Natural Computing (2022) 21:39–51
https://doi.org/10.1007/s11047-020-09812-4

Iterative arrays with self-verifying communication cell

Martin Kutrib[1]

Accepted: 3 October 2020 / Published online: 5 November 2020

Abstract
We study the computational capacity of self-verifying iterative arrays (SVIA). A self-verifying device is a nondeterministic device whose nondeterminism is symmetric in the following sense. Each computation path can give one of the answers *yes*, *no*, or *do not know*. For every input word, at least one computation path must give either the answer *yes* or *no*, and the answers given must not be contradictory. It turns out that, for any time-computable time complexity, the family of languages accepted by SVIAs is a characterization of the so-called complementation kernel of nondeterministic iterative array languages, that is, languages accepted by such devices whose complementation is also accepted by such devices. SVIAs can be sped-up by any constant multiplicative factor as long as the result does not fall below realtime. We show that even realtime SVIA are as powerful as lineartime self-verifying cellular automata and vice versa. So they are strictly more powerful than the deterministic devices. Closure properties and various decidability problems are considered.

Keywords Iterative arrays · Self-verification · Computational capacity · Speed-up · Closure properties · Decidability problems

1 Introduction

One of the central questions in complexity and language theory asks for the power of nondeterminism in bounded-resource computations. Traditionally, nondeterministic devices have been viewed as having as many nondeterministic guesses as time steps. The studies of this concept of unlimited nondeterminism led, for example, to the famous open LBA-problem or the unsolved question whether or not P equals NP. In order to gain a better understanding of the nature of nondeterminism, in Fischer and Kintala (1979) and Kintala (1977) it has been viewed as an additional limited resource at the disposal of time or space bounded computations.

The concept of so-called *self-verification* at least dates back to the paper (Duris et al. 1997). It applies to automata for decision problems and makes use of stronger notions of acceptance and rejection of inputs.

A self-verifying device is a nondeterministic device whose nondeterminism is symmetric in the following sense. Each computation path can give one of the answers *yes*, *no*, or *unknown*. For every input word, at least one computation path must give either the answer *yes* or *no*, and the answers given must not be contradictory. So, if a computation path gives the answer *yes* or *no*, in both cases the answer is definitely correct. This justifies the notion *self-verifying* and is in contrast to the general case, where an answer different from *yes* does not allow to conclude whether or not the input belongs to the language. Here we study the computational capacity of self-verifying iterative arrays (SVIA).

Self-verifying finite automata have been introduced and studied in Duris et al. (1997) and Hromkovic and Schnitger (2001, (2003) mainly in connection with randomized Las Vegas computations. Descriptional complexity issues for self-verifying finite automata have been studied in Jirásková and Pighizzini (2011). In particular, the conversion of self-verifying finite automata to deterministic finite automata from a state complexity point of view is solved there. The main results are matching upper and lower bounds growing like $3^{n/3}$ for the costs, in terms of the number of states, of such simulations.

A preliminary version of this work was presented at the *25th International Workshop on Cellular Automata and Discrete Complex Systems*, Guadalajara, Mexico, June 26–28, 2019, and it is published in Kutrib and Worsch (2019).

✉ Martin Kutrib
kutrib@informatik.uni-giessen.de

[1] Institut für Informatik, Universität Giessen, Arndtstr. 2, 35392 Giessen, Germany

Another question that motivates the concept of self-verification is as follows. Given a language such that also its complement belongs to the same family, the description of which of both is more economic (Ilie et al. 2000)? For example, in Jirásková (2005) it is shown that a nondeterministic finite automaton can require 2^n states to accept the complement of a language accepted by an n-state nondeterministic finite automaton. So, a representation of the complement by the n-state automaton together with a bit that says that actually the complement of the language accepted is meant is much more economic.

The computational and descriptional complexity of self-verifying pushdown automata has been studied in Fernau et al. (2017). Self-verifying cellular automata have recently been introduced in Kutrib and Worsch (2020), where it turned out that realtime self-verifying one-way cellular automata (SVOCA) are strictly more powerful than realtime deterministic one-way cellular automata, since they can accept non-semilinear unary languages. Moreover, SVOCAs as well as their two-way variant can strongly be sped-up from lineartime to realtime and they are even capable to simulate any lineartime computation of deterministic two-way cellular automata in realtime. Closure properties of the family of languages accepted by realtime SVOCAs are considered as well, where the closure under concatenation, iteration (Kleene star), and non-erasing homomorphisms are left open. Furthermore, decidability problems are studied.

The paper is organized as follows. In Sect. 2 we present the basic notation and the definitions of self-verifying iterative arrays as well as an introductory example. In general, the symmetric conditions for acceptance/rejection of self-verifying devices imply immediately the effective closures of the language families accepted under complementation. In Sect. 3 this observation is turned in a characterization. Moreover, the strong speed-up by a multiplicative constant is derived for any time-computable time complexity. In Sect. 4 we explore the computational capacity of realtime SVIAs. In particular, its is shown that even realtime SVIAs are as powerful as lineartime self-verifying cellular automata and vice versa. So they are strictly more powerful than deterministic iterative arrays. By mutual simulations of iterative arrays and cellular automata it turns out that, in fact, the sequential input mode of iterative arrays and the parallel input mode of cellular automata as well as realtime and lineartime all lead to the same language family for nondeterministic devices and for self-verifying devices. This is certainly not true for larger time complexities, since cellular automata obey a linear space bound where iterative arrays are semi-infinite.

Several closure properties of the family of languages accepted by realtime SVIAs are inherited from the results on realtime SVOCAs. Moreover, the open property under concatenation can be shown in terms of iterative arrays in Sect. 5. Finally, decidability problems are considered in Sect. 6. In particular, by a reduction of the emptiness problem it is shown that the property of being self-verifying is non-semidecidable.

2 Preliminaries and definitions

We denote the non-negative integers by $\mathbb{N}$. Let Σ denote a finite set of letters. Then we write Σ^* for the *set of all finite words* (strings) consisting of letters from Σ. The *empty word* is denoted by λ, and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. For the *reversal of a word* w we write w^R and $|w|$ denotes its *length*. A subset of Σ^* is called a *language* over Σ. In general, we use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*.

A one-dimensional iterative array is a linear, semi-infinite array of finite state machines (sometimes called cells) that are identical except for the leftmost one. All but the leftmost cells are connected to their both nearest neighbors, respectively (see Fig. 1). For convenience we identify the cells by their coordinates, that is, by non-negative integers. The leftmost cell is distinguished. This so-called communication cell is connected to its right neighbor and, additionally, to the input supply which feeds the input sequentially. We assume that once the whole input is consumed an end-of-input symbol is supplied permanently. At the outset of a computation all cells are in the so-called quiescent state. The cells work synchronously at discrete time steps. Here we assume that the communication cell is a nondeterministic finite automaton while all the other cells are deterministic ones (cf. Buchholz et al. 2003). Although this is a very restricted case, for easier writing we call such devices nondeterministic.

More formally, a *nondeterministic iterative array* (NIA, for short) is a system $M = \langle S, \Sigma, F_+, s_0, \triangleleft, \delta_{nd}, \delta_d \rangle$, where S is the finite, nonempty set of *cell states*, Σ is the finite, nonempty set of *input symbols*, $F_+ \subseteq S$ is the set of *accepting states*, $s_0 \in S$ is the *quiescent state*, $\triangleleft \notin \Sigma$ is the *end-of-input symbol*, $\delta_{nd} : (\Sigma \cup \{\triangleleft\}) \times S \times S \to (2^S \setminus \emptyset)$ is the *nondeterministic local transition function for the communication cell*, $\delta_d : S \times S \times S \to S$ is the *deterministic local transition function for non-communication cells* satisfying $\delta_d(s_0, s_0, s_0) = s_0$.

A configuration of M at time $t \geq 0$ is a pair (w_t, c_t), where $w_t \in \Sigma^*$ is the remaining input sequence and c_t :

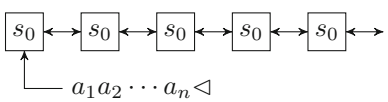

Fig. 1 Initial configuration of an iterative array

$\mathbb{N} \to S$ is a mapping that maps the single cells to their current states. The initial configuration (w_0, c_0) is defined by the given input $w_0 \in \Sigma^*$ and the mapping $c_0(i) = s_0$, $i \geq 0$. Subsequent configurations are computed by the *global transition function* Δ that is induced by δ_d and δ_{nd} as follows: Let (w_t, c_t), $t \geq 0$, be a configuration. Then the set of its possible successor configurations (w_{t+1}, c_{t+1}) is defined as follows:

$$(w_{t+1}, c_{t+1}) \in \Delta((w_t, c_t)) \iff \begin{cases} c_{t+1}(0) \in \delta_{nd}(a, c_t(0), c_t(1)) \\ c_{t+1}(i) = \delta_d(c_t(i-1), c_t(i), c_t(i+1)) \end{cases}$$

for all $i \geq 1$, where $a = \lhd$, $w_{t+1} = \lambda$ if $w_t = \lambda$, and $a = a_1$, $w_{t+1} = a_2 a_3 \cdots a_n$ if $w_t = a_1 a_2 \cdots a_n$.

An input w is accepted by an NIA M if at some time step during the course of at least one computation for w the communication cell enters an accepting state. The *language accepted by* M is denoted by $L(M) = \{ w \in \Sigma^* \mid w \text{ is accepted by } M \}$. Let $t : \mathbb{N} \to \mathbb{N}$, $t(n) \geq n + 1$ be a mapping. If for each $w \in L(M)$ there is an accepting computation with at most $t(|w|)$ time steps, then M and $L(M)$ are said to be of time complexity t.

In general, the family of all languages which are accepted by some type of device X with time complexity t is denoted by $\mathscr{L}_t(X)$. If t is the function $n + 1$, acceptance is said to be in *realtime*. Since for nontrivial computations an iterative array has to read at least one end-of-input symbol, realtime has to be defined as $(n + 1)$-time. We write $\mathscr{L}_{rt}(X)$ for realtime and $\mathscr{L}_{lt}(X)$ for lineartime.

Now we turn to *self-verifying* iterative arrays (SVIA). Basically, an SVIA is an NIA, but the definition of acceptance is different. There are now three disjoint sets of states representing answers *yes*, *no*, and *neutral*. Moreover, for every input word, at least one computation path must give either the answer *yes* or *no*, and the answers given must not be contradictory. In order to implement the three possible answers the state set is partitioned into three disjoint subsets $S = F_+ \dot{\cup} F_- \dot{\cup} F_0$, where F_+ is the set of accepting states, F_- is the set of rejecting states, and $F_0 = S \backslash (F_+ \cup F_-)$ is referred to as the set of neutral states. We specify F_+ and F_- in place of the set F_+ in the definition of SVIAs. Let $M = \langle S, \Sigma, F_+, F_-, s_0, \lhd, \delta_{nd}, \delta_d \rangle$ be an SVIA. For each input word $w \in \Sigma^*$, let S_w denote the set of states entered by the communication cell during any computation on w, that is, $S_w = \{ s \in S \mid s \in (\Delta^{[t]}(w, c_0))(0), \text{ for some } t \geq 0 \}$, where $\Delta^{[t]}$ denotes the t-fold composition of Δ, in other words, the set of configurations reachable in t time steps. For the "self-verifying property" it is required that for each $w \in \Sigma^*$, $S_w \cap F_+$ is empty if and only if $S_w \cap F_-$ is nonempty. So, similarly as for acceptance, an input w is *rejected* by an SVIA if at some time step during the course of at least one computation on w the communication cell enters a rejecting state. Note that by the self-verifying property every input is either accepted or rejected. If all $w \in L(M)$ are accepted and all $w \notin L(M)$ are rejected after at most $t(|w|)$ time steps, then the self-verifying iterative array M is said to be of time complexity t.

In the sequel we will often utilize the possibility of iterative arrays to simulate the data structures pushdown stores (stacks) (Buchholz and Kutrib 1997; Čulik and Yu 1984), queues, and rings (Kutrib 2008) without any loss of time. Here a ring is a queue that can write and erase at the same time. For pushdown stores the communication cell simulates the top of the store, for queues it simulates the front, and for rings the front and the end of the store.

For the sake of completeness, we recall exemplarily the principle of a pushdown store simulation from Kutrib (2008). It suffices to use three additional racks for the simulation. Let the three pushdown registers of each cell be numbered one, two, and three from top to bottom, and suppose that the third register is connected to the first register of the right neighbor. The content of the pushdown store is identified by scanning the registers in their natural ordering beginning in the communication cell, whereby empty registers are ignored. The pushdown store dynamics of the transition function is defined such that each cell prefers to have only the first two registers filled. The third register is used as a buffer. In order to reach that charge it obeys the following rules (cf. Fig. 2).

1. If all three registers of its left (upper) neighbor are filled, it takes over the symbol from the third register of the neighbor and stores it in its first register. The old contents of the first and second registers are shifted to the second and third register.
2. If the second register of its left neighbor is free, it erases its own first register. Observe that the erased symbol is taken over by the left neighbor. In addition, the cell stores the content of its second register into its first one, if the second one is filled. Otherwise, it takes the symbol of the first register of its right neighbor, if this register is filled.
3. Possibly more than one of these actions are superimposed.

We illustrate the definitions with an example.

Example 1 The nondeterministic context-free language $\{ w \in \{a, b\}^* \mid w = w^R \}$ is accepted by the SVIA $M = \langle S, \{a, b\}, F_+, F_-, s_0, \lhd, \delta_{nd}, \delta_d \rangle$. The basic idea is to simulate a stack whose top is the communication cell.

So, we set $S = (\{s_0, s_1, s_2, s_3, s_+, s_-\} \times S_{pd}) \cup \{s_0\} \cup \hat{S}_{pd}$ with $F_+ = \{s_+\} \times S_{pd}$ and $F_- = \{s_-\} \times S_{pd}$. Here S_{pd} are the register contents used by the communication cell to manage the top entries of the stack, while $\hat{S}_{pd}$ are the

non-quiescent states of all but the communication cell, that realize the stack. So, the transition function δ_d just realizes the interior of the stack and is omitted here.

The idea of the construction of δ_{nd} is summarized in the following table and described below. See Fig. 3 for a transition diagram of the communication cell. Since we did not make S_{pd} explicit in detail and since the state of the right neighbor of the communicating cell is only needed for updating its part of the stack, the right neighbor state is left out in the table and the figure. The current state of the communication cell is indicated as $(s_i, y\ldots)$ or $(s_i, \bot)$ meaning that at the top of the stack is a symbol $y \in \{a, b\}$ or the stack is empty. A transition to $xy\ldots$ means that x has been pushed onto, and a transition to $\cdots$ means that y has been popped from the stack.

The communication cell enters the accepting state if it is in the quiescent state while reading the end-of-input symbol, that is, if the input is the empty word (Transition 1). If the input is non-empty, the communication cell enters state s_1 (in its first state register) which is used to read and push the symbols of the first part of the input (Transitions 2–5). Then by Transitions 6–9 it is guessed whether the length of the input is odd or even and whether the center of the input is reached. If the guess is odd then the sole symbol without matching mate at the center is simply read without pushing it (Transitions 6–7). If the guess is even then the last symbol of the left part has to be matched and is pushed (Transitions 8–9). Now, state s_2 is entered.

Once in state s_2, the SVIA M compares the pushdown contents with the remaining input by Transitions 10 and 11. If a mate is not matching, state s_3 is entered (Transitions 12 and 13). Once in state s_3, the remaining input is further read, whereby one symbol is popped in each step (Transitions 14 and 15).

Consider the point in time after M is in state s_2 for the first time and let v denote the part of the input that has been pushed to the stack at that time and let u denote the part of the input that still has not been read. Then the input is vxu if the length of the input has been guessed to be odd, and the input is vu if the length of the input has been guessed to be even. If the communication cell switched from s_1 to s_2 "at the right time and in the right way", that is $|u| = |v|$, then it will for the first time see input $\lhd$ and the empty stack $\bot$ simultaneously. In this situation, M knows that the guess was correct and it can accept (Transition 16) or reject (Transition 17).

If the communication cell switched from s_1 to s_2 "at the wrong time or in a wrong way", that is $|u| \neq |v|$, then it

(1)	$\delta_{nd}(\lhd, (s_0, \bot), \_)$	$\ni$	$(s_+, \bot)$	accept λ
(2)	$\delta_{nd}(a, (s_0, \bot), \_)$	$\ni$	(s_1, a)	push first symbol a
(3)	$\delta_{nd}(b, (s_0, \bot), \_)$	$\ni$	(s_1, b)	push first symbol b
(4)	$\delta_{nd}(a, (s_1, y\cdots), \_)$	$\ni$	$(s_1, ay\cdots)$	in s_1 continue pushing input symbols
(5)	$\delta_{nd}(b, (s_1, y\cdots), \_)$	$\ni$	$(s_1, by\cdots)$	in s_1 continue pushing input symbols
(6)	$\delta_{nd}(a, (s_1, y\cdots), \_)$	$\ni$	$(s_2, y\cdots)$	switch to s_2, dropping symbol a
(7)	$\delta_{nd}(b, (s_1, y\cdots), \_)$	$\ni$	$(s_2, y\cdots)$	switch to s_2, dropping symbol b
(8)	$\delta_{nd}(a, (s_1, y\cdots), \_)$	$\ni$	$(s_2, ay\cdots)$	switch to s_2, without dropping a
(9)	$\delta_{nd}(b, (s_1, y\cdots), \_)$	$\ni$	$(s_2, by\cdots)$	switch to s_2, without dropping b
(10)	$\delta_{nd}(a, (s_2, a\cdots), \_)$	$\ni$	$(s_2, \cdots)$	continue in s_2 for matching symbol a
(11)	$\delta_{nd}(b, (s_2, b\cdots), \_)$	$\ni$	$(s_2, \cdots)$	continue in s_2 for matching symbol b
(12)	$\delta_{nd}(a, (s_2, b\cdots), \_)$	$\ni$	$(s_3, \cdots)$	switch to s_3 if mismatch
(13)	$\delta_{nd}(b, (s_2, a\cdots), \_)$	$\ni$	$(s_3, \cdots)$	switch to s_3 if mismatch
(14)	$\delta_{nd}(a, (s_3, y\cdots), \_)$	$\ni$	$(s_3, \cdots)$	drop remaining input symbols
(15)	$\delta_{nd}(b, (s_3, y\cdots), \_)$	$\ni$	$(s_3, \cdots)$	drop remaining input symbols
(16)	$\delta_{nd}(\lhd, (s_2, \bot), \_)$	$\ni$	$(s_+, \bot)$	accept if everything matched
(17)	$\delta_{nd}(\lhd, (s_3, \bot), \_)$	$\ni$	$(s_-, \bot)$	reject since there was a mismatch

Fig. 2 Principle of a pushdown store simulation. Subfigures are in row-major order

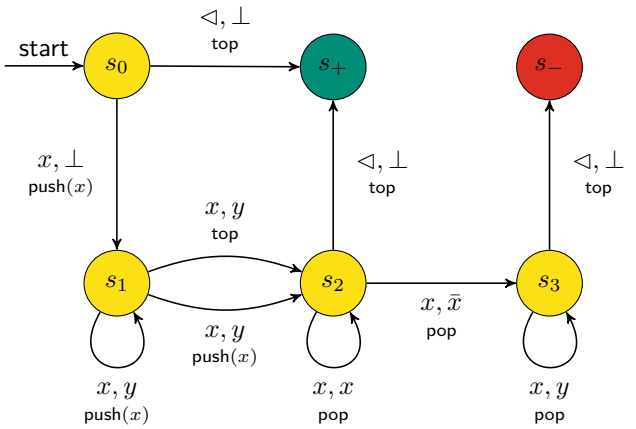

Fig. 3 Transition diagram of the communication cell of an SVIA accepting the language $\{ w \in \{a,b\}^* \mid w = w^R \}$. Edges corresponding to transitions $\delta_{nd}(x,(s,y),\_)$ are labeled x, y in the first line, and by the corresponding operation on the pushdown store in the second line. Here, $x \in \{a,b\}$ and $x, \bar{x}$ means a, b or b, a

cannot decide whether the input is a palindrome. This can be recognized by M since either all input symbols have been consumed but the stack is not empty or the stack is empty but not all input symbols have been consumed. In this case the computation blocks in one of the states from $\{s_1, s_2, s_3\}$ which are neutral states.

So, the communication cell comes to a decision *yes* or *no* if and only if it has guessed the center of the input correctly. Moreover, the answer is correct.

3 Structural properties and speed-up

Though we are mainly interested in fast computations, that is, realtime and lineartime computations, we allowed general time complexities in the definition of the devices (see Kutrib (2009) for a discussion of this general treatment of time complexity functions). However, it seems to be reasonable to consider only time complexities t that allow the communication cell to recognize the time step $t(n)$. Such functions are said to be *time-computable*. For example, the function $t(n) = n + 1$ is trivially a time-computable time complexity for IAs.

Other examples are time complexities $\lfloor \frac{y}{x} \cdot n \rfloor$, for any positive integers $x < y$, polynomials $t(n) = n^k$, and exponential time complexities $t(n) = k^n$, for any integer $k \geq 2$. More details can be found in Mazoyer and Terrier (1999).

In general, the symmetric conditions for acceptance/rejection of self-verifying devices imply immediately the effective closures of the language families accepted under complementation. In order to turn this observation in a characterization, we first give evidence that self-verifying iterative arrays are in fact a generalization of deterministic

iterative arrays. The proof of a corresponding result for cellular automata (Kutrib and Worsch 2020) applies here almost literally.

Lemma 2 *Any deterministic iterative array with a time-computable time complexity t can effectively be converted into an equivalent self-verifying iterative array with the same time complexity t.*

The proper inclusion $\mathscr{L}_{rt}(\mathrm{IA}) \subset \mathscr{L}_{rt}(\mathrm{NIA})$ is well known (Buchholz et al. 2003). So, nondeterminism strengthens the computational capacity of iterative arrays. On the other hand, it is an open problem whether the family $\mathscr{L}_{rt}(\mathrm{NIA})$ is closed under complementation. Therefore, the question whether the family $\mathscr{L}_{rt}(\mathrm{SVIA})$ is properly included in $\mathscr{L}_{rt}(\mathrm{NIA})$, or whether both families coincide, is of natural interest. Next we turn to relate it to the open complementation closure of $\mathscr{L}_{rt}(\mathrm{NIA})$.

Proposition 3 *Let t be a time-computable time complexity. The family of languages $L \in \mathscr{L}_t(\mathrm{NIA})$, such that the complement $\overline{L}$ belongs to $\mathscr{L}_t(\mathrm{NIA})$ as well, coincides with the family $\mathscr{L}_t(\mathrm{SVIA})$.*

Proof Given a t-time SVIA M, it is straightforward to construct an NIA that accepts the complement of $L(M)$ with the same time complexity t.

Conversely, let M_1 be an NIA accepting L and M_2 be an NIA accepting $\overline{L}$ with time complexity t. Now a t-time self-verifying iterative array M simulates M_1 and M_2 on different tracks, that is, it uses the same two channel technique of Dyer (1980) and Smith (1972).

Then it remains to define the set of accepting states as $F_+ = \{ (s,s') \mid s \in F_1 \}$ and the set of rejecting states as $F_- = \{ (s,s') \mid s' \in F_2 \}$, where F_1 is the set of accepting states of M_1 and F_2 is the set of accepting states of M_2. □

Proposition 3 implies that $\mathscr{L}_{rt}(\mathrm{SVIA})$ is properly included in $\mathscr{L}_{rt}(\mathrm{NIA})$ if and only if $\mathscr{L}_{rt}(\mathrm{NIA})$ is not closed under complementation; otherwise both families coincide.

While iterative arrays fetch their input sequentially through the communication cell, so-called cellular automata obey a parallel input mode. In a preinitial step their cells fetch an input symbol. That is, there are as many cells as input symbols. So, a deterministic two-way cellular automaton (CA) is a linear array of identical deterministic finite automata which are numbered $1, 2, \ldots, n$. Except for border cells the state transition depends on the current state of a cell itself and those of its both nearest neighbors. Border cells receive a boundary symbol # on their free input lines. In a one-way cellular automaton (OCA) the next state of each cell only depends on the state of the cell itself and the state of its immediate neighbor to the right. An input w is accepted by a cellular automaton if at some time step during some computation the leftmost cell enters an accepting state. For cellular automata, realtime is defined to be $t(n) = n$. Cellular automata whose first state transitions are nondeterministic and whose further transitions are deterministic (NCA, NOCA) are studied in Buchholz et al. (2002). Cellular automata that have the self-verifying property (SVCA, SVOCA) are considered in Kutrib and Worsch (2020).

It is well known that deterministic (one-way) cellular automata and deterministic iterative arrays can be sped-up from $(n + t(n))$-time to $(n + \frac{t(n)}{k})$-time, for any integer $k > 0$ (Bucher and Čulik 1984; Ibarra and Jiang 1987; Ibarra et al. 1985; Ibarra and Palis 1985). Thus, lineartime is close to realtime. However, for deterministic one-way cellular automata and iterative arrays, lineartime is strictly more powerful than realtime. The problem is still open for deterministic two-way cellular automata. Concerning nondeterministic cellular automata, the results in Buchholz et al. (2002) show that, in fact, one-way or two-way information flow and realtime or lineartime do not make a difference, that is, the family of languages accepted by realtime NOCAs coincides with the family of languages accepted by lineartime NCAs. However, to our knowledge, these properties and relations are not known for iterative arrays. In order to study them, next, we turn to strong speed-up results for self-verifying iterative arrays.

Theorem 4 *Let $k \geq 1$ be a constant and t be a time complexity. Then the families $\mathscr{L}_{k \cdot t}(\mathrm{SVIA})$ and $\mathscr{L}_t(\mathrm{SVIA})$ coincide.*

Proof A given $(k \cdot t)$-time SVIA M is simulated by a t-time SVIA M' as follows. Basically, M' performs two tasks in parallel on different tracks.

For the first task, assume that the input is fed to the communication cell of M' in k-symbol blocks, that is, k input symbols in each step. Then each k cells of M' are grouped together into one cell. In this well-known way the iterative array M' can simulate k steps of M in one step. That is, this task of M' has time complexity t and the self-verifying property, since M has time complexity $k \cdot t$ and the self-verifying property.

The second task of M' is to make the assumption for the first task true. To this end, it simulates a ring store whose front (and end) is the communication cell. Now the communication cell starts to guess k input symbols in every step. These symbols are fed to the first task. Additionally, the communication step guesses when the end-of-input symbol appears. From that time step on no further input symbols are guessed. In order to verify that the guesses are correct, the k symbols are entered at the end of the ring store respectively. In each step, the symbol at the front of the ring is removed and compared with the actual input symbol. If both match, the guessed symbol is

correct, otherwise it is not. In case of a mismatch or a wrongly guessed number of input symbols the second task remains in a neutral state. If it has guessed the input correctly, it enters a positive state.

Finally, M' accepts if and only if the second task guesses the input correctly and the first task accepts. That is, if the actual input is accepted by M. The iterative array M' rejects if and only if the second task guesses the input correctly and the first task rejects. That is, if the actual input is rejected by M. So, M' has the self-verifying property. □

In particular, we conclude:

Corollary 5 *The families* $\mathscr{L}_{rt}(\text{SVIA})$ *and* $\mathscr{L}_{lt}(\text{SVIA})$ *coincide.*

4 Computational capacity

The first question in connection with the computational capacity of realtime SVIA is the impact of the (restricted) nondeterminism. Does it increase the capacity? More precisely, we are interested in the question whether the computing power of realtime SVIA is strictly stronger than that of realtime IA.

Example 1 shows that the mirror language is accepted by a realtime SVIA. However, by using a completely different algorithm the language is accepted by some deterministic realtime IA as well (Cole 1969). So, it cannot be used as a witness for the strictness of the inclusion $\mathscr{L}_{rt}(\text{IA}) \subset \mathscr{L}_{rt}(\text{SVIA})$. Nevertheless, the strictness follows from a more general result below and is stated in Proposition 11.

The context-free languages form another important language family. The possibility to simulate a pushdown store whose top is simulated by the self-verifying communication cell suggests to compare the family $\mathscr{L}_{rt}(\text{SVIA})$ with the family of context-free languages. In Fernau et al. (2017) self-verifying pushdown automata are considered which accept a strict sub-family of the context-free languages. The simulation of a self-verifying pushdown automaton by a realtime SVIA is straightforward. On the other hand, even deterministic realtime iterative arrays accept non-context-free languages, for example the "copy language" $\{ ww \mid w \in \{a,b\}^* \}$ (Cole 1969).

Corollary 6 *The family of languages accepted by self-verifying pushdown automata is strictly included in the family* $\mathscr{L}_{rt}(\text{SVIA})$.

In order to discuss further comparisons we now turn to results that show the strong computational capacity of realtime SVIAs. First, we turn to nondeterministic devices and show that sequential or parallel input mode induce the same computational capacity as long as the time complexity is linearly bounded.

Lemma 7 *The family* $\mathscr{L}_{lt}(\text{NIA})$ *is included in* $\mathscr{L}_{lt}(\text{NCA})$.

Proof Let M be some lineartime NIA with state set S and set of input symbols Σ. By the known speed-up results we can safely assume that M accepts in $2n$-time. Now we construct an equivalent $2n$-time NCA M' as follows (see Fig. 4).

Since M works in $2n$-time, no more than $2n$ cells of the array are used (n denotes the length of the input). So, each of the n cells of M' simulates two cells of M. In particular, the leftmost cell of M' simulates the communication cell of M together with the first deterministic cell of M. Cell $2 \leq i \leq n$ of M' simulates cells $2(i-1)$ and $2(i-1)+1$ of M (which are initially quiescent).

Basically, M' uses three tracks after the first time step. On the first track the input is successively shifted to the left to feed the simulated communication cell of M, where # symbols are appended at the right. On the third track, the cells of M are simulated, that is, each cell of M' provides two registers on the third track. Also on the second track each cell of M' is split into two registers. During the first time step, all cells on the second track nondeterministically compute two mappings from the finite set of mappings $(\Sigma \cup \{\triangleleft\}) \times S \times S \to S$ and store them into their two registers. These are the first $2n$ guesses of the communication cell of M. The leftmost cell of M' additionally applies the first of its guessed mappings to its actual input symbol and the states s_0 and s_0 in order to determine the new state of the communication cell of M.

In subsequent transitions, the guessed mappings are successively shifted to the left, register by register. The leftmost cell of M' deterministically simulates the nondeterministic behavior of the communication cell of M by applying the next mapping to the current local configuration. When the leftmost cell received the border symbol # in the nth step it simulates further steps of the communication cell of M on the end-of-input symbol $\triangleleft$. In all but the left register of the leftmost cell M' simulates the deterministic transitions of M.

In this way, M' accepts if and only if M accepts. □

Lemma 7 gives an upper bound for languages accepted by lineartime NIAs. It shows that a sequential input mode and one nondeterministic cell can be traded for parallel input mode and all cells nondeterministic once only. In fact, it will turn out that the upper bound is sharp.

Lemma 8 *The family* $\mathscr{L}_{lt}(\text{NCA})$ *is included in* $\mathscr{L}_{rt}(\text{NIA})$.

Proof The possibility to speed-up NCAs by a constant factor is shown in Buchholz et al. (2002). That is, $\mathscr{L}_{lt}(\text{NCA}) = \mathscr{L}_{rt}(\text{NCA})$. So, given some realtime NCA M

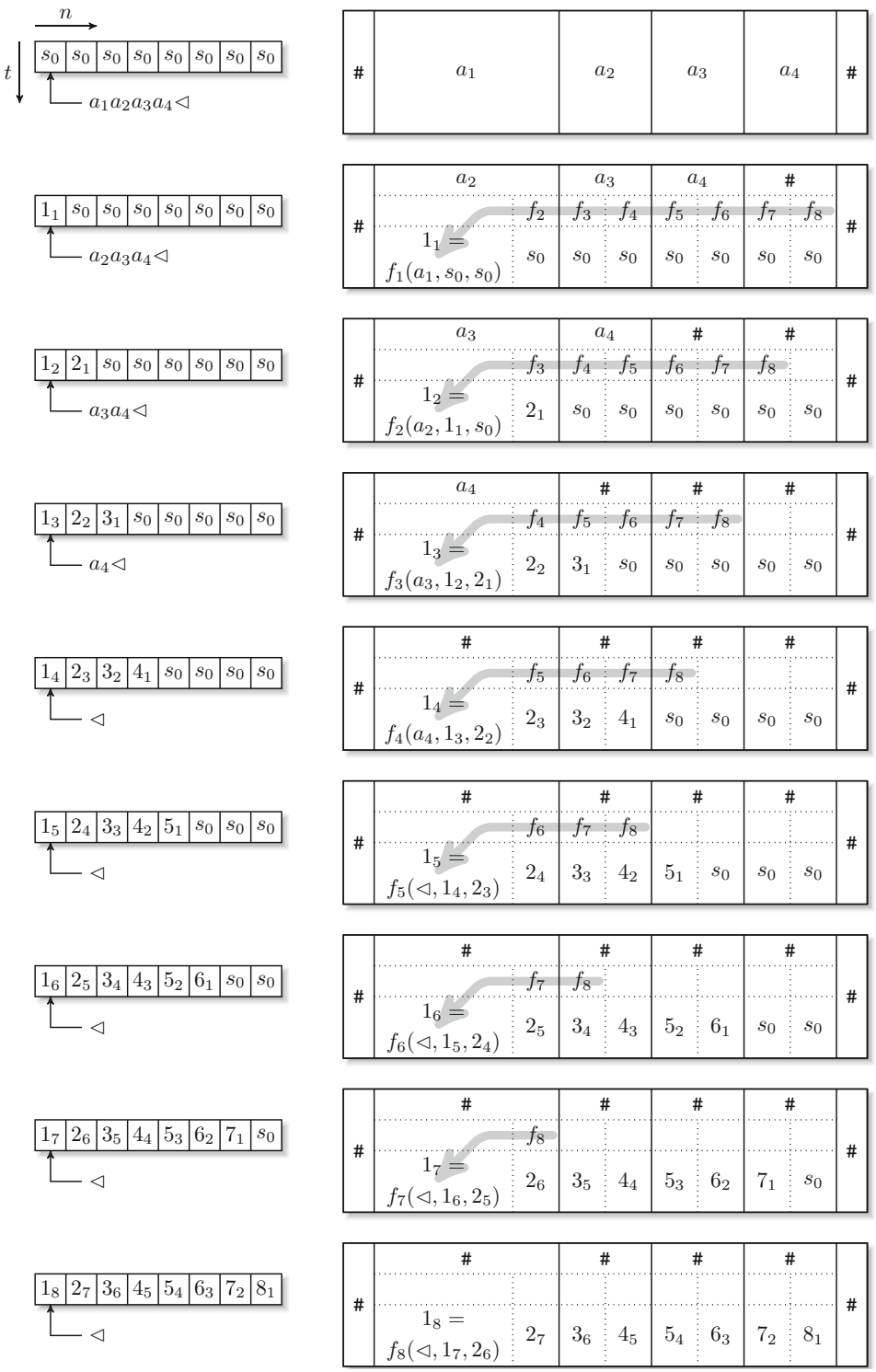

Fig. 4 Simulation of a lineartime (2*n*-time) NIA (left) by a lineartime (2*n*-time) NCA (right)

with state set S and set of input symbols Σ, we construct an equivalent realtime NIA M' as follows. Basically, M' works in three phases. First it guesses and generates a configuration that represents the fourfold packed initial configuration of M. Then this packed part is synchronized. Finally, the synchronized cells simulate M whereby four steps are simulated in one step. The phases are depicted in Fig. 5. In parallel, the guesses are verified.

Phase 1 Let n denote the length of the input. In the following we assume that $n + 1$ is a multiple of four. The generalization of the simulation to the other cases is a straightforward adaption.

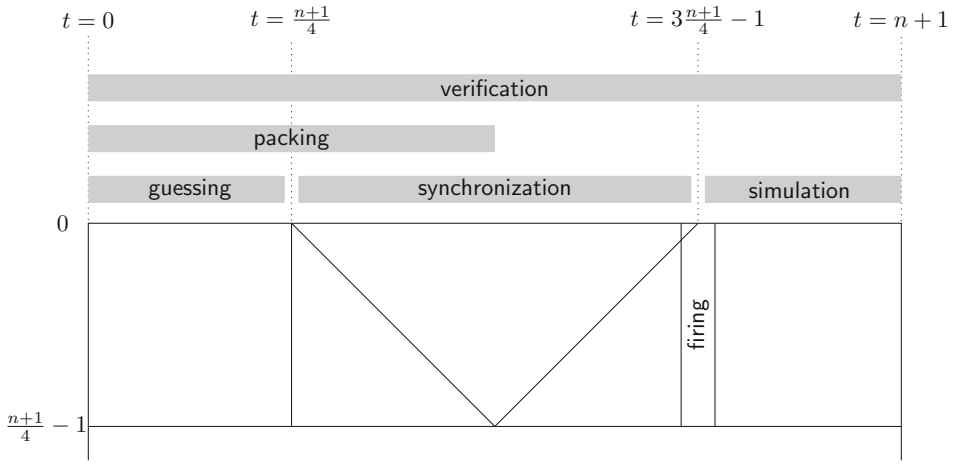

Fig. 5 Simulation phases

So, during the first $\frac{n+1}{4}$ time steps the communication cell of M' guesses four input symbols in every step. Additionally, the communication cell guesses a mapping $(\Sigma \cup \{\#\}) \times \{a\} \times (\Sigma \cup \{\#\}) \to S$ for each (guessed) input symbol $a \in \Sigma$. These mappings are used for the simulation of the nondeterministic first transitions of the cells of M. The blocks of four input symbols together with the corresponding mappings are shifted to the right such that each of the leftmost $\frac{n+1}{4}$ cells gets one block of symbols and mappings. When the communication cell guesses the end-of-input symbol the first phase ends.

In order to verify that the guesses are correct, the four symbols are entered at the end of a ring store respectively. In each step, the symbol at the front of the ring is removed and compared with the actual input symbol. If both match, the guessed symbol is correct, otherwise it is not. In case of a mismatch or a wrongly guessed number of input symbols the computation blocks in a neutral state.

Phase 2 At time step $\frac{n+1}{4} + 1$ the communication cell initiates an FSSP synchronization of the leftmost $\frac{n}{4}$ cells. The blocks of four input symbols together with the corresponding mappings arrive at their destination cells $0 \leq i \leq \frac{n+1}{4} - 1$ at time $2i + 1$. The initial signal for the FSSP arrives at cell i at time $\frac{n+1}{4} + 1 + i > 2i + 1$. So, each cell starts Phase 2 after finishing Phase 1. Altogether, Phase 2 is finished when all cells are synchronized at time $\frac{n+1}{4} + 1 + 2 \cdot \frac{n+1}{4} - 2 = 3 \cdot \frac{n+1}{4} - 1$.

Phase 3 Due to the compressed representation, M' can simulate M with fourfold speed. In order to simulate the nondeterministic transitions which the cells of M perform during the first time step, the cells of M' apply the nondeterministically guessed mappings to their local configurations. Thus, M' simulates the nth step of M at time step $3 \cdot \frac{n+1}{4} - 1 + \frac{n+1}{4} = n$.

Since the verification of the guessed input takes $n + 1$ time steps, we conclude that the total time complexity of M' is $t(n) = n + 1$, that is realtime.

Finally, M' accepts if and only if the input has been guessed correctly and M accepts. So, we have $L(M') = L(M)$. □

Lemmas 7 and 8 reveal $\mathscr{L}_{lt}(\text{NIA}) \subseteq \mathscr{L}_{lt}(\text{NCA}) \subseteq \mathscr{L}_{rt}(\text{NIA})$. Together with the trivial inclusion $\mathscr{L}_{rt}(\text{NIA}) \subseteq \mathscr{L}_{lt}(\text{NIA})$ and the results in Buchholz et al. (2002), we obtain the equality of the next theorem.

Theorem 9 $\mathscr{L}_{rt}(\text{NIA}) = \mathscr{L}_{lt}(\text{NIA}) = \mathscr{L}_{lt}(\text{NCA}) = \mathscr{L}_{rt}(\text{NCA}) = \mathscr{L}_{rt}(\text{NOCA}) = \mathscr{L}_{lt}(\text{NOCA})$.

Since the linear functions $t(n) \geq n$ are time computable by one-way cellular automata as well as by iterative arrays, Proposition 3 and a similar proposition for cellular automata from Kutrib and Worsch (2020) can be used to build the bridge from nondeterministic to self-verifying devices.

Theorem 10 $\mathscr{L}_{rt}(\text{SVIA}) = \mathscr{L}_{lt}(\text{SVIA}) = \mathscr{L}_{lt}(\text{SVCA}) = \mathscr{L}_{rt}(\text{SVCA}) = \mathscr{L}_{rt}(\text{SVOCA}) = \mathscr{L}_{lt}(\text{SVOCA})$.

Proof By Theorem 9, all language families $\mathscr{L}_{rt}(\text{NIA})$, $\mathscr{L}_{lt}(\text{NIA})$, $\mathscr{L}_{lt}(\text{NCA})$, $\mathscr{L}_{rt}(\text{NCA})$, $\mathscr{L}_{rt}(\text{NOCA})$, and $\mathscr{L}_{lt}(\text{NOCA})$ coincide. So, all families consisting of the complements of the languages from the original families coincide as well. Now Proposition 3 and a similar proposition for cellular automata from Kutrib and Worsch (2020) imply that all language families $\mathscr{L}_{rt}(\text{SVIA})$, $\mathscr{L}_{lt}(\text{SVIA})$, $\mathscr{L}_{lt}(\text{SVCA})$, $\mathscr{L}_{rt}(\text{SVCA})$, $\mathscr{L}_{rt}(\text{SVOCA})$, and $\mathscr{L}_{lt}(\text{SVOCA})$ are identical. □

In particular, now we can deduce that even the restricted nondeterminism gained in considering a self-verifying communication cell strictly increases the computational capacity of realtime iterative arrays. That is, the inclusion $\mathscr{L}_{rt}(\text{IA}) \subset \mathscr{L}_{rt}(\text{SVIA})$ is strict.

Proposition 11 *The family* $\mathscr{L}_{rt}(\mathrm{IA})$ *is strictly included in* $\mathscr{L}_{rt}(\mathrm{SVIA})$.

Proof The relations $\mathscr{L}_{rt}(\mathrm{IA}) \subset \mathscr{L}_{lt}(\mathrm{IA}) = \mathscr{L}_{lt}(\mathrm{CA})$ are known (see, for example, Kutrib 2009). Since $\mathscr{L}_{lt}(\mathrm{IA}) \subseteq \mathscr{L}_{lt}(\mathrm{SVIA}) = \mathscr{L}_{rt}(\mathrm{SVIA})$ the assertion follows. □

Theorem 10 shows that a sequential input mode and one nondeterministic cell can be traded for parallel input mode and all cells nondeterministic once only, and vice versa. To this end, it does not matter whether the computations are in realtime or lineartime. But what about the world beyond lineartime? Are self-verifying arrays stronger than deterministic ones? Or weaker than nondeterministic ones? The open question of the strictness of one of the inclusions

$$\mathscr{L}_{lt}(\mathrm{IA}) = \mathscr{L}_{lt}(\mathrm{CA}) \subseteq \mathscr{L}_{rt}(\mathrm{SVCA}) \subseteq \mathscr{L}(\mathrm{SVCA}) \subseteq \mathscr{L}(\mathrm{NCA})$$

is strongly related to famous open problems in complexity theory (see Kutrib 2015). Note that at the top of this hierarchy are devices that may have an exponential time complexity (due to the space bound).

5 Closure of the realtime self-verifying language family under concatenation

The closure properties of the family $\mathscr{L}_{rt}(\mathrm{SVOCA})$ have been investigated in Kutrib and Worsch (2020). By Theorem 10 we know that these are the closure properties of the identical language family $\mathscr{L}_{rt}(\mathrm{SVIA})$. It is known that the family is closed under the Boolean operations and reversal. In particular, the closure under reversal is of crucial importance. It is an open problem for $\mathscr{L}_{rt}(\mathrm{CA})$ and, equivalently, for $\mathscr{L}_{lt}(\mathrm{OCA})$. Moreover, it is linked with the open closure property under concatenation for the same family and, hence, with the question whether lineartime CAs are more powerful than realtime CAs. It is known that the family $\mathscr{L}_{rt}(\mathrm{IA})$ is not closed under reversal, while the family $\mathscr{L}_{lt}(\mathrm{IA})$ is closed.

Concerning the operations homomorphism and inverse homomorphism, the closure of $\mathscr{L}_{rt}(\mathrm{SVIA})$ under inverse homomorphism and the non-closure under homomorphisms is known.

The question whether the family $\mathscr{L}_{rt}(\mathrm{SVOCA})$ is closed under concatenation is stated as an open problem in Kutrib and Worsch (2020). Here we can solve the problem in terms of self-verifying iterative arrays.

Proposition 12 *The family of languages accepted by realtime* SVIA *is closed under concatenation.*

Proof Let $L_1, L_2 \in \mathscr{L}_{rt}(\mathrm{SVIA})$. If the empty word belongs to L_1 then language L_2 belongs to the concatenation and vice versa. Since the family of languages accepted by realtime SVIA is closed under union, it remains to consider languages $L_1, L_2 \in \mathscr{L}_{rt}(\mathrm{SVIA})$ that do not contain the empty word. Let M_1 and M_2 be realtime SVIA that accept L_1 and L_2. Since the family $\mathscr{L}_{rt}(\mathrm{SVIA})$ is closed under reversal, there is a realtime SVIA M_2^R that accepts the reversal L_2^R of L_2.

A realtime SVIA M that accepts the concatenation $L_1 \cdot L_2$ works as follows. First we describe two tasks that are performed by M in parallel.

Basically, the first task is to read the input and to simulate M_1. In addition, the input is stored into a ring whose front is the communication cell. Moreover, in any simulation step, M tests whether it would accept or reject the input prefix read so far by checking if it would accept or reject when the next input symbol were the end-of-input symbol $\lhd$. If the current input prefix is accepted or rejected, the input symbol stored into the ring is marked suitably.

The second task is to guess the reversal of the input symbol by symbol. The guessed reversal is stored into a pushdown store whose top is the communication cell. Additionally, the realtime SVIA M_2^R is simulated on the guessed input. Similarly as for the first task, if the current prefix of the guessed input would be accepted or rejected, the guessed input symbol stored into the pushdown store is marked suitably.

Let $x_1x_2\cdots x_n$ be the actual input. It is stored in the ring when the end-of-input symbol appears. At that time, let $y_1y_2\cdots y_n$ be the content of the pushdown store (from top to bottom). Clearly, M has guessed the reversal of the input correctly if and only if $x_1x_2\cdots x_n = y_1y_2\cdots y_n$. So, after having read the end-of-input symbol, the SVIA M verifies the guessed reversal of the input by successively removing symbols from the ring and pushdown store and testing whether they match. If M detects any mismatch it blocks in a neutral state.

Now, assume that the reversal of the input has been guessed correctly.

Additionally, while verifying the guesses by successively scanning the ring and the pushdown store, the SVIA M tests whether for some $1 \leq i \leq n-1$ input symbol x_i is marked by the simulation of M_1 and symbol y_{i+1} is marked by the simulation of M_2^R.

Case 1 $x_1\cdots x_n \in L_1L_2$, say $x_1\cdots x_i \in L_1$. Then there are computations by M_1 accepting $x_1\cdots x_i$ and by M_2^R accepting

$$y_ny_{n-1}\cdots y_{i+1} = x_nx_{n-1}\cdots x_{i+1} = (x_{i+1}\cdots x_n)^R.$$

Having M accept an input if and only if x_i is marked by M_1 and symbol y_{i+1} is marked by M_2^R makes M accept all words in L_1L_2. In detail, if input symbol x_i is marked accepting by the simulation of M_1 the word $x_1x_2\cdots x_i$

belongs to the language L_1. If input symbol y_{i+1} is marked accepting by the simulation of M_2^R, the word $y_n y_{n-1} \cdots y_{i+1} = x_n x_{n-1} \cdots x_{i+1}$ belongs to the language L_2^R and, thus, $x_{i+1} x_{i+2} \cdots x_n$ belongs to the language L_2. So, the input $x_1 x_2 \cdots x_n$ belongs to the concatenation $L_1 \cdot L_2$. In this way, M can accept any input from $L_1 \cdot L_2$ and only inputs from the concatenation $L_1 \cdot L_2$.

Case 2 $x_1 \cdots x_n \notin L_1 L_2$. In this case for each i either $x_1 \cdots x_i \notin L_1$ or $x_n \cdots x_{i+1} \notin L_2^R$ or both. That means that for each i there are computations by M_1 and M_2^R for the respective inputs such that at least one of both rejects. Hence, there will be a computation of M which correctly explicitly rejects an input, if for any two adjacent cells always at least one of them is marked rejecting. In detail, if the test finds neither two adjacent cells marked accepting, nor two adjacent cells that are marked accepting and unmarked, nor two adjacent cells unmarked the communication cell enters a rejecting state. In this case, no matter between which two adjacent symbols one assumes the cut between first and second factor, M has explicitly rejected at least one of them. Clearly, in this case the input cannot belong to the concatenation. On the other hand, if some input does not belong to the concatenation, then there is always a computation of M that results in such a marking. So, M rejects any input that does not belong to $L_1 \cdot L_2$ and only inputs that do not belong to $L_1 \cdot L_2$. In any other case, the leftmost cell remains in a neutral state.

For the computation on input of length n the SVIA M takes $n+1$ steps to read (and guess) the input for the tasks, and further $n+1$ steps to verify the guesses and test the markings. So, M works in lineartime which can be sped-up to realtime. □

The closure properties of $\mathscr{L}_{rt}(\text{SVIA})$ with respect to iteration (Kleene star) and non-erasing homomorphisms are still open problems. They are settled for nondeterministic devices since, basically, for iteration it is sufficient to guess the positions in the input at which words are concatenated, and for non-erasing homomorphism it is sufficient to guess the pre-image of the input. However, self-verifying devices have to reject explicitly if the input does not belong to the language. Intuitively, this means that they have to 'know' that all possible guesses either do not lead to accepting computations or are 'wrong.' The closure properties are summarized in Table 1.

6 Decidability questions

First we note that the membership problem is obviously decidable for SVIAs obeying a time-computable time complexity.

On the other hand, in Kutrib (2009) it is observed that for any language family that effectively contains $\mathscr{L}_{rt}(\text{IA})$, the problems emptiness, universality, finiteness, infiniteness, regularity, and context-freeness are not semidecidable. Since we know

$$\mathscr{L}_{rt}(\text{IA}) \subset \mathscr{L}_{lt}(\text{IA}) \subseteq \mathscr{L}_{lt}(\text{SVIA}) = \mathscr{L}_{rt}(\text{SVIA})$$

we derive the next corollary.

Corollary 13 *The problems emptiness, universality, finiteness, infiniteness, inclusion, equivalence, regularity, and context-freeness are not semidecidable for realtime* IA*s and thus for realtime* SVIA*s.*

In Kutrib and Worsch (2020) it is shown that the problem to decide whether a given realtime one-way cellular automaton is self-verifying or not is undecidable. Unfortunately, the result has no direct implications for the same question for iterative arrays. However, the undecidability for cellular automata is shown by a reduction of the emptiness problem. We turn to prove the undecidability for iterative arrays as well. Moreover, we use a reduction of the emptiness and universality problem, but the reduction itself is different. Since general iterative arrays do not have neutral or rejecting states (only accepting and non-accepting states), there is no partitioning of the state set. So, the decidability can be asked for a given fixed partitioning or for the existence of a partitioning. We first consider the latter question.

Theorem 14 *Given a realtime (non)deterministic iterative array M with state set S and accepting states F_+, it is not semidecidable whether there exists $F_- \subseteq (S \setminus F_+)$ such that M is an* SVIA *with respect to the sets F_+ and F_-.*

Table 1 Closure properties of the language family $\mathscr{L}_{rt}(\text{SVIA}) = \mathscr{L}_{lt}(\text{SVCA})$ in comparison with the families $\mathscr{L}_{rt}(\text{IA})$, $\mathscr{L}_{lt}(\text{IA}) = \mathscr{L}_{lt}(\text{CA})$, and $\mathscr{L}_{rt}(\text{OCA})$, where h_λ denotes λ-free homomorphisms

Family	——	$\cup$	$\cap$	R	$\cdot$	$*$	h_λ	h	h^{-1}
$\mathscr{L}_{rt}(\text{SVIA}) = \mathscr{L}_{lt}(\text{SVCA})$	✔	✔	✔	✔	✔	?	?	✗	✔
$\mathscr{L}_{rt}(\text{IA})$	✔	✔	✔	✗	✗	✗	✗	✗	✔
$\mathscr{L}_{lt}(\text{IA}) = \mathscr{L}_{lt}(\text{CA})$	✔	✔	✔	✔	?	?	?	✗	✔
$\mathscr{L}_{rt}(\text{OCA})$	✔	✔	✔	✔	✗	✗	✗	✗	✔

Proof Let $M_0 = \langle S, \Sigma, F_+, s_0, \triangleleft, \delta_{nd}, \delta_d \rangle$ be an arbitrary realtime IA. We safely may assume that a cell which has left the quiescent state will never enter the quiescent state again. This behavior can be implemented by adding a new state that plays the role of the quiescent state. If necessary, the new state can be entered instead of s_0.

We modify M_0 to $M_1 = \langle S', \Sigma', F'_+, s_0, \triangleleft, \delta'_{nd}, \delta'_d \rangle$ by adding a new input symbol \$ and two new states p_+ and p_0. So, we set $S' = S \cup \{p_+, p_0\}$, $\Sigma' = \Sigma \cup \{\$\}$, and $F'_+ = F_+ \cup \{p_+\}$. The intention is that a \$ in the input causes the IA to simulate a step on the end-of-input symbol $\triangleleft$ (in restricted form) and to reinitialize the computation by letting the cells enter the quiescent state again (which is impossible in M_0). Therefore, the transition function δ'_{nd} is basically δ_{nd} extended by transitions for the input symbol \$ and the states p_+ and p_0. When a \$ appears in the input, the communication cell enters state p_+ if it could enter an accepting state on the end-of-input symbol $\triangleleft$. For all $s_1, s_2 \in S$,

$$\delta'_{nd}(\$, s_1, s_2) = \{p_+\} \text{ if } \delta_{nd}(\triangleleft, s_1, s_2) \cap F_+ \neq \emptyset.$$

Otherwise, it enters state p_0: $\delta'_{nd}(\$, s_1, s_2) = \{p_0\}$ if $\delta_{nd}(\triangleleft, s_1, s_2) \cap F_+ = \emptyset$. In state p_+ or p_0 the computation continues as it would from the very beginning. For $p \in \{p_+, p_0\}$, all $a \in \Sigma' \cup \{\triangleleft\}$, and all $s \in S'$, $\delta'_{nd}(a, p, s) = \delta'_{nd}(a, s_0, s_0)$. In order to implement the reinitialization of the other cells, recall that δ_d drives no non-quiescent cell into the quiescent state. So, we can utilize the quiescent state as a signal sent by the communication cell. The signal causes the reinitialization of the cells passed through. So, the transition function δ'_d is basically δ_d extended as follows. For $p \in \{p_+, p_0\}$ and all $s_1, s_2 \in S$,

$$\delta'_d(p, s_1, s_2) = s_0, \quad \delta'_d(s_0, s_1, s_2) = s_0, \quad \delta'_d(s_1, s_0, s_2) = \delta(s_1, s_0, s_0).$$

Therefore, $L(M_1)$ consists of all concatenations of \$ separated words u_i such that at least one u_i is in $L(M_0)$. In particular $L(M_1) \cap \Sigma^* = L(M_0)$.

We claim that there exists $F'_- \subseteq (S' \setminus F'_+)$ such that the iterative array $M_2 = \langle S', \Sigma', F'_+, F'_-, s_0, \triangleleft, \delta'_{nd}, \delta'_d \rangle$ is self-verifying if and only if $L(M_0)$ is empty or coincides with Σ^*. Observe that $L(M_2) = L(M_1)$ because both have the same set of accepting states.

If $L(M_0)$ is empty then $L(M_1)$ is empty. Therefore, the communication cell will never enter an accepting state from F'_+. So, we safely may set $F'_- = (S' \setminus F'_+)$ and obtain that M_2 is self-verifying. Similarly, if $L(M_0) = \Sigma^*$ then $L(M_1) = \Sigma'^*$, and we safely may set $F'_- = \emptyset$ to obtain a self-verifying IA.

Now assume that $L(M_0)$ and, thus, $L(M_2)$ neither be empty nor contain all words over the input alphabet. Then there exists some $u \in L(M_0)$ and some $v \notin L(M_0)$. We consider the computation of M_2 on input $u\$v$. Since M_0 accepts u, the IA M_2 enters an accepting state while processing the input prefix $u\$$ (its computation is a simulation of M_0 on $u\triangleleft$). Then the computation of M_2 is reinitialized and continues with a simulation of M_0 on input v. Since $v \notin L(M_0)$, in this phase, M_2 cannot accept v. However, since it already was in an accepting state and its overall answer is already *yes*, M_2 cannot enter a contradictory rejecting state in this phase. This implies that the communication cell of M_2 on input v will only assume neutral states and, thus, neither accept nor reject v. That is, M_2 is not self-verifying and the claim follows.

From the construction of M_2 and the claim we conclude that the semidecidability of the problem in question implies the semidecidability of the emptiness or universality problem for realtime IAs contradicting Corollary 13. □

What about the undecidability if we provide a partitioning of its state set? Can we test if this partitioning makes the IA self-verifying? The answer is no, since for a given realtime iterative array with accepting state set F_+ there are only finitely many partitions induced by setting $F_- \subseteq (S \setminus F_+)$. All these could be tested in parallel. Now the problem in question can be semidecided if the test is successful for at least one partitioning.

Corollary 15 *Given a realtime (non)deterministic iterative array M with state set S and partitioning $S = F_+ \dot\cup F_- \dot\cup F_0$, it is not semidecidable whether M is an* SVIA *with respect to the partitioning.*

By Lemma 2, any deterministic iterative with a time-computable time complexity can effectively be made self-verifying. But it is non-semidecidable whether it already *is* self-verifying. This non-semi-decidability carries immediately over to nondeterministic iterative arrays. However, it is an open problem whether any nondeterministic iterative with a time-computable time complexity can effectively be made self-verifying. In fact, it is an open problem whether the family of languages accepted by realtime nondeterministic iterative arrays is closed under complementation or not.

Acknowledgements We would like to thank Thomas Worsch. He participated in the discussions and his ideas were significant contributions to this paper. We consider him truly a co-author, but he insisted not to put his name on the author list.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

Bucher W, Čulik K II (1984) On real time and linear time cellular automata. RAIRO Inform. Théor. 18:307–325

Buchholz T, Kutrib M (1997) Some relations between massively parallel arrays. Parallel Comput. 23:1643–1662

Buchholz T, Klein A, Kutrib M (2002) On interacting automata with limited nondeterminism. Fund. Inform. 52:15–38

Buchholz T, Klein A, Kutrib M (2003) Iterative arrays with limited nondeterministic communication cell. In: Words, languages and combinatorics III. World Scientific Publishing, pp 73–87

Cole SN (1969) Real-time computation by n-dimensional iterative arrays of finite-state machines. IEEE Trans. Comput. C–18:349–365

Čulik K II, Yu S (1984) Iterative tree automata. Theor. Comput. Sci. 32:227–247

Duris P, Hromkovic J, Rolim JDP , Schnitger G (1997) Las vegas versus determinism for one-way communication complexity, finite automata, and polynomial-time computations. In: Theoretical aspects of computer science (STACS 1997), LNCS, vol. 1200. Springer, pp 117–128

Dyer CR (1980) One-way bounded cellular automata. Inform. Control 44:261–281

Fernau H, Kutrib M, Wendlandt M (2017) Self-verifying pushdown automata. In: Non-classical models of automata and applications (NCMA 2017), *books@ocg.at*, vol 329. Austrian Computer Society, Vienna, pp 103–117

Fischer PC, Kintala CMR (1979) Real-time computations with restricted nondeterminism. Math. Syst. Theory 12:219–231

Hromkovic J, Schnitger G (2001) On the power of las vegas for one-way communication complexity, OBDDs, and finite automata. Inf. Comput. 169:284–296

Hromkovic J, Schnitger G (2003) Nondeterministic communication with a limited number of advice bits. SIAM J. Comput. 33:43–68

Ibarra OH, Jiang T (1987) On one-way cellular arrays. SIAM J. Comput. 16:1135–1154

Ibarra OH, Palis MA (1985) Some results concerning linear iterative (systolic) arrays. J. Parallel Distrib. Comput. 2:182–218

Ibarra OH, Kim SM, Moran S (1985) Sequential machine characterizations of trellis and cellular automata and applications. SIAM J. Comput. 14:426–447

Ilie L, Păun G, Rozenberg G, Salomaa A (2000) On strongly context-free languages. Discrete Appl. Math. 103:158–165

Jirásková G (2005) State complexity of some operations on binary regular languages. Theor. Comput. Sci. 330(2):287–298

Jirásková G, Pighizzini G (2011) Optimal simulation of self-verifying automata by deterministic automata. Inf. Comput. 209:528–535

Kintala CMR (1977) Computations with a restricted number of nondeterministic steps. Ph.D. thesis, Pennsylvania State University

Kutrib M (2008) Cellular automata—a computational point of view. In: Bel-Enguix G, Jiménez-López MD, Martín-Vide C (eds) New developments in formal languages and applications, chap. 6. Springer, Berlin, pp 183–227

Kutrib M (2009) Cellular automata and language theory. In: Encyclopedia of complexity and system science. Springer, pp 800–823

Kutrib M (2015) Complexity of one-way cellular automata. In: Cellular automata and discrete complex systems (AUTOMATA 2014), LNCS, vol 8996. Springer, pp 3–18

Kutrib M, Worsch T (2019) Iterative arrays with self-verifying communication cell. In: Cellular automata and discrete complex systems (AUTOMATA 2019), LNCS, vol 11525. Springer, pp 77–90

Kutrib M, Worsch T (2020) Self-verifying cellular automata. J. Cell. Autom. 15:223–242

Mazoyer J, Terrier V (1999) Signals in one-dimensional cellular automata. Theor. Comput. Sci. 217:53–80

Smith AR III (1972) Real-time language recognition by one-dimensional cellular automata. J. Comput. System Sci. 6:233–253