String Generation by Cellular Automata

Martin Kutrib
Andreas Malcher
Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
kutrib@informatik.uni-giessen.de
andreas.malcher@informatik.uni-giessen.de

In contrast to many investigations of cellular automata with regard to their ability to *accept* inputs under certain time constraints, in this paper we are studying cellular automata with regard to their ability to *generate* strings in real time. Structural properties such as speedup results and closure properties are investigated. On the one hand, constructions for the closure under intersection, reversal and length-preserving homomorphism are presented, whereas on the other hand the nonclosure under union, complementation and arbitrary homomorphism are obtained. Finally, decidability questions such as emptiness, finiteness, equivalence, inclusion, regularity and context-freeness are addressed.

Keywords: cellular automata; pattern generation; real-time computation; speedup; closure properties; decidability problems

1. Introduction

Cellular automata (CAs) are a widely used model to describe, analyze and understand parallel processes. They are particularly applicable to massive parallel systems, since they are arrays of identical copies of deterministic finite automata where, in addition, the single nodes are homogeneously connected to both their immediate neighbors. Furthermore, they work synchronously at discrete timesteps processing a parallel distributed input, where every cell is fed with an input symbol in a pre-initial step.

A standard approach to measure the computational power of some model is to study its ability to accept formal languages (see, for example, [1]). In general, the given input is accepted if there is a timestep at which a designated cell, usually the leftmost one, enters an accepting state. Commonly studied models are two-way CAs with time restrictions such as real time or linear time, which means that the available time for accepting an input is restricted to the length of the input or to a multiple of the length of the input. An important restricted class is the real-time one-way CAs [2], where every cell is connected only with its right neighbor. Hence, the flow of

https://doi.org/10.25088/ComplexSystems.30.2.111

information is only from right to left. A survey of results concerning the computational capacity, closure properties and decidability questions for these models and references to the literature may be found, for example, in [3, 4].

In the language accepting approach, each computation can be considered as producing a yes or no answer for every possible input within a certain time. A much broader view on cellular automaton (CA) computations is taken in [5, 6], where CA computations are considered as transducing actions; that is, they produce an output of size n for every input of size n under time constraints such as real and linear time. Thus, the point of view changes from a parallel language accepting device to a parallel language transforming device. Grandjean et al. [5] discusses for CAs the time constraints of linear time and real time. Moreover, the inclusion relationships based on these constraints, closure properties and relations to CAs considered as formal language acceptors are established. An important technical result is a speedup theorem stating that every computation beyond real time can be sped up by a linear factor. Kutrib and Malcher [6] also considers CAs with sequential input mode, called iterative arrays, as transducing devices. In particular, these devices are compared with the CA counterpart with parallel input mode. In addition, the cellular transducing models are compared with classical sequential transducing devices such as finite-state transducers and pushdown transducers.

In this paper, we will look at computations with CAs from yet another perspective. Rather than computing a yes or no answer or computing an output, we are considering CAs as generating devices. This means in detail that the CA starts with an arbitrary number of cells all being in a quiescent state and, subsequently, works by applying its transition function synchronously to all cells. Finally, if the configuration reaches a fixed point, the sequence of cell states is considered as the *string generated*. From this point of view, CAs compute a (partial) function mapping an initial length n to a string of length n over some alphabet. First investigations for this model have been made in [7]. In particular, the real-time generation of unary patterns is studied in depth and a characterization by time-constructible functions and their corresponding unary formal languages is given. In this paper, we will continue these investigations for real-time CAs and study, in particular, speedup possibilities, closure properties and decidability questions of the model.

It should be remarked that the notion of pattern generation is also used for CAs in other contexts. For example, in [8] the sequence of configurations produced by a CA starting with some input is considered as a two-dimensional pattern. Kari [9] describes a CA as a universal pattern generator in the sense that starting from a finite configuration, all finite patterns over the state alphabet are generated,

which means here that these patterns occur as infixes in the sequence of configurations computed.

The paper is organized as follows. In Section 2, we formally define how CAs accept and generate formal languages and we present a detailed example generating prefixes of the Thue–Morse sequence. Section 3 is devoted to structural properties of cellular string generators. Two speedup results are presented, which are used in the subsequent constructions showing closure under intersection, reversal and length-preserving homomorphism. In Section 4, we deal with the problems of deciding emptiness, finiteness, infiniteness, inclusion, equivalence, regularity and context-freeness of real-time cellular string generators. By showing that suitably encoded computations of a Turing machine can be generated by CAs in real time, all decidability problems mentioned turn out to be not semidecidable.

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 strings* consisting of letters from Σ. The *empty string* is denoted by λ, and we set $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. A subset of Σ^* is called a *language* over Σ. For the *length of a string* w we write $|w|$. In general, we use $\subseteq$ for *inclusions* and $\subset$ for *strict inclusions*. For convenience, we use $S_{\#}$ to denote $S \bigcup \{\#\}$.

A two-way CA is a linear array of identical finite automata, called cells, numbered 1, 2, …, n. Except for border cells, each one is connected to both of its 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 timesteps.

Formally, a *deterministic two-way CA* 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 the 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 $2 \leq i \leq n-1$,

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

https://doi.org/10.25088/ComplexSystems.30.2.111

and for the outermost cells we set

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

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

Here, a CA M can operate as a decider or a generator of strings.

A CA *accepts* a string (or word) $a_1a_2\ldots a_n \in \Sigma^+$ if at some timestep during the course of the computation starting in the *initial configuration* $c_0(i) = a_i$, $1 \le i \le n$, 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 CA *generates* a string $a_1a_2\ldots a_n$ if at some timestep t during the computation on the initial configuration $c_0(i) = s_0$, $1 \le i \le n$: (1) the string appears as configuration (i.e., $c_t(i) = a_i$, $1 \le i \le n$) and (2) configuration c_t is a fixed point of the global transition function Δ (i.e., 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 CA operates as a 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 $t(|w|)$ timesteps, or if all $w \in P(M)$ are generated with at most $t(|w|)$ timesteps, then M is said to be of time complexity t. If $t(n) = n$ then M operates in *real time*. The family of all patterns generated by a CA in real time is denoted by $\mathcal{P}_{rt}(M)$.

We illustrate the definitions with an example.

Example 1. The Thue–Morse sequence is an infinite sequence over the alphabet $\{0, 1\}$. The well-known sequence has applications in numerous fields of mathematics and its properties are nontrivial. There are 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 and yields the prefixes $p_0 = 0$, $p_1 = 01$, $p_2 = 0110$, $p_3 = 01\,101\,001$, $p_4 = 0\,110\,100\,110\,010\,110$ and so on. We remark that the length of the prefix p_i is 2^i.

The pattern $P_{\text{Thue}} = \{p_i \mid i \ge 0\}$ derived therefrom can be generated by some CA in real time [7]. However, this means that nothing is

generated whenever the length of the CA is not a power of two. Here, we are going to generalize this construction and extend the pattern generated in such a way that prefixes p_i of the Thue–Morse sequence are also generated even when the length of the CA is not a power of two. The goal is to generate on initial length n the prefix p_i with $i = \lceil \log_2(n) \rceil$. Since the length of the generated prefix is larger than the number of cells if n is not a power of two, we will group two adjacent symbols $x, y \in \{0, 1\}$ of the sequence into states $x \mid y$ where appropriate. Note that in the special case of an input length being a power of two, the construction described here is the construction given in [7].

The basic idea is to work with a real-time version of the firing squad synchronization problem (FSSP) based on the time-optimal solution of Waksman [10]. The latter solution starts with one general at the left end of the array and it takes $n-1$ timesteps (n being the length of the array) to reach the right end. If we start instead with two generals at both ends, where the left general symmetrically behaves like the right general, we save $n-1$ timesteps. Since we need one additional timestep to initialize the generals at both ends, we can realize the FSSP within $2n-2-(n-1)+1 = n$ timesteps, that is, within real time.

In the construction of the FSSP, the initial length is iteratively divided into halves, whereby dependent on the arity, one or two middle points and thus one or two new generals are generated. In the first case, here the single middle cell is virtually split into two. These cells will represent two grouped adjacent symbols of the sequence. Next, we identify these cells. The following recursive formula $f(n, m)$ gives 2 if the m^{th} cell on initial length n contains a compressed symbol, and otherwise gives 1:

$$f(n, m) = \begin{cases} 1 & \text{if } n = 1 \text{ or } m \leq 1, \\ 2 & \text{if } n > 1 \text{ odd and } m = \dfrac{n+1}{2}, \\ f\left(\dfrac{n+1}{2}, \left(m \bmod \dfrac{n+1}{2}\right) + \left(m \operatorname{div} \dfrac{n+1}{2}\right)\right) & \text{if } n > 1 \text{ odd and } m \neq \dfrac{n+1}{2}, \\ f\left(\dfrac{n}{2}, m \bmod \dfrac{n}{2}\right) & \text{if } n > 1 \text{ even.} \end{cases}$$

For example, in Figure 1 we have $f(15, 4) = f(8, 4) = f(4, 0) = 1$, $f(15, 8) = 2$ and $f(15, 13) = f(8, 6) = f(4, 2) = f(2, 0) = 1$.

https://doi.org/10.25088/ComplexSystems.30.2.111

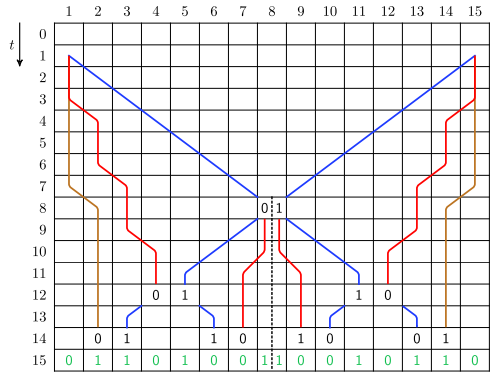


Figure 1. Generation of the prefix $p_4 = 0\,110\,100\,110\,010\,110$ of the Thue–Morse sequence on initial length $n = 15$.

In Figure 2 we have $f(17, 2) = f(9, 2) = f(5, 2) = f(3, 2) = 2$, $f(17, 8) = f(9, 8) = f(5, 4) = f(3, 2) = 2$ and $f(17, 17) = f(9, 9) = f(5, 5) = f(3, 3) = f(2, 2) = f(1, 0) = 1$.

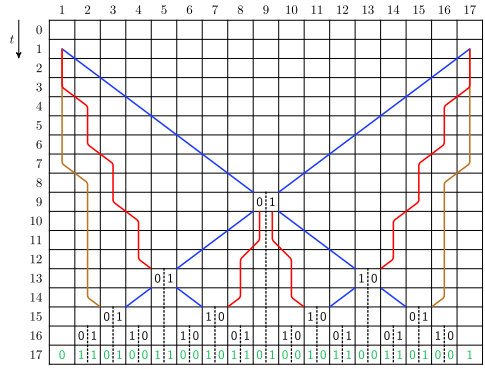


Figure 2. Generation of the prefix p_5 of the Thue–Morse sequence on initial length $n = 17$.

The function f can be used to define the generated compressed Thue–Morse sequence as follows. Let $f_n(m) = \sum_{j=1}^{m} f(n, j)$ and let $p_i = w_{i,1}w_{i,2}\ldots w_{i,2^i}$ be an enumeration of the symbols of prefix p_i. Then, the corresponding compressed prefix $p'_i(n)$ on initial length n with $i = \lceil \log_2(n) \rceil$ is $p'_i(n, 1)p'_i(n, 2)\ldots p'_i(n, n)$, where we define $p'_i(n, m) = w_{i,f_n(m)}$ if $f(n, m) = 1$, and $p'_i(n, m) = w_{i,f_n(m)-1}w_{i,f_n(m)}$ if $f(n, m) = 2$. For example, if $n = 15$, we have $f(15, 8) = 2$ and

$f(15, j) = 1$ if $j \neq 8$. Hence, $f_{15}(8) = 9$ and $f_{15}(13) = 14$, which gives $p'_4(15, 8) = 11$ and $p'_4(15, 13) = 1$, respectively.

The general generation procedure is as follows (see Figures 1 and 2 for examples). Initially, at time $\lceil n/2 \rceil + 1$, the left new middle cell obtains the information 0 and the right new middle cell obtains the information 1, or the new single and grouped middle cell obtains the information $0 \mid 1$. In the latter case, the cell simulates two cells from now on. The signals sent out from the new middle cells to the left and right, respectively, have this information attached. If these signals meet some other signal so that another one or two new middle cell(s) is (are) generated, the left cell gets the information 0 and the right cell gets the information 1 if the signal carried the information 0. Otherwise, the left cell gets the information 1 and the right cell gets the information 0. This behavior is iterated up to the next to last timestep in which all cells have become a general. In the last timestep, in which all cells are synchronized, a left signal 0 (1) in cell i at time $n-1$ leads to 0 (1) in cell $i-1$ and 1 (0) in cell i at time n. Analogously, a right signal 0 (1) in cell i at time $n-1$ leads to 0 (1) in cell i and 1 (0) in cell $i+1$ at time n. Note that here a split cell simulates two cells that are counted to determine the numbers i. Finally, after the synchronization, all states of the array represent the string generated and become permanent.

Let $\Sigma = \{0, 1, 0 \mid 0, 0 \mid 1, 1 \mid 0, 1 \mid 1\}$ be the alphabet and π be a projection to the original letters, that is, $\pi(x) = x$ and $\pi(x \mid y) = xy$, for $x, y \in \{0, 1\}$. Then, the pattern generated by the constructed real-time CA M has the desired property: $w \in P(M)$ implies $\pi(w) = p_i$ with $i = \lceil \log_2(|w|) \rceil$.

3. Structural Properties

For unary patterns of the form

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

or even

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

where $\varphi : \mathbb{N} \to \mathbb{N}$ is a time-constructible function, the three notions of language acceptance, time-constructibility and string generation are characterizing each other; that is, they coincide. Here, we now turn to structural properties of the string generators as speedup and closures of the set of generated strings under certain operations.

https://doi.org/10.25088/ComplexSystems.30.2.111

3.1 Speedup

Several types of cellular language acceptors can be sped up as long as the remaining time complexity does not fall below real time. A proof in terms of trellis automata can be found in [11]. In [12, 13] the speedup results are shown for deterministic and nondeterministic cellular and iterative language acceptors. The proofs are based on sequential machine characterizations of the parallel devices. In particular, for all $k \geq 1$, deterministic CAs can be sped up from $(n + t(n))$-time to $\left(n + \lceil t(n) / k \rceil\right)$-time [12–14]. The question of whether every linear-time cellular language acceptor can be sped up to real time is an open problem.

The situation for string generators is more involved. While for language acceptors usually only the states of one distinguished cell determine acceptance or rejection, the states of all cells are the result of a string generation. So, these known speedup results do not apply here. However, in terms of transducers that given an input of size n compute an output of size n, a speedup result is known [5] that can be adapted to our notion.

As a preliminary lemma, we prove that cellular string generators working in real time plus a constant amount of time can always be sped up to real time.

Lemma 1. Let M be a cellular string generator with time complexity $n + k$, where $k \geq 1$ is a constant integer. Then an equivalent real-time string generator M' can effectively be constructed.

Proof. The basic idea for the construction of M' is to have in every cell $k + 1$ tracks such that at time t the tracks in M' are filled with states from M at time $t, t + 1, \ldots, t + k$. Hence, timestep $n + k$ in M is simulated at time n in M' in the last track. The construction works in two phases. In the first phase, we start the simulation of M at both ends, but we simulate $k + 1$ timesteps of M at once in every cell leaving the quiescent state s_0. In the remaining timesteps, each cell shifts the contents of its i^{th} track to its $(i-1)^{\text{st}}$ track for $2 \leq i \leq k + 1$ and updates the entry in the $(k+1)^{\text{st}}$ track. Hence, every cell keeps track of the last $k + 1$ timesteps.

We have a left part and a right part of the computation in this first phase. For the left part, we may assume that all missing information from the right can be replaced by the information s_0, since in the beginning all cells are in the quiescent state s_0. In addition, we have to provide each cell in the left part with the necessary information from the left, namely, with the states of the k cells to the left. This can be realized by using k additional tracks. Analogously, we can assume for the right part of the computation that missing information from the

left can be replaced by s_0 and we can provide information about the states of the k cells to the right by using k tracks.

At time $n/2+1$, if the initial length n is even and at time $(n+1)/2$ if n is odd, the left and right part of the computation meet in the middle cell(s), from which the second phase starts. In this phase, all information to update the entries in the $(k+1)^{\text{st}}$ track is available, taking into account both neighbors. Hence, the simulation of the last $k+1$ timesteps can be continued. At timestep n in M', we have eventually simulated the desired $(n+k)^{\text{th}}$ timestep of M in the $(k+1)^{\text{st}}$ track. To extract this information in the n^{th} timestep of M', we start an instance of the FSSP in the beginning of another track. If the FSSP fires at timestep n, we just output the $(k+1)^{\text{st}}$ track that would be calculated at timestep n. □

Example 2. Let us discuss the speedup from $n+2$ to n starting with an initial length of $n=7$. Figure 3 shows the original computation where configurations are numbered from 1 to 63, as well as the sped-up computation.

t	1	2	3	4	5	6	7
0	s_0	s_0	s_0	s_0	s_0	s_0	s_0
1	1	s_0	s_0	s_0	s_0	s_0	7
2	8	9	s_0	s_0	s_0	13	14
3	15	16	17	s_0	19	20	21
4	22	23	24	25	26	27	28
5	29	30	31	32	33	34	35
6	36	37	38	39	40	41	42
7	43	44	45	46	47	48	49
8	50	51	52	53	54	55	56
9	57	58	59	60	61	62	63

(a)

t	1	2	3	4	5	6	7
0	s_0	s_0	s_0	s_0	s_0	s_0	s_0
1	# # 1 8 15	s_0	s_0	s_0	s_0	s_0	# # 7 14 21
2	# # 8 15 22	# 8 9 16 23	s_0	s_0	s_0	14 # 13 20 27	# # 14 21 28
3	# # 15 22 29	# 15 16 23 ⊔	15 16 17 24 ⊔	s_0	20 21 19 26 ⊔	21 # 20 27 ⊔	# # 21 28 35
4	# # 22 29 ⊔	# 22 23 ⊔ ⊔	22 23 24 ⊔ ⊔	25 32 39	27 28 26 ⊔ ⊔	28 # 27 ⊔ ⊔	# # 28 35 ⊔
5	# # 29 ⊔ ⊔	# 29 30 ⊔ ⊔	31 38 45	32 39 46	33 40 47	35 # 34 ⊔ ⊔	# # 35 ⊔ ⊔
6	# # 36 ⊔ ⊔	37 44 51	38 45 52	39 46 53	40 47 54	41 48 55	# # 42 ⊔ ⊔
7	43 50 57	44 51 58	45 52 59	46 53 60	47 54 61	48 55 62	49 56 63

(b)

Figure 3. (a) Original computation of $n+2$ timesteps on initial length $n=7$ as discussed in Example 2. (b) The sped-up real-time computation.

Here, the lower three tracks are used to keep track of the states in the last three timesteps of the original computation. The upper two tracks are used to send information from left to right in the left part and from right to left in the right part. This information is used to update the lower three tracks. In the first phase, states are computed

under the assumption that enough cells to the right or left, respectively, are in the quiescent state s_0. If this assumption fails, since the middle cell(s) are approached, we still have to enter some state according to the assumption, but this state may not be the state of the original computation and is denoted by $\_$. However, such states are not necessary for the computation in the second phase.

Next, we turn to showing how to speed up the part of the time complexity beyond real time by a constant factor. As mentioned before, to this end we consider the involved and tricky construction shown for transductions in [5]. Before we turn to the adaption to our notion, we sketch the underlying idea of [5] to speed up the part of the time complexity beyond real time by a factor 2 (see Figure 4). The basic idea is to compress the input of length n into the $\frac{n}{2}$ cells $\frac{1}{4}n+1$, $\frac{1}{4}n+2$, …, $\frac{3}{4}n$, to simulate the given $(n+t(n))$-time CA with double speed on the compressed input, and to decompress the simulation result. The most involved part is the compression depicted in the red parts of the space-time diagram in Figure 4. Roughly speaking, in these areas the compression takes place in addition to as many as possible simulation steps. The cells in the green area of the space-time

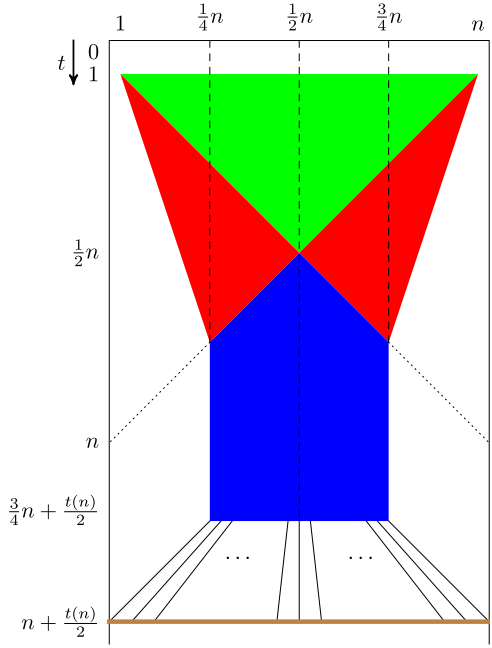


Figure 4. Principle of speeding up the part of the time complexity beyond real time by a factor 2 according to [5].

diagram are in the quiescent state. The simulation in the blue area is with double speed. Concerning the time, once cell $\frac{1}{2}n$ has left the quiescent state, it has to perform a further $\frac{1}{2}n + t(n)$ steps. Since the simulation is with double speed, this takes $\frac{1}{4}n + \frac{1}{2}t(n)$ time. The decompression of the computation result takes another $\frac{1}{4}n$ timesteps.

Theorem 1. Let M be a cellular string generator with time complexity $n + t(n)$, where $t : \mathbb{N} \to \mathbb{N}$ is a function such that M is synchronizable at timestep $n + t(n)$. Then, for all $k \geq 1$, an equivalent string generator M' with time complexity $n + \lceil t(n) / k \rceil$ can effectively be constructed.

Proof. The principle of the construction is as shown in [5] and has been described above. However, since a cellular string generator has to end a successful computation in a stable configuration, there are differences with the transductions considered in [5]. So, basically, the simulation sketched in Figure 4 has to be stopped synchronously at time $\frac{3}{4}n + \frac{1}{2}t(n)$, and the decompression has to be stopped synchronously at time $n + \frac{1}{2}t(n)$.

The first synchronization justifies the condition that M has to be synchronizable at timestep $n + t(n)$. So, some FSSP that synchronizes M at this time is implemented on an extra track of M. Therefore, its simulation is finished by firing the cells $\frac{1}{4}n + 1, \frac{1}{4}n + 2, \ldots, \frac{3}{4}n$ at time $\frac{3}{4}n + \frac{1}{2}t(n)$ as required. Additionally, the decompression phase is started synchronously.

The second synchronization is achieved by using two more extra tracks. On the first track, an FSSP is implemented that synchronizes all cells at timestep n. On the second track, an FSSP is implemented that would synchronize all cells at time $n + t(n)$. But now this second FSSP runs with speed one until the first FSSP fires, and continues with half speed. So it fires at time $n + \frac{1}{2}t(n)$ as required. □

3.2 Closure Properties

This subsection is devoted to investigating the closure properties of the family $\mathcal{P}_{rt}(\mathrm{CA})$. We start with the Boolean operations.

Proposition 1. The family $\mathcal{P}_{rt}(\mathrm{CA})$ is closed under intersection. It is closed neither under union nor under complementation.

Proof. The first result is the closure under intersection, which can be realized by the well-known Cartesian product construction using two tracks, in each of which the given real-time string generators are

https://doi.org/10.25088/ComplexSystems.30.2.111

simulated. Additionally, in a third track an instance of the FSSP is started. In the last timestep, when the FSSP is firing, it is checked in each cell whether or not in the first and second track the same state s would be entered. If so, the single state s is entered. Otherwise, some new state p is entered that alternates with another new state q. It is clear that in this way all strings are generated that are generated by both given real-time string generators. On the other hand, states p and q ensure that no string is generated that is not generated by both given real-time string generators.

Next we study the union operation. Each real-time string generator M with state set S can be considered to define a (partial) function f_M between $\mathbb{N}$ and S^+. Hence, a real-time string generator M that generates the union $P(M_1) \bigcup P(M_2)$ for real-time string generators M_1 and M_2 defines such a function f_M as well. This means that there must not be any $n \in \mathbb{N}$ such that $f_{M_1}(n) \neq f_{M_2}(n)$. In other words, if two real-time string generators produce different strings on the same initial length, the union cannot be described by a function and hence cannot be realized by a real-time string generator. This shows that the family $\mathcal{P}_{rt}(\mathrm{CA})$ is not closed under union. By De Morgan's laws we obtain that the family $\mathcal{P}_{rt}(\mathrm{CA})$ is not closed under complementation as well. □

We would like to remark that in [7] it has been shown that the pattern

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

is generated by some real-time CA if φ is a time-constructible function. Define the pattern

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

By interchanging the roles of the symbols a and b, we immediately obtain that the pattern $\hat{P}^c_\varphi$ is generated by some real-time CA as well. Clearly, $\hat{P}^c_\varphi$ is in general not the set-theoretic complement of $\hat{P}_\varphi$, but can be interpreted to be the opposite of $\hat{P}_\varphi$.

For the union of two patterns generated, we may consider the special case that the union can be described by a function. The basic idea to show the closure of $\mathcal{P}_{rt}(\mathrm{CA})$ in this special case would be similar to the construction for the intersection. That is, in two tracks the given real-time string generators are simulated and in the third track an instance of the FSSP is started. Analogously, in the last timestep it is checked whether the first two tracks would generate the same stable configuration or exactly one stable configuration. However, an

instability can occur, for example, in one cell only, say on the first track. If the configuration on the second track is stable, then we should generate it. Otherwise, the unstable cells should stay unstable so that no string is generated. Thus, the instability should be made stable if and only if the other configuration is stable. This check probably costs more than constant time. Hence, it is currently an open question whether a real-time CA generating the union can be constructed even if the union of patterns can be described by a function.

Next, we look at the reversal operation. By interchanging the left-hand and right-hand entries of the local transition function, we immediately obtain that the reversal of a real-time pattern can also be generated by a real-time string generator.

Proposition 2. The family $\mathcal{P}_{rt}(\mathrm{CA})$ is closed under reversal.

Finally, we investigate homomorphisms and obtain that the family $\mathcal{P}_{rt}(\mathrm{CA})$ is not closed under arbitrary homomorphisms, whereas we can show that the family $\mathcal{P}_{rt}(\mathrm{CA})$ is closed under length-preserving homomorphisms.

Proposition 3. The family $\mathcal{P}_{rt}(\mathrm{CA})$ is not closed under arbitrary homomorphism, but is closed under length-preserving homomorphism.

Proof. For the nonclosure under arbitrary homomorphism, we consider the language $L = \{a, bb\}$, which can be generated by a real-time CA, and the homomorphism h, which maps a to aa and b to b. Then, $h(L) = \{aa, bb\}$, which cannot be generated by any real-time CA.

Now, let $M = \langle S, s_0, \#, \delta \rangle$ be a real-time CA generating the pattern $P \subseteq \Sigma^*$, where $\Sigma \subseteq S$. Moreover, let $h : \Sigma^* \to \Gamma^*$ be a length-preserving homomorphism. For the construction of a CA that generates $h(P)$ in real time, we consider a primed version S' of the state set S, where the primed version of Σ is denoted by Σ', and we extend h to a length-preserving homomorphism h' mapping from S'^* to $(S' \cup \Gamma)^*$ such that $h'(a') = h(a)$ if $a' \in \Sigma'$, and $h'(s') = s'$ if $s' \in S' \setminus \Sigma'$.

Next, we will sketch the construction of a CA M' generating $P(M') = h'(P(M)) = h'(P) = h(P)$ in real time plus one that is subsequently sped up with Lemma 1 to work in real time. The basic idea is to work with three tracks and to start in the first track the simulation of M with state set S', and in the second track an instance of the FSSP. At timestep n, when the FSSP fires, we generate in the third track the homomorphic image under h' of the first track. At timestep $n + 1$, we check in every cell whether its state in the first track, that is, in the simulated configuration of M at time n, would also remain the same in the next timestep, that is, in the simulated configuration of M at time $n + 1$. If so, each such cell enters the state from the third track,

https://doi.org/10.25088/ComplexSystems.30.2.111

that is, the homomorphic image under h' of the state in the simulated configuration of M at time n. Otherwise, a new state p is entered that alternates with some other new state q. Furthermore, the transition function of M' is suitably complemented so that configurations containing only symbols from Γ remain stable. If the configuration of M at time n is stable, a string from Σ^* is generated by M and we obtain that the homomorphic image under h', which is equivalent to h on Σ, of this string is generated by M'. Otherwise, if M generates no string due to an unstable configuration, then M' enters an unstable configuration as well. Therefore, it is ensured that M' generates no string as well in such cases. □

4. Decidability Problems

In order to generate some sophisticated patterns, we provide a technique that is based on the possibility of CAs to simulate certain data structures without any loss of time [3]. 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 queues). 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 [3].

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 on 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. Figure 5).

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.

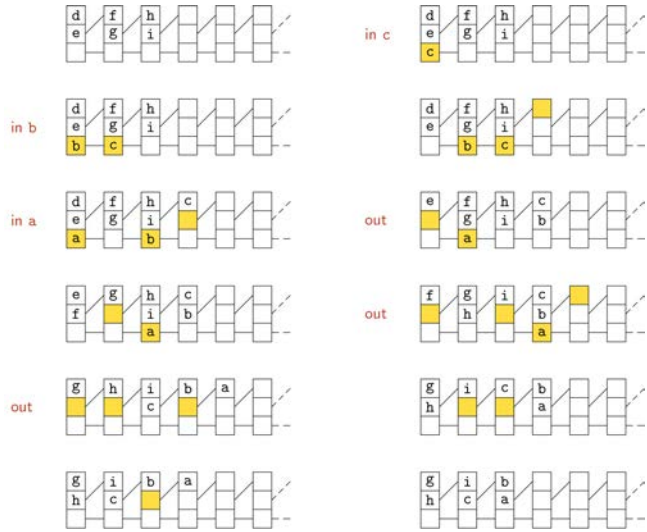


Figure 5. Principle of a ring (queue) simulation. Subfigures are in row-major order.

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.

Undecidability results. The first decidability problem we are dealing with is the question of whether a given CA M generates a string at all, or whether the pattern $P(M)$ is empty. In order to show that the emptiness problem is not even semidecidable, even for CAs that generate patterns in real time, we reduce the problem to decide whether a Turing machine *does not halt* on empty input. It is well known that this problem is not semidecidable.

For the reduction, a technique from [15] is utilized. Basically, the history of a Turing machine computation is encoded into a string. It suffices to consider deterministic Turing machines with one single tape and one single read-write head. Without loss of generality and for technical reasons, we assume that the Turing machines can either

print a symbol on the current tape square or move the head one square to the right or left. Moreover, they neither can print blanks nor leave a square containing a blank. Finally, a Turing machine is assumed to perform at least one transition. So, let Q be the state set of some Turing machine M, where q_0 is the initial state and $T \cap Q = \emptyset$ is the tape alphabet containing the blank symbol $\_$. Then a configuration of M can be written as a string of the form $T^*(Q, T)T^*$ such that $x_1x_2\ldots x_i(q, y)x_{i+1}x_{i+2}\ldots x_n$ is used to express that M is in state q, scanning tape symbol y, and $x_1x_2\ldots x_iyx_{i+1}x_{i+2}\ldots x_n$ is the support of the tape inscription.

Dependent on M, we define the string v_M. Let $\$ \notin T \cup Q$, $m \geq 1$ and $w_i \in T^*(Q, T)T^*$, $0 \leq i \leq m$ be configurations of M. If M does not halt on empty input, then $v_M = \lambda$. Otherwise, we set

$$v_M = \$w_0\$w_1\$w_2\$\cdots\$w_m\$,$$

where w_0 is the initial configuration $(q_0, \_)$, w_m is a halting configuration and w_i is the successor configuration of w_{i-1}, $1 \leq i \leq m$. Now we define a pattern as $P_M = \{v_M\} \setminus \{\lambda\}$.

Proposition 4. Let M be a Turing machine. Then the pattern P_M is generated by some CA in real time.

Proof. The basic idea of the construction of a real-time CA that generates P_M is as follows. The rightmost cell emits one symbol of the string v_M in every timestep onto a special track. The content of this initially empty track is successively shifted to the left by all cells. In order to compute the correct symbols, the rightmost cell simulates a queue. In addition, it uses two auxiliary registers, R_1 and R_2.

In more detail, the rightmost cell does the following. Whenever it emits a symbol onto the special track, this symbol is additionally entered into the initially empty queue. Let δ be the transition function of M, and $\delta(q_0, \_) = (q', y)$ with $y \in T$ (recall that M cannot leave a blank square). Then, in the first three steps, the rightmost cell emits the symbols \$, $(q_0, \_)$ and \$ (the first three symbols of v_M) and fills R_1 with (q', y) and R_2 with \$. In general, registers R_1 and R_2 contain two symbols that can be seen as at the front of the queue. So, after the third step the actual queue content is empty and its extension by the two registers is $(q', y)\$$ (the first symbol is at the front of the queue), which is basically the second configuration of M. From now on, the queue content is revolved symbol by symbol, whereby the successor configuration is determined, emitted and entered into the queue again.

We distinguish two cases. First, let R_2 contain a single symbol from $T \cup \{\$\}$. This means that the short part of the configuration that

may change due to an action of M is yet unreached or already processed. So in this case, the content of R_1 is emitted, the content of R_2 is moved to R_1, and the first symbol is removed from the actual queue and stored into R_2 (see Figure 6).

In the second case, register R_2 contains a symbol (q, x) from $Q \times T$. This means that the part of the configuration that may change due to an action of M is reached. In order to compute the changes, a step of M has to be simulated. Dependent on the head movement of M, we distinguish subcases. If $\delta(q, x) = (q', y)$, that is, M writes without moving the head, then the action is as shown in Figure 7.

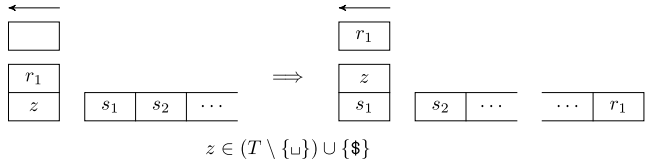


Figure 6. Action of the rightmost cell. On the left the action is depicted symbolically. For the cell structure, the register of the left-shifting track to which symbols are emitted (indicated by the left arrow on top), register R_1 above register R_2 and the idealized queue are depicted.

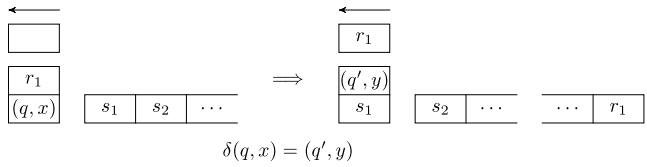


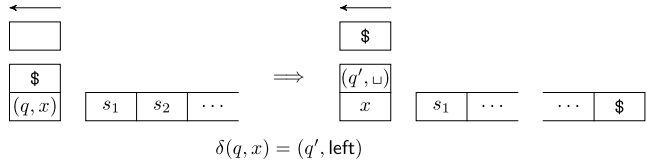
Figure 7. Action of the rightmost cell when the head of the Turing machine does not move. The cell structure is as in Figure 6.

If $\delta(q, x) = \big(q', \text{left}\big)$ and R_1 contains \$, that is, M moves its head to the left from the leftmost visited square to a blank square, then the action is as shown in Figure 8 at the top.

If $\delta(q, x) = \big(q', \text{left}\big)$ and R_1 does not contain \$, that is, M moves its head to the left to a nonblank square, then the action is as shown in Figure 8 at the bottom.

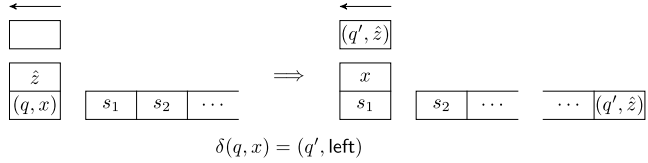
If $\delta(q, x) = \big(q', \text{right}\big)$ and the symbol at the front of the actual queue is \$, that is, M moves its head to the right from the rightmost visited square to a blank square, then the action is as shown in Figure 9 at the top.

https://doi.org/10.25088/ComplexSystems.30.2.111



$\delta(q, x) = (q', \mathsf{left})$

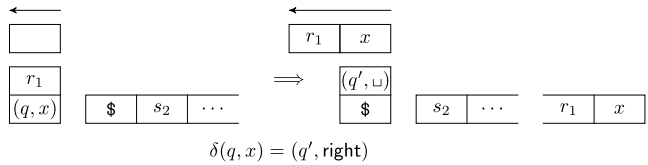
Emitand enqueue : $R_1, R_1 \leftarrow (q', \_), \ R_2 \leftarrow x.$



$\delta(q, x) = (q', \mathsf{left})$

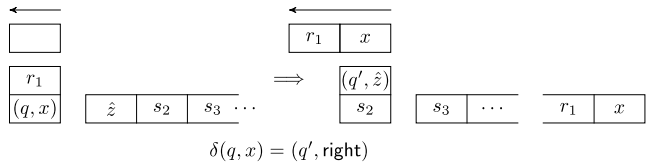
Emit and enqueue : $(q', R_1), \ R_1 \leftarrow x, \ R_2 \leftarrow$ dequeue.

Figure 8. Action of the rightmost cell when the head of the Turing machine moves to the left. $\hat{z} \in T \backslash \{\_\}$. The cell structure is as in Figure 6.



$\delta(q, x) = (q', \mathsf{right})$

Emit and enqueue : $R_1 x, \ R_1 \leftarrow (q', \_), \ R_2 \leftarrow$ dequeue.



$\delta(q, x) = (q', \mathsf{right})$

Emit and enqueue : $R_1 x, \ R_1 \leftarrow (q',$ dequeue$), \ R_2 \leftarrow$ dequeue.

Figure 9. Action of the rightmost cell when the head of the Turing machine moves to the right. $\hat{z} \in T \backslash \{\_\}$. The cell structure is as in Figure 6.

Finally, if $\delta(q, x) = \big(q', \text{right}\big)$ and the symbol at the front of the actual queue is not \$, that is, M moves its head to the right to a nonblank square, then the action is as shown in Figure 9 at the bottom.

For easier writing, in the last two subcases two symbols are emitted. These subcases are actually realized by two steps.

In order to conclude the construction of the real-time CA, we add an additional track on which an FSSP is set up that synchronizes the array at real time. At that time, the leftmost cell receives the first symbol of the generated sequence, and all but possibly the rightmost cell enter a state that is their current symbol of the sequence. However,

the generated sequence is the valid string v_M only if the last configuration computed by the rightmost cell is a halting configuration of M. So the rightmost cell is extended such that it always checks whether the current configuration (of M) is halting. If not, the computation continues to emit symbols as before. If yes, the computation continues to emit the current configuration and a trailing \$. After that, it just emits some error symbol. Finally, the rightmost cell enters the stable state \$ (the last symbol of v_M) if it emits this \$ as the trailing symbol of a halting configuration exactly at the synchronization time, that is, at real time. Otherwise it starts to alternate between entering two different states, thus preventing the computation from generating a pattern.

We conclude that the real-time CA constructed generates the string v_M if and only if M halts on empty tape and the length of the array is $|v_M|$. That is, it generates pattern P_M. □

Basically, Proposition 4 is already the reduction of the problem to decide whether a Turing machine *does not halt* on empty input to our emptiness problem.

Theorem 2. Given a real-time CA M, it is not semidecidable whether $P(M)$ is empty.

Proof. Given an arbitrary Turing machine M', by Proposition 4 we construct a real-time CA M that generates the pattern $P_{M'}$. Pattern $P_{M'}$ is empty if and only if M' does not halt on empty input. So if the emptiness of $P(M)$ were semidecidable, then the problem to decide whether a Turing machine does not halt on empty input would be semidecidable, a contradiction. □

From the undecidability of the emptiness problem, the undecidability of the equivalence and inclusion problem follows immediately.

Corollary 1. Given two real-time CAs M_1 and M_2, it is neither semidecidable whether they generate the same pattern, that is, $P(M_1) = P(M_2)$, nor whether $P(M_1) \subseteq P(M_2)$.

Proof. It is easy to construct a real-time CA that generates the empty pattern. So if equivalence would be semidecidable, then emptiness would be semidecidable as well. Since $P(M_1) = P(M_2)$ if and only if $P(M_1) \subseteq P(M_2)$ and $P(M_2) \subseteq P(M_1)$, the semidecidability of inclusion would imply the semidecidability of equivalence, a contradiction. □

Theorem 3. Given a real-time CA M, it is neither semidecidable whether $P(M)$ is finite nor whether $P(M)$ is infinite.

Proof. Let M' be an arbitrary Turing machine.

https://doi.org/10.25088/ComplexSystems.30.2.111

In order to show the assertion for finiteness, the pattern $P_{M'}$ is extended to $\hat{P}_{M'} = \{v_{M'}a^n \mid n \geq 0\} \setminus \{a^n \mid n \geq 0\}$. Since $\hat{P}_{M'}$ is empty and thus finite if and only if M' does not halt on empty input, the non-semidecidability of finiteness follows.

For infiniteness, the pattern $P_{M'}$ is changed differently. We set

$$\tilde{P}_{M'} = \{\, \$w_0\$w_1\$\cdots\$w_m\$ \mid m \geq 1, \\ w_0 = (q_0, \_), \text{ and } w_i \text{ is successor of } w_{i-1} \,\}.$$

Since pattern $\tilde{P}_{M'}$ is infinite if and only if M' does not halt on empty input, the non-semidecidability of infiniteness follows. □

Next we turn to two decidability problems that, to some extent, relate pattern generation with language acceptance. It is of natural interest if a given language is regular or context free. So here we can ask whether the strings of a pattern form a regular or context-free language.

Theorem 4. Given a real-time CA M, it is neither semidecidable whether $P(M)$ forms a regular nor whether $P(M)$ forms a context-free language.

Proof. Let L be an arbitrary recursively enumerable language. Then there is a Turing machine M' that enumerates the words of L; that is, it produces a list of not necessarily distinct words from L such that any word in L appears in the list. Modify M' as follows. Whenever a new word w is to be put into the list, the new machine M'' first checks if w already appears in the list. If yes, it is not put into the list again and M'' continues to enumerate the next word. If not, the word w is put into the list and after that M'' enters a state from some set Q_+ that indicates that a new word has been enumerated.

Similarly to the proof of Theorem 3 we set

$$\overline{P}_{M''} = \{\, \$w_0\$w_1\$\cdots\$w_m\$ \mid m \geq 1,\ w_0 = (q_0, \_), \\ w_i \text{ is successor of } w_{i-1}, \text{ and the state in } w_m \text{ belongs to } Q_+ \,\}.$$

So if L is finite, the pattern $\overline{P}_{M''}$ is finite and thus regular and context free. If L is infinite, the pattern $\overline{P}_{M''}$ is infinite, and a simple application of the pumping lemma shows that it is not regular and not context free. In particular, pattern $\overline{P}_{M''}$ is regular and context free if and only if L is finite. Since finiteness of recursively enumerable language is not semidecidable, the assertion follows. □

Acknowledgments

A preliminary version of this work was presented at the 26th International Workshop on Cellular Automata and Discrete Complex Systems (AUTOMATA 2020), August 10–12, 2020, and is published in [16].

References

[1] A. R. Smith, "Cellular Automata and Formal Languages," in *11th Annual Symposium on Switching and Automata Theory (SWAT 1970)*, Piscataway, NJ: IEEE, 1970 pp. 216–224. doi:10.1109/SWAT.1970.4.

[2] C. R. Dyer, "One-Way Bounded Cellular Automata," *Information and Control*, **44**(3), 1980 pp. 261–281. doi:10.1016/S0019-9958(80)90164-3.

[3] M. Kutrib, "Cellular Automata: A Computational Point of View," *New Developments in Formal Languages and Applications* (G. Bel-Enguix, M. D. Jiménez-López and C. Martín-Vide, eds.), Berlin: Springer, 2008 pp. 183–227. doi:10.1007/978-3-540-78291-9_6.

[4] M. Kutrib, "Cellular Automata and Language Theory," *Encyclopedia of Complexity and Systems Science* (R. A. Meyers, ed.), New York: Springer, 2009 pp. 800–823.

[5] A. Grandjean, G. Richard and V. Terrier, "Linear Functional Classes over Cellular Automata," in *Proceedings of the 18th International Workshop on Cellular Automata and Discrete Complex Systems and the 3rd International Symposium Journées Automates Cellulaires (AUTOMATA & JAC 2012)* (E. Formenti, ed.), EPTCS, **90**, 2012 pp. 177–193. doi:10.4204/EPTCS.90.15.

[6] M. Kutrib and A. Malcher, "One-Dimensional Cellular Automaton Transducers," *Fundamenta Informaticae*, **126**(2), 2013 pp. 201–224. doi:10.3233/FI-2013-878.

[7] M. Kutrib and A. Malcher, "One-Dimensional Pattern Generation by Cellular Automata," in *Proceedings of the 14th International Conference on Cellular Automata for Research and Industry (ACRI 2020)*, Lodz, Poland (T. M. Gwizdałła, L. Manzoni, G. C. Sirakoulis, S. Bandini and K. Podlaski, eds.), Cham, Switzerland: Springer, 2020 pp. 46–55. doi:10.1007/978-3-030-69480-7_6.

[8] S. Wolfram, "Random Sequence Generation by Cellular Automata," *Advances in Applied Mathematics*, **7**(2), 1986 pp. 123–169. doi:10.1016/0196-8858(86)90028-X.

[9] J. Kari, "Universal Pattern Generation by Cellular Automata," *Theoretical Computer Science*, **429**, 2012 pp. 180–184. doi:10.1016/j.tcs.2011.12.037.

[10] A. Waksman, "An Optimum Solution to the Firing Squad Synchronization Problem," *Information and Control*, **9**(1), 1966 pp. 66–78. doi:10.1016/S0019-9958(66)90110-0.

[11] C. Choffrut and K. Čulik II, "On Real-Time Cellular Automata and Trellis Automata," *Acta Informatica*, **21**(4), 1984 pp. 393–407. doi:10.1007/BF00264617.

[12] O. H. Ibarra, S. M. Kim and S. Moran, "Sequential Machine Characterizations of Trellis and Cellular Automata and Applications," *SIAM Journal on Computing*, **14**(2), 1985 pp. 426–447. doi:10.1137/0214033.

[13] O. H. Ibarra and M. A. Palis, "Some Results Concerning Linear Iterative (Systolic) Arrays," *Journal of Parallel and Distributed Computing*, **2**(2), 1985 pp. 182–218. doi:10.1016/0743-7315(85)90034-6.

[14] W. Bucher and K. Čulik II, "On Real Time and Linear Time Cellular Automata," *RAIRO Informatique Théorique et Applications*, **18**(4), 1984 pp. 307–325. www.numdam.org/item/ITA_1984__18_4_307_0.

[15] J. Hartmanis, "On the Succinctness of Different Representations of Languages," in *Proceedings of the 6th International Colloquium on Automata, Languages and Programming (ICALP 1979)*, Graz, Austria (H. Maurer, ed.), Berlin, Heidelberg: Springer, 1979 pp. 282–288. doi:10.1007/3-540-09510-1_22.

[16] M. Kutrib and A. Malcher, "Cellular String Generators," in *Proceedings of the 26th International Workshop on Cellular Automata and Discrete Complex Systems (AUTOMATA 2020)*, Stockholm, 2020 (H. Zenil, ed.), Cham, Switzerland: Springer, 2020 pp. 59–70. doi:10.1007/978-3-030-61588-8_5.