and $\mathsf{run}_{E'}$, as well as a counting gadget (partially depicted in the left of Figure 10) composed of locations $2^0, \ldots, 2^k, 2_c^0, \ldots, 2_c^k$ and $2(k+1)$ dummy locations (the locations without label in Figure 10). The initial locations of the E'-gadget are $\mathsf{in}_{E'}$ as well as the counting gadget's locations $2^0, 2_c^1, \ldots, 2_c^k$, corresponding to the binary representation of 1. Entering the E'-gadget hence places an x-token in each of these locations.

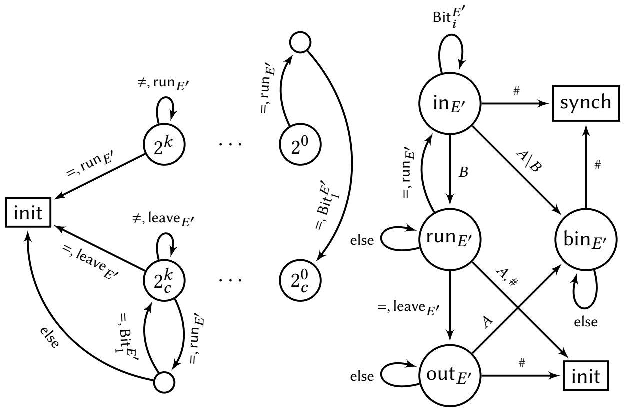


Fig. 10. Gadget $E' = B^{2^k}$ with $B \subseteq A$. To avoid crossing edges, we depict two copies of location init.

Since letters not in B do not contribute to generating any string in $L(E') = B^{2^k}$, the x-token in location $\mathsf{in}_{E'}$ moves to $\mathsf{bin}_{E'}$ whenever the next processed letter is not in B. For all following a-transitions, with $a \in A$, that x-token stays in $\mathsf{bin}_{E'}$. On the other hand, the x-token in location $\mathsf{in}_{E'}$ moves to to $\mathsf{run}_{E'}$ whenever the next processed letter is in B. The gadget is constructed such that upon the j-th visit of the x-token in $\mathsf{run}_{E'}$, for all $1 \leq j \leq 2^k$, the distribution of the x-tokens in the counting gadget represents the binary number j. Note that the only consistent way for the x-token to leave $\mathsf{run}_{E'}$ is by taking some equality-guarded $\mathsf{run}_{E'}$- or $\mathsf{leave}_{E'}$-transitions.

The equality-guarded $\mathsf{leave}_{E'}$-transition, however, is inconsistent as long as there is an x-token in 2_c^k; hence, in this case the only consistent way for the x-token to leave $\mathsf{run}_{E'}$ is by firing an equality-guarded $\mathsf{run}_{E'}$-transition. This moves the x-token in $\mathsf{run}_{E'}$ back to $\mathsf{in}_{E'}$, while the x-tokens in the counting gadget are moved to dummy locations. From the dummy locations, the only consistent transitions are equality-guarded $\mathsf{Bit}_i^{E'}$-transitions that do a correct incrementation of the counter for the x-token. There are $\mathsf{Bit}_i^{E'}$-transitions in $\mathsf{in}_{E'}$ so that the x-token will simply stay in $\mathsf{in}_{E'}$. In the following, we regard a single $\mathsf{run}_{E'}$-transition as doing an implicit increment of the counter. Hence, whenever the x-token is moved back to $\mathsf{in}_{E'}$, its *age* has increased by one. Finally, the inconsistent equality-guarded $\mathsf{run}_{E'}$-transition from 2^k ensures that the x-token will finally leave $\mathsf{run}_{E'}$ via some equality-guarded $\mathsf{leave}_{E'}$-transition to $\mathsf{out}_{E'}$ when its age is equal to 2^k. Summarising, in the counting gadget there are equality-guarded $\mathsf{run}_{E'}$-self-loops in all locations $2^0, \ldots, 2^{k-1}, 2^0_c, \ldots, 2^k_c$, and an equality-guarded $\mathsf{run}_{E'}$ from 2^k to init; there are equality-guarded $\mathsf{leave}_{E'}$-self-loops in all locations $2^0, \ldots, 2^k, 2^0_c, \ldots, 2^{k-1}_c$ and an equality-guarded $\mathsf{leave}_{E'}$ transition from 2^k_c to init. Note that the equality guards for $\mathsf{run}_{E'}$- and $\mathsf{leave}_{E'}$-transitions are necessary to treat situations in which a y-token, for all $y \neq x$, is entering the squaring gadget (with age 1) while the x-token is still in the squaring gadget with some age greater than 1. The (partially depicted) inequality-guarded $\mathsf{run}_{E'}$- and $\mathsf{leave}_{E'}$-self-loops in each of $2^1_c, \ldots, 2^k_c$ ensure that incrementing the age of the y-token does not interfere with incrementing the age of the x-token. Observe that due to the restriction that every token must pass $\mathsf{run}_{E'}$ exactly 2^k times, there will always be at most one token in $\mathsf{out}_{E'}$ between two letters from A, even if some y-token enters the gadget when the x-token is already residing in the gadget. Analogously to the other gadgets, firing another a-transition, for $a \in A$, guides a token from $\mathsf{out}_{E'}$ to $\mathsf{bin}_{E'}$.

Firing #-transitions from $\mathsf{in}_{E'}$ or $\mathsf{bin}_{E'}$ guides a token to synch, while a token in $\mathsf{out}_{E'}$ or $\mathsf{run}_{E'}$ would be guided to init to enforce a new simulation.

- $(B+\varepsilon)^{2^k}$ **-gadget:** Figure 11 depicts the gadget for the case $E' = (B+\varepsilon)^{2^k}$ with $B \subseteq A$. The gadget is very similar to the squaring gadget for B^{2^k}. In particular, the gadget is defined over the alphabet $A \cup \{\star, \#\} \cup \Sigma_{E'}$, where $\Sigma_{E'}$ consists of distinguished letters $\mathsf{run}_{E'}, \mathsf{leave}_{E'}, \mathsf{Bit}^{E'}_0, \mathsf{Bit}^{E'}_1, \ldots, \mathsf{Bit}^{E'}_k$ plus some additional distinguished letter $\mathsf{mrg}_{E'}$. Moreover, the gadget consists of a counting gadget and the idea of using equality-guarded $\mathsf{run}_{E'}$- and $\mathsf{leave}_{E'}$-transitions to force a token to visit the location $\mathsf{run}_{E'}$ for exactly 2^k times is the same as it is for the B^{2^k}-gadget. However, the empty word ε causes some complications that require a careful treatment. First of all note that a token, when entering the gadget, should not be forced to stay until its age becomes *exactly* 2^k but rather *at most* 2^k: the remaining "years" may be compensated by the empty word. The extreme case is the empty word itself, for which a corresponding token should be able to move to $\mathsf{out}_{E'}$ without passing through $\mathsf{run}_{E'}$ at all. For this reason, besides $\mathsf{in}_{E'}$ and locations $2^0, 2^1_c, \ldots, 2^k_c$, the gadget contains a further initial location $\mathsf{bag}_{E'}$: whenever an x-token wants to enter the gadget, (besides $2^0, 2^1_c, \ldots, 2^k_c$) we put a copy of the x-token into $\mathsf{in}_{E'}$ (in order to pass through $\mathsf{run}_{E'}$ for at least once and at most 2^k times) and into $\mathsf{bag}_{E'}$ (in order to not pass through $\mathsf{run}_{E'}$ at all).

  Further, after an equality-guarded $\mathsf{run}_{E'}$-transition was fired in $\mathsf{run}_{E'}$ For the x-token with age i (where i must be between 1 and 2^k), a copy of the x-token moves back to $\mathsf{in}_{E'}$ (with its age increased by 1, thanks to the counting gadget), and another copy moves to $\mathsf{bag}_{E'}$. For the second copy of the x-token in $\mathsf{bag}_{E'}$ with age i we assume that the missing $2^k - i$ visits to $\mathsf{run}_{E'}$ are compensated by the empty word.

  While an x-token with age i is still in the gadget, another distinct y-token may enter the initial locations of the gadget. Observe that these two tokens may arrive in $\mathsf{bag}_{E'}$ simultaneously. As a core property of the construction, we require to guarantee that there is always at most one token in $\mathsf{out}_{E'}$. For guaranteeing this property, we use $\mathsf{mrg}_{E'}$-transitions, explained in the following.

First of all note that firing a-transitions, for $a \in A \cup \{\#\}$, while some token is in $\mathsf{bag}_{E'}$, is inconsistent. In order to avoid the restart of a new simulation, all tokens must leave $\mathsf{bag}_{E'}$ before a next letter from $A \cup \{\#\}$ is processed. The only consistent way to leave $\mathsf{bag}_{E'}$ is via some inequality-guarded $\mathsf{mrg}_{E'}$-transition to $\mathsf{out}_{E'}$, with an update on the register (even firing some equality-guarded $\mathsf{mrg}_{E'}$ is inconsistent). Hence, whenever there are tokens residing in $\mathsf{bag}_{E'}$, they will be merged into a single *locally fresh* token. For the simulation to be faithful, however, we need that the token must be even *globally fresh* with respect to all other tokens residing somewhere in the gadget.

This is accomplished by the distinguished location bag, see Figure 7: whenever an $\mathsf{mrg}_{E'}$-transition is fired in bag: if the input datum is not globally fresh, the inconsistent transition bag $\xrightarrow{=r \ \ \mathsf{mrg}_{E'}}$ init is executed; otherwise, the two following transitions

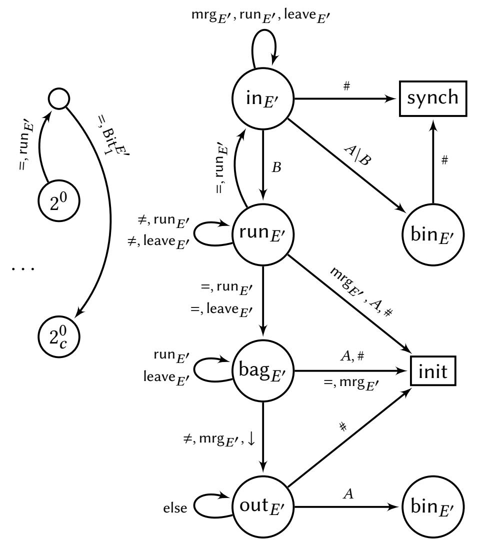


Fig. 11. Gadget $E' = (B + \varepsilon)^{2^k}$ for some $B \subseteq A$. We depict two copies of $\mathsf{bin}_{E'}$ in order to avoid crossing edges.

$$\mathsf{bag} \xrightarrow{\neq r \ \ \mathsf{mrg}_{E'}} \mathsf{bag} \text{ and } \mathsf{bag} \xrightarrow{\neq r \ \ \mathsf{mrg}_{E'} \ \ r\downarrow} \mathsf{bag}$$

produce a globally fresh token, that will be added to bag for future calls of global freshness.

As we mentioned previously, a core property of the construction is that there is always at most one token in $\mathsf{out}_{E'}$. Another crucial feature of the construction, complementary to this property, is the following: for every string generated by a subexpression, even though it may be generated in several ways, it is always a *single* token that leaves the subexpression to be processed further. This feature is vital: a consistent simulation may enfore the usage of equality-guarded transitions. Hence, the time of the simulation (*i.e.*, the length of the shortest synchronizing data words) depends on the number of tokens scattered in the RA. For this feature to work, whenever there is an x-token in $\mathsf{bag}_{E'}$ and some y-token in $\mathsf{run}_{E'}$ at the same time, the y-token must first be guided from $\mathsf{run}_{E'}$ to $\mathsf{bag}_{E'}$ via the equality-guarded $\mathsf{run}_{E'}$- or $\mathsf{leave}_{E'}$-transition. Only *after* that, both the x- and the y-token can be merged into a single fresh token via the $\mathsf{mrg}_{E'}$-transition without evoking an inconsistency. This guarantees that, for each initial token in the initial locations of the gadget, and for all string generated by the subexpression, only a single token leaves the gadget.

We proceed with the inductive definition of the expression gadget, and explain how one can construct from the building blocks described above more complex expression gadgets corresponding to the operations of concatenation and union.

- **concatenation gadget:** Figure 12 depicts the gadget for the expression $E' = E_1 \cdot E_2$. Without loss of generality we assume that $E_1 \neq \varepsilon$. We further assume that the expression gadgets for E_1 and E_2 consist of a single initial location in_{E_1} and in_{E_2}, respectively; the construction for the case that they consist of multiple initial locations is analogous.

The gadget is defined over the alpabet $A \cup \{\star, \#\} \cup \Sigma_{E'}$, where $\Sigma_{E'} = \Sigma_{E_1} \cup \Sigma_{E_2}$. The gadget consists of a distinguished initial location $\mathsf{in}_{E'}$ which is identified with in_{E_1}, a distinguished location $\mathsf{out}_{E'}$, which is identified with the distinguished location out_{E_2} of the E_2-gadget, and a distinguished location $\mathsf{bin}_{E'}$, which is identified with the locations bin_{E_1} and bin_{E_2} from the E_1-gadget, respectively, the E_2-gadget, illustrated in Figure 12 by dotted lines surrounding the corresponding locations. Transitions moving to out_{E_1} of the E_1-gadget are replaced by a corresponding transition to the initial location in_{E_2} of the E_2-gadget. (Note that this means that out_{E_1} is not reachable by any transition, and thus we can also remove out_{E_1} from the gadget.) All other transitions are overtaken from the gadgets for the subexpressions E_1 and E_2 and not depicted here. We further add to every location from the E_1-gadget a self-loop for every letter $\sigma \in \Sigma_{E_2}$; likewise, we add to every location from the E_2-gadget a self-loop for every letter $\sigma \in \Sigma_{E_1}$ (not depicted here).

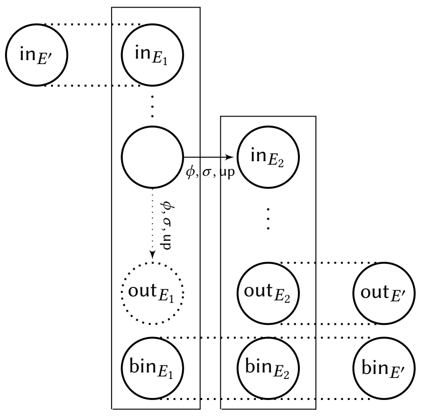


Fig. 12. Gadget for $E' = E_1 \cdot E_2$

- **union gadget:** Figure 13 depicts the gadget for the expression $E' = E_1 + E_2$. We assume that $E_i \neq \varepsilon$ and that both E_1-gadget and E_2-gadget have a single initial location in_{E_1} and in_{E_2}, respectively. For the cases with multiple initial locations, the construction of the E'-gadget is analogous. The gadget is defined over the alphabet $A \cup \{\star, \#\} \cup \Sigma_{E'}$, where $\Sigma_{E'} = \Sigma_{E_1} \cup \Sigma_{E_2} \cup \{\mathsf{mrg}_{E'}\}$. The gadget consists of a distinguished location $\mathsf{in}_{E'}$, which is identified with in_{E_1} and in_{E_2}, as well as a distinguished location $\mathsf{bin}_{E'}$, which is identified with bin_{E_1} and bin_{E_2} from the E_1-gadget respectively the E_2-gadget. There are two further distinguished locations $\mathsf{bag}_{E'}$ and $\mathsf{out}_{E'}$.
When an x-token enters the initial location $\mathsf{in}_{E'}$, it moves along the two subexpression gadgets for E_1 and E_2 in parallel. A token that would reach location out_{E_i}, with $i \in \{1, 2\}$, in the E_i-gadget, is redirected to the location $\mathsf{bag}_{E'}$. When a token arrives in $\mathsf{bag}_{E'}$, the only consistent way to leave $\mathsf{bag}_{E'}$ is by firing an inequality-guarded $\mathsf{mrg}_{E'}$-transition, which puts a *globally fresh* token into $\mathsf{out}_{E'}$. Note that if the two subexpressions E_1 and E_2 contribute to generating a string u so that two distinct tokens arrive in $\mathsf{bag}_{E'}$ at the same time, they are *merged* into a single token and the result is moved into $\mathsf{out}_{E'}$.

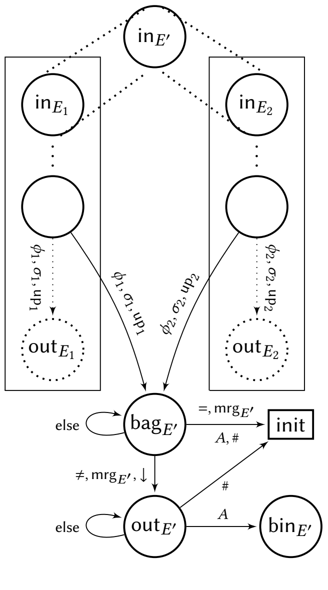


Fig. 13. Gadget $E' = E_1 + E_2$

In order to ensure above, for each initial token in the initial locations of the gadget, and for all strings generated by the subexpression, only a single token leaves the gadget, we proceed as follows: whenever there is some x-token in $\mathsf{bag}_{E'}$ and some y-token in a location $\mathsf{run}_{E''}$ or $\mathsf{bag}_{E''}$ at the same time, where $y \neq x$ and E'' is a subexpression of E_1 or E_2, then the y-token must first leave its current location, and only *after* that the x-token can leave $\mathsf{bag}_{E'}$. For this to achieve, we define for all locations $\ell \in \{\mathsf{run}_{E''}, \mathsf{bag}_{E''} \mid E''$ is a subexpression of E_1 or $E_2\}$ transitions $\ell \xrightarrow{\mathsf{mrg}_{E'}}$ init, and we define an unguarded self-loop with label $\mathsf{mrg}_{E'}$ for all other locations in the E_i-gadget, for $i = 1, 2$. For all letters in Σ_{E_1}, we add self-loops in all locations of the E_1-gadget as well as to $\mathsf{bag}_{E'}$ and $\mathsf{out}_{E'}$, and analogously, for all letters in Σ_{E_2}, we add self-loops in all locations of the E_2-gadget as well as to $\mathsf{bag}_{E'}$ and $\mathsf{out}_{E'}$ (not depicted here).

At the end, to complete the construction of $\mathcal{R}$, we give the formal definition of the alphabet of $\mathcal{R}$. Define $\Sigma_{\mathcal{R}} = A \cup \{\#, \star, \mathsf{next}, \mathsf{Bit}_0, \mathsf{Bit}_1, \ldots, \mathsf{Bit}_n\} \cup \Sigma_E$, where $\Sigma_a = \Sigma_\varepsilon = \emptyset$. We proceed with a detailed example.

Example 5.3. Figure 14 partially depicts the 1-NRA $\mathcal{R}$ for $E = E_1 \cdot E_2 \cdot b$ and $N = 11$, where $E_1 = a + ab + \varepsilon$ and $E_2 = (a + b + \varepsilon)^{2^3}$. The initial locations of the expression gadget for E are in_E and bag_{E_1}.

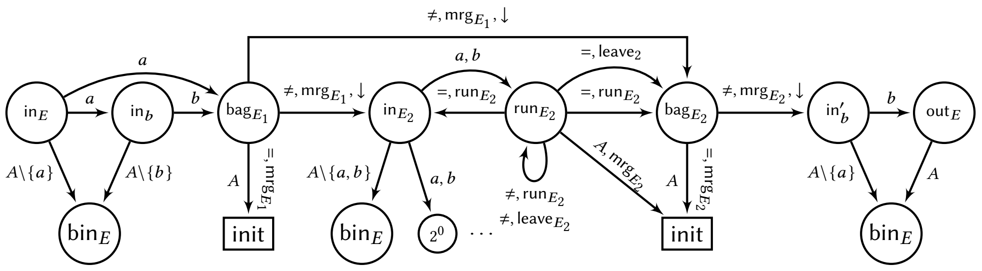


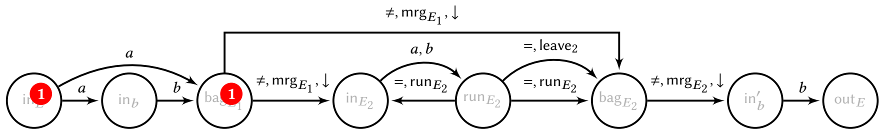
Fig. 14. The expression gadget for the regular-like expression $E = E_1 \cdot E_2 \cdot b$, where $E_1 = (a + ab + \varepsilon)$ and $E_2 = (\{a, b\} + \varepsilon)^{2^3}$. A new simulation puts an initial token into both in_E and bag_{E_1}. We depict locations bin_E and init more than once in order to avoid crossing edges. Table 1 lists the omitted transitions in the automaton.

Table 1. Unconstrained self-loops in locations that are omitted in Figure 14

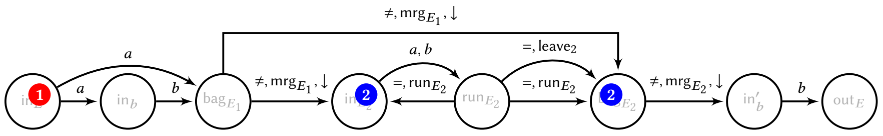
Location	Letters
in_E	$\mathsf{mrg}_{E_1}, \mathsf{mrg}_{E_2}, \mathsf{run}_{E_2}, \mathsf{leave}_{E_2}$
in_b	$\mathsf{mrg}_{E_1}, \mathsf{mrg}_{E_2}, \mathsf{run}_{E_2}, \mathsf{leave}_{E_2}$
bag_{E_1}	$\mathsf{mrg}_{E_2}, \mathsf{run}_{E_2}, \mathsf{leave}_{E_2}$
in_{E_2}	$\mathsf{mrg}_{E_1}, \mathsf{mrg}_{E_2}, \mathsf{run}_{E_2}, \mathsf{leave}_{E_2}$
run_{E_2}	mrg_{E_1}
bag_{E_2}	$\mathsf{mrg}_{E_1}, \mathsf{run}_{E_2}$

In the following, we explain three possible scenarios.

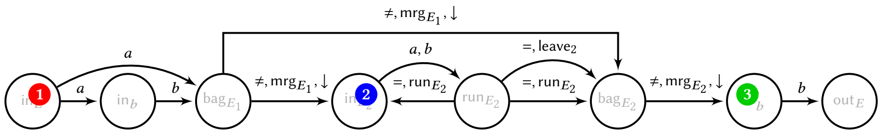
We guess a wrong witness $u \in L(E)$ such as $u = ab$. After a reset, there is a red 1-token in in_E and bag_{E_1}.



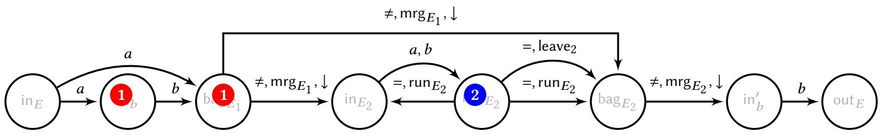
Firing a-transitions while there is a token in bag_{E_1} is inconsistent. The red 1-token in bag_{E_1} must first be removed from bag_{E_1} by an inequality-guarded mrg_{E_1}-transition. Firing a mrg_{E_1}-transition requires inputting a globally fresh datum. We use the blue datum 2, which simultaneously puts a blue 2-token into in_{E_2} and into bag_{E_2}. (Actually, an additional blue 2-token should be placed into each of the locations 2^0, 2^1_c, 2^2_c, and 2^3_c of the squaring gadget of the E_2-gadget. As the mode of operation of the counting gadget is clear from Lemma 4.1, we do not depict these locations here, and neither do we mention the $\mathsf{Bit}_i^{E_2}$-transitions that increase the value of the counter. Note that the "age" of the blue 2-token, *i.e.*, the value of the counter within the squaring counting gadget, is one.) The red 1-token in in_E does not move due to the self-loop on mrg_{E_1}.



Now, firing an a-transition is inconsistent as long as the blue 2-token is residing in bag_{E_2}; it must be removed using the outgoing inequality-guarded mrg_{E_2}-transition. Using a fresh green datum 3, the blue 2-token in bag_{E_2} changes its datum into green 3 and moves to in'_b. Note that the green 3-token reaches in'_b as the empty word is in $L(E_1 \cdot E_2)$.

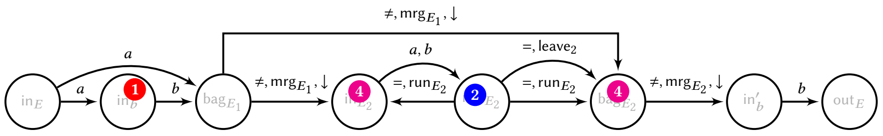


Now we can fire a-transitions without causing an inconsistency. The situation thereafter is depicted as follows.

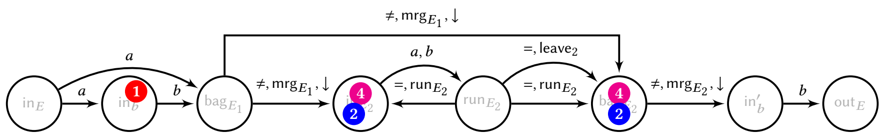


The red 1-token has replicated in order to move along the two gadgets for the subexpressions a and ab. The blue 2-token has moved to run_{E_2}. The green 3-token has moved to bin_E (not depicted here). The intuition behind discarding the green 3-token is as follows: recall that the green 3-token corresponds to the empty word that is generated by $E_1 \cdot E_2$. However, the input letter a does not contribute to a prefix generated by the "next" subexpression b.

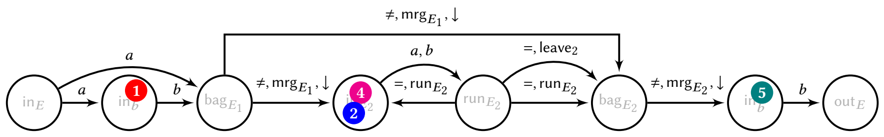
Now, before we can consistently process the next letter b, we have to fire transitions that remove the red 1-token from bag_{E_1}, and that remove the blue 2-token from run_{E_2}. We start by firing a mrg_{E_1}-transition together with a fresh magenta 4-datum, which places one magenta 4-token into in_{E_2}, and one magenta 4-token into bag_{E_2}. The red 1-token in in_b and the blue 2-token in run_{E_2} do not move.



Now we aim to make the blue 2-token leave run_{E_2}. Recall that the blue 2-token has age one and can thus leave run_{E_2} only via some equality-guarded run_{E_2}-transition (leave_{E_2} is only possible if the age of the token is equal to eight). Firing run_{E_2}-transitions replicates the blue 2-token into two tokens: one in bag_{E_2} and one in in_{E_2}. The age of the blue 2-token in in_{E_2} has increased to two.

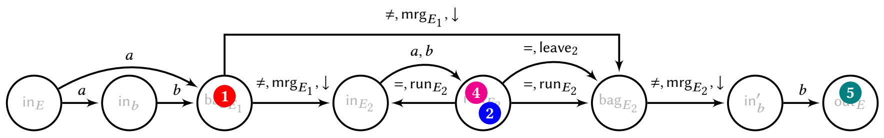


There are hence two distinct tokens in bag_{E_2} at the same time, which is due to the fact that the string a can be generated in two ways by $E_1 \cdot E_2$: the magenta 4-token was generated via $a \cdot \varepsilon$, where $a \in L(E_1)$ and $\varepsilon \in L(E_2)$, and the blue 2-token was generated via $\varepsilon \cdot a$, where $\varepsilon \in L(E_1)$ and $a \in L(E_2)$. Now again, before we can fire b-transitions, these two tokens have to leave bag_{E_2} to avoid an inconsistency. This is only possible via some mrg_{E_2}-transition, together with some fresh datum. Here we choose a turquoise 5-datum.

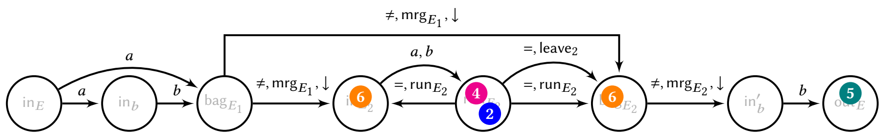


Note how the blue 2-token and the magenta 4-token in bag_{E_2} *merge* into a single turquoise 5-token in in'_b. The incentive to merge multiple tokens residing in bag_{E_2} is indeed to arrive in in'_b (or any other winning location of subexpression gadgets) with at most one token, no matter what is the degree of ambiguity of the subexpression in generating a string (here: a). Observe how the inconsistent mrg_{E_2}-transition from run_{E_2} guarantees that the magenta 4-token in bag_{E_2} has to wait for the blue 2-token to arrive in bag_{E_2} before it can leave bag_{E_2} via some mrg_{E_2}-transition. This ensures that no two consecutive mrg_{E_2}-transitions, resulting in multiple tokens in in'_b, can be done.

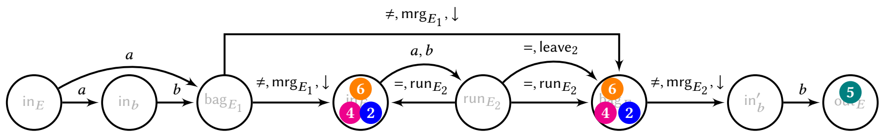
We can finally fire a consistent b-transition. The result is as follows:



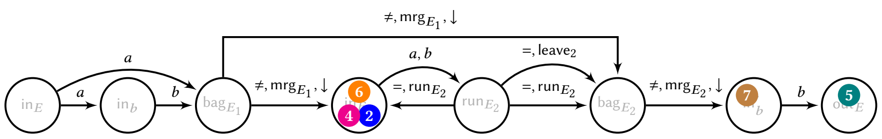
Before the RA $\mathcal{R}$ can check whether the guessed string ab is a witness for bounded non-universality by firing #-transitions, the red 1-token has to be removed from bag_{E_1}, and the blue 2-token and the magenta 4-token have to be removed from run_{E_2}. Inputting $(\mathsf{mrg}_{E_1}, ⑥)$ places one orange 6-token into in_{E_2} and one orange 6-token into bag_{E_2}.



Before we can fire mrg_{E_2} to remove the orange 6-token from bag_{E_2}, the magenta 4-token of age one and the blue 2-token of age two must leave run_{E_2}. These two tokens can only leave run_{E_2} individually via some equality-guarded run_{E_2}-transition. Inputting $(\mathsf{run}_{E_2}, ②)$ places a blue 2-token into bag_{E_2}, and a blue 2-token of age three into in_{E_2}. Inputting $(\mathsf{run}_{E_2}, ④)$ places a magenta 4-token into bag_{E_2}, and a magenta 4-token of age two into in_{E_2}.



There are now three distinct tokens in bag_{E_2}. This is due to three possible ways of generating ab by E: the orange 6-token (with no age) represents $ab \in L(E_1)$ and $\varepsilon \in L(E_2)$, the 1-aged magenta 4-token of age one represents $a \in L(E_1)$ and $b \in L(E_2)$, and the blue 2-token of age two represents $\varepsilon \in L(E_1)$ and $ab \in L(E_2)$. These tokens must leave bag_{E_2} via some inequality-guarded mrg_{E_2}-transition. Inputting $(\mathsf{mrg}_{E_2}, \textcircled{7})$ merges the three tokens in bag_{E_2} into a single brown 7-token in in'_b.

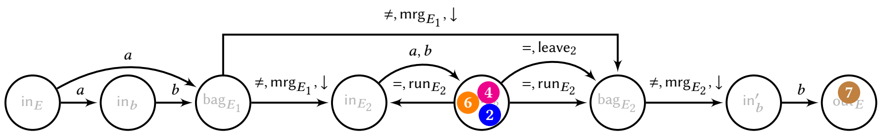


The RA $\mathcal{R}$ can now fire #-transitions without causing some inconsistency. The turquoise token in out_E moves to init, while all other tokens move to synch. The data word

$$\begin{aligned} w_{ab} = {} & (\star, \textcircled{1})(\mathsf{mrg}_{E_1}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{3})(a, \textcircled{0})(\mathsf{mrg}_{E_1}, \textcircled{4})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{5})(b, \textcircled{0}) \\ & (\mathsf{mrg}_{E_1}, \textcircled{6})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{run}_{E_2}, \textcircled{4})(\mathsf{mrg}_{E_2}, \textcircled{7})(\#, \textcircled{0}) \end{aligned}$$

is thus not synchronizing.

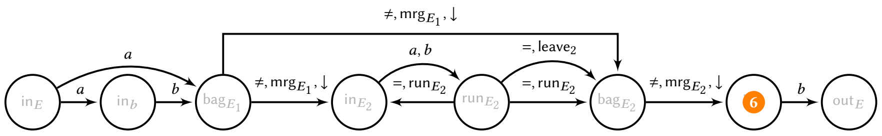
We guess a right witness $u \notin L(E)$, such as ab^9a. We continue with the situation after the data word w_{ab} is processed. First of all recall that the age of the blue 2-token is three, the age of the magenta 4-token is two, and the age of the orange 6-token is one. After firing a b-transition, we yield the following situation:



It is now easy to see that inputting the data word

$$(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{run}_{E_2}, \textcircled{4})(\mathsf{run}_{E_2}, \textcircled{6})(b, \textcircled{0})$$

five times in a row (to simulate the string ab^7) results in the same situation (except for the brown 7-token, which will be moved to bin_E), but the ages of all tokens have increased by five. The blue 2-token is now of age eight and must leave run_{E_2} via some equality-guarded leave_{E_2}-transition. After removing the orange 6-token and the magenta 4-token into in_{E_2} via individual run_{E_2}-transitions, we must further fire an inequality-guarded mrg_{E_2}-transition to move the blue 2-token from bag_{E_2} to in'_b. Then we can fire a consistent b-transition. We proceed similarly for two more rounds, yielding



Firing the last letter $(a, \textcircled{0})$ now places the orange 6-token into bin. From here, if the RA $\mathcal{R}$ checks whether the guessed string is a witness for bounded non-universality of E, all tokens will be moved to synch. The so constructed data word is synchronizing.

We cheat by guessing strings that are longer than N, such as ab^{11}. Note that $ab^{11} \notin L(E)$, and indeed, continuing from the situation depicted in the last figure (with the orange token in in'_b), inputting $(b, ⓪)(b, ⓪)$ places all tokens in the checking gadget into bin. Now, #-transitions would move all tokens from the checking gadget to synch. However, the length gadget has counted 11, and thus the token in the corresponding location in the counting gadget is moved to init. This spoils our attempt to cheat by exceeding the bound $N = 11$.

5.1 Defining the bound on the length of the synchronizing data word

In this subsection, we define the constant N' to bound the length of synchronizing data words for $\mathcal{R}$. By construction, all synchronizing data words w of $\mathcal{R}$ consist of a subword over A, which is the witness for bounded non-universality of the expression E, and some additional letters such as, for example, $\mathsf{mrg}_{E'}$ or $\mathsf{run}_{E'}$ for a squaring subexpression E'. We will in the following show that the number of additional letters is bounded.

Let E' be a regular-like expression with squaring, and let $u \in A^*$ be a string. We use $|E'|$ to denote the size of E', that is the number of concatenation, union and squaring operations occurring in E'. We denote by $\mathsf{maxExtra}_{E',u}$ the maximum number of additional letters, that must be added to a non-universality witness string u to synchronize R without invoking any inconsistent transitions.

Observe that the longest string generated by E' is of length at most $2^{|E'|}$. We show that

$$\mathsf{maxExtra}_{E',u} \leq |u|^{2|E'|+2}.$$

For this, observe that on inputting a data word w, if an E'-gadget starts with a single token in each of its initial locations, it ends with at most a single token in $\mathsf{out}_{E'}$ (*c.f.* Claim A below). Hence,

- $\mathsf{maxExtra}_{E',u} = 0$ if $E' = \varepsilon$ or $E' = a$ for some $a \in A$.
- $\mathsf{maxExtra}_{E',u} \leq 2 \cdot \min(2^k, |u|)$ if $E' = B^{2^k}$. The B^{2^k}-gadget uses the initial token for each non-empty prefix of the non-universality witness up to length 2^k. For every such prefix, there are two extra letters due to one of $\mathsf{run}_{E'}$ and $\mathsf{leave}_{E'}$, and exactly one of the $\mathsf{Bit}_i^{E'}$-letters (used for the counter).
- $\mathsf{maxExtra}_{E',u} \leq 3 \cdot (\min(2^k, |u|) + 1)$ if $E' = (B + \varepsilon)^{2^k}$ The $(B + \varepsilon)^{2^k}$-gadget uses the initial token for each prefix of the non-universality witness up to length 2^k, including the empty word. For every such prefix, there are three extra letters due to one of $\mathsf{run}_{E'}$ and $\mathsf{leave}_{E'}$, $\mathsf{mrg}_{E'}$, and exactly one of the $\mathsf{Bit}_i^{E'}$-letters (used for the counter).
- $\mathsf{maxExtra}_{E',u} \leq (\min(2^{|E'|}, |u|) + 1)(\mathsf{maxExtra}_{E_1,u} + \mathsf{maxExtra}_{E_2,u})$ if $E' = E_1 \cdot E_2$. The longest string generated by the expression E' has length $2^{|E'|}$. There are $(\min(2^{|E'|}, |u|) + 1)$ pairs of prefixes and suffixes that may form a decomposition of u. Here, we take into the account the maximum extra letters for each such pair.
- $\mathsf{maxExtra}_{E',u} \leq (\min(2^{|E'|}, |u|) + 1) + \mathsf{maxExtra}_{E_1,u} + \mathsf{maxExtra}_{E_2,u}$ if $E' = E_1 + E_2$. The expression E' may possibly generate all strings with length less than $\min(2^{|E|}, |u|)$ while reading the non-universality witness u. Here, we must consider an extra letter due to the empty word as well. After each such a string, an extra letter $\mathsf{mrg}_{E'}$ is processed to avoid an inconsistency.

We use $\mathsf{maxExtra}_E$ to denote the maximum number of additional letters required to synchronize $\mathcal{R}$ based on some witness string $u \in A^{\leq N}$ (where N is encoded in binary). Note that the longest string generated by E is of length at most $2^{|E|}$. We can hence assume $N \leq 2^{|E|}$. The bound $\mathsf{maxExtra}_E$ is, loosely speaking, derived from $|E|$ concatenations with consecutive squaring operations. Hence, $\mathsf{maxExtra}_E \leq (2N(N+1))^{|E|} \leq N^{2|E|+2}$ for $N \neq 0$. Now, we are ready to define $N' = 2N + n + 2 + N^{2|E|+2}$, where $2N$ comes from the fact that a synchronizing data word must contain the witness string u (which is bounded by N), together with $|u|$ many Bit_i-letters required in the length-gadget. In the length-gadget, at most $n = log(N)$ occurrences of next are required (to move along from one counting gadget to the next binary counting gadget). We further need an initial $\star$ to guide all tokens to the initial locations of the gadgets, and a final # in order to guide all tokens to synch. Observe that N' is polynomial in N, and thus can be encoded with polynomially many bits.

5.2 Correctness of the Construction

For proving the correctness of the construction, we present two technical claims. We say that a data word w over $\Sigma_E \cup A$ is *consistent* if there is no token in init if we process w starting with a single token in each of the initial locations of the E-gadget. We define the projection of data word w onto A, denoted by $\mathsf{proj}(w)$, to be the mapping which first projects w onto $\Sigma_{\mathcal{R}}$ and then removes from the obtained string all letters not in A. For instance,

$$\mathsf{proj}\left((\star,\textcircled{1})(\mathsf{mrg}_1,\textcircled{2})(\mathsf{mrg}_2,\textcircled{3})(a,\textcircled{4})(\mathsf{mrg}_1,\textcircled{5})(\mathsf{run}_2,\textcircled{2})(\mathsf{mrg}_2,\textcircled{6})(b,\textcircled{4})\right) = ab.$$

Claim A. For every regular expression E, for every consistent data word w over $A \cup \Sigma_E$, if we have started with a single token in the initial locations of the E-gadget and have processed w, then

- there is exactly a single token in out_E if $\mathsf{proj}(w) \in L(E)$, and
- there is no token in out_E if $\mathsf{proj}(w) \notin L(E)$.

Claim B. For every regular expression E, for every string $u \in A^{\leq N}$, there exists a data word w over $A \cup \Sigma_E$ such that $\mathsf{proj}(w) = u$, $|w| \leq \mathsf{maxExtra}_{E,u} + |u|$, and w is consistent.

The proofs of these two claims are technical and can be found in Appendix 9.

Finally, we prove that there exists some string $u \in A^*$ with $u \notin L(E)$ and $|u| \leq N$ if, and only if, there exists some synchronizing data word w_{synch} over $\Sigma_{\mathcal{R}}$ such that $|w_{\mathsf{synch}}| \leq N'$.

First assume that there exists some string $u \in A^*$ with $u \notin L(E)$ and $|u| \leq N$. By Claim B, there exists a data word w over $\Sigma_E \cup A$ such that $\mathsf{proj}(w) = u$, $|w| \leq \mathsf{maxExtra}_{E,u} + |u|$, and w is consistent. Recall that this implies that after processing w starting with a single token in each of the initial locations of the E-gadget, there will be no token in init. Claim A and $u \notin L(E)$ further imply that there will be neither a token in out_E. Hence, after processing w, all tokens are "good-for-bet", that is in locations from which #-transitions go to synch. Now let w' be the result of "meshing up" w with letters from the length gadget (next, $\mathsf{Bit}_0, \ldots$), so that no inconsistency arises from counting. Note that this adds to the length of w at most $|u| + n$ letters. Finally observe that for all $w_{\mathsf{synch}} = (\star, x) \cdot w' \cdot (\#, x)$, with $x \in D$, the claim holds.

Now assume for all $u \in A^*$ that $u \in L(E)$ or $|u| > N$. Assume by contradiction that there exists some synchronizing data word over $\Sigma_{\mathcal{R}}$ with length bounded by N'. Let w_{synch} be a shortest synchronizing data word over $\Sigma_{\mathcal{R}}$ with $|w_{\mathsf{synch}}| \leq N'$, *i.e.*, $w_{\mathsf{synch}} = (\star, x) \cdot w \cdot (\#, y)$ for some $x, y \in D$. Note that, since w_{synch} is synchronizing, before the last letter $(\#, y)$ is processed, all tokens must be in some location different from init (as firing #-transitions while some token is in init is inconsistent). Hence we can assume that w is consistent. By assumption $\mathsf{proj}(w) \in L(E)$. But then, by Claim A, there is a token in out_E before a #-transition is fired. Contradiction.

This finishes the proof of NEXPTIME-hardness for the bounded synchronization problem for NRAs. Note that NEXPTIME-hardness already holds for 1-NRAs.

The *bounded universality* problem asks, given an RA and $N \in \mathbb{N}$ encoded in binary, whether all data words w with $|w| \leq N$ are in the language of the automaton. We state that the bounded non-universality problem in NRAs is

NEXPTIME-complete. The membership in NEXPTIME follows by guessing a witness w letter-by-letter; and checking if the successor configurations after reading w are all non-accepting. A variant of the presented reduction for the bounded synchronizing problem allows to prove that the bounded non-universality problem in NRAs is NEXPTIME-hard: equip $\mathcal{R}$ with the initial location init and set L_f of accepting locations including all locations but synch.

THEOREM 5.4. *The bounded universality problem for NRAs is co-*NEXPTIME*-complete.*

Acknowledgements We thank Sylvain Schmitz for helpful discussions on well-structured systems and non-elementary complexity classes.

REFERENCES

[1] Renzo Angles and Claudio Gutierrez. 2008. Survey of Graph Database Models. *ACM Comput. Surv.* 40, 1, Article 1 (Feb. 2008), 39 pages. DOI: http://dx.doi.org/10.1145/1322432.1322433

[2] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4, Article 31 (Dec. 2012), 46 pages. DOI: http://dx.doi.org/10.1145/2389241.2389250

[3] Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, and E. Shapiro. 2003. DNA molecule provides a computing machine with both data and fuel. *Proc. National Acad. Sci. USA* 100 (2003), 2191–2196.

[4] Miko laj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. 2006. Two-Variable Logic on Words with Data. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 7–16. DOI: http://dx.doi.org/10.1109/LICS.2006.51

[5] Mikolaj Bojanczyk and Pawel Parys. 2011. XPath Evaluation in Linear Time. *J. ACM* 58, 4, Article 17 (July 2011), 33 pages. DOI: http://dx.doi.org/10.1145/1989727.1989731

[6] Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. 2007. Rewriting Systems with Data. In *Fundamentals of Computation Theory, 16th International Symposium, FCT 2007, Budapest, Hungary, August 27-30, 2007, Proceedings (Lecture Notes in Computer Science)*, Erzsébet Csuhaj-Varjú and Zoltán Ésik (Eds.), Vol. 4639. Springer, 1–22. DOI: http://dx.doi.org/10.1007/978-3-540-74240-1_1

[7] Patricia Bouyer, Antoine Petit, and Denis Thérien. 2003. An algebraic approach to data languages and timed languages. *Inf. Comput.* 182, 2 (2003), 137–162. DOI: http://dx.doi.org/10.1016/S0890-5401(03)00038-5

[8] Ján Černý. 1964. Poznámka k homogénnym experimentom s konečnými automatmi. *Matematicko-fyzikálny časopis* 14, 3 (1964), 208–216.

[9] Dmitry Chistikov, Pavel Martyugin, and Mahsa Shirmohammadi. 2016. Synchronizing Automata over Nested Words. In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Vol. 9634. Springer, 252–268.

[10] Lorenzo Clemente and Slawomir Lasota. 2015. Timed Pushdown Automata Revisited. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. IEEE, 738–749. DOI: http://dx.doi.org/10.1109/LICS.2015.73

[11] Stéphane Demri and Ranko Lazic. 2009. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.* 10, 3 (2009). DOI: http://dx.doi.org/10.1145/1507244.1507246

[12] Stéphane Demri, Ranko Lazic, and David Nowak. 2007. On the freeze quantifier in Constraint LTL: Decidability and complexity. *Inf. Comput.* 205, 1 (2007), 2–24. DOI: http://dx.doi.org/10.1016/j.ic.2006.08.003

[13] Stéphane Demri, Ranko Lazic, and Arnaud Sangnier. 2010. Model checking memoryful linear-time logics over one-counter automata. *Theor. Comput. Sci.* 411, 22-24 (2010), 2298–2316. DOI: http://dx.doi.org/10.1016/j.tcs.2010.02.021

[14] Laurent Doyen, Line Juhl, Kim Guldstrand Larsen, Nicolas Markey, and Mahsa Shirmohammadi. 2014. Synchronizing Words for Weighted and Timed Automata. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India (LIPIcs)*, Venkatesh Raman and S. P. Suresh (Eds.), Vol. 29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 121–132. DOI: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2014.121

[15] Laurent Doyen, Thierry Massart, and Mahsa Shirmohammadi. 2011. Infinite Synchronizing Words for Probabilistic Automata. In *Mathematical Foundations of Computer Science 2011 - 36th International Symposium, MFCS 2011, Warsaw, Poland, August 22-26, 2011. Proceedings (Lecture Notes in Computer Science)*, Vol. 6907. Springer, 278–289.

[16] Diego Figueira. 2009. Satisfiability of Downward XPath with Data Equality Tests. In *Proceedings of the Twenty-eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '09)*. ACM, New York, NY, USA, 197–206. DOI: http://dx.doi.org/10.1145/1559795.1559827

[17] Diego Figueira. 2012. Alternating register automata on finite words and trees. *Logical Methods in Computer Science* 8, 1 (2012). DOI: http://dx.doi.org/10.2168/LMCS-8(1:22)2012

[18] Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. 2011. Ackermannian and Primitive-Recursive Bounds with Dickson's Lemma. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada.* IEEE Computer Society, 269–278. DOI: http://dx.doi.org/10.1109/LICS.2011.39

[19] Michael Kaminski and Nissim Francez. 1994. Finite-Memory Automata. *Theor. Comput. Sci.* 134, 2 (1994), 329–363. DOI: http://dx.doi.org/10.1016/0304-3975(94)90242-9

[20] Jan Kretínský, Kim Guldstrand Larsen, Simon Laursen, and Jirí Srba. 2015. Polynomial Time Decidability of Weighted Synchronization under Partial Observability. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015 (LIPIcs)*, Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 142–154.

[21] Kim Guldstrand Larsen, Simon Laursen, and Jirí Srba. 2014. Synchronizing Strategies under Partial Observability. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings (Lecture Notes in Computer Science)*, Vol. 8704. Springer, 188–202.

[22] Alexei Lisitsa and Igor Potapov. 2005. Temporal Logic with Predicate lambda-Abstraction. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA.* IEEE Computer Society, 147–155. DOI: http://dx.doi.org/10.1109/TIME.2005.34

[23] Pavel V. Martyugin. 2010. Complexity of Problems Concerning Carefully Synchronizing Words for PFA and Directing Words for NFA. In *Computer Science - Theory and Applications, 5th International Computer Science Symposium in Russia, CSR 2010, Kazan, Russia, June 16-20, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6072. Springer, 288–302.

[24] Frank Neven, Thomas Schwentick, and Victor Vianu. 2004. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 5, 3 (2004), 403–435. DOI: http://dx.doi.org/10.1145/1013560.1013562

[25] Jean-Eric Pin. 1978. Sur les mots synthronisants dans un automate fini. *Elektronische Informationsverarbeitung und Kybernetik* 14, 6 (1978), 297–303.

[26] Hiroshi Sakamoto and Daisuke Ikeda. 2000. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.* 231, 2 (2000), 297–308. DOI: http://dx.doi.org/10.1016/S0304-3975(99)00105-X

[27] Sylvain Schmitz. 2016. Complexity Hierarchies Beyond Elementary. *ACM Trans. Comput. Theory* 8, 1 (2016), 3:1–3:36.

[28] Mahsa Shirmohammadi. 2014. PhD thesis: Qualitative Analysis of Probabilistic Synchronizing Systems. (2014).

[29] Larry J. Stockmeyer and Albert R. Meyer. 1973. Word Problems Requiring Exponential Time: Preliminary Report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong (Eds.). ACM, 1–9. DOI: http://dx.doi.org/10.1145/800125.804029

[30] Nikos Tzevelekos. 2011. Fresh-register automata. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 295–306. DOI: http://dx.doi.org/10.1145/1926385.1926420

[31] Mikhail V. Volkov. 2008. Synchronizing Automata and the Cerny Conjecture. In *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers (Lecture Notes in Computer Science)*, Carlos Martín-Vide, Friedrich Otto, and Henning Fernau (Eds.), Vol. 5196. Springer, 11–27. DOI: http://dx.doi.org/10.1007/978-3-540-88282-4_4

APPENDIX

6 PROOFS FOR DETERMINISTIC REGISTER AUTOMATA

Lemma 3.2. *For all DRAs for which there exist synchronizing data words, there exists a synchronizing data word w such that $|w| \leq 2|R| + 1$.*

Proof. Let $\mathcal{R} = \langle L, R, \Sigma, T\rangle$ be an RA on the data domain D and with $k \geq 1$ registers. Recall that we denote by $\mathsf{data}(w)$ the data occurring in data words w; for configurations $q = (\ell, \nu)$ we use the same notation $\mathsf{data}(q) = \{\nu(r) \mid r \in R\}$ to denote the data appearing in the valuation of q. Let $\pi : Y_1 \to Y_2$ be a bijection on data where $Y_1, Y_2 \subseteq D$. For every configuration $q = (\ell, \nu)$, define $\pi(q) = (\ell, \nu')$, where ν' satisfies $\nu'(r) = \pi(\nu(r))$ for all $r \in R$. For every data word $w = (a_1, d_1) \ldots (a_n, d_n)$, define $\pi(w) = (a_1, \pi(d_1)) \ldots (a_n, \pi(d_n))$. Note that the application of π on q and w preserves the reachability property, *i.e.*, $\mathsf{post}(\pi(q), \pi(w)) = \{\pi(q') \mid q' \in \mathsf{post}(q, w)\}$.

Assuming that $\mathcal{R}$ has some synchronizing data words, we first prove the following claim by an induction.

Claim. For all pairs of configurations q_1, q_2, if there exists w such that $|\mathsf{post}(\{q_1, q_2\}, w)| = 1$, then

- for all sets $X = \{x_1, x_2, \cdots, x_{2k+1}\} \subset D$ with $\mathsf{data}(q_1), \mathsf{data}(q_2) \subseteq X$,
- there exists some data word $w_{q_1,q_2} \in (\Sigma \times X)^*$ such that $|\mathsf{post}(\{q_1, q_2\}, w_{q_1,q_2})| = 1$.

Note that by $|X| = 2k + 1$, the data efficiency of w_{q_1,q_2} is at most $2k + 1$.

Proof of Claim. Let q_1 and q_2 be two configurations of $\mathcal{R}$ and define $\mathsf{data}(q_1, q_2) = \mathsf{data}(q_1) \cup \mathsf{data}(q_2)$. Since $\mathcal{R}$ has some synchronizing data words, there exists w such that $|\mathsf{post}(\{q_1, q_2\}, w)| = 1$. The proof is by an induction on the length of w.
Base of induction. Assume $w = (a, d)$ have length $|w| = 1$. Let X be any arbitrary set of data such that $|X| = 2k + 1$ and $\mathsf{data}(q_1, q_2) \subseteq X$. There are two cases:

- $d \in X$: This entails that $\mathsf{data}(w) \subseteq X$. Observe that $w_{q_1,q_2} = w$ satisfies the induction statement.
- $d \notin X$: Since $|\mathsf{data}(q_1, q_2)| \leq 2k$, there exists data $x \neq d$ such that $x = X \setminus \mathsf{data}(q_1, q_2)$. Since $x \neq d$, we can define the bijection $\pi : \{d\} \cup \mathsf{data}(q_1, q_2) \to \{x\} \cup \mathsf{data}(q_1, q_2)$ such that $\pi(d) = x$ and $\pi(d') = d'$ for all $d' \in \mathsf{data}(q_1, q_2)$. Observe that $\pi(q_i) = q_i$ for all $i \in \{1, 2\}$. Then

$$|\mathsf{post}(\{q_1, q_2\}, (a, d))| = |\mathsf{post}(\{\pi(q_1), \pi(q_2)\}, (a, \pi(d))| = |\mathsf{post}(\{q_1, q_2\}, (a, x))|.$$

  This and the assumption $|\mathsf{post}(\{q_1, q_2\}, (a, d))| = 1$ yields $|\mathsf{post}(\{q_1, q_2\}, (a, x))| = 1$. The word $w_{q_1,q_2} = (a, x)$ satisfies the induction statement.

Base of induction holds.
Step of induction. Assume that the induction hypothesis holds for $i - 1$. Consider some word $(a, d) \cdot w$ such that $|w| = i - 1$ and $|\mathsf{post}(\{q_1, q_2\}, (a, d) \cdot w)| = 1$.

Consider some set X which has cardinality $2k + 1$ and $\mathsf{data}(q_1, q_2) \subseteq X$, we construct the data word w_{q_1,q_2} as follows. Let $p_1 = \mathsf{post}(q_1, (a, d))$ and $p_2 = \mathsf{post}(q_2, (a, d))$, and let $\mathsf{data}(p_1, p_2) = \mathsf{data}(p_1) \cup \mathsf{data}(p_2)$. Due to the fact that p_1, p_2 are successors of q_1, q_2 after inputting (a, d), we know that if $d \in \mathsf{data}(q_1, q_2)$ then $d \in \mathsf{data}(p_1, p_2)$. There are two cases:

- $d \in \mathsf{data}(q_1, q_2)$ or $d \notin \mathsf{data}(p_1, p_2)$. These guarantee that $\mathsf{data}(p_1, p_2) \subseteq \mathsf{data}(q_1, q_2)$ if $d \in \mathsf{data}(q_1, q_2)$, and that $\mathsf{data}(p_1, p_2) = \mathsf{data}(q_1, q_2)$ if $d \notin \mathsf{data}(p_1, p_2)$. As a result, $\mathsf{data}(p_1, p_2) \subseteq X$. By induction hypothesis, there exists some data word w_{p_1,p_2} over data domain X such that $|\mathrm{post}(\{p_1,p_2\},w_{p_1,p_2})| = 1$. For $w_{q_1,q_2} = (a,d)\cdot w_{p_1,p_2}$ the statement of induction holds, as $|\mathrm{post}(\{q_1,q_2\},w_{q_1,q_2})| = 1$.

- $d \notin \mathrm{data}(q_1,q_2)$ and $d \in \mathrm{data}(p_1,p_2)$. Without loss of generality, we assume that $d \notin X$. Otherwise $d \in X$ would imply $\mathrm{data}(p_1,p_2) \subseteq X$, and we simply let $w_{q_1,q_2} = w_{p_1,p_2}$. Since $|\mathrm{data}(q_1,q_2)| \leq 2k$, there exists data $x \neq d$ such that $x = X \setminus \mathrm{data}(q_1,q_2)$. Since $x \neq d$, we can define the bijection $\pi : \{d\} \cup \mathrm{data}(q_1,q_2) \to \{x\} \cup \mathrm{data}(q_1,q_2)$ such that $\pi(d) = x$ and $\pi(d') = d'$ for all $d' \in \mathrm{data}(q_1,q_2)$. Since $\mathrm{data}(p_1,p_2) \setminus \{d\} \subseteq \mathrm{data}(q_1,q_2)$, having d in the domain of π, the bijection π ranges over $\mathrm{data}(p_1,p_2)$. By induction hypothesis, there exists some data word w_{p_1,p_2} over data domain $(X \setminus \{x\}) \cup (\{d\})$ such that $|\mathrm{post}(\{p_1,p_2\},w_{p_1,p_2})| = 1$. Then, $|\mathrm{post}(\{\pi(p_1),\pi(p_2)\},\pi(w_{p_1,p_2}))| = 1$. For all $1 \leq i \leq 2$, we have $\pi(p_i) \in \mathrm{post}(q_i,(a,x))$ since $p_i \in \mathrm{post}(q_i,(a,d))$ and $x = \pi(d)$. By above arguments, we conclude that $|\mathrm{post}((\{q_1,q_2\},(a,x)\pi(w_{p_1,p_2})| = 1$. As $\{x\} \cup \mathrm{data}(q_1,q_2) \subseteq X$, thus the data word $w_{q_1,q_2} = (a,x)\pi(w_{p_1,p_2})$ satisfies the statement of induction.

The above arguments prove that in all cases, there exists $w_{q_1,q_2} \in (\Sigma \times X)^*$ that merges two configurations q_1 and q_2 into a singleton, which completes the proof of **Claim**.

Since $\mathcal{R}$ has some synchronizing data word, using Lemma 3.1, we know that there exists some word w with data efficiency k such that $\mathrm{post}(L \times D^k, w) \subseteq L \times \mathrm{data}(w)^k$. Consider some set $X = \{x_1, x_2, \cdots, x_{2k+1}\} \subset D$ such that $\mathrm{data}(w) \subseteq X$. We use the pairwise synchronization technique as follows. Define $S_n = L \times X^k$ and $n = |L|(2k+1)^k$, *i.e.*, $|S_n| = n$. For all $i = n-1, \cdots, 1$ repeat the following:

(1) Take a pair of configurations $q_1, q_2 \in S_{i+1}$. By the **Claim** above, one can find some word $w_{q_1,q_2} \in (\Sigma \times X)^*$ such that $|\mathrm{post}(\{q_1,q_2\},w_{q_1,q_2})| = 1$,
(2) Define $v_i = w_{q_1,q_2}$ and $S_i = \mathrm{post}(S_{i+1}, v_i)$.

Note that by determinism of $\mathcal{R}$, for every $i \in \{1,\cdots,n-1,\}$, we have $|S_i| \leq |S_{i+1}| - 1$. Thus the word $w_{\mathsf{synch}} = w \cdot v_{n-1} \cdots v_2 \cdot v_1$ is a synchronizing data word for $\mathcal{R}$. Since $\mathrm{data}(w) \subseteq X$ and $\mathrm{data}(v_i) \subseteq X$ for all $i \in \{1,\cdots,n-1\}$, the data efficiency of w_{synch} is at most $2k+1$. The proof is complete. □

Lemma 3.5. *The synchronizing problem for k-DRAs is* PSPACE*-complete.*

Proof. We prove PSPACE-hardness by a reduction from the non-emptiness problem for k-DRA. Let $\mathcal{R} = (L, R, \Sigma, T)$ be a k-DRA equipped with an initial location ℓ_i and an accepting location ℓ_f, where, without loss of generality, we assume that all outgoing transitions from ℓ_i update all registers, and that ℓ_f has no outgoing edges. We also assume that $\mathcal{R}$ is complete, otherwise, we add some non-accepting location and direct all undefined transitions to it.

The reduction is such that from $\mathcal{R}$ we construct another k-DRA $\mathcal{R}_{\mathsf{syn}}$ such that the language of $\mathcal{R}$ is not empty if, and only if, $\mathcal{R}_{\mathsf{syn}}$ has some synchronizing data word. We define $\mathcal{R}_{\mathsf{syn}} = (L_{\mathsf{syn}}, R, \Sigma_{\mathsf{syn}}, T_{\mathsf{syn}})$ as follows. The set of locations is $L_{\mathsf{syn}} = L \cup \{\mathsf{reset}\}$, where $\mathsf{reset} \notin L$ is a new location; the alphabet is $\Sigma_{\mathsf{syn}} = \Sigma \cup \{\star\}$, where $\star \notin \Sigma$. To define T_{syn}, we add the following transitions to T.

- $\ell_f \xrightarrow{a\ R\downarrow} \ell_f$ for all letters $a \in \Sigma_{\mathsf{syn}}$,
- $\ell_i \xrightarrow{\star\ R\downarrow} \ell_i$
- $\mathsf{reset} \xrightarrow{a\ R\downarrow} \ell_i$ for all letters $a \in \Sigma_{\mathsf{syn}}$,
- $\ell \xrightarrow{\star\ R\downarrow} \mathsf{reset}$ for all $\ell \in L_{\mathsf{syn}}$ except for $\mathsf{reset}, \ell_i, \ell_f$.

Note that $\mathcal{R}_{\mathsf{synch}}$ is indeed deterministic and complete. To establish the correctness of the reduction, we prove that the language of $\mathcal{R}$ is not empty if, and only if, $\mathcal{R}_{\mathsf{syn}}$ has a synchronizing data word.

First, assume that the language of $\mathcal{R}$ is not empty. Then there exists a data word $w = (a_1, d_1) \ldots (a_n, d_n)$ such that $w \in L(\mathcal{R})$. Hence there exists a run starting from (ℓ_i, v_i) and ending in (ℓ_f, v_f) for some $v_i, v_f \in D^{|R|}$. The data word $(\star, d)(\star, d)w(\star, d)$ for some $d \in D$ synchronizes $\mathcal{R}_{\mathsf{syn}}$ in location ℓ_f.

Second, assume that $\mathcal{R}_{\mathsf{syn}}$ has some synchronizing data word. Let $w \in (\Sigma_{\mathsf{syn}} \times D)^*$ be one of the shortest data synchronizing data words. All transitions in ℓ_f are self-loops with update on all registers; Hence, $\mathcal{R}_{\mathsf{syn}}$ can only be synchronized in ℓ_f. Hence, we also have $\mathsf{post}((\ell_i, v_i), w) = \{(\ell_f, v_f)\}$ (for some $v_i, v_f \in D^{|R|}$). By the fact that w is a shortest synchronizing data word, we can infer that the corresponding run does not contain any $\star$-transitions except for two self-loops in ℓ_i in the very beginning. Hence there exists a run from (ℓ_i, v_i) to ℓ_f and thus $L(\mathcal{R}) \neq \emptyset$. □

7 PROOFS FOR NON-DETERMINISTIC REGISTER AUTOMATA

Lemma 4.1. *There is a family of 1-NRAs $(\mathcal{R}_{\mathsf{counter}(n)})_{n\in\mathbb{N}}$ with $O(n)$ locations, such that for all synchronizing data words w, some datum $d \in \mathsf{data}(w)$ appears in w at least 2^n times.*

Proof. The family of 1-NRAs $(\mathcal{R}_{\mathsf{counter}(n)})_{n\in\mathbb{N}}$ is defined as follows. The alphabet of RA $\mathcal{R}_{\mathsf{counter}(n)}$ is $\Sigma = \{\#, \star, \mathsf{Bit}_0, \mathsf{Bit}_1, \cdots, \mathsf{Bit}_n\}$. The structure of $\mathcal{R}_{\mathsf{counter}(n)}$ is composed of three distinguished locations synch, reset, zero and locations $2^n, 2^{n-1}, \cdots, 2^1, 2^0$ and $2^n_c, 2^{n-1}_c, \cdots, 2^1_c, 2^0_c$. The general structure of $\mathcal{R}_{\mathsf{counter}(n)}$ is partially depicted in Figure 4. The RA $\mathcal{R}_{\mathsf{counter}(n)}$ is constructed such that for all synchronizing data words w, some datum $x \in \mathsf{data}(w)$ appears in w at least 2^n times. A counting feature is thus embedded in $\mathcal{R}_{\mathsf{counter}(n)}$: intuitively, the set of all reached configurations represents the counter value. Starting from $\{(\mathsf{zero}, x)\}$, the first increment results in $\{2^n_c, \cdots, 2^2_c, 2^1_c, 2^0\} \times \{x\}$, where location 2^i means that the i-th least significant bit in the binary representation of the counter value is set to 1, and location 2^i_c means that the i-th bit is set to $\mathtt{0}$. Informally, we say that there is an x-token in every reached location. Here, $2^n_c, \cdots, 2^2_c, 2^1_c, 2^0$ have x-tokens. A sequence of counter increments is encoded by re-placing the x-tokens, as shown in the following sequence of sets of locations: $\{2^n_c, \cdots, 2^2_c, 2^1, 2^0_c\}$, $\{2^n_c, \cdots, 2^2_c, 2^1, 2^0\}$, $\{2^n_c, \cdots, 2^3_c, 2^2, 2^1_c, 2^0_c\}$, etc. The transitions of $\mathcal{R}_{\mathsf{counter}(n)}$ are defined in such a way that, starting from $\{(\mathsf{zero}, x)\}$, *either* 2^i *or* 2^i_c have tokens, but never both of them at the same time. We now present a detailed explanation of the structure of $\mathcal{R}_{\mathsf{counter}(n)}$.

All transitions in synch are self-loops with an update on the register synch $\xrightarrow{\Sigma\ r\downarrow}$ synch. Thus, $\mathcal{R}_{\mathsf{counter}(n)}$ can only be synchronized in synch. Moreover, synch is only accessible by #-transitions. Similarly, all transitions except for those with label $\star$, are self-loops in location reset; thus, $\mathcal{R}_{\mathsf{counter}(n)}$ can only be synchronized by leaving reset by reading $\star$. We use this also to avoid transitions which are *incorrect* with respect to the binary incrementing process: all incorrect actions are guided to reset to enforce another $\star$. Assuming w to be one of the shortest synchronizing words, we see that $\mathsf{post}(L \times D, w) = \{(\mathsf{synch}, x)\}$, where w starts with $(\star, x)$ and ends with $(\#, x)$.

The counting involves an *initializing process* and several *incrementing* processes.

- *initializing the counter to* zero: the $\star$-transitions are devised to place a token in zero: from all locations $\ell \in L \setminus \{\mathsf{synch}\}$ we have $\ell \xrightarrow{\star\ r\downarrow}$ zero. This sets the counter to $\mathtt{0}$.
- *incrementing the counter*: we use $\mathsf{Bit}_0, \ldots, \mathsf{Bit}_n$-transitions with equality guards to control the increment. Intuitively, an equality-guarded Bit_i-transition is taken to set the i-th bit in the binary representation of the counter value according to the standard rules of binary incrementation.

  Initially, the token in zero splits in 2^0 and $2^n_c, \cdots 2^1_c$ to represent $\mathtt{0} \cdots \mathtt{01}$, by taking the transitions zero $\xrightarrow{=r\ \mathsf{Bit}_0}$

2^0 and zero $\xrightarrow{=r\ \mathsf{Bit}_0} 2^j_c$ for all $1 \le j \le n$. Equality-guarded Bit_i-transitions for $i \in \{1, \dots, n\}$ are incorrect for zero and thus guided to reset. Whenever data different from x is processed, $\mathcal{R}_{\mathsf{counter}(n)}$ takes self-loops (omitted in Figure 4) and keeps the x-tokens unmoved.

The equality-guarded Bit_i-transitions should only be taken if the i-th bit is not set, or, equivalently, if the location 2^i contains no token. This is guaranteed by a Bit_i-transition $2^i \xrightarrow{=r\ \mathsf{Bit}_i}$ reset, for every $0 \le i \le n$, which results in an incorrect transition and should be avoided. (Otherwise the counting process has to restart from `0`.) In Figure 4, we depict the corresponding transitions for $i = 2$ and $i = n$.

Further, we need to guarantee that for all $i \ge 1$ a Bit_i-transition is taken only if all less significant bits are set, or, equivalently, if all locations $2^{i-1}, \cdots 2^0$ contain a token. This is ensured by a Bit_i-transition $2^j_c \xrightarrow{=r\ \mathsf{Bit}_i}$ reset, for every $0 \le j < i$, which again results in an incorrect transition. See, *e.g.*, the transition $2^2_c \xrightarrow{=r\ \mathsf{Bit}_i}$ reset in Figure 4 for every $3 \le i \le n$.

Finally, Bit_i-transitions must produce tokens in 2^i and $2^0_c, \cdots 2^{i-1}_c$, thus $2^i_c \xrightarrow{=r\ \mathsf{Bit}_i} 2^i$ and $2^j \xrightarrow{=r\ \mathsf{Bit}_i} 2^j_c$ for all $0 \le j < i$. All tokens in locations 2^j and 2^j_c, respectively, for $j > i$ remain where they are, which is implemented by equality-guarded Bit_i-self-loops in 2^j and 2^j_c, respectively.

By construction, it is easy to see that Bit_i-transitions are the only way to produce a token in 2^i, which can be fired if 2^i_c has a token. The Bit_i-transitions then consume the token in 2^i_c. This guarantees that after the first $\star$-transition, which puts a token into zero, the two locations 2^i and 2^i_c will never have a token at the same time.

Finally, all equality-guarded #-transitions in 2^n_c and 2^i for all $0 \le i < n$ are sent to reset. In contrast, all #-transitions in 2^n and 2^i_c for all $0 \le i < n$ are sent to synch, with an update on the register. This guarantees that the counter must correctly count from `0` to `10···0`, meaning that at least one datum x appears at least 2^n times while synchronizing $\mathcal{R}_{\mathsf{counter}(n)}$. □

Lemma 4.3. *The non-universality problem is reducible to the synchronizing problem for NRAs.*

Proof. The reduction is based on the construction presented in Theorem 17 in (14).

Let $\mathcal{R} = \langle L, R, \Sigma, T \rangle$ be an NRA equipped with an initial location ℓ_{in} and a set L_{f} of accepting locations, where, without loss of generality, we assume that all outgoing transitions from ℓ_{in} update all registers. We also assume that $\mathcal{R}$ is complete, otherwise, we add some non-accepting location and direct all undefined transitions to it.

We construct an NRA $\mathcal{R}_{\mathsf{syn}}$ such that there exists some data word that is not in $L(\mathcal{R})$ if, and only if, $\mathcal{R}_{\mathsf{syn}}$ has some synchronizing data word. We define $\mathcal{R}_{\mathsf{syn}} = \langle L_{\mathsf{syn}}, R, \Sigma_{\mathsf{syn}}, T_{\mathsf{syn}} \rangle$ as follows. The set of locations is $L_{\mathsf{syn}} = L \cup \{\mathsf{reset}, \mathsf{synch}\}$ where $\mathsf{synch}, \mathsf{reset} \notin L$ are two new locations. The alphabet is $\Sigma_{\mathsf{synch}} = \Sigma \cup \{\#, \star\}$ where $\#, \star \notin \Sigma$. The transition relation T_{syn} is the union of T and set containing the following transitions:

- $\mathsf{synch} \xrightarrow{a\ R\downarrow} \mathsf{synch}$ for all letters $a \in \Sigma_{\mathsf{syn}}$,
- $\mathsf{reset} \xrightarrow{\star\ R\downarrow} \ell_{\mathsf{in}}$ and $\mathsf{reset} \xrightarrow{a\ R\downarrow} \mathsf{reset}$ for all letters $a \in \Sigma_{\mathsf{syn}} \backslash \{\star\}$,
- $\ell \xrightarrow{\star\ R\downarrow} \ell_{\mathsf{in}}$ for all locations $\ell \in L$,
- $\ell \xrightarrow{\#\ R\downarrow}$ synch for all non-accepting locations $\ell \in L \backslash L_{\mathsf{f}}$,
- $\ell \xrightarrow{\#\ R\downarrow}$ reset for all accepting locations $\ell \in L_{\mathsf{f}}$.

Next, we prove the correctness of the reduction.

First, assume there exists a data word $w = (a_1, d_1) \dots (a_n, d_n)$ such that $w \notin L(\mathcal{R})$. Hence, all runs starting in (ℓ_{in}, v_i) with $v_i \in D^{|R|}$ end in some configuration (ℓ, v) with $\ell \notin L_{\text{f}}$. The data word $(\star, d) \cdot w \cdot (\#, d)$ with $d \in D$ synchronizes $\mathcal{R}_{\text{syn}}$ in location synch, proving that $\mathcal{R}_{\text{syn}}$ has some synchronizing data word.

Second, assume that $\mathcal{R}_{\text{syn}}$ has some synchronizing data word. All transitions in synch are self-loops with update on all registers; thus, $\mathcal{R}_{\text{syn}}$ can only synchronize in synch. Moreover, synch is only accessible with #-transitions; assuming w is one of the shortest synchronizing data words, we see that $\text{post}(L \times D, w) = \{(\text{synch}, v))\}$ for some $v \in D^{|R|}$. From all locations $\ell \in L$ we have $\ell \xrightarrow{\star\ R\downarrow} \ell_{\text{in}}$; we say that $\star$-transitions *reset* $\mathcal{R}_{\text{syn}}$. Moreover, the only outgoing transition in location reset is the $\star$-transition. Thus, a *reset* followed by some # must occur while synchronizing. Let $w = w_0(\star, d_\star)w_1(\#, d_\#)w_2$, where $w_1 \in (\Sigma \times D)^+$ is the data word between the last occurrence of $\star$ and the first following occurrence of #, and $w_2 \in (\Sigma' \backslash \{\star\})^*$. We prove that $w_1 \notin L(\mathcal{R})$. By contradiction, assume that w_1 is in the language; thus, there exist valuations $v_i, v_f \in D^{|R|}$ such that $\mathcal{R}_{\text{syn}}$ has a run over w_1, *i.e.*, starting in (ℓ_{in}, v_i) and ending in (ℓ_f, v_f) where $\ell_f \in L_{\text{f}}$. In fact, since all outgoing transitions in ℓ_{in} update all registers, then for all valuations v_i, $\mathcal{R}_{\text{syn}}$ has an accepting run over w_1.

Note that w_0 cannot be a synchronizing word for $\mathcal{R}_{\text{syn}}$, because this would contradict the assumption that w is one of the shortest synchronizing data word. It implies that there must be some configuration q such that $\text{post}_{\mathcal{R}_{\text{syn}}}(q, w_0)$ contains some configuration (ℓ, v) with $\ell \neq \text{synch}$. From (ℓ, v), inputting the next $(\star, d_\star)$ (that is after w_0 in synchronizing word w), we reach $(\ell_{\text{in}}, \{d_\star\}^{|R|})$. Since for all valuations v_i, starting in (ℓ_{in}, v_i), $\mathcal{R}_{\text{synch}}$ has an accepting run over w_1, it must have an accepting run from $(\ell_{\text{in}}, \{d_\star\}^{|R|})$ to some accepting configuration (ℓ_f, v_f) too. Reading the last # (that is after w_1 in synchronizing word w), reset is reached. Since w_2 does not contain any $\star$, reset is never left, meaning that $\mathcal{R}_{\text{syn}}$ cannot synchronize in synch, a contradiction. The proof is complete.

Note that the reduction preserves the number of registers in the NRAs. □

8 CONSTRUCTION OF THE LENGTH GADGET

Length gadget: this gadget is constructed based on the family of counting RAs described in Lemma 4.1. We refer to members of the counting family with $\mathcal{R}_{\text{counter}(i)}$, where $\mathcal{R}_{\text{counter}(i)}$ counts until 2^i (see Figure 5). The length gadget helps the gambler; it counts the number of letters in A that are already read since the last reset. It is indeed important to keep track of this number before betting on a string, since the gambler only wins if the *bounded* non-universality is proved. If the counter value is at most N, then #-transitions go to synch placing a bet that the guessed string u with $|u| \leq N$ is not in $L(E)$. Otherwise, *i.e.*, if the counter value is larger than N, then the #-transition activates a reset and lets the gambler restart with a new guess.

To construct the length gadget, we assume, without loss of generality, that $2^n \leq N + 1 < 2^{n+1}$, so that $n + 1$ bits are sufficient to encode $N + 1$ in binary. Consider the binary representation of $N + 1$ (the least significant bit first). The length gadget is a chain of (modified) counting RAs, where $\mathcal{R}_{\text{counter}(i)}$ is the j-th member in the chain, if the bit with 2^i-significance in the binary representation of N is the j-th bit set to 1. For example, if $N = 9$ and $N + 1 = 10 = 2^1 + 2^3 = (\mathtt{1010})$, then the length gadget is a chain composed of RAs $\mathcal{R}_{\text{counter}(1)}$ and $\mathcal{R}_{\text{counter}(3)}$. Since $2^n \leq N + 1 < 2^{n+1}$, the chain always ends with the counter RA $\mathcal{R}_{\text{counter}(n)}$. We use a new letter next to move along the chain; see Figure 7 We set $\Sigma_{\text{count}} = \{\text{next}, \text{Bit}_0, \text{Bit}_1, \dots, \text{Bit}_n\}$.

Recall that the counting RAs are defined over input letters $\{\text{Bit}_0, \text{Bit}_1, \cdots, \text{Bit}_n, \#, \star\}$ and have three distinguished locations, zero, reset and synch. We modify the counting gadgets as follows:

- The initial location of the length gadget is the zero location in the first counting RA in the chain; in Figure 7, the gadget starts in zero of $\mathcal{R}_{\mathsf{counter}(1)}$. All zero locations of other counting RAs in the chain are omitted.
- The $\star$-transitions are as in the other locations in $\mathcal{R}$; thus all $\star$-transitions are directed to zero, bag and 1_E (initial locations of the length, freshness- and checking gadgets), with an update on r.
- Since the gadget is aimed to count the number of letters in A, the Bit_i-transitions ($0 \leq i \leq n$) which trigger the *increment process*, before entering to locations $2^0, 2^0_c, 2^1, 2^1_c, \cdots, 2^n, 2^n_c$, reach a new location (say $\mathsf{dum}_{2^0}, \mathsf{dum}_{2^0_c}, \cdots, \mathsf{dum}_{2^n}, \mathsf{dum}_{2^n_c}$ whose names are omitted in Figure 7) where the only outgoing transitions are on letters in A. This small modification guarantees that an increment of the counter is only done by an input of letters in A. Moreover, more technically, the Bit_i-transitions ($0 \leq i \leq n$) triggering the increment process do not have equality guards, as we do not need them here.
- Locations synch and reset of all counting RAs in the chain are merged with the synch and reset in $\mathcal{R}$. All *inconsistent transitions with increment*, as before, are directed to reset.
- Recall that in the counting RA $\mathcal{R}_{\mathsf{counter}(i)}$, the #-transitions from all locations were directed to reset, except for locations $2^i, 2^{i-1}_c, \cdots, 2^0_c$. From $2^i, 2^{i-1}_c, \cdots, 2^0_c$ locations, for when the counter value is $\mathtt{10\cdots00} = 2^i$, the #-transitions were directed to synch to synchronize $\mathcal{R}_{\mathsf{counter}(i)}$.
  Here, we replace the #-transitions with next-transitions such that, in the j-th counter RA $\mathcal{R}_{\mathsf{counter}(i)}$ in the chain, from all locations the next-transition is directed to reset, except for $2^i, 2^{i-1}_c, \cdots, 2^0_c$ locations. From those locations, it acts as the Bit_0-transitions of the (omitted) zero location in the $(j+1)$-th counter RA in the chain. Thus, the next-transitions go from one counter RA to the next counter in the chain, and set the initial configuration for the second counter RA.
- As explained above, the last counter RA in the chain is always $\mathcal{R}_{\mathsf{counter}(n)}$. From all locations in the length gadget, #-transitions are directed to synch, except for locations zero and 2^n in $\mathcal{R}_{\mathsf{counter}(n)}$ where it is directed to reset. The #-transition in zero is inconsistent due to the fact that the empty word is not produced by E; recall that there is no Kleene-star operation in regular-like expressions. The #-transition in 2^n in $\mathcal{R}_{\mathsf{counter}(n)}$ is inconsistent due to the fact that the length of the guessed string u exceeds N; thus the gambler is not allowed to win by witnessing u.

Lemma 4.1. *There is a family of 1-NRAs $(\mathcal{R}_{\mathsf{counter}(n)})_{n\in\mathbb{N}}$ with $O(n)$ locations, such that for all synchronizing data words w, some datum $d \in \mathsf{data}(w)$ appears in w at least 2^n times.*

Proof. The family of 1-NRAs $(\mathcal{R}_{\mathsf{counter}(n)})_{n\in\mathbb{N}}$ is defined as follows. The alphabet of RA $\mathcal{R}_{\mathsf{counter}(n)}$ is $\Sigma = \{\#, \star, \mathsf{Bit}_0, \mathsf{Bit}_1, \cdots, \mathsf{Bit}_n\}$. The structure of $\mathcal{R}_{\mathsf{counter}(n)}$ is composed of three distinguished locations synch, reset, zero and locations $2^n, 2^{n-1}, \cdots, 2^1, 2^0$ and $2^n_c, 2^{n-1}_c, \cdots, 2^1_c, 2^0_c$. The general structure of $\mathcal{R}_{\mathsf{counter}(n)}$ is partially depicted in Figure 4. The RA $\mathcal{R}_{\mathsf{counter}(n)}$ is constructed such that for all synchronizing data words w, some datum $x \in \mathsf{data}(w)$ appears in w at least 2^n times. A counting feature is thus embedded in $\mathcal{R}_{\mathsf{counter}(n)}$: intuitively, the set of all reached configurations represents the counter value. Starting from $\{(\mathsf{zero}, x)\}$, the first increment results in $\{2^n_c, \cdots, 2^2_c, 2^1_c, 2^0\} \times \{x\}$, where location 2^i means that the i-th least significant bit in the binary representation of the counter value is set to 1, and location 2^i_c means that the i-th bit is set to $\mathtt{0}$. Informally, we say that there is an x-token in every reached location. Here, $2^n_c, \cdots, 2^2_c, 2^1_c, 2^0$ have x-tokens. A sequence of counter increments is encoded by re-placing the x-tokens, as shown in the following sequence of sets of locations: $\{2^n_c, \cdots, 2^2_c, 2^1, 2^0_c\}$, $\{2^n_c, \cdots, 2^2_c, 2^1, 2^0\}$, $\{2^n_c, \cdots, 2^3_c, 2^2, 2^1_c, 2^0_c\}$, etc. The transitions of $\mathcal{R}_{\mathsf{counter}(n)}$ are defined in such a way that, starting from $\{(\mathsf{zero}, x)\}$, *either* 2^i *or* 2^i_c have tokens, but never both of them at the same time. We now present a detailed explanation of the structure of $\mathcal{R}_{\mathsf{counter}(n)}$.

All transitions in synch are self-loops with an update on the register synch $\xrightarrow{\Sigma\ r\downarrow}$ synch. Thus, $\mathcal{R}_{\mathsf{counter}(n)}$ can only be synchronized in synch. Moreover, synch is only accessible by #-transitions. Similarly, all transitions except for those with label $\star$, are self-loops in location reset; thus, $\mathcal{R}_{\mathsf{counter}(n)}$ can only be synchronized by leaving reset by reading $\star$. We use this also to avoid transitions which are *incorrect* with respect to the binary incrementing process: all incorrect actions are guided to reset to enforce another $\star$. Assuming w to be one of the shortest synchronizing words, we see that $\mathsf{post}(L \times D, w) = \{(\mathsf{synch}, x)\}$, where w starts with $(\star, x)$ and ends with $(\#, x)$.

The counting involves an *initializing process* and several *incrementing* processes.

- *initializing the counter to* zero: the $\star$-transitions are devised to place a token in zero: from all locations $\ell \in L \setminus \{\mathsf{synch}\}$ we have $\ell \xrightarrow{\star\ r\downarrow}$ zero. This sets the counter to $\mathtt{0}$.
- *incrementing the counter*: we use $\mathsf{Bit}_0, \ldots, \mathsf{Bit}_n$-transitions with equality guards to control the increment. Intuitively, an equality-guarded Bit_i-transition is taken to set the i-th bit in the binary representation of the counter value according to the standard rules of binary incrementation.
  Initially, the token in zero splits in 2^0 and $2^n_c, \cdots 2^1_c$ to represent $\mathtt{0}\cdots\mathtt{01}$, by taking the transitions zero $\xrightarrow{=r\ \mathsf{Bit}_0}$ 2^0 and zero $\xrightarrow{=r\ \mathsf{Bit}_0}$ 2^j_c for all $1 \leq j \leq n$. Equality-guarded Bit_i-transitions for $i \in \{1, \ldots, n\}$ are incorrect for zero and thus guided to reset. Whenever data different from x is processed, $\mathcal{R}_{\mathsf{counter}(n)}$ takes self-loops (omitted in Figure 4) and keeps the x-tokens unmoved.
  The equality-guarded Bit_i-transitions should only be taken if the i-th bit is not set, or, equivalently, if the location 2^i contains no token. This is guaranteed by a Bit_i-transition $2^i \xrightarrow{=r\ \mathsf{Bit}_i}$ reset, for every $0 \leq i \leq n$, which results in an incorrect transition and should be avoided. (Otherwise the counting process has to restart from $\mathtt{0}$.) In Figure 4, we depict the corresponding transitions for $i = 2$ and $i = n$.
  Further, we need to guarantee that for all $i \geq 1$ a Bit_i-transition is taken only if all less significant bits are set, or, equivalently, if all locations $2^{i-1}, \cdots 2^0$ contain a token. This is ensured by a Bit_i-transition $2^j_c \xrightarrow{=r\ \mathsf{Bit}_i}$ reset, for every $0 \leq j < i$, which again results in an incorrect transition. See, *e.g.*, the transition $2^2_c \xrightarrow{=r\ \mathsf{Bit}_i}$ reset in Figure 4 for every $3 \leq i \leq n$.
  Finally, Bit_i-transitions must produce tokens in 2^i and $2^0_c, \cdots 2^{i-1}_c$, thus $2^i_c \xrightarrow{=r\ \mathsf{Bit}_i} 2^i$ and $2^j \xrightarrow{=r\ \mathsf{Bit}_i} 2^j_c$ for all $0 \leq j < i$. All tokens in locations 2^j and 2^j_c, respectively, for $j > i$ remain where they are, which is implemented by equality-guarded Bit_i-self-loops in 2^j and 2^j_c, respectively.

By construction, it is easy to see that Bit_i-transitions are the only way to produce a token in 2^i, which can be fired if 2^i_c has a token. The Bit_i-transitions then consume the token in 2^i_c. This guarantees that after the first $\star$-transition, which puts a token into zero, the two locations 2^i and 2^i_c will never have a token at the same time.

Finally, all equality-guarded #-transitions in 2^n_c and 2^i for all $0 \leq i < n$ are sent to reset. In contrast, all #-transitions in 2^n and 2^i_c for all $0 \leq i < n$ are sent to synch, with an update on the register. This guarantees that the counter must correctly count from $\mathtt{0}$ to $\mathtt{10}\cdots\mathtt{0}$, meaning that at least one datum x appears at least 2^n times while synchronizing $\mathcal{R}_{\mathsf{counter}(n)}$. □

Lemma 4.3. *The non-universality problem is reducible to the synchronizing problem for NRAs.*

Proof. The reduction is based on the construction presented in Theorem 17 in (14).

Let $\mathcal{R} = \langle L, R, \Sigma, T\rangle$ be an NRA equipped with an initial location ℓ_{in} and a set L_{f} of accepting locations, where, without loss of generality, we assume that all outgoing transitions from ℓ_{in} update all registers. We also assume that $\mathcal{R}$ is complete, otherwise, we add some non-accepting location and direct all undefined transitions to it.

We construct an NRA $\mathcal{R}_{\text{syn}}$ such that there exists some data word that is not in $L(\mathcal{R})$ if, and only if, $\mathcal{R}_{\text{syn}}$ has some synchronizing data word. We define $\mathcal{R}_{\text{syn}} = \langle L_{\text{syn}}, R, \Sigma_{\text{syn}}, T_{\text{syn}}\rangle$ as follows. The set of locations is $L_{\text{syn}} = L \cup \{\text{reset}, \text{synch}\}$ where synch, reset $\notin L$ are two new locations. The alphabet is $\Sigma_{\text{synch}} = \Sigma \cup \{\#, \star\}$ where $\#, \star \notin \Sigma$. The transition relation T_{syn} is the union of T and set containing the following transitions:

- synch $\xrightarrow{a\ R\downarrow}$ synch for all letters $a \in \Sigma_{\text{syn}}$,
- reset $\xrightarrow{\star\ R\downarrow} \ell_{\text{in}}$ and reset $\xrightarrow{a\ R\downarrow}$ reset for all letters $a \in \Sigma_{\text{syn}} \setminus \{\star\}$,
- $\ell \xrightarrow{\star\ R\downarrow} \ell_{\text{in}}$ for all locations $\ell \in L$,
- $\ell \xrightarrow{\#\ R\downarrow}$ synch for all non-accepting locations $\ell \in L \setminus L_{\text{f}}$,
- $\ell \xrightarrow{\#\ R\downarrow}$ reset for all accepting locations $\ell \in L_{\text{f}}$.

Next, we prove the correctness of the reduction.

First, assume there exists a data word $w = (a_1, d_1) \ldots (a_n, d_n)$ such that $w \notin L(\mathcal{R})$. Hence, all runs starting in (ℓ_{in}, v_i) with $v_i \in D^{|R|}$ end in some configuration (ℓ, v) with $\ell \notin L_{\text{f}}$. The data word $(\star, d) \cdot w \cdot (\#, d)$ with $d \in D$ synchronizes $\mathcal{R}_{\text{syn}}$ in location synch, proving that $\mathcal{R}_{\text{syn}}$ has some synchronizing data word.

Second, assume that $\mathcal{R}_{\text{syn}}$ has some synchronizing data word. All transitions in synch are self-loops with update on all registers; thus, $\mathcal{R}_{\text{syn}}$ can only synchronize in synch. Moreover, synch is only accessible with #-transitions; assuming w is one of the shortest synchronizing data words, we see that $\text{post}(L \times D, w) = \{(\text{synch}, v))\}$ for some $v \in D^{|R|}$. From all locations $\ell \in L$ we have $\ell \xrightarrow{\star\ R\downarrow} \ell_{\text{in}}$; we say that $\star$-transitions *reset* $\mathcal{R}_{\text{syn}}$. Moreover, the only outgoing transition in location reset is the $\star$-transition. Thus, a *reset* followed by some # must occur while synchronizing. Let $w = w_0(\star, d_\star)w_1(\#, d_\#)w_2$, where $w_1 \in (\Sigma \times D)^+$ is the data word between the last occurrence of $\star$ and the first following occurrence of #, and $w_2 \in (\Sigma' \setminus \{\star\})^*$. We prove that $w_1 \notin L(\mathcal{R})$. By contradiction, assume that w_1 is in the language; thus, there exist valuations $v_i, v_f \in D^{|R|}$ such that $\mathcal{R}_{\text{syn}}$ has a run over w_1, *i.e.*, starting in (ℓ_{in}, v_i) and ending in (ℓ_f, v_f) where $\ell_f \in L_{\text{f}}$. In fact, since all outgoing transitions in ℓ_{in} update all registers, then for all valuations v_i, $\mathcal{R}_{\text{syn}}$ has an accepting run over w_1.

Note that w_0 cannot be a synchronizing word for $\mathcal{R}_{\text{syn}}$, because this would contradict the assumption that w is one of the shortest synchronizing data word. It implies that there must be some configuration q such that $\text{post}_{\mathcal{R}_{\text{syn}}}(q, w_0)$ contains some configuration (ℓ, v) with $\ell \neq$ synch. From (ℓ, v), inputting the next $(\star, d_\star)$ (that is after w_0 in synchronizing word w), we reach $(\ell_{\text{in}}, \{d_\star\}^{|R|})$. Since for all valuations v_i, starting in (ℓ_{in}, v_i), $\mathcal{R}_{\text{synch}}$ has an accepting run over w_1, it must have an accepting run from $(\ell_{\text{in}}, \{d_\star\}^{|R|})$ to some accepting configuration (ℓ_f, v_f) too. Reading the last # (that is after w_1 in synchronizing word w), reset is reached. Since w_2 does not contain any $\star$, reset is never left, meaning that $\mathcal{R}_{\text{syn}}$ cannot synchronize in synch, a contradiction. The proof is complete.

Note that the reduction preserves the number of registers in the NRAs. □

9 PROOFS OF CLAIMS A AND B

For the rest of this section, we fix some regular-like expression with squaring E. Given some subexpression E' of E and some datum $x \in D$, we say that a data word w over $\Sigma_E \cup A$ is *consistent with respect to E' and x*, if there is no $y \in D$ such that $(\text{init}, y) \in \text{post}_E((\text{in}_{E'}, x), w \cdot (a, d))$ for all $(a, d) \in A \times D$. Note that the notion of consistency indeed depends on the considered expression and the initial datum of the token in $\mathsf{in}_{E'}$. For instance, let E and E_1 be as in Example 5.3 on page 24. Then the data word $(\mathsf{mrg}_{E_1}, \textcircled{2})$ is consistent with respect to E_1 (alone) and the red 1-token. But it is not consistent with respect to E and the red 1-token, because after $(\mathsf{mrg}_{E_1}, \textcircled{2})$, before we can fire some a-transition, where $a \in A$, we need to guide the blue 2-token out of bag_{E_2}. It is neither consistent with respect to E_1 and the blue 2-token, because the mrg_{E_1}-transition must be fired with a datum that is locally (and globally) fresh.

9.1 Proof of Claim A

For every subexpression E' of E and for every data word over $\Sigma_E \cup A$ that is consistent with respect to E' and x, for some $x \in D$, if we process the data word w starting with a single x-token in each of the initial locations of the E'-gadget, then

- there is a single token in $\mathsf{out}_{E'}$ if $\mathsf{proj}(w) \in L(E')$,
- there is no token in $\mathsf{out}_{E'}$ if $\mathsf{proj}(w) \notin L(E')$.

The proof of this claim is by induction on the structure of the expression E'.

- Assume $E' = a$ for some $a \in A$. First assume $\mathsf{proj}(w) \in L(E')$. Hence $\mathsf{proj}(w) = a$ and $w = v_1 \cdot (a, d) \cdot v_2$ for some $d \in D$ and v_1, v_2 are (possibly empty) data words over Σ_E. Starting with an x-token in $\mathsf{in}_{E'}$, the x-token stays where it is while processing v_1 (unguarded self-loops at $\mathsf{in}_{E'}$ for all letters in Σ_E), but as soon as the a-transition is fired, the x-token is guided to $\mathsf{out}_{E'}$. It stays there until the complete suffix v_2 is processed (unguarded self-loops at $\mathsf{in}_{E'}$ for all letters in Σ_E). No other token is generated; hence the claim holds. Second assume $\mathsf{proj}(w) \notin L(E')$. Hence $\mathsf{proj}(w) = bu'$ for some $u' \in A^*$, or $\mathsf{proj}(w) = au'$ for some $u' \in A^+$. If $\mathsf{proj}(w) = bu'$ for some $u' \in A^*$, then $w = v_1 \cdot (b, d) \cdot v_2$ for some $d \in D$, some data word v_1 over Σ_E, and some data word v_2 over $\Sigma_E \cup A$. Starting with an x-token in $\mathsf{in}_{E'}$, the x-token stays where it is while processing v_1 (unguarded self-loops at $\mathsf{in}_{E'}$for all letters in Σ_E), but as soon as the a-transition is fired, the x-token is guided to $\mathsf{bin}_{E'}$. It stays there until the complete suffix v_2 is processed (unguarded self-loops at $\mathsf{bin}_{E'}$ for all letters in $\Sigma_E \cup A$). If $\mathsf{proj}(w) = au'$ for some $u' \in A^+$, then $w = v_1 \cdot (a, d) \cdot v_2 \cdot (\sigma, d') \cdot v_3$, for some $d, d' \in D$, data words v_1, v_2 over Σ_E, $\sigma \in A$, and data word v_3 over $\Sigma_E \cup A$. Using the same reasoning as above, we know that after processing the prefix $v_1 \cdot (a, d) \cdot v_2$ the x-token (and only the x-token) is in $\mathsf{out}_{E'}$. Firing the σ-transition, however, guides this token to $\mathsf{bin}_{E'}$. It stays there until the complete suffix v_3 is processed (unguarded self-loops at $\mathsf{bin}_{E'}$ for all letters in $\Sigma_E \cup A$). Hence, in both cases there is no token in $\mathsf{out}_{E'}$. This finishes the proof.
- The proof for the case $E' = \varepsilon$ is very similar to the case $E' = a$ and therefore left to the reader.
- Assume $E' = B^{2^k}$ for some $B \subseteq A$ and $k \geq 1$.
  - Assume that $\mathsf{proj}(w) \in L(E')$. This implies $\mathsf{proj}(w) = a_1 \ldots a_m$, where $m = 2^k$ and $a_i \in B$ for all $1 \leq i \leq m$. The consistency of w with respect to E' and x implies that w contains the following substring (probably interleaved with some other letters, which do not contribute in relocating any tokens in the RA):

    $$(a_1, d_1)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_1^{E'}, x)(a_2, d_2)(\mathsf{run}_{E'}, x) \ldots (\mathsf{Bit}_k^{E'}, x)(a_m, d_m)(\mathsf{leave}_{E'}, x),$$

    where $d_i \in D$ are arbitrary. This data word will guide the x-token to $\mathsf{out}_{E'}$. No other token is generated, hence the claim holds.
  - Otherwise, assume that $\mathsf{proj}(w) \notin L(E')$. If $a_i \notin B$ for some $1 \leq i \leq m$, then consistency of w with respect to E' and x implies that w contains the following substring (probably interleaved with some other letters,

which do not contribute in relocating any tokens in the RA):

$$(a_1, d_1)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_1^{E'}, x)(a_2, d_2) \dots (\mathsf{run}_{E'}, x)(\mathsf{Bit}_j^{E'}, x)(a_i, d_i)(a_{i+1}, d_{i+1}) \dots (a_n, d_n),$$

where $d_i \in D$ are arbitrary and j is the least significant bit in the binary representation of i. The a_i-transition will guide the x-token from $\mathsf{in}_{E'}$ to $\mathsf{bin}_{E'}$, where it will stay until the rest of w is processed.

If $m < 2^k$, then consistency of w with respect to E' and x implies that w contains the following substring (probably interleaved with some other letters, which do not contribute in relocating any tokens in the RA):

$$(a_1, d_1)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_1^{E'}, x)(a_2, d_2)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_0^{E'}, x) \dots (a_m, d_m)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_j^{E'}, x),$$

where $d_i \in D$ are arbitrary and j is the least significant bit in the binary representation of m. This will guide the x-token into $\mathsf{in}_{E'}$.

If $m > 2^k$, then consistency of w with respect to E' and x implies that w contains the following substring (probably interleaved with some other letters, which do not contribute in relocating any tokens in the RA):

$$(a_1, d_1)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_1^{E'}, x)(a_2, d_2) \dots (a_{2^k}, d_{2^k})(\mathsf{leave}_{E'}, x)(a_{2^k+1}, d_{2^k+1}) \dots (a_m, d_m),$$

where $d_i \in D$ are arbitrary. The a_{2^k+1}-transition will guide the x-token from $\mathsf{out}_{E'}$ to $\mathsf{bin}_{E'}$, where it will stay until the rest of w is processed.

In each of these cases, the x-token is not in $\mathsf{out}_{E'}$. There is no other token generated, hence the claim holds.

- Assume $E' = (B + \varepsilon)^{2^k}$ for some $B \subseteq A$ and $k \geq 1$. The proof is very similar to the case $E' = B^{2^k}$, and therefore left to the reader.
- Let $E' = E_1 \cdot E_2$.
  - Let $u = \mathsf{proj}(w)$ and assume $u \in L(E')$. Hence there exist u_1, u_2 such that $u = u_1 \cdot u_2$ where $u_1 \in L(E_1)$ and $u_2 \in L(E_2)$. Let w_1 be the longest prefix of w such that $\mathsf{proj}(w_1) = u_1$ and after the last occurrence of some letter $a \in A$ in w_1, there does not occur any letter from Σ_{E_2} (♠). (If $u_1 = \varepsilon$, then w_1 does not occur any letter from Σ_{E_2}.) Note that if w is consistent with respect to E' and x, it is also consistent with respect to E_1 and x (where we mean the E_1-gadget running independently of the E'-gadget). Furthermore, w_1 is also consistent with respect to E_1 and x. By induction hypothesis on E_1 and w_1, after processing w_1 starting with a single token in each of the initial locations of the E_1-gadget, there will be a single token, called the y-token, in out_{E_1}. By construction, after processing w_1 starting with a single token in each of the initial locations of the E'-gadget, this token (and no other) will be placed into each of the initial locations of the E_2-gadget. (Note that by assumption ♠, the token will indeed be placed into the initial locations of the E_2-gadget, as they it cannot have been removed by any σ-transition for some $\sigma \in \Sigma_{E_2}$.) Let w_2 be such that $w = w_1 \cdot w_2$. Note that every consistent transition in the E'-gadget is also consistent in the E_2-gadget. Hence, by induction hypothesis on E_2 and y, and by $\mathsf{proj}(w_2) \in L(E_2)$, after processing w_2 starting with a single token in each of the initial locations in the E_2-gadget, there will be a single token in out_{E_2}. By construction, after processing w_2 in the E'-gadget starting with a single token in each of the initial locations of the E_2-gadget, there will be a token in $\mathsf{out}_{E'}$. This proves the first half of the claim.

    For the second half we need to prove that there will be no more than one token in $\mathsf{out}_{E'}$. Note that if the y-token reaches the initial locations of the E_2-gadget within the E'-gadget, there may be some other token, in the following called the z-token, be residing in some location of the E_1-gadget or in

some location of the E_2-gadget. For instance, for the data word w_{ab} in Example 5.3, after processing $(\mathsf{mrg}_{E_1},\textcircled{2})(\mathsf{mrg}_{E_2},\textcircled{3})(a,\textcircled{0})(\mathsf{mrg}_{E_1},\textcircled{4})$, there is a magenta 4-token in each of in_{E_2} and bag_{E_2} (the initial locations of the E_2-gadget), but there is also some red 1-token in in_b of the E_1-gadget, and there is some blue 2-token in run_{E_2} of the E_2-gadget. However, the construction is such that whenever two tokens arrive in the same location at the same time, they are either of non-equal "age" for some squaring subexpression E'' so that they will not be guided to $\mathsf{out}_{E''}$ at the same time, or they arrive in location $\mathsf{bag}_{E''}$ for some subexpression E'' of E', from which they can only leave by firing some inequality-guarded $\mathsf{mrg}_{E''}$-transition, which merges the two tokens into a single token. As a consequence, there will be only a single token in $\mathsf{out}_{E'}$ after processing w.
  - The proof for $\mathsf{proj}(w) \notin L(E')$ is very similar to the case for $\mathsf{proj}(w) \in L(E')$ and left to the reader.
- Let $E' = E_1 + E_2$.
  - Assume $\mathsf{proj}(w) \in L(E)$. Let us first consider the case $\mathsf{proj}(w) \in L(E_1)$ and $\mathsf{proj}(w) \in L(E_2)$. Note that each transition that is consistent in the E'-gadget is consistent also in the E_i-gadget for $i = 1, 2$ (we mean here the E_i-gadget running independently of the E'-gadget). We can conclude that w is also consistent with respect to E_i and x, for $i = 1, 2$. Hence, by induction hypothesis, after processing w there will a single token, called the x_i-token, be in out_{E_i}, for $i = 1, 2$. Note that the x_1-token may not necessarily be placed into out_{E_1} at the same time as the x_2-token is placed into out_{E_2}; however, as soon as the x_i-token has been placed into out_{E_i}, for some $i = 1, 2$, there are no further letters from A processed. Formally, if w_{pre} is the shortest prefix of w such that after processing w_{pre} at least one of x_1 or x_2 is placed into out_{E_1} or out_{E_2}, then we have $w_{\mathsf{suf}} \in (\Sigma_E \times D)^*$ for the corresponding suffix of w. By construction, after processing w_{pre} in the E'-gadget, there will be at least one of the x_1- or the x_2-token be placed in $\mathsf{bag}_{E'}$; without loss of generality assume it is the x_1-token. Since w is consistent with respect to E', we know that w_{suf} contains some letter $(\mathsf{mrg}_{E'}, y)$, with $y \neq x_1$ and $y \neq x_2$ (by local and global freshness), so that the $\mathsf{mrg}_{E'}$-transition removes the x_1-token out of $\mathsf{bag}_{E'}$ and puts a fresh y-token into $\mathsf{out}_{E'}$. Since $\mathsf{out}_{E'}$ has self-loops for all letters $\sigma \in \Sigma_{E'}$, and $w_{\mathsf{suf}} \in (\Sigma_E \times D)^*$, this proves the first half of the claim. For the second half, we distinguish two cases. (i) the x_2-token arrives in $\mathsf{bag}_{E'}$ at the same time as the x_1-token: firing the $\mathsf{mrg}_{E'}$-transition removes the x_2-token together with the x_1-token out of $\mathsf{bag}_{E'}$ and puts a fresh y-token into $\mathsf{out}_{E'}$. (ii) the x_2-token does not arrive in $\mathsf{bag}_{E'}$ at the same time as the x_1-token: since w_{suf} does not contain any letter from A, the x_2-token can only reside in a location within the E_2-gadget that can reach $\mathsf{bag}_{E'}$ without firing a-transitions, for all $a \in A$. Hence, the x_2-token must reside in a location $\mathsf{run}_{E''}$ or $\mathsf{bag}_{E''}$, for some subexpression E'' of E_2. By construction, firing $\mathsf{mrg}_{E'}$-transitions while the x_2-token is in any of these locations is inconsistent. Hence, the x_2-token is guided to $\mathsf{bag}_{E'}$ by firing suitable $\mathsf{leave}_{E''}$- or $\mathsf{mrg}_{E''}$-transitions *before* the $\mathsf{mrg}_{E'}$-transition is fired. Note that the x_1-token is waiting for the x_2-token in $\mathsf{bag}_{E'}$, due to unguarded self-loops at $\mathsf{bag}_{E'}$ for $\mathsf{leave}_{E''}$ and $\mathsf{mrg}_{E''}$. Hence, firing $\mathsf{mrg}_{E'}$ (together with the fresh y-datum) merges the x_1- and the x_2-token indeed into a single fresh y-token placed into $\mathsf{out}_{E'}$. The reasoning for the case that $\mathsf{proj}(w) \in L(E_1)$ and $\mathsf{proj}(w) \notin L(E_2)$ (or vice versa) is very similar.
  - Assume $\mathsf{proj}(w) \notin L(E)$. Hence $\mathsf{proj}(w) \notin L(E_1)$ and $\mathsf{proj}(w) \notin L(E_2)$. Note that each transition that is consistent in the E'-gadget is consistent also in the E_i-gadget for $i = 1, 2$ (we mean here the E_i-gadget running independently of the E'-gadget). We can conclude that w is also consistent with respect to E_i and x, for $i = 1, 2$. Hence, by induction hypothesis, after processing w there will be no token in out_{E_i}, for

$i = 1, 2$. It is easy to see that, by construction, there can hence be no token in $\mathsf{out}_{E'}$ after processing w in the E'-gadget.

9.2 Proof of Claim B

We prove a stronger version of Claim B: for every regular expression E', for every $x \in D$, and for every finite string $u \in A^{\leq N}$, there exists a data word w in $((\Sigma_{E'} \times D) \cup (A \times \{d\}))^*$ such that $\mathsf{proj}(w) = u$, $|w| \leq \mathsf{maxExtra}_{E',u} + |u|$, and w is consistent with respect to E' and x, for some $d \in D$.

Let E' be a regular expression, let $x \in D$, and let $u = a_1 a_2 \ldots a_m \in A^{\leq N}$. The proof is by induction on the structure of E'. The proof for the case that $E' = a$ or $E' = \varepsilon$ is easy and left to the reader.

- Assume $E' = B^{2^k}$ for some $B \subseteq A$. We distinguish two cases.
  (1) If $m < 2^k$, then define
  $$w = (a_1, d)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_1^{E'}, x)(a_2, d)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_0^{E'}, x) \ldots (a_m, d)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_j^{E'}, x),$$
  where j is the least significant bit in the binary representation of m.
  (2) If $m \geq 2^k$, then define
  $$\begin{aligned} w =&(a_1, d)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_1^{E'}, x)(a_2, d)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_0^{E'}, x) \ldots \\ &(a_{k-1}, d)(\mathsf{run}_{E'}, x)(\mathsf{Bit}_k^{E'}, x)(a_k, d)(\mathsf{leave}_{E'}, x)(a_{k+1}, d) \ldots (a_m, d). \end{aligned}$$

  That $\mathsf{proj}(w) = u$ and $|w| \leq \mathsf{maxExtra}_{E',u} + |u|$ can be easily seen. We prove that w is consistent with respect to E' and x. In case (1), after processing w, there will be an x-token in $\mathsf{in}_{E'}$ and there will be k other x-tokens in the counting gadget: one in each of the locations corresponding to the binary representation of m. In all these locations, firing an a-transition is consistent with respect to E' and x, for all $a \in A$. In case (2), after processing the prefix of w up to (and including) the letter $(\mathsf{leave}_{E'}, x)$, there will be a single x-token in $\mathsf{out}_{E'}$, and there will be k other x-tokens in the counting gadget: one in each of the locations $2^k, 2_c^{k-1}, \ldots, 2_c^0$, corresponding to the binary represention of the number k. Note that firing a-transitions from each of these locations is consistent with respect to E' and x, for all $a \in A$, and thus for the case that $m = k$ the claim holds. For the case that $m > k$, the token in $\mathsf{out}_{E'}$ will be guided to $\mathsf{bin}_{E'}$ (where it will stay until processing w is finished), whereas the tokens in the counting gadget will not move until processing w is finished. Note that also in $\mathsf{bin}_{E'}$, firing a-transitions, for all $a \in A$, is consistent with respect to E' and x; hence the claim holds.
- Assume $E' = (B + \varepsilon)^{2^k}$ for some $B \subseteq A$. The proof is very similar to the case $E' = B^{2^k}$, except for that w must contain $\mathsf{mrg}_{E'}$-letters at suitable positions.
- Assume $E' = E_1 \cdot E_2$. In order to give the intuition and to avoid unnecessary technical complications, we illustrate the proof for this case by a running example, partially reusing Example 5.3. For illustration, the reader may consider Figures 14 *ff.* Let $E' = E_1 \cdot E_2$, where $E_1 = (a + ab + \varepsilon)$, $E_2 = (\{a, b\} + \varepsilon)^{2^3}$, let $u = ab$, and let the initial token be a red 1-token.

  By induction hypothesis on E_1, u and the red 1-token, there exists a data word w_1 such that $\mathsf{proj}(w_1) = u$, $|w_1| \leq \mathsf{maxExtra}_{E_1,u} + |u|$, and w_1 is consistent with respect to E_1 and x. A possible candidate for our example is, *e.g.*,
  $$w_1 = (\mathsf{mrg}_{E_1}, \textcircled{2})(a, \textcircled{0})(\mathsf{mrg}_{E_1}, \textcircled{4})(b, \textcircled{0})(\mathsf{mrg}_{E_1}, \textcircled{6}).$$

For every $0 \leq i \leq m$, let $u[: i]$ denote the prefix of u up to position i, and let $u[i :]$ denote the (corresponding) suffix of u starting at position $i + 1$. In particular, $u[: 0] = \varepsilon$ and $u[0 :] = u$.

Note that $u[: 0] = \varepsilon \in L(E_1)$. By Claim A, there is thus exactly one token in location out_{E_1} in the E_1-gadget. From w_1, we can conclude that this token must be the blue 2-token, because $(\mathsf{mrg}_{E_1}, \textcircled{2})$ is the last letter before the first letter $(a, \textcircled{0})$ is processed, and mrg_{E_1}-transitions necessarily guide tokens to out_{E_1}. In general, to obtain the token being in out_{E_1}, consider the last letter in $\{\mathsf{mrg}_{E''}, \mathsf{leave}_{E''} \mid E'' \text{ is a subexpression of } E_1\}$ before the first letter $(a, \textcircled{0})$, for some $a \in A$, is processed. If this set is empty, then it is the initial token that will be in out_{E_1}.

By construction of the E'-gadget, the blue 2-token will be guided to in_{E_1}. By induction hypothesis on E_2, $u[0 :]$ and the blue 2-token, there exists a data word w_2^0 satisfying $\mathsf{proj}(w_2^0) = u[0 :]$, $|w_2^0| \leq \mathsf{maxExtra}_{E_2, u[0:]}$, and w_2^0 is consistent with respect to E_2 and the blue 2-token. For instance, we may choose

$$w_2^0 = (\mathsf{mrg}_{E_2}, \textcircled{3})(a, \textcircled{0})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{5})(b, \textcircled{0})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{7}).$$

Similarly, $u[: 1] = a \in L(E_1)$. By Claim A, there is thus exactly one token in location out_{E_1} in the E_1-gadget after processing the corresponding prefix of w_1. From w_1, we can conclude that this token must be the magenta 4-token, because $(\mathsf{mrg}_{E_1}, \textcircled{2})$ is the last letter before the second letter $(b, \textcircled{0})$ is processed. By induction hypothesis on E_2, $u[1 :]$ and the magenta 4-token there exists a data word w_2^1 satisfying $\mathsf{proj}(w_2^1) = u[1 :]$, $|w_2^1| \leq \mathsf{maxExtra}_{E_2, u[1:]}$, and w_2^1 is consistent with respect to E_2 and the magenta 4-token. For instance, we may choose

$$w_2^1 = (\mathsf{mrg}_{E_2}, \textcircled{8})(b, \textcircled{0})(\mathsf{run}_{E_2}, \textcircled{4})(\mathsf{mrg}_{E_2}, \textcircled{9}).$$

Since also $u[: 2] = aa \in L(E_1)$, we can follow the same reasoning to obtain, for the suffix $u[2 :]$ and the orange 6-token, a data word w_2^2 of the form

$$w_2^2 = (\mathsf{mrg}_{E_2}, \textcircled{10}).$$

It remains to explain what happens if $u[: i] \notin L(E_1)$ for some $0 \leq i \leq m$. In this case, by Claim A, there will be no token in out_{E_1} after processing the corresponding prefix of w_1. By construction of the $E_1 \cdot E_2$-gadget, there will be no token guided to in_{E_2}, so that we do not need to consider the corresponding suffix $u[i :]$ at all.

In order to define w, we will now combine the data words w_1, w_2^i, for $0 \leq i \leq 2$, in a suitable way. For this we define a partial binary *mash-up*-operation $\odot$ on data words that, roughly speaking, defines from two data words of, *e.g.*, the form $(a_1, \textcircled{0}) \cdot v_1 \cdot (a_2, \textcircled{0}) \cdot v_2 \cdot (a_3, \textcircled{0})$ and $(a_2, \textcircled{0}) \cdot v_2' \cdot (a_3, \textcircled{0})$, where $a_1, a_2, a_3 \in A$ and $v_1, v_2, v_2' \in (\Sigma_{E'} \times D)^*$, a data word $(a_1, \textcircled{0}) \cdot v_1 \cdot (a_2, \textcircled{0}) \cdot v_2 \cdot v_2' \cdot (a_3, \textcircled{0})$. The mash-up operation is done for corresponding suffixes of the two data words, *i.e.*, from right to left, where letters in $A \times \{d\}$ are used as delimiters. For instance, the result of the mash-up operation applied on w_1 and w_2^0 is

$$\begin{aligned} w_{10} = {} & (\mathsf{mrg}_{E_1}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{3})(a, \textcircled{0})(\mathsf{mrg}_{E_1}, \textcircled{4})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{5})(b, \textcircled{0}) \\ & (\mathsf{mrg}_{E_1}, \textcircled{6})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{7}). \end{aligned}$$

However, there is one additional rule: note that, by construction, between two letters from $A \times \{\textcircled{0}\}$, for every subexpression E'' of E', a letter $(\mathsf{mrg}_{E''}, \bullet)$ may occur at most once. We will thus additionally identify all letters with symbol $\mathsf{mrg}_{E''}$ between two letters from $A \times \{\textcircled{0}\}$, even if they use different data. For instance, the result of this operation applied on w_{10} and w_2^1 is

$$
\begin{aligned}
w_{11} = {} & (\mathsf{mrg}_{E_1}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{3})(a, \textcircled{0})(\mathsf{mrg}_{E_1}, \textcircled{4})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{mrg}_{E_2}, \textcircled{5})(b, \textcircled{0}) \\
& (\mathsf{mrg}_{E_1}, \textcircled{6})(\mathsf{run}_{E_2}, \textcircled{2})(\mathsf{run}_{E_2}, \textcircled{4})(\mathsf{mrg}_{E_2}, \textcircled{7}).
\end{aligned}
$$

Observ that how $(\mathsf{mrg}_{E_2}, \textcircled{8})$ and $(\mathsf{mrg}_{E_2}, \textcircled{9})$, respectively, from w_2^1 are identified with $(\mathsf{mrg}_{E_2}, \textcircled{5})$ and $(\mathsf{mrg}_{E_2}, \textcircled{7})$, respectively. Finally, set $w = w_{11} \odot w_2^2 = w_{11}$. By construction, all three properties of the claim are satisfied.

- Assume $E' = E_1 + E_2$. By induction hypothesis, there exist, for $i = 1, 2$, data words w_i such that $\mathsf{proj}(w_i) = u$, $|w_i| \leq \mathsf{maxExtra}_{E_i,u} + |u|$, and w_i is consistent with respect to E_i and x. Let $p = \min(2^{|E'|}, |u|)$. Define w to be the result of inserting into the data word $w_1 \odot w_2$, for every $1 \leq i \leq p$, a letter $(\mathsf{mrg}_{E'}, y_i)$ directly before the occurrence of (a_i, d), where the y_i's are globally fresh and pairwise distinct data from D. Clearly $|w| \leq \mathsf{maxExtra}_{E',|u|} + |u|$. We prove that w is consistent with respect to E' and x. The main issue here is to guarantee that
  - no equality-guarded $\mathsf{mrg}_{E'}$-transition is fired while some token resides in $\mathsf{bag}_{E'}$. This is guaranteed by choosing globally fresh data y_i.
  - no $\mathsf{mrg}_{E'}$-transition is fired while some token resides in $\mathsf{run}_{E''}$ or $\mathsf{bag}_{E''}$, for some subexpression E'' of E'. This is guaranteed by inserting $(\mathsf{mrg}_{E'}, y_i)$ directly before the letter (a_i, d) and by the consistency of w_i with respect to E_i and x, for $i = 1, 2$: this ensures that all necessary $\mathsf{run}_{E''}$-, $\mathsf{leave}_{E''}$-, and $\mathsf{mrg}_{E''}$-transitions (that guide all tokens out of $\mathsf{run}_{E''}$ or $\mathsf{bag}_{E''}$) are fired *before* a $\mathsf{mrg}_{E'}$-transition is fired.

  Note that firing $\mathsf{mrg}_{E'}$-transitions while tokens are in any other location inside the E_1-gadget or the E_2 is consistent. Hence w is consistent with respect to E' and x.