Cellular String Generators

Martin Kutrib, Andreas Malcher

Cellular String Generators

Martin Kutrib[0000−0002−9564−2625] and Andreas Malcher[0000−0002−9589−5833]

Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
{kutrib,andreas.malcher}@informatik.uni-giessen.de

Abstract. In contrast to many investigations of cellular automata with regard to their ability to *accept* inputs under certain time constraints, we are studying here cellular automata towards their ability to *generate* strings in real time. Structural properties such as speed-up 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 non-closure under union, complementation, and arbitrary homomorphism is obtained. Finally, decidability questions such as emptiness, finiteness, equivalence, inclusion, regularity, and context-freeness are addressed.

1 Introduction

Cellular automata (CA) are a widely used model to describe, analyze, and understand parallel processes. They are in particular applied for 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 time steps 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, [13]). In general, the given input is accepted if there is a time step at which a designated cell, usually the leftmost one, enters an accepting state. Commonly studied models are two-way cellular automata 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 are the real-time one-way cellular automata [3], where every cell is connected with its right neighbor only. Hence, the flow of information is from right to left only. A survey on results concerning the computational capacity, closure properties, and decidability questions for these models and references to the literature may be found, for example, in [9, 10].

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 automata computations is taken in [4, 11], where cellular automata 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. The paper [4] discusses for cellular automata the time constraints of linear time and real time. Moreover, the inclusion relationships based on these constraints, closure properties, and relations to cellular automata considered as formal language acceptors are established. An important technical result is a speed-up theorem stating that every computation beyond real time can be sped up by a linear factor. The paper [11] considers also cellular automata with sequential input mode, called iterative arrays, as transducing devices. In particular, these devices are compared with the cellular automata 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 on computations with cellular automata from yet another perspective. Rather than computing a yes or no answer or computing an output we are considering cellular automata as generating devices. This means in detail that the cellular automaton starts with an arbitrary number of cells being all in a quiescent state and, subsequently, works by applying its transition function synchronously to all cells. Finally, if the configuration reaches a fixpoint the sequence of cell states is considered as the *string generated*. From this point of view cellular automata 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 [12]. 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 cellular automata and study, in particular, speed-up possibilities, closure properties, and decidability questions of the model.

It should be remarked that the notion of pattern generation is used for cellular automata also in other contexts. For example, in [15] the sequence of configurations produced by a cellular automaton starting with some input is considered as a two-dimensional pattern generated. Kari [8] describes a cellular automaton as 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 cellular automata 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 speed-up 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 cellular automata 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 $S \cup \{\#\}$.

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

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

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

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

A cellular automaton *accepts* a string (or 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 string $a_1 a_2 \cdots a_n$, 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 string 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 $t\colon \mathbb{N} \to \mathbb{N}$ 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*. The family of all patterns generated by a cellular automaton in real time is denoted by $\mathscr{P}_{rt}(M)$.

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 non-trivial. 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 and yields the prefixes $p_0 = 0$, $p_1 = 01$, $p_2 = 0110$, $p_3 = 01101001$, $p_4 = 0110100110010110$, and so on. We remark that the length of the prefix p_i is 2^i.

The pattern $P_{Thue} = \{\, p_i \mid i \geq 0 \,\}$ derived therefrom can be generated by some cellular automaton in real time [12]. However, this means that nothing is generated whenever the length of the cellular automaton is not a power of two. Here, we are going to generalize this construction and extend the pattern generated in such a way that also prefixes p_i of the Thue-Morse sequence are generated even when the length of the cellular automaton 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 $x|y$ 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 [12].

The basic idea is to work with a real-time version of the FSSP based on the time optimal solution of Waksman [14]. 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 left general symmetrically behaves as the right 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.

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 mth cell on initial length n contains a compressed

symbol and, otherwise, gives 1.

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

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

The function f can be used to define the generated compressed Thue-Morse sequence. Let $f_n(m) = \sum_{j=1}^{m} f(n,j)$ and $p_i = w_{i,1}w_{i,2}\cdots w_{i,2^i}$ 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 $p'_i(n,1)p'_i(n,2)\cdots p'_i(n,n)$, 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 Figure 1 for an example). 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|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, are attached with this information. 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

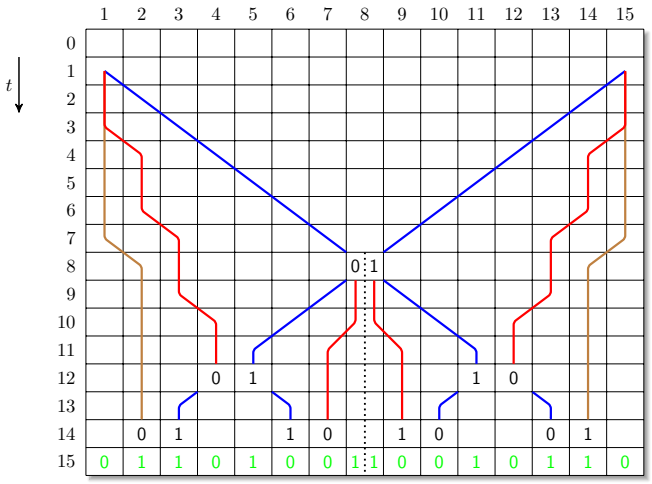

Fig. 1. Generation of the prefix $p_4 = 0110100110010110$ of the Thue-Morse sequence on initial length $n = 15$.

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 last but one time step in which all cells have become a general. In the last time step, 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 $\Sigma = \{0, 1, 0|0, 0|1, 1|0, 1|1\}$ be the alphabet and π be a projection to the original letters, that is, $\pi(x) = x$ and $\pi(x|y) = xy$, for $x, y \in \{0, 1\}$. Then, the pattern generated by the constructed real-time cellular automaton 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 $P_\varphi = \{\, a^n \mid \text{there is an } m \text{ with } n = \varphi(m) \,\}$, or even $\hat{P}_\varphi = \{\, x^n \mid x = a \text{ if there is an } m \text{ with } n = \varphi(m), \text{ and } x = b \text{ otherwise} \,\}$, where $\varphi\colon \mathbb{N} \to \mathbb{N}$ 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 speed-up and closures of the set of generated strings under certain operations.

3.1 Speed-Up

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 [2]. In [6, 7] the speed-up 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 cellular automata can be sped-up from $(n + t(n))$-time to $(n + \lceil t(n)/k \rceil)$-time [1, 6, 7]. 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 speed-up results do not apply here. However, in terms of transducers that given an input of size n compute an output of size n, a speed-up result is known [4] 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 2. *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, time step $n + k$ 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 $k + 1$ time steps of M at once in every cell leaving the quiescent state s_0. In the remaining time steps, each cell shifts the contents of its ith track to its $(i-1)$st track for $2 \leq i \leq k+1$ and updates the entry in the $(k+1)$st track. Hence, every cell keeps track of the last $k + 1$ time steps.

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 where the second phase starts. In this phase, all information to update the entries in the $(k + 1)$st track is available taking into account both neighbors. Hence, the simulation of the last $k+1$ time steps can be continued. At time step n in M' we have eventually simulated the desired $(n + k)$th time step of M in the $(k + 1)$st track. To extract this information in the nth time step of M' we start in the beginning in another track an instance of the FSSP. If the FSSP would fire at time step n, we just output the $(k + 1)$st track that would be calculated at time step n. □

Next, we turn to show 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 [4]. Before we turn to the adaption to our notion, we sketch the underlying idea of [4] to speed up the part of the time complexity beyond real time by a factor 2 (see Figure 2). The basic idea is to compress the input of length n into the $\frac{n}{2}$ cells $\frac{1}{4}n + 1, \frac{1}{4}n + 2, \ldots, \frac{3}{4}n$, to simulate the given $(n + t(n))$-time cellular automaton 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 2. 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 diagram are in the quiescent state. The simulation in the blue area is with double speed. Concerning the time,

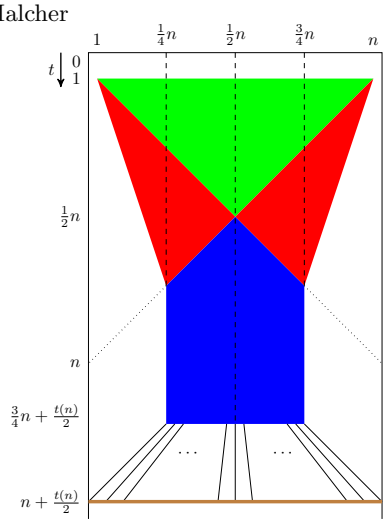

Fig. 2. Principle of speeding up the part of the time complexity beyond real time by a factor 2 according to [4].

once cell $\frac{1}{2}n$ has left the quiescent state it has to perform 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 $\frac{1}{4}n$ time steps.

Theorem 3. *Let M be a cellular string generator with time complexity $n + t(n)$, where $t\colon \mathbb{N} \to \mathbb{N}$ is a function such that M is synchronizable at time step $n+t(n)$. 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 [4] 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 [4]. So, basically, the simulation sketched in Figure 2 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 time step $n + t(n)$. 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 time step 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 $\mathscr{P}_{rt}(\mathrm{CA})$. We start with the Boolean operations.

Proposition 4. *The family $\mathscr{P}_{rt}(CA)$ is closed under intersection. It is neither closed under union nor under complementation.*

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 also the reversal of a real-time pattern can be generated by a real-time string generator.

Proposition 5. *The family $\mathscr{P}_{rt}(CA)$ is closed under reversal.*

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

Proposition 6. *The family $\mathscr{P}_{rt}(CA)$ is not closed under arbitrary homomorphism, but is closed under length-preserving homomorphism.*

Proof. For the non-closure under arbitrary homomorphism we consider the language $L = \{a, bb\}$, that 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 cellular automaton generating the pattern $P \subseteq \Sigma^*$, where $\Sigma \subseteq S$. Moreover, let $h\colon \Sigma^* \to \Gamma^*$ be a length-preserving homomorphism. For the construction of a cellular automaton 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 cellular automaton M' generating $P(M') = h'(P(M)) = h'(P) = h(P)$ in real time plus one that is subsequently sped up with Lemma 2 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 time step n, when the FSSP fires, we generate in the third track the homomorphic image under h' of the first track. At time step $n+1$, we check in every cell whether its state in the first track, that is in the simulated configuration of M at time n, would remain the same also in the next time step, that is, in the simulated configuration of M at time $n+1$. If so, each such cell enters the state from the third track, 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 instable configuration, then M' enters an instable configuration as well. Therefore, it is ensured that M' generates no string as well in such cases. □

4 Decidability Problems

The first decidability problem we are dealing with is the question whether a given cellular automaton 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 cellular automata 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 [5] 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 $\sqcup$. Then a configuration of M can be written as a string of the form $T^*(Q,T)T^*$ such that $x_1x_2\cdots x_i(q,y)x_{i+1}x_{i+2}\cdots x_n$ is used to express that M is in state q, scanning tape symbol y, and $x_1x_2\cdots x_iyx_{i+1}x_{i+2}\cdots x_n$ 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 $(q_0, \sqcup)$, 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 7. *Let M be a Turing machine. Then the pattern P_M is generated by some cellular automaton in real time.*

Basically, Proposition 7 is already the reduction of the problem to decide whether a Turing machine *does not halt* on empty input to our emptiness problem.

Theorem 8. *Given a real-time cellular automaton M, it is not semidecidable whether $P(M)$ is empty.*

Proof. Given an arbitrary Turing machine M', by Proposition 7 we construct a real-time cellular automaton 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 9. *Given two real-time cellular automata M_1 and M_2, it is neither semidecidable whether $P(M_1) = P(M_2)$ nor whether $P(M_1) \subseteq P(M_2)$.*

Theorem 10. *Given a real-time cellular automaton M, it is neither semidecidable whether $P(M)$ is finite nor whether $P(M)$ is infinite.*

Proof. Let M' be an arbitrary Turing machine.

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

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

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 11. *Given a real-time cellular automaton 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.

Similar as in the proof of Theorem 10 we set

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

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

□

Acknowledgements. The authors would like to thank the anonymous referees for their useful comments and remarks.

References

1. Bucher, W., Čulik II, K.: On real time and linear time cellular automata. RAIRO Inform. Théor. 18, 307–325 (1984)
2. Choffrut, C., Čulik II, K.: On real-time cellular automata and trellis automata. Acta Inform. 21, 393–407 (1984)
3. Dyer, C.R.: One-way bounded cellular automata. Inform. Control 44, 261–281 (1980)
4. Grandjean, A., Richard, G., Terrier, V.: 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 (2012)
5. Hartmanis, J.: On the succinctness of different representations of languages. In: International Colloquium on Automata, Languages and Programming (ICALP 1979). LNCS, vol. 71, pp. 282–288. Springer (1979)
6. Ibarra, O.H., Kim, S.M., Moran, S.: Sequential machine characterizations of trellis and cellular automata and applications. SIAM J. Comput. 14, 426–447 (1985)
7. Ibarra, O.H., Palis, M.A.: Some results concerning linear iterative (systolic) arrays. J. Parallel Distributed Comput. 2, 182–218 (1985)
8. Kari, J.: Universal pattern generation by cellular automata. Theor. Comput. Sci. 429, 180–184 (2012)
9. Kutrib, M.: Cellular automata – a computational point of view. In: New Developments in Formal Languages and Applications, chap. 6, pp. 183–227. Springer (2008)
10. Kutrib, M.: Cellular automata and language theory. In: Encyclopedia of Complexity and Systems Science, pp. 800–823. Springer (2009)
11. Kutrib, M., Malcher, A.: One-dimensional cellular automaton transducers. Fundam. Inform. 126, 201–224 (2013)
12. Kutrib, M., Malcher, A.: One-dimensional pattern generation by cellular automata. Accepted at ACRI 2020. LNCS, Springer, to appear.
13. Smith III, A.R.: Cellular automata and formal languages. In: Symposium on Switching and Automata Theory (SWAT 1970). pp. 216–224. IEEE (1970)
14. Waksman, A.: An optimum solution to the firing squad synchronization problem. Inform. Control 9, 66–78 (1966)
15. Wolfram, S.: Random sequence generation by cellular automata. Adv. Appl. Math. 7, 123–169 (1986)