Natural Computing (2022) 21:361–375
https://doi.org/10.1007/s11047-022-09887-1

One-dimensional pattern generation by cellular automata

Martin Kutrib[1] · Andreas Malcher[1]

Accepted: 26 April 2022 / Published online: 24 May 2022

Abstract
To determine the computational capacity of cellular automata they are often investigated towards their ability to accept formal languages within certain time constraints. In this paper, we take up an opposite position and look at cellular automata towards their ability to *generate* patterns, within certain time constraints. As an example we describe a construction of a cellular automaton that generates prefixes of the Oldenburger–Kolakoski sequence within real time. Furthermore, we study the real-time generation of unary and non-unary patterns in depth. In the unary case, we obtain a characterization by time-constructible functions and their corresponding unary formal languages. In the non-unary case, we provide constructions that generate any arbitrary given properly thin context-free language as well as all prefixes of any given automatic sequence.

Keywords Cellular automata · Pattern generation · Thin languages · Automatic sequences

1 Introduction

Parallel computational models are appealing and widely used in order to describe, understand, and manage parallel processes occurring in real life. Cellular automata (CA) are a model which allows to describe massively parallel systems, since they are arrays of identical copies of deterministic finite automata. Furthermore, the single nodes are homogeneously connected to both their immediate neighbors, and they work synchronously at discrete time steps. In general, cellular automata work on a given input which is provided in a parallel way, that is, every cell is fed with an input symbol in a pre-initial step.

The computational power of cellular automata can be measured by their ability to accept formal languages. In this context, the given input is accepted if there is a time step at which the leftmost cell enters an accepting state. Usually studied models comprise the real-time one-way cellular automata (Dyer 1980), where every cell is connected with its right neighbor only which restricts the flow of information from right to left. Moreover, the available time for accepting an input is restricted to the length of the input. Other models studied are real-time two-way cellular automata and linear-time two-way cellular automata (Smith 1970, 1972; Choffrut and Čulik 1984). A survey on formal language results concerning in particular the computational capacity, closure properties and decidability questions for these models and references to the literature may be found, for example, in Kutrib (2008), Kutrib (2009).

Another perspective on computations with cellular automata is taken in Grandjean et al. (2012), Kutrib and Malcher (2013), where cellular automata are used as transducers, that is, they transform an input into an output obeying time constraints such as real and linear time. The paper (Grandjean et al. 2012) discusses for cellular automata several time constraints and inclusion relationships based on these constraints. Moreover, closure properties and relations to cellular automata considered as formal language acceptors are established. In Kutrib and Malcher (2013) also cellular automata with sequential input mode, called iterative arrays, are considered as transducing devices and compared with the cellular automata counterpart with parallel input mode. Additionally, the cellular

Some of the results of this paper have been announced at ACRI 2020 in Łódź, December 2020. An extended abstract appeared in the proceedings of that conference (Kutrib and Malcher 2020).

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

Andreas Malcher
andreas.malcher@informatik.uni-giessen.de

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

transducing models are compared with classical sequential transducing devices such as finite state transducers and pushdown transducers.

In this paper, we will take yet another view on computations with cellular automata. Namely, we will consider cellular automata not as devices processing an input and computing a yes or no answer as in the case of a language accepting device or computing an output as in the case of a transducing device, but we consider cellular automata as generating devices. This means that the cellular automaton starts with an arbitrary number of cells being all in a quiescent state and being bordered by a permanent boundary symbol at both ends. Subsequently, it works synchronously according to its transition function. Finally, if the configurations reach a fixpoint, we consider such configurations as the *patterns generated* by the automaton. Thus, cellular automata considered this way compute a (partial) function mapping an initial length n to a pattern of length n over the alphabet of the automaton. In Kutrib and Malcher (2021), first investigations on the basic ability of cellular automata to compute such functions within real time have been made. The results obtained concern basically structural properties such as speed-up constructions, closure properties, and decidability questions. Here, we will study the real-time generation of unary and non-unary patterns in more depth.

It should be remarked that the notion of pattern generation is used for cellular automata also in other contexts, but with a different meaning. For example, in Wolfram (1986) the sequence of configurations produced by a cellular automaton starting with some input is considered as a two-dimensional pattern generated and the complexity of such patterns is investigated for example for the regular case in de Oliveira et al. (2016). In Kari (2012) a cellular automaton is studied as universal pattern generator in the sense that starting from a finite configuration all finite patterns over the state alphabet are generated. This means that these patterns occur as infixes in the sequence of configurations computed.

The paper is organized as follows. In Sect. 2 we provide a formal definition of cellular automata and describe how they accept formal languages as well as they generate patterns within time constraints, in particular, within real time and linear time. The section is concluded with an illustrating example generating prefixes of the Oldenburger–Kolakoski sequence. In Sect. 3, we study the ability of real-time cellular automata to generate unary patterns in depth. A given function $f : \mathbb{N} \to \mathbb{N}$ defines a unary pattern in a natural way, namely, the pattern consists of all strings a^n, where n is in the range of f, and is undefined otherwise. A first result is that such patterns can be generated in real time under the condition that the function f has some constructibility properties. If the pattern is modified so that strings b^n are generated if n is not in the range of f, then another result shows that the generation is still possible in real time, but the construction is much more involved, since the information whether or not n belongs to the range of f is locally computed, but has to be transported to all cells in due time. Moreover, it can be shown that the notions of time-constructibility, real-time language acceptance, and real-time pattern generation are equivalent in the unary case. In Sect. 4, we study real-time cellular automata that generate non-unary patterns in more detail. It is a basic observation that every pattern generated by a real-time cellular automaton is a properly thin language, that is, it contains at most one word of length n for all $n \geq 1$. Hence it is of interest which thin languages are in turn generated by real-time cellular automata. In Sect. 4.1 we first show that every finite properly thin language can be generated in real time. Subsequently, the construction can be enlarged to generate every regular properly thin language and, finally, an additional refinement gives that also every context-free properly thin language can be generated in real time. In Sect. 4.2 we consider another class of non-unary patterns, namely those which are described by prefixes of automatic sequences such as the well-know Thue-Morse sequence. Automatic sequences are introduced and intensely studied in Allouche and Shallit (2003). Our main result here is that every pattern defined by the prefixes of any given automatic sequence can be generated by a real-time cellular automaton.

2 Preliminaries

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 we set $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. A subset of Σ^* is called a *language* over Σ. For the *reversal of a word* w we write w^R and for its *length* we write $|w|$. In general, we use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*. For convenience, we use $S_\#$ to denote $S \cup \{\#\}$.

A two-way cellular automaton is a linear array of identical finite automata, called cells, numbered $1, 2, \ldots, n$. Except for border cells each one is connected to its both nearest neighbors. The state transition depends on the current state of a cell itself and the current states of its two neighbors, where the outermost cells receive a permanent boundary symbol on their free input lines. The cells work synchronously at discrete time steps.

Formally, a *deterministic two-way cellular automaton* (CA, for short) is a system $M = \langle S, \Sigma, F, s_0, \#, \delta \rangle$, where S is the finite, nonempty set of *cell states*, $\Sigma \subseteq S$ is set of *input symbols*, $F \subseteq S$ is the set of *accepting states*, $s_0 \in S$ is the *quiescent state*, $\# \notin S$ is the permanent *boundary symbol*, and $\delta : S_\# \times S \times S_\# \to S$ is the *local transition function* satisfying $\delta(s_0, s_0, s_0) = s_0$.

A *configuration* c_t of M at time $t \geq 0$ is a mapping $c_t : \{1, 2, \ldots, n\} \to S$, for $n \geq 1$, occasionally represented as a word over S. Given a configuration c_t, $t \geq 0$, its successor configuration is computed according to the global transition function Δ, that is, $c_{t+1} = \Delta(c_t)$, as follows. For $2 \leq i \leq n-1$,

$$c_{t+1}(i) = \delta(c_t(i-1)), c_t(i), c_t(i+1)),$$

and for the outermost cells we set

$$c_{t+1}(1) = \delta(\#, c_t(1), c_t(2)) \quad \textit{and} \quad c_{t+1}(n) = \delta(c_t(n-1), c_t(n), \#).$$

Thus, the global transition function Δ is induced by δ.

Here, a cellular automaton M can operate as decider or generator of one-dimensional patterns (or words, or strings).

A cellular automaton *accepts* a word $a_1 a_2 \cdots a_n \in \Sigma^+$, if at some time step during the course of the computation starting in the *initial configuration* $c_0(i) = a_i$, $1 \leq i \leq n$, the leftmost cell enters an accepting state, that is, the leftmost symbol of some reachable configuration is an accepting state. If the leftmost cell never enters an accepting state, the input is *rejected*. The *language accepted by M* is denoted by

$$L(M) = \{\, w \in \Sigma^+ \mid w \text{ is accepted by } M \,\}.$$

A cellular automaton *generates* a word $a_1 a_2 \cdots a_n$, if at some time step t during the computation on the initial configuration $c_0(i) = s_0$, $1 \leq i \leq n$, (i) the word appears as configuration (that is, $c_t(i) = a_i$, $1 \leq i \leq n$) and (ii) configuration c_t is a fixpoint of the global transition function Δ (that is, the configuration is stable from time t on). The *pattern generated by M* is

$$P(M) = \{\, w \in S^+ \mid w \text{ is generated by } M \,\}.$$

Since the set of input symbols and the set of accepting states are not used when a cellular automaton operates as generator, we may safely omit them from its definition.

Let $t : \mathbb{N} \to \mathbb{N}$ be a mapping. If all $w \in L(M)$ are accepted with at most $t(|w|)$ time steps, or if all $w \in P(M)$ are generated with at most $t(|w|)$ time steps, then M is said to be of time complexity t. If $t(n) = n$ then M operates in *real time*. If $t(n) = k \cdot n$ for a rational number $k \geq 1$ then M operates in *linear time*.

Before we illustrate the definitions with an example we recall a basic technique which is useful for constructing cellular automata that generate some sophisticated patterns. The technique is basically the possibility of cellular automata to simulate certain data structures without any loss of time (Kutrib 2008). Here we consider in particular the data structures queues and rings, where a ring is a queue that can write and erase at the same time (for convenience we will call both queue in the sequel). For the simulation, some designated cell (the rightmost one, for example) simulates the front and the end of the queues. For the sake of completeness, we next recall exemplarily the principle of this simulation from (Kutrib 2008).

2.1 Simulation of a queue and ring store

It suffices to use three additional tracks for the simulation. Let the three registers of each cell be numbered one, two, and three from top to bottom, and suppose that the second register is connected to the first register of the right neighbor, and the third register is connected to the third register of the right neighbor. The content of the store is identified by scanning the registers as connected. That is, beginning in the designated cell, first the first register is scanned and then the second register. Next the first and second register of the right neighbor, and so one until the last cell participating in the simulation is reached. The scanning continues with its third register and then with the third register of its left neighbor, and so on. Empty registers are ignored.

The 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 to move the entered symbols to the end of the store. Altogether, it obeys the following rules (cf. Fig. 1).

1. If the third register of a left neighbor is filled, a cell takes over the symbol from that register. The cell stores the symbol into its first free register, if possible. Otherwise, it stores the symbol into its own third register.
2. If the third register of its left neighbor is free, it marks its own third register as free.
3. 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.
4. If the second register of its left neighbor is filled and its own second register is free, then the cell takes the symbol from the first register of its right neighbor and stores it into its own second register.
5. Possibly, more than one of these actions are superimposed.

Now we are prepared to turn to an example.

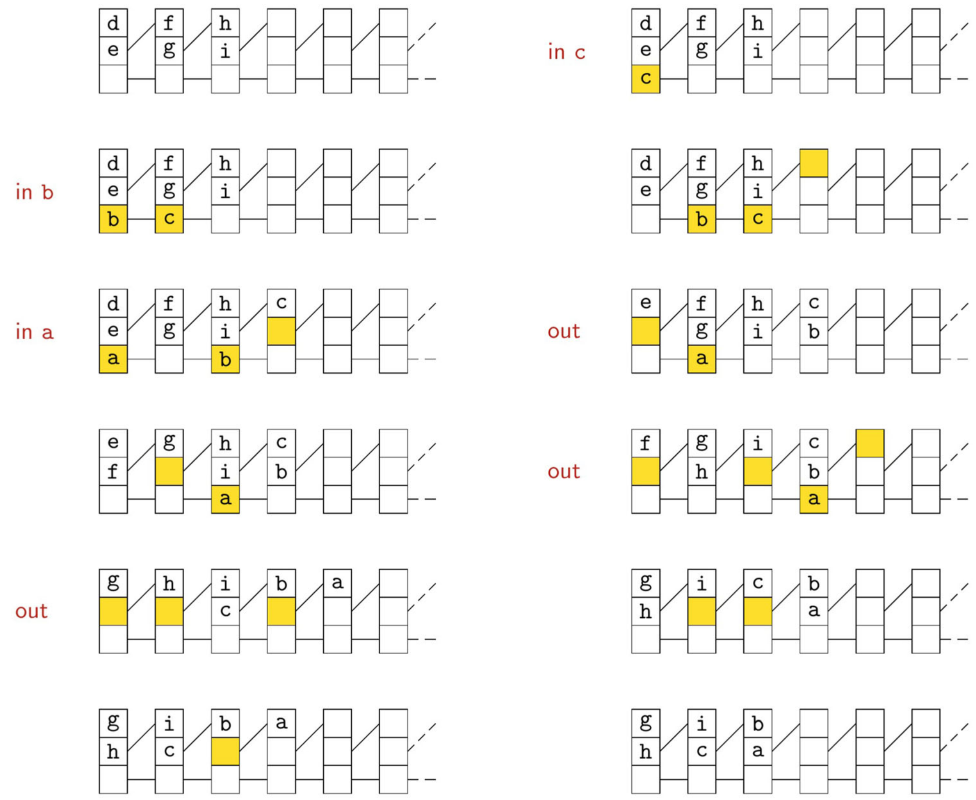


Fig. 1 Principle of a ring (queue) simulation. Subfigures are in row-major order

Example 1 We consider the Oldenburger–Kolakoski sequence (Kolakoski 1965; Oldenburger 1939) that is an infinite sequence over the alphabet $\{1, 2\}$. Basically, it is the sequence of run lengths in its own run-length encoding, where the sequence starts with 1 and consists of alternating blocks of 1's and 2's. The sequence is described as follows: The first symbol is 1. For $n \geq 1$, the nth symbol is the length of the nth block. So, the Oldenburger–Kolakoski sequence can be constructed by setting the first symbol to 1. The 1 means that the first block has length 1, that is the 1 itself. The next block starts with 2 which means that the second block has length 2. This gives 122 Next, the 2 at position three means that a block of length 2 has to be appended to the sequence, which yields 12211. Now the 1 at the fourth position means to append 2 and the 1 at the fifth position to append a further 1 yielding 1221121, and so on.

Since the nth run of the sequence is generated by its nth symbol, and the nth run generates the nth symbol, the Oldenburger–Kolakoski sequence is self-generating or self-describing. In fact, not much is known about the Oldenburger–Kolakoski sequence and several of its properties are non-trivial and open problems (Allouche and Shallit 2003).

Next, we want to construct a real-time CA M that will generate the pattern $P_{OK} = \{ p \mid p$ is prefix of the Oldenburger–Kolakoski sequence$\}$. In order to ensure a real-time generation, the CA M works with a real-time version of the Firing Squad Synchronization Problem (FSSP) based on the time optimal solution of Waksman (1966) on one of its tracks. The latter solution starts with one general at the left end of the array and it takes $n - 1$ time steps (n being the length of the array) to reach the right end. If we start instead with two generals at both ends, where the right general symmetrically behaves as the left general, we save $n - 1$ time steps. Since we need one additional time step to initialize the generals at both ends, we can realize the FSSP within $2n - 2 - (n - 1) + 1 = n$ time steps, that is, within real time.

So, the cells of the CA M consist of four registers forming four tracks. On the first track, the mentioned real-time version of the FSSP is executed. When the FSSP fires every cell enters instead a stable state 1 or 2. The second track is used for some control information. The third track (which is split into sub-tracks) is used to simulate a queue, whose front and end is attached to the rightmost cell of M. The content of the fourth track is shifted to the left in each time step. The rightmost cell feeds this track with the next symbol of the sequence to be generated.

In particular, in the very first time step, all cells split into the four tracks, the rightmost cell sets up the queue, feeds the fourth track with the first symbol 1 of the sequence, and enters 1 into the queue (see Fig. 2 for an example).

Subsequently, the rightmost cell behaves as follows. It maintains an internal counter that can count from one to

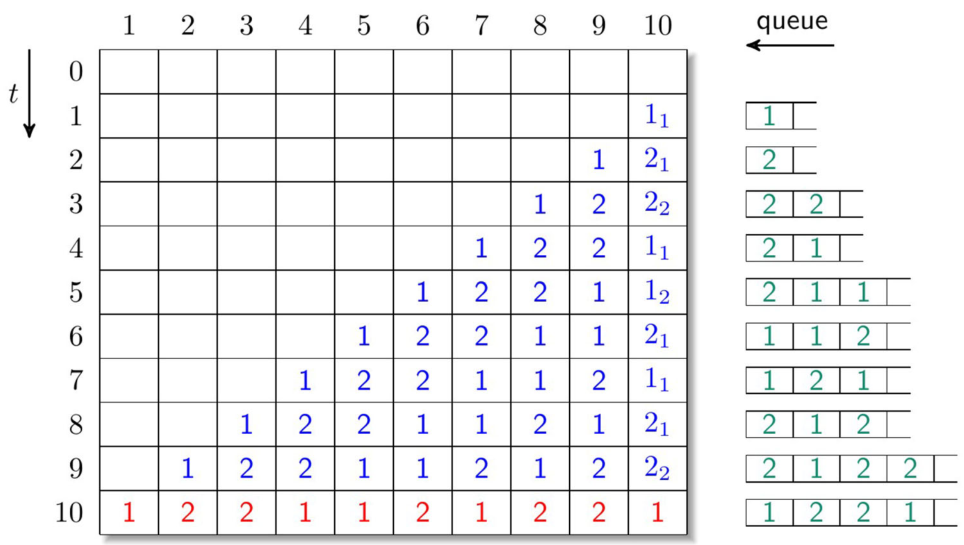


Fig. 2 Space-time diagram of the generation of the prefix 1221121221 of the Oldenburger–Kolakoski sequence. The fourth track and the counter of the rightmost cell are depicted. For the sake of readability, the queue on the third track is depicted separately to the right of the space-time diagram

two. The counter is set to one in the first time step. Whenever the counter coincides with the digit at the front of the queue, the rightmost cell feeds the fourth track with the first symbol of the next block of the sequence (1 or 2), sets its internal counter to one, deletes the front of the queue, and enters the symbol fed to the fourth track to the queue. Whenever the counter is less than the digit at the front of the queue, the rightmost cell feeds the fourth track with another symbol of the current block, increments its internal counter to two, keeps the front of the queue, and enters the symbol fed to the fourth track to the queue.

In this way, each position from left to right of the already generated sequence is interpreted, and, accordingly, the next block of length one or two is fed to the fourth track. This means, that the Oldenburger–Kolakoski sequence is successively fed to the fourth track, which is successively shifted to the left. At time step n when the FSSP fires, the current contents of the fourth track become stable states of the cells. Since the first symbol fed to the fourth track arrives exactly at time step n at the leftmost cell, the stable configuration reached at time step n is a prefix of the Oldenburger–Kolakoski sequence. □

3 Unary pattern generation, languages acceptance, and time constructibility

In order to explore the capabilities and properties of real-time cellular pattern generators, we start with unary words. At a first glance, to generate a unary pattern is trivial. In fact, this is true for patterns of the form $\{\, a^n \mid n \geq 1 \,\}$. In such cases it is sufficient that any cell enters state a in its first step and remains in this state. However, these patterns are total which means that the pattern contains a word for any $n \geq 1$. So, let us make the task a little harder. Now we define the pattern to be generated as a partial function on n. More precisely, we impose a condition on n such that the pattern a^n is to be generated if and only if n meets the condition. Let, for example, $\varphi : \mathbb{N} \to \mathbb{N}$ be some function. Then pattern P_φ is defined to be a^n if there is some m such that $n = \varphi(m)$, and undefined otherwise. Clearly, such a pattern can only be generated by a cellular automaton in real time if the function φ is constructible in some sense. If it is uncomputable, trivially P_φ cannot be generated. In order to fix the notion of constructibility the notion of time-constructibility is widely used.

In particular, a strictly increasing function $\varphi : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if there is a semi-infinite cellular automaton $M = \langle S, \Sigma, F, s_0, \#, \delta \rangle$, that is, n is infinite, whose leftmost cell is in some state of F at time $t \geq 0$ if and only if $t = \varphi(i)$ for some $i \geq 0$. The initial configuration of M is quiescent, that is, $c_0(i) = s_0$, $1 \leq i$.

The investigation of time-constructible functions in cellular automata originates in Fischer (1965), where a cellular automaton is constructed that time-constructs the function $i \mapsto p_i$ where p_i denotes the ith prime number. In Choffrut and Čulik (1984) a time-constructor for the function $i \mapsto 2^i$ is given. The systematic study of this concept was started in Mazoyer and Terrier (1999). The family of time-constructible functions is denoted by $\mathscr{F}(\mathrm{CA})$.

As a simple example, in Fig. 3 the time-construction of the function $i \mapsto i^2$ is given. Basically, the necessary signals can be derived from $(i+1)^2 = i^2 + 2i + 1$. In particular, after being designated at time i^2, the leftmost cell has to wait for $2i$ time steps before it is designated again at time $i^2 + 2i + 1$. This delay is exactly the time needed by an auxiliary signal α that moves from the leftmost cell 1 to cell

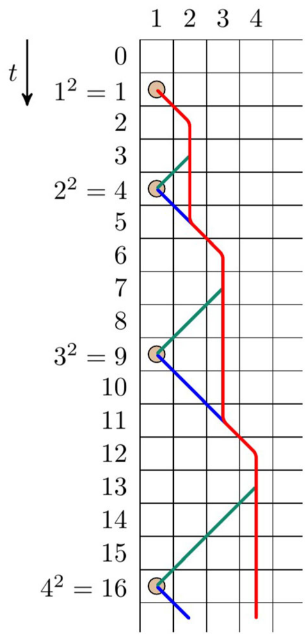


Fig. 3 Time construction of the function i^2. Signal α is depicted in blue and green, signal β is depicted in red. (Color figure online)

$i + 1$, stays there for one time step, and moves back to the leftmost cell. To this end, another auxiliary signal β is used that stays in cell i until it is hit by α and moves to cell $i + 1$.

Let us come back to the pattern P_φ, where φ is a time-constructible function. Since the pattern is undefined for lengths n that are not in the range of φ, the pattern is easily generated by a CA.

Proposition 2 *Let $\varphi : \mathbb{N} \to \mathbb{N}$ be a time-constructible function. Then the pattern*

$$P_\varphi = \{\, a^n \mid \text{ there is an } m \text{ with } n = \varphi(m)\,\}$$

is generated by some real-time CA.

Proof A real-time CA that generates P_φ essentially simulates a time constructor for φ. In addition, its rightmost cell initially sends a signal with maximal speed to the left. Each cell passed through by this signal enters state a and remains in it. The signal arrives at the leftmost cell at time n. If this cell is simultaneously distinguished by the time construction, the number of cells n is in the range of φ. In this case the leftmost cell enters state a and remains in it. This means that the pattern a^n is generated in real time. If at arrival of the signal the leftmost cell is not distinguished by the time construction, it starts to enter alternating some states b and c. So, the configuration will never be stable and no pattern is generated in this case. □

The simple construction of the generator of P_φ works fine since, in principle, only the leftmost cell decides whether a pattern has to be generated or not. Moreover, the other cells can safely enter the pattern states in advance without violating the overall result. So, let us make the task again a little harder. Now we define the pattern to be generated as a total function on n and impose a condition on n such that the pattern a^n is to be generated if n meets the condition, but the pattern b^n otherwise. In this case, the leftmost cell can still decide which pattern has to be generated, but all the other cells cannot enter the pattern state in advance. Instead, they have to know whether the condition is met or not. Since an FSSP synchronization of the array (starting at both ends simultaneously) cannot be done in less than $n - 1$ steps, for a real-time generation there is not enough time for the leftmost cell to inform the other cells about whether or not the condition is met. For a time-constructible function $\varphi : \mathbb{N} \to \mathbb{N}$, we define the pattern $\hat{P}_\varphi$ to be a^n if there is some m such that $n = \varphi(m)$, and b^n otherwise. The next theorem shows that even these unary patterns are generated in real time.

Theorem 3 *Let $\varphi : \mathbb{N} \to \mathbb{N}$ be a time-constructible function. Then the pattern*

$$\hat{P}_\varphi = \{\, x^n \mid x = a \text{ if there is an } m \text{ with } n = \varphi(m) \text{ and } x = b \text{ otherwise}\,\}$$

is generated by some real-time CA.

Proof The basic idea of a real-time CA M that generates $\hat{P}_\varphi$ is as follows. In order to check the condition on the length of the input, a time-constructor for φ is simulated. In order to gain enough time to synchronize the cells, it is sped-up by a factor of two. This is done by grouping two cells of the time-construction into one cell. So, the leftmost cell now simulates cell 1 and cell 2 of the original time-constructor. More precisely, if originally the leftmost cell is distinguished at an even time step i, now it is distinguished at time step $i/2$ by the simulation of the original cell 1. If originally the leftmost cell is distinguished at an odd time step i, now it is distinguished at time step $(i - 1)/2$ by the simulation of the original cell 2.

Note that all computations apart from the compressed simulation of the time-constructor are done without grouping the cells.

Next, the compressed simulation of the time-constructor is expanded again (see Fig. 4). To this end, a signal α with speed 1/3 is initially sent to the right by the leftmost cell.

Case 1 Whenever the leftmost cell of the compressed simulation is distinguished at time step $i/2$ by the simulation of the original cell 1, that is, i is even, then it sends a signal β to the right. When this signal meets α, it follows α for one time step and runs back to the left. It arrives at the leftmost cell at time i again. This is inductively seen as follows.

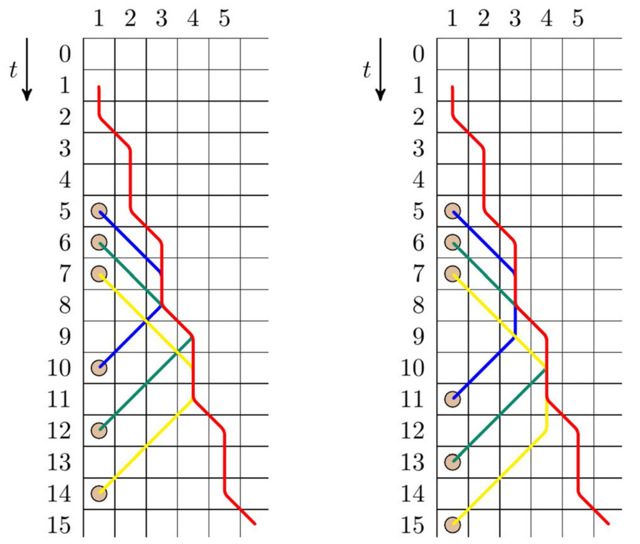


Fig. 4 Principles of designating time $2i$ (left) and $2i+1$ (right) starting at time i. The signal α with speed 1/3 is depicted in red. The three computations starting at time steps 5, 6, and 7 are depicted in blue, green, and yellow. (Color figure online)

Case 1a Let $j = i/2$ be even. Then β meets α in cell $j/2$ at time $j + j/2 - 1$. This is true for $j = 2$ in cell 1 at time 2. Now assume it is true for some even j. Since j is even, $j + j/2 - 1$ is congruent 2 modulo 3. That is, signal α moves to cell $j/2 + 1$ at time $j + j/2$ and stays there until time $j + j/2 + 2$. Signal β is sent by the leftmost cell at time $j + 2$ and arrives in cell $j/2 + 1 = (j + 2)/2$ at time $j + 2 + j/2 + 1 - 1 = j + j/2 + 2 = (j + 2) + (j + 2)/2 - 1$.

Next, when signal β meets α it follows α for one time step, that is, when sent at time j it enters cell $j/2 + 1$ at time $j + j/2$, and moves back to the leftmost cell, where it arrives at time $j + j/2 + j/2 = 2j = i$ as claimed.

Case 1b Now let $j = i/2$ be odd. Then β meets α in cell $(j + 1)/2$ at time $j + (j - 1)/2$. This is true for $j = 1$ in cell 1 at time 1. Now assume it is true for some odd j. Since j is odd, $j + (j - 1)/2$ is congruent 1 modulo 3. That is, signal α moves to cell $(j + 1)/2 + 1$ at time $j + (j - 1)/2 + 2$ and stays there until time $j + (j - 1)/2 + 4$. Signal β sent by the leftmost cell at time $j + 2$ arrives in cell $(j + 1)/2 + 1 = (j + 2 + 1)/2$ at time $j + 2 + (j + 2 + 1)/2 - 1 = (j + 2) + (j + 1)/2$.

Next, when signal β meets α it follows α for one time step, that is, stays in cell $(j + 1)/2$ at time $j + (j - 1)/2 + 1$, and moves back to the leftmost cell, where it arrives at time $j + (j - 1)/2 + 1 + (j + 1)/2 - 1 = 2j = i$ as claimed.

Case 2 Whenever the leftmost cell of the compressed simulation is distinguished at time step $(i - 1)/2$ by the simulation of the original cell 2, that is, i is odd, then it sends a signal γ to the right. When this signal meets α, it follows α for one time step, stays in that cell for another time step, and runs back to the left. Case 2 is basically the same as Case 1 with the exception that the signal which bounces at α is delayed for one time step. So, it arrives at the leftmost cell one time step later than in Case 1, that is, at time $2(i - 1)/2 + 1 = i$.

Now we continue the construction of the real-time CA that generates $\hat{P}_\varphi$ (see Fig. 5 for an example). In addition to the simulation of the time-constructor, the rightmost cell initially sends a signal τ with maximal speed to the left. If the number of cells n is not congruent 3 modulo 4, this signal meets signal α of the time constructor in cell $\lfloor n/4 \rfloor + 1$. At this point it can be determined if some signal β or γ of the time constructor joins τ on its way to the leftmost cell. Since τ arrives at the leftmost cell at time n and β or γ distinguishes the leftmost cell, exactly in this case the pattern a^n has to be generated, and b^n otherwise. So, cell $\lfloor n/4 \rfloor + 1$ can send this information to the right and to the left to cause all cells reached to enter the correct pattern state. In this way, the cells 1 to $2\lfloor n/4 \rfloor + 1$ are reached in the remaining $\lfloor n/4 \rfloor$ time steps. Since n is not congruent 3 modulo 4 we have $2\lfloor n/4 \rfloor + 1 \geq \lceil n/2 \rceil$. So, the left half of the array generates the correct pattern. In order to achieve the same for the right half it is sufficient, additionally to implement the whole procedure symmetrically on the right end of the array.

Finally, the case where n is congruent 3 modulo 4 has to be considered. In this case, the signals α and τ meet in the adjacent cells $\lfloor n/4 \rfloor + 1$ and $\lfloor n/4 \rfloor + 2$. So, these cells can send the information which pattern is to be generated one time step after the meeting, for which then $\lfloor n/4 \rfloor$ time steps are left. This means that also in this case the cells 1 to $2\lfloor n/4 \rfloor + 2 \geq \lceil n/2 \rceil$ will receive this information in due time. □

The family of time-constructible functions $\mathscr{F}(\mathrm{CA})$ is very rich. Several examples and the closure of the family under a bunch of operations are shown in Mazoyer and Terrier (1999). This fact transfers immediately to the family of unary patterns of the form $\hat{P}_\varphi$ generated by cellular automata in real time. Moreover, in Mazoyer and Terrier (1999) the following relation between time-constructible functions and unary languages is shown: a function $\varphi : \mathbb{N} \to \mathbb{N}$ is time constructible if and only if the language $L_\varphi = \{ a^{\varphi(m)} \mid m \geq 1 \}$ is accepted by a real-time cellular automaton.

Proposition 4 *Let $\varphi : \mathbb{N} \to \mathbb{N}$ be a function and $\hat{P}_\varphi$ be generated by some cellular automaton in real time. Then language $L_\varphi = \{ a^{\varphi(m)} \mid m \geq 1 \}$ is accepted by a real-time cellular automaton.*

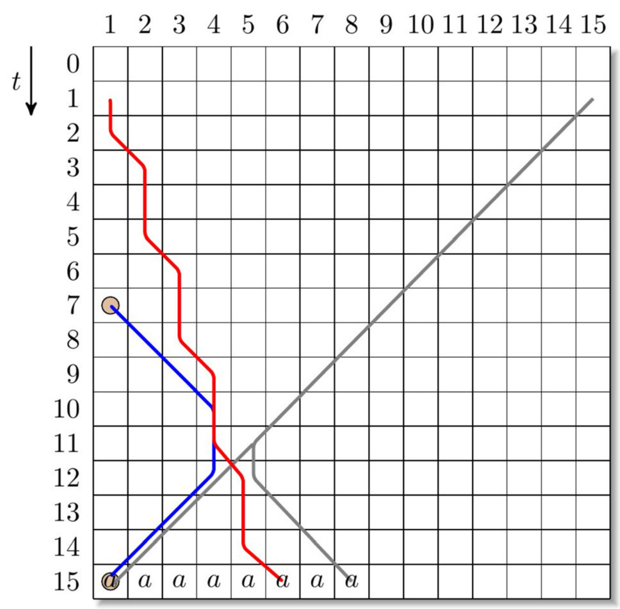


Fig. 5 Example of a $\hat{P}_\varphi$ generator, where $a^{15} \in \hat{P}_\varphi$. The signal α with speed 1/3 is depicted in red, signal γ that designates time 15 is depicted in blue, and signal τ is depicted in gray. Since $15 \equiv 3 (\mathrm{mod}\ 4)$, the signals α and τ meet in the adjacent cells 4 and 5. (Color figure online)

Now by Theorem 3, Proposition 4, and the result from (Mazoyer and Terrier 1999), we have shown that for unary languages/patterns the three different notions of language acceptance, time-constructibility, and pattern generation, in fact, coincide.

Theorem 5 *A function $\varphi : \mathbb{N} \to \mathbb{N}$ is time constructible if and only if the language L_φ is accepted by a real-time cellular automaton if and only if the pattern $\hat{P}_\varphi$ can be generated by a cellular automaton in real time.*

4 Generating non-unary patterns

In this section, we consider non-unary patterns and identify some classes of non-unary patterns that can be generated by cellular automata in real time. Since the cellular automata work deterministically, it is clear that for every initial length of n cells at most one pattern of length n is generated. Patterns or languages having this property of containing at most one word of length n for every $n \geq 1$ are called *properly thin* in Păun and Salomaa (1995).

Since every pattern generated by a cellular automaton is a properly thin language, it is an interesting question which classes of properly thin languages can be generated by real-time cellular automata. Here, we can identify two large classes of such properly thin languages. First, we consider in Sect. 4.1 finite, regular, and context-free properly thin languages and eventually show that every context-free properly thin language is generated by a cellular automaton in real time. Second, we consider in Sect. 4.2 non-unary patterns which are defined by the set of all prefixes of an infinite sequence. For example, the prefixes of the Oldenburger–Kolakoski sequence from Example 1 or the prefixes of the Thue-Morse sequence. Clearly, every such pattern is properly thin. We present a construction of a real-time cellular automaton that generates the prefixes of every given *automatic sequence* (Allouche and Shallit 2003).

4.1 Generating context-free properly thin languages

We start with some definitions. A language is called *properly thin*, if it contains at most one word of every length. It is called *properly k-thin*, if it contains at most k words of every length for a fixed $k \geq 1$. A language is called *slender*, if it is properly k-thin for some $k \geq 1$. If L is properly thin, then $|\{w \in L \mid |w| = n\}| \leq 1$ for all $n \geq 1$. It should be noted that thin languages and generalizations are studied in detail in Ilie (1994), Păun and Salomaa (1995) and that there is some related older literature (Kunze et al. 1981; Latteux and Thierrin 1983) in which thin languages are studied under the name of semi-discrete languages.

To obtain our main result we first show how to generate every finite properly thin language.

Theorem 6 *Every finite properly thin language can be generated by a cellular automaton in real time.*

Proof Let L be a finite properly thin language. Then, $L = \{ u_1, u_2, \ldots, u_m \}$ for some $m \geq 1$ and $|u_i| \neq |u_j|$ for all $1 \leq i < j \leq m$. Hence, the basic idea for a CA generating L is to count the number ℓ of initial cells and to generate string u_i, if there is some u_i of length ℓ. Otherwise, nothing is generated by entering an instable configuration. To be more precise, let $t = \max\{ |u_i| \mid 1 \leq i \leq m \}$. Then, the leftmost cell starts a signal with maximum speed to the right counting from 1 to $\lceil t/2 \rceil$, whereas the rightmost cell starts a signal with maximum speed to the left counting from 1 to $\lceil t/2 \rceil$ as well. If the counting would be larger than $\lceil t/2 \rceil$, then nothing has to be generated. This is realized by stopping both signals and entering a new state p that alternates with some other new state q and creates an instable configuration. If both signals meet in the middle, then the number of initial cells, say ℓ, is known. Since there are only finitely many strings in L, it can be checked using the state set whether there exists some u_i with $|u_i| = \ell$. If so, the signal to the left continues and generates the first half of u_i, whereas the signal to the right continues and generates the second half of u_i. If such an u_i does not exist, state p alternating with state q is entered and an instable configuration is created. An example computation illustrating the construction is given in Fig. 6. □

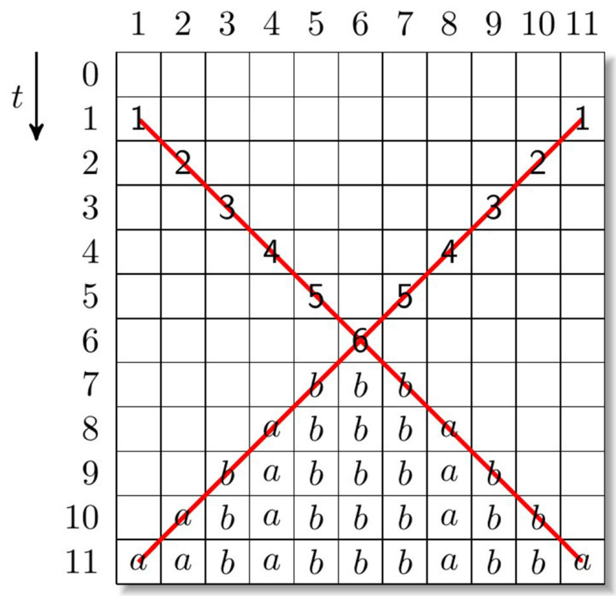


Fig. 6 Example of generating the string $w = aababbbabba$ of length 11. At time step 6 the number of initial cells is known since both signals meet. Hence, these signals can be used to generate the first and the second half of string w

The next step is to generalize the result from finite properly thin languages to regular properly thin languages.

Theorem 7 *Every regular properly thin language can be generated by a cellular automaton in real time.*

Proof According to Păun and Salomaa (1995) every slender regular language L can be represented as the union $L = \bigcup_{i=1}^{k} L_i$ of pairwise disjoint sets $L_i = \{\, u_i v_i^n w_i \mid n \geq 0 \,\}$ with strings u_i, v_i, w_i, for $1 \leq i \leq k$. This implies in particular for regular properly thin languages L that every $w \in L$ is contained in exactly one set L_i with $1 \leq i \leq k$. Hence, L can be divided into a "finite" part and an "infinite part," namely, $L = L' \cup \bigcup_{i=1}^{k} L'_i$ with $L' = \{u_1 w_1, u_2 w_2, \ldots, u_k w_k\}$ and $L'_i = \{\, u_i v_i^n w_i \mid n \geq 1 \,\}$, if $v_i \neq \lambda$, and $L'_i = \emptyset$, otherwise. Again, every $w \in L$ is contained in exactly one set from L' and L'_i with $1 \leq i \leq k$. The basic idea to generate a word from L is again to count the number ℓ of initial cells. In case of words from the finite part L', we check this in one track, say track 0, in the same way as in the construction given in the proof of Theorem 6. If the check is successful, stable states representing the word from L' are entered. In case of words from the infinite part, it has to be checked whether there is some $1 \leq i \leq k$ and some integer $n \geq 1$ such that $\ell = |u_i w_i| + n \cdot |v_i|$. This is realized in k additional tracks whereby in track i the check whether $\ell = |u_i w_i| + n \cdot |v_i|$ $(1 \leq i \leq k)$ is performed. If $L'_i = \emptyset$, nothing has to be checked. Thus, we may assume that $L'_i \neq \emptyset$. In this case, we first have to consider u_i and w_i. Let $t = \max\{|u_i|, |w_i|\}$. Then, we start in the leftmost cell a signal R_1 to the right which moves $|u_i|$ cells in t time steps. Similarly, we start in the rightmost cell a signal L_1 to the left which moves $|w_i|$ cells in t time steps. At time step $t+1$ we start in cell $|u_i|+1$ a signal R_2 with maximum speed to the right which counts modulo $|v_i|$. Additionally, we start in the $(|w_i|+1)$st cell from the right a signal L_2 with maximum speed to the left which counts modulo $|v_i|$ as well.

If both signals meet correctly, then we know that $\ell - |u_i w_i|$ is divisible by $|v_i|$. Hence, there is an $n \geq 1$ such that $\ell = |u_i w_i| + n \cdot |v_i|$. Then, the signal to the left continues and generates, using stable states, the first half of v_i. To this end, it generates v_i possibly multiple times as long as cells passed through by R_2 are visited, and finally generates u_i as long as cells passed through by R_1 are visited. Similarly, the signal to the right continues and generates the second half of v_i. If both signals meet incorrectly, then there is no $n \geq 1$ such that $\ell = |u_i w_i| + n \cdot |v_i|$. In this case, an instable configuration is created by entering a new state p that alternates with another new state q.

Since every $w \in L$ is contained in exactly one set from L' and L'_i with $1 \leq i \leq k$, there is at most one track in which a check is successful and in which stable states are entered. Hence, these stable states overwrite all tracks and produce the word from L to be generated. An example computation illustrating the construction is given in Fig. 7.

The time needed for the construction described so far is bounded by $\ell + \max\{|u_1|, |w_1|, |u_2|, |w_2|, \ldots, |u_k|, |w_k|\}$ which is real time plus a constant. By using the speed-up result for cellular string generators shown in Kutrib and Malcher (2021) we obtain that our construction can be sped-up to work in real time. □

Now, we have all prerequisites and can show the main result of this subsection.

Theorem 8 *Every context-free properly thin language can be generated by a cellular automaton in real time.*

Proof According to Ilie (1994) every slender context-free language L can be represented as the union $L = \bigcup_{i=1}^{k} L_i$ of pairwise disjoint sets $L_i = \{\, u_i v_i^n w_i x_i^n y_i \mid n \geq 0 \,\}$ with strings u_i, v_i, w_i, x_i, y_i for $1 \leq i \leq k$. This implies for context-free properly thin languages L that every $w \in L$ is contained in exactly one set L_i with $1 \leq i \leq k$.

Hence, L can be divided into a "regular" part L' and a "non-regular" part L'', namely, $L = L' \cup L''$ with $L' = \{u_1 w_1 y_1, u_2 w_2 y_2, \ldots, u_k w_k y_k\} \cup \bigcup_{i=1}^{k} L'_i$ and $L'_i = \{\, u_i v_i^n w_i x_i^n y_i \mid n \geq 1 \,\}$, if either $v_i = \lambda$ and $x_i \neq \lambda$ or $v_i \neq \lambda$ and $x_i = \lambda$. Otherwise, $L'_i = \emptyset$. The non-regular part L'' is defined as $L'' = \bigcup_{i=1}^{k} L''_i$ with $L''_i = \{\, u_i v_i^n w_i x_i^n y_i \mid n \geq 1 \,\}$, if $v_i \neq \lambda$

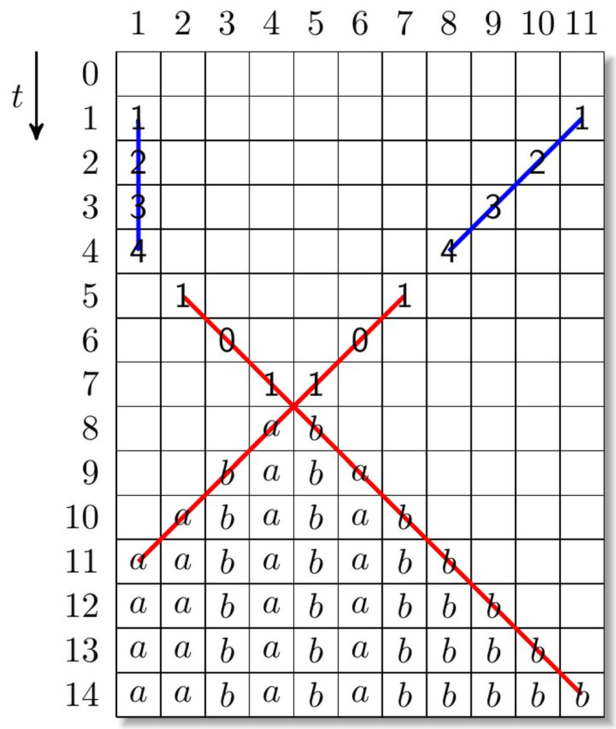


Fig. 7 Example of generating the string $w = u_1 v_1^3 w_1 = aababababbbbb$ of length 11 with $u_1 = a$, $v_1 = ab$, and $w_1 = bbbb$. At time step 1 the signals L_1 and R_1 (blue) are started. At time step $\max\{|u_1|, |w_1|\} + 1 = 5$ the signals L_2 and R_2 (red) are started which meet correctly at time step 7. Subsequently, L_2 continues to generate the prefix *aaba* and R_2 continues to generate the suffix *babbbbb*. (Color figure online)

and $x_i \neq \lambda$, and $L_i'' = \emptyset$, otherwise. Again, every $w \in L$ is contained in exactly one set from L' and L''. The basic idea to generate a word from L is again to count the number ℓ of initial cells. In case of words from the regular part L', we check this in a finite number of tracks in the same way as in the construction given in the proof of Theorem 7. If the check is successful in one track, stable states representing the word from L' are entered. In case of words from the non-regular part, it has to be checked whether there is some $1 \leq i \leq k$ and some integer $n \geq 1$ such that $\ell = |u_i w_i y_i| + n \cdot |v_i x_i|$. This is realized in k additional tracks whereby in track i the check whether $\ell = |u_i w_i y_i| + n \cdot |v_i x_i|$ is performed. If $L_i'' = \emptyset$, nothing has to be checked. Thus, we may assume that $L_i'' \neq \emptyset$.

In the following construction we will use signals to subtract $|u_i|$ and $|w_i y_i|$ from ℓ. Then, it will be tested whether $\ell - |u_i w_i y_i|$ is divisible by $|v_i x_i|$ which can be realized by a signal counting modulo $|v_i x_i|$. If so, there is an $n \geq 1$ such that $\ell = |u_i w_i y_i| + n \cdot |v_i x_i|$ and w has to be generated. If not, there is no such n and alternating states leading to an instable configuration are entered in the track.

To be more detailed, let $t = \max\{|u_i|, |w_i y_i|\}$. Then, we start in the leftmost cell a signal R_1 to the right which moves $|u_i|$ cells in t time steps, whereby the first $|u_i|$ cells are suitably marked. Similarly, we start in the rightmost cell a signal L_1 to the left which moves $|w_i y_i|$ cells in t time steps, whereby the last $|y_i|$ cells are suitably marked. At time step $t+1$ we start in cell $|u_i| + 1$ a signal R_2 with maximum speed to the right which counts modulo $|v_i x_i|$. Additionally, we start in the $(|w_i y_i| + 1)$st cell from the right a signal L_2 with maximum speed to the left which counts modulo $|v_i x_i|$ as well. If both signals meet correctly, we have found an n such that $\ell = |u_i w_i y_i| + n \cdot |v_i x_i|$. Both signals continue to the left and right, respectively, informing the remaining cells that a word $w \in L$ is to be generated.

For the generation of w we already know due to signals L_1 and R_1 in which cells u_i and y_i have to be generated. For the generation of the infix $v_i^n w_i x_i^n$ we assume for a moment that $w_i = \lambda$. Then, we additionally start at time step $t+1$ in cell $|u_i| + 1$ a signal R_3 with speed $|v_i|/|v_i x_i|$ to the right, and we also start in the $(|w_i y_i| + 1)$st cell from the right a signal L_3 with speed $|v_i|/|v_i x_i|$ to the left. If both signals meet correctly, the cells touched by R_3 should generate v_i^n and the cells touched by L_3 should generate x_i^n. However, if $w_i \neq \lambda$, then w_i should be inserted in between v_i^n and x_i^n which can in principle be realized by shifting the contents of the cells touched by L_3 $|w_i|$ cells to the right while generating w_i. Observe that signal L_1 has left free $|w_i|$ cells after marking the cells containing y_i. Hence, there are sufficient free cells into which the contents of the cells touched by L_3 can be shifted while generating w_i. To synchronize the cells to be shifted we start at time step $t+1$ in an additional subtrack an instance of the FSSP with initial generals at both ends that synchronizes the array at time step $t + \ell$, and the right shift can be done within $|w_i|$ additional time steps. Moreover, it can be checked whether all cells have been touched by signals L_2 and R_2 which gives the information whether a word is indeed to be generated. Only in this case, all cells enter a stable state at time step $t + \ell + |w_i| + 1$. Hence, the word w is generated. Again, since every $w \in L$ is contained in exactly one set from L' and L_i'' with $1 \leq i \leq k$, there is at most one track in which a check is successful and in which stable states are entered. Hence, these stable states overwrite all tracks and produce the word from L to be generated. An example computation illustrating the construction is given in Fig. 8.

The time needed for the construction can be calculated as follows. We need t time steps for signals L_1 and R_1. We need at most ℓ additional time steps for signals L_2, L_3, R_2, R_3, and the FSSP. Finally, $|w_i| + 1$ time steps are needed for the right shift and the generation step. Altogether, the time needed is bounded by $\ell + t + |w_i| + 1 \leq \ell + 2 \cdot \max\{|u_1|, |w_1 y_1|, |u_2|, |w_2 y_2|, \ldots, |u_k|, |w_k y_k|\} + 1$ which is real time plus a constant. By using again the speed-up result for cellular string generators we obtain that our construction can be sped-up to work in real time. □

4.2 Generating automatic sequences

Now, we turn to non-unary patterns whose elements are prefixes of infinite sequences of symbols. Example 1 already showed how the prefixes of the infinite Oldenburger–Kolakoski sequence are generated. We are going to consider a wide class of sequences, that is, we consider *automatic sequences* (Allouche and Shallit 2003). So, what is an automatic sequence? It may be defined in a number of

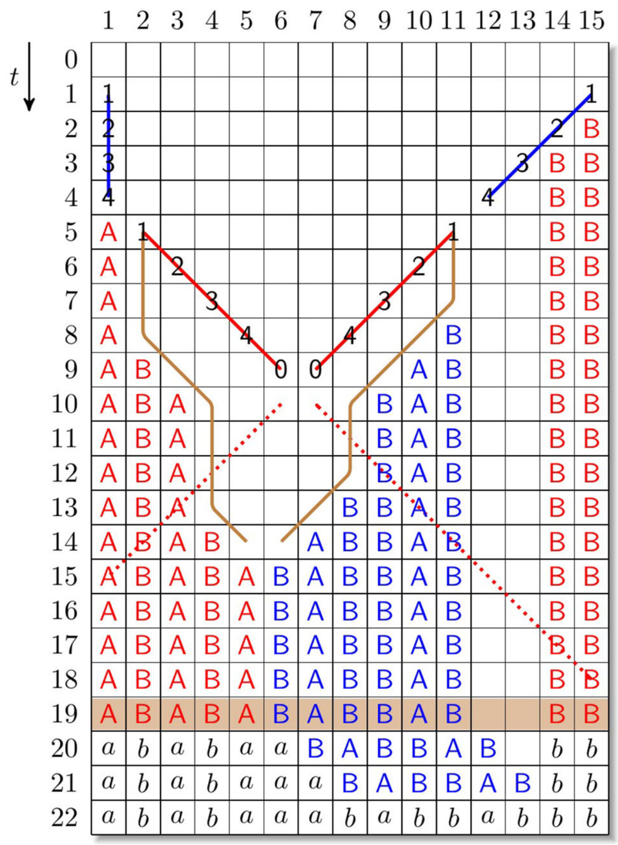


Fig. 8 Example of generating the string $w = u_1 v_1^2 w_1 x_1^2 y_1 = ababaaababbabbb$ of length 15 with $u_1 = a$, $v_1 = ba$, $w_1 = aa$, $x_1 = bab$, and $y_1 = bb$. At time step 1 the signals L_1 and R_1 (blue) are started which mark the cells carrying u_1 and y_1. At time step $\max\{|u_1|, |w_1 y_1|\} + 1 = 5$ the signals L_2 and R_2 (red) counting modulo $|v_1 x_1| = 5$ are started which meet correctly at time step 9. Subsequently, L_2 and R_2 continue (red dotted lines) to inform all cells that a word from L is to be generated. In addition, at time step 5 the signal R_3 (brown) with speed 2/5 and signal L_3 (brown) with speed 3/5 are started which meet correctly at time step 14 having marked cells carrying v_1^n and x_1^n. The FSSP fires at time step 19. Hence, the cells carrying u_1, v_1^n, and y_1 enter stable states while the cells carrying x_1^n are shifted $|w_1| = 2$ cells to the right and generate w_1. Eventually, at time step 22 all cells have entered a stable state. (Color figure online)

ways. For our purposes the automata-theoretic definition is well suited.

Let $k \geq 1$ be a positive integer and $A = \langle S, \Sigma_k, U, s_0, \delta, \tau \rangle$ be a deterministic finite automaton with output (Moore automaton), where S is the finite, nonempty set of *cell states*, $\Sigma_k = \{0, 1, \ldots, k-1\}$ is the set of *input symbols*, $s_0 \in S$ is the *initial state*, $\delta : S \times \Sigma_k \to S$ is the *transition function*, U is the set of *output symbols*, and $\tau : S \to U$ is the *output function*.

The automaton defines a *k-automatic sequence* $(u_m)_{m \geq 0}$ as follows. For each $m \geq 0$, the automaton is fed with the k-ary expansion of m (the unary expansion is 1^m). After processing this input word the automaton reaches one of its states, whose image under the output function τ gives the mth element of the sequence. Basically, the automaton can be fed with the k-ary expansions starting with the most significant digit or starting with the least significant digit. In the former case the automaton is said to be *direct reading* and in the latter case it is *reverse reading* (Allouche and Shallit 2003). However, since every automatic sequence that is defined by a direct reading automaton can be defined by a reverse reading automaton and vice versa, we safely may assume reverse reading automata in the sequel.

Many well-known infinite sequences with applications in numerous fields of mathematics and non-trivial properties are automatic. The next example mentions three of them (see (Allouche and Shallit 2003) for further details).

Example 9 The *Thue-Morse sequence* is a 2-automatic sequence over the alphabet $\{0, 1\}$. There a several ways of generating the Thue-Morse sequence one of which is given by a Lindenmayer system with axiom 0 and rewriting rules $0 \to 01$ and $1 \to 10$. The generation of strings can be described as follows: starting with the axiom every symbol 0 (symbol 1) is in parallel replaced by the string 01 (10). This procedure is iteratively applied to the resulting strings. The first steps of this procedure yield the strings 0, 01, 0110, 01101001, and so on. So, the first terms of the sequence are 0, 1, 1, 0, 1, 0, 0, 1.

The *Rudin-Shapiro sequence* is a 2-automatic sequence over $\{1, -1\}$. Let $e(m)$ be the number of (possibly overlapping) occurrences of 11 in the binary expansion of m. Then the Rudin-Shapiro sequence $(u_m)_{m \geq 0}$ can be defined by the formula $u_m = (-1)^{e(m)}$. The first terms of the sequence are $1, 1, 1, -1, 1, 1, -1, 1, 1, 1, 1, -1, -1, -1, 1, -1$.

The *Regular Paperfolding sequence* takes its name from the following fact. Assume that a strip of paper is folded repeatedly in half in the same direction. If each fold is now opened out to create a right-angled corner then a right turn is represented by 1 and a left turn by a 0. The first terms of the sequence are 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0. The *Regular Paperfolding sequence* is 2-automatic. □

We now turn to show that the prefixes of automatic sequences are patterns generated by cellular automata in real time. The case $k = 1$ is often omitted in the definition of automatic sequences. However, it is treated separately in Allouche and Shallit (2003). So, we treat it separately here as well.

Lemma 10 *Let* $u = (u_m)_{m \geq 0}$ *be a* 1*-automatic sequence. Then the pattern*

$P_u = \{\, p \mid p \text{ is prefix of } u \,\}$

can be generated by a cellular automaton in real time.

Proof Recall that the cells of a cellular automaton are numbered from 1 to n, whereas the first term of the automatic sequence is u_0. So, the prefix to be generated is $u_0 u_1 \cdots u_{n-1}$.

Let A be the deterministic finite automaton that defines the 1-automatic sequence u. Then the state reached by A after processing the input 1^n has to be mapped to the nth term of the sequence.

Basically, the simple idea of M is to start a signal at the leftmost cell at time step 1 that moves with unit speed to the right. The signal simulates the finite automaton A along its way assuming the input symbols are 1. Since the initial state of A has to be mapped to u_0, the leftmost cell of M can simply apply the output function τ of A to enter the stable state u_0. Then cell 2 of M simulates the successor state of A under input 1. Again, it can apply the output function τ of A to enter the stable state u_1. When the signal arrives at the rightmost cell of M, it simulates the state of A after processing the input 1^{n-1}. So, after applying the output function τ of A it enters the stable state u_{n-1}. □

Next we consider the general case, where k may be an arbitrary positive integer.

Theorem 11 *Let* $k \geq 1$ *be a positive integer and* $u = (u_m)_{m \geq 0}$ *be a* k*-automatic sequence. Then the pattern* $P_u = \{\, p \mid p \text{ is prefix of } u \,\}$ *can be generated by a cellular automaton in real time.*

Proof The case $k = 1$ has been shown in Lemma 10. So, in the rest of the proof we assume $k \geq 2$.

We will construct a real-time CA M that generates the pattern P_u. The cellular automaton M performs several tasks in parallel on different tracks. We present the tasks one by one. Let $A = \langle S, \Sigma_k, U, s_0, \delta, \tau \rangle$ be the deterministic finite automaton that defines the k-automatic sequence u.

Task 1 is already known from Example 1. In order to ensure a real-time generation, the CA M establishes an instance of the FSSP with initial generals at both ends, that synchronizes the array at time step n in real time. If the FSSP is about to fire, all cells enter their stable states from the set U corresponding to their position in the sequence. In order to know what states these are, essentially, two further tasks are performed.

Task 2 is to realize a k-ary counter. The counter is initially set up at the leftmost cell, that is cell 1. In each time step, it is incremented by one and moves one cell to the right with the least significant digit in front. The digits (partial states) used for the counter are $\{{}^{+}0, 0, 1, \ldots, k-1\}$, where ${}^{+}0$ indicates digit 0 when a carry-over occurred. Since the counter is moving, a cell carrying such a carry-over can process it in the next step when it receives the next, more significant digit. Figure 9 shows the first few steps of a binary counter ($k = 2$).

Now we can see that the counter digits that are flowing through some cell x are representing exactly the number $x - 1$ in base k. Evidently, this is true for cell 1 through which the counter 0 flows. In Fig. 9, exemplarily, the green frames show that the counter flows through cell 3 entirely with the digits 0 followed by 1. This means 2 in base 2. Similarly, the green frame of cell 5 contains the digits 0, 0, and 1. This means 4 in base 2. So, each cell x has simply to simulate the deterministic finite automaton A on the input provided by the counter digits in order to figure out the

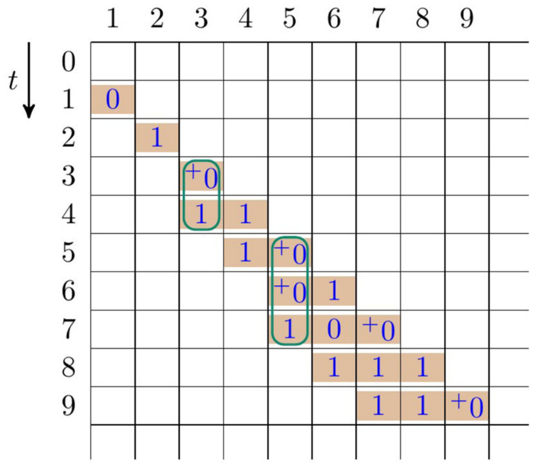


Fig. 9 Example of a moving binary counter that is incremented by one in each step. The cells carrying digits of the counter are shaded. For example, when cell 3 sees the incoming 1 from the left at time 2, it increments the counter by 1 resulting in digit 0. This incrementation generates a carry-over that is indicated by the plus in the new state ${}^{+}0$. In the next step cell 4 sees an incoming 0 and increments it to 1 without carry-over. So, the state of cell 4 at time 4 is 1. Cell 3, however was in state ${}^{+}0$ and, thus, knows that the carry-over indicated by the plus has still to be processed. Since it sees that the next incoming digit does not exist, that is, it is treated as 0, it processes the carry-over by incrementing the 0 to 1, which is the new state of the cell. The green frames show exemplarily that cell 3 is flowed through by the digits 0 followed by 1, that is the binary expansion of 2, and cell 5 is flowed through by the digits 0, 0, and 1, that is the binary expansion of 4

state of A reached by input of the k-ary expansion of $x-1$. Since each cell can trivially apply the function τ to this state, each cell x that is entirely flown through by the counter knows the symbol u_{x-1} to which it has to enter upon firing of the FSSP. However, this works fine only for cells that are entirely flown through by the counter. More precisely, it does not apply for the last $\lceil \log n \rceil$ cells of the array. To cope with these cells we have to consider the most sophisticated task of M.

We first describe Task 3 in a way that is not implemented. Instead, our first description has to be extended, but is provided for clarity. The idea is to realize a k-ary counter initially set up at the rightmost cell, that is cell n. Initially, the counter gets value 1. Then, in each time step, it is incremented by two and moves one cell to the left with the least significant digit in front. However, the movement of the counter has to be stopped when it meets the right-moving counter of Task 2 in the center of the array. To stop the movement cell by cell, the counter is realized by providing an additional cell between each two cells that carry a digit. As before, the digits (partial states) used for the counter are $\{0, 1, \ldots, k-1\}$, where a superscript $^+$ indicates that a carry-over occurred. The additional cells between two digits enter the blank state, where a superscript $^+$ remembers that a carry-over occurred one step before and is still to be processed in the next step.

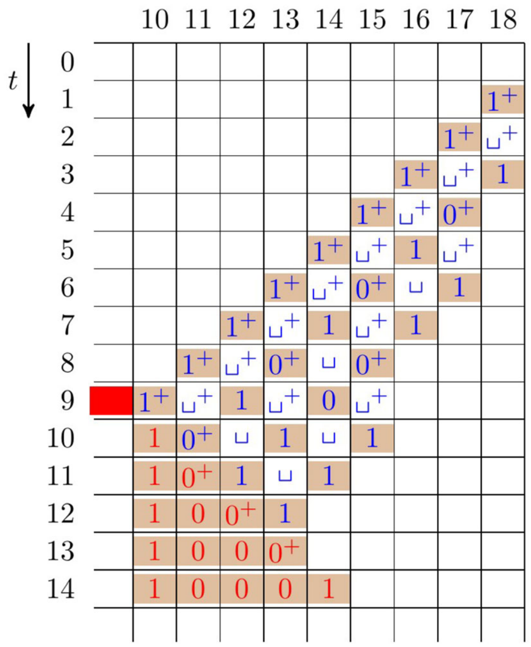


Fig. 10 Example of a left-moving binary counter that is incremented by two in each step. The cells carrying digits of the counter are shaded. The red cell 9 at $t = 9$ indicates the front of the right-moving counter of Task 2 that stops the movement of the left-moving counter cell by cell

Figure 10 shows an example of a binary counter ($k = 2$) for $n = 18$.

Next, Task 3 has to be extended. To this end, we distinguish whether the length of the array n is even or odd. If n is even, the counter is stopped in cell $\frac{n}{2}+1$ at time $\frac{n}{2}$. Since it is set up with value 1, it is stopped representing the value $1 + 2(\frac{n}{2} - 1) = n - 1$. In the following time step cell $\frac{n}{2}+1$ (the front of the counter) starts to send signals with unit speed to the right. Along their way, these signals simulate the deterministic finite automaton A by reading the digits of the stopped counter. After passing through the counter, the cells apply function τ to the state reached at the end of the simulation. In this way, the first signal reads the k-ary expansion of $n-1$ and after applying τ knows the symbol u_{n-1}. Since the signal is started at time $\frac{n}{2}+1$ in cell $\frac{n}{2}+1$, it reaches the rightmost cell n at time n. Thus, cell n can enter the correct state u_{n-1} when the FSSP is about to fire. Finally, after having sent the first signal, cell $\frac{n}{2}+1$ starts to decrement the counter by 1 at each time step.

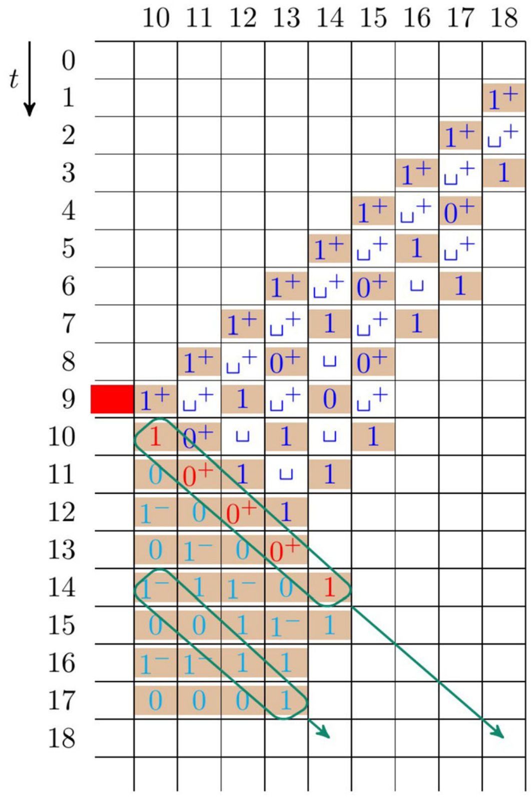
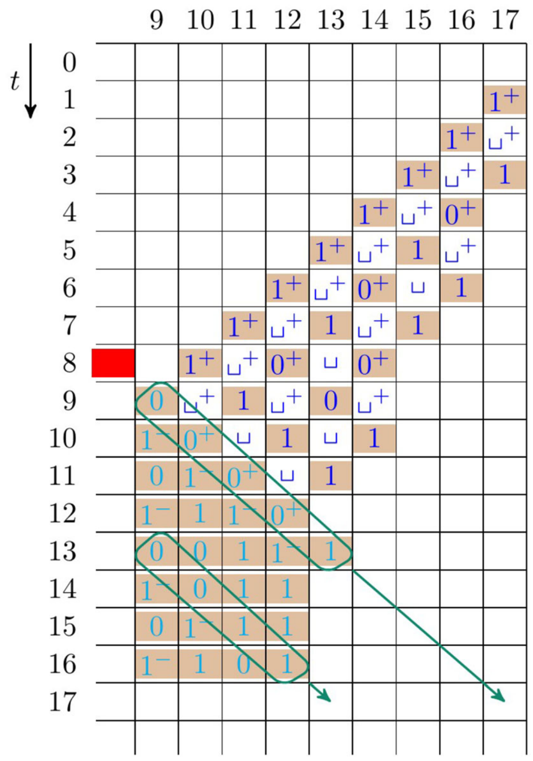


Fig. 11 Example of Task 3. The cells carrying digits of the counter are shaded. The red cell 9 at $t = 9$ indicates the front of the right-moving counter of Task 2 that stops the movement of the left-moving counter cell by cell. The digits involved in the decrementation are shown in green. The green frames and arcs show exemplarily that the first signal sees the digits 1, 0, 0, 0, 1, that is the binary expansion of 17, and reaches cell $n = 18$ at time n, and that the $\lceil \log n \rceil$th signal sees the digits 1, 0, 1, 1, that is the binary expansion of 13, and reaches cell 14 at time n, as well

Therefore, the second signal sent reads the k-ary expansion of $n-2$. It reaches cell $n-1$ at time n and, thus, cell $n-1$ can enter the correct state u_{n-2} when the FSSP is about to fire as well. This works fine as long as the signals can pass through the counter entirely. It is not hard to see that the signal sent at time $\frac{n}{2}+\lceil\log n\rceil$ has passed through the counter entirely at time n at the latest, if $n\geq 16$. Therefore, at least the last $\lceil\log n\rceil$ cells of the array not covered by Task 2 get their symbols from U in due time. Figure 11 shows an example of a binary counter ($k=2$) for $n=18$.

Now we turn to the case where n is odd. The case is detected by the cell in front of the left-moving counter by means of meeting the counter of Task 2 in one center cell instead of two neighboring center cells. Since the further behavior of Task 3 is controlled by this cell this is early enough. If n is odd, the counter is stopped in cell $\lceil\frac{n}{2}\rceil$ at time $\lceil\frac{n}{2}\rceil$ representing the value $1+2(\lceil\frac{n}{2}\rceil-1)=n$. In the case where n is even, the counter is first stopped and then decremented. Here its value is one too much. Therefore, these two steps are merged. So, the counter is already decremented by 1 at its arrival in cell $\lceil\frac{n}{2}\rceil$. This means that the first signal simulates the deterministic finite automaton A by reading the k-ary expansion of $n-1$ as it should. The rest of the task is as in the case where n is even. Figure 12 shows an example of a binary counter ($k=2$) for $n=17$.

So far, we have shown the theorem for $n\geq 16$. However, the finitely many missing cases $1\leq n\leq 15$ can be treated by M on an extra track. To this end, initially a signal is sent by the leftmost cell. The signal associates the cells passed through with their corresponding symbols from U. It stops when it arrives at the rightmost cell or when it reaches cell 15. Should the FSSP fire within the first 15 steps, each cell can enter its stable state from U. Otherwise, the cells ignore the information on the extra track. □



	9	10	11	12	13	14	15	16	17
0									
1									1^+
2								1^+	$\sqcup^+$
3							1^+	$\sqcup^+$	1
4						1^+	$\sqcup^+$	0^+	
5					1^+	$\sqcup^+$	1	$\sqcup^+$	
6				1^+	$\sqcup^+$	0^+	$\sqcup$	1	
7			1^+	$\sqcup^+$	1	$\sqcup^+$	1		
8		1^+	$\sqcup^+$	0^+	$\sqcup$	0^+			
9	0	$\sqcup^+$	1	$\sqcup^+$	0	$\sqcup^+$			
10	1	0^+	$\sqcup$	1	$\sqcup$	1			
11	0	1	0^+	$\sqcup$	1				
12	1^-	1	1	0^+					
13	0	0	1	1	1				
14	1	0	1	1					
15	0	1	1	1					
16	1^-	1	0	1					
17									

Fig. 12 Example of Task 3 for odd n. The cells carrying digits of the counter are shaded. The red cell 8 at $t=8$ indicates the front of the right-moving counter of Task 2 that stops the movement of the left-moving counter cell by cell when both reach cell 9. The digits involved in the decrementation are shown in green. The green frames and arcs show exemplarily that the first signal sees the digits 0, 0, 0, 0, 1, that is the binary expansion of 16, and reaches cell $n=17$ at time n, and that the $\lceil\log n\rceil$th signal sees the digits 0, 0, 1, 1, that is the binary expansion of 12, and reaches cell 13 at time n, as well

Funding Open Access funding enabled and organized by Projekt DEAL.

References

Allouche J, Shallit JO (2003) Automatic sequences—theory, applications, generalizations. Cambridge University Press, Cambridge

Choffrut C, Čulik K II (1984) On real-time cellular automata and trellis automata. Acta Inf. 21:393–407

de Oliveira PPB, Ruivo ELP, Costa WL, Miki FT, Trafaniuc VV (2016) Advances in the study of elementary cellular automata regular language complexity. Complex 21(6):267–279. https://doi.org/10.1002/cplx.21686

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

Fischer PC (1965) Generation of primes by a one-dimensional real-time iterative array. J. ACM 12:388–394

Grandjean A, Richard G, Terrier V (2012) Linear functional classes over cellular automata. In: Formenti E (ed) International workshop on Cellular Automata and Discrete Complex Systems and Journées Automates Cellulaires (AUTOMATA & JAC 2012), EPTCS, vol 90, pp 177–193

Ilie L (1994) On a conjecture about slender context-free languages. Theor Comput Sci 132:427–434

Kari J (2012) Universal pattern generation by cellular automata. Theor Comput Sci 429:180–184

Kolakoski W (1965) Problem 5304: self generating runs. Am Math Monthly 72:674

Kunze M, Shyr HJ, Thierrin G (1981) *h*-Bounded and semidiscrete languages. Inf. Control 51:174–187

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, New York, pp 183–227. https://doi.org/10.1007/978-3-540-78291-9_6

Kutrib M (2009) Cellular automata and language theory. In: Meyers R (ed) Encyclopedia of complexity and system science. Springer, New York, pp 800–823. https://doi.org/10.1007/978-0-387-30440-3_54

Kutrib M, Malcher A (2013) One-dimensional cellular automaton transducers. Fund Inf 126:201–224. https://doi.org/10.3233/FI-2013-878

Kutrib M, Malcher A (2020) One-dimensional pattern generation by cellular automata. In: Gwizdalla TM, Manzoni L, Sirakoulis GC, Bandini S, Podlaski K (eds) Cellular Automata for Research and Industry (ACRI 2020), LNCS, vol 12599, pp 46–55. Springer. https://doi.org/10.1007/978-3-030-69480-7_6

Kutrib M, Malcher A (2021) String generation by cellular automata. Complex Syst 30:111–132

Latteux M, Thierrin G (1983) Semidiscrete context-free languages. Int J Comput Math 14:3–18

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

Oldenburger R (1939) Exponent trajectories in symbolic dynamics. Trans Am Math Soc 46:453–466

Păun G, Salomaa A (1995) Thin and slender languages. Discrete Math 61:257–270. https://doi.org/10.1016/0166-218X(94)00014-5

Smith III AR (1970) Cellular automata and formal languages. In: 11th Ann. IEEE Symposium on Switching and Automata Theory, pp 216–224. IEEE

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

Waksman A (1966) An optimum solution to the firing squad synchronization problem. Inf Control 9:66–78

Wolfram S (1986) Random sequence generation by cellular automata. Adv Appl Math 7:123–169