Different Kinds of Neighborhood-Varying Cellular Automata

Jean-Christophe Dubacq

October 24, 2013

Abstract

The mainframe of this work has been the study of cellular automata as very simple SIMD systems. We introduce some variations over the basic definition of cellular automata in order to compare these new models with other models of cellular automata. The neighborhood-varying cellular automata is shown to be as powerful as time-varying cellular automata. The dynamically reconfigurable cellular automata are more interesting because they effectively reduce the running-time for some normal CA computations. Then some possible applications of these works are browsed.

Contents

1 Introduction

In this report, we want to study the properties of cellular automata as very simple interconnected computers, for which it would be possible to change its own topology. This has already been used by Wu and Rosenfeld in [8], and the initial focus of this work was the study of *graph cellular automata*, which are an extension of cellular automata. The interesting possibilities of these automata in graph recognition induced the desire of having a reconfiguration of some networks, in other that were better fitted for other computations.

The first idea was to have a certain topology (array or line) and to change the neighborhood with the time. As some work had already been done with automata changing of function according to a time table, we compared the resulting possibilities of these two classes of automata, with adding the combination of both, *i.e.* automata that can change both their neighborhood and their functions, as time varies.

The second idea was to have only local transformations, as cellular automata are essentially dealing with local processes. So we would allow one cell to use its neighbors or the neighbors of its neighbors only. Moreover, the change would have to be embedded in the definition of the automata itself. With these constraints, it is possible to prove that these automata, that are still entirely deterministic (tough it is possible to define non-deterministic versions), are for some computations, much faster than normal cellular automata.

A choice has been made in the second part, it is to suppose one-way communication. Hence, each cell can be considered as having an infinite fan-out, such as an output bus ; meanwhile, the fan-in (number of arguments for the computation) is constant.

The other choice that could have been done would have been to suppose two-way communications. But this would be more complicated to define, as deadlocks could result from certain configurations. Hence, this possibility has not been considered for the present.

Some applications of all these new classes are exposed. The first part doesn't lead to many applications, because of the equivalence results proved in theorem 1, while the second part allows a greater number of possibilities due to the instrinsic differences with the usual models.

2 Definitions

2.1 Time-Varying Cellular Automata (TVCA)

A Time-varying CA is a CA for which the transition function can vary in the time. The connections between the cells (*i.e.* the neighborhood) remain constant. The power of the TVCA changes according to the complexity of the function that gives the number of the neighborhood for each iteration. A survey of these possibilities can be found in [4]. More specific results about the power of these automata are given in [6] and [5].

A TVCA is given by $\mathcal{T} = (d, \Sigma, N, \mathcal{L}_1, \ldots, \mathcal{L}_{n-1}, \delta_1, \ldots, \delta_n)$ where :

- $d \in \mathbb{N}$ is an integer, the dimension of the TVCA.
- Σ is the set of states of $\mathcal{T}$.
- N is the neighborhood of the CA. It is a finite subset of $\mathbb{Z}^d$.
- $\mathcal{L}_1, \ldots, \mathcal{L}_{n-1}$ are tally languages, such that

  $$\forall i, j \leq n-1, \mathcal{L}_i \cap \mathcal{L}_j = \emptyset$$

  The set of all these languages is sometimes called oracle of the TVCA because it can be seen as an implicit question to an oracle.
- $\delta_1, \ldots, \delta_n$ are n functions of $\Sigma^{\sharp N} \to \Sigma$

2.2 Neighborhood-Varying Cellular Automata (NVCA)

The idea of the Neighborhood-Varying CA is that the function will not change, but the local connections will depend on the time. The size of the different neighborhoods has to be the same, as the function is of constant arity.

A NVCA is given by $\mathcal{N} = (d, \Sigma, N_1, \ldots, N_n, \mathcal{L}_1, \ldots, \mathcal{L}_{n-1}, \delta)$ where :

- $d \in \mathbb{N}$ is an integer, the dimension of the NVCA.
- Σ is the set of states of $\mathcal{T}$.
- $N_1, \ldots, N_n$ are the neighborhoods of the NVCA. These are subsets of $\mathbb{Z}^d$, such that $\sharp N_1 = \ldots = \sharp N_n$. The neighborhoods are considered as a $\sharp N_1$-uple. It means there is an order on the neighbors, such that a permutation on a neighborhood changes it. For example, neighborhood $\{0, -1, 1\}$ is not considered as the same as $\{-1, 0, 1\}$.
- $\mathcal{L}_1, \ldots, \mathcal{L}_{n-1}$ have the same definition than for TVCA
- δ is a function of $\Sigma^{\sharp N_1} \to \Sigma$

2.3 Varying Cellular Automata (VCA)

A Varying-CA combines the properties of NVCA and TVCA. Both function and neighborhood depend on the time, through a control similar to these of NVCA and TVCA.

A VCA is given by $\mathcal{V} = (d, \Sigma, N_1, \ldots, N_n, \mathcal{L}_1, \ldots, \mathcal{L}_{n-1}, \delta_1, \ldots, \delta_n)$ where :

- $d \in \mathbb{N}$ is an integer, the dimension of the NVCA.
- Σ is the set of states of $\mathcal{T}$.

- $N_1, \ldots, N_n$ are the neighborhoods of the VCA. These need to be of the same size[1].
- $\mathcal{L}_1, \ldots, \mathcal{L}_{n-1}$ have the same definition than for TVCA
- $\forall i, \delta_i$ is a function of $\Sigma^{\sharp N_i} \to \Sigma$

2.4 Global function

We can define for each of the preceding CA, a configuration as being a mapping from $Z^d \to \Sigma$. It means, we consider we have a bunch of cells put in an infinite array of dimension d, each containing a value of Σ. The set of all configurations will be denoted by $C_{\mathcal{A}}$ for the CA $\mathcal{A}$.

Applying a CA to a configuration is considering that each cell will apply a function to a certain neighborhood to guess what will be his next state, in a synchronous way (*i.e.* all cells change of state at the same time). In our definition, the neighborhood and the function can depend on the time t. This defines a mapping from $C_{\mathcal{A}}$ into $C_{\mathcal{A}}$ that is said to be the *global function of* $\mathcal{A}$ and that will be denoted by $F_{\mathcal{A}}$

Applying a CA k times to a configuration means iterating the application of the CA from time $t = 0$ to time $t = k - 1$.

TVCA For the TVCA, the neighborhood is set. The function applied by the CA is :

- $\delta_i, 1 \leq i \leq n-1$, if $0^t \in \mathcal{L}_i$.
- δ_n, if $0^t \notin \bigcup_{i=1}^{i=n-1} \mathcal{L}_i$.

NVCA For the NVCA, the function is set. Note that it is the reason why the size of the neighborhood is a constant. The neighborhood used to compute the arguments of the function by the CA is :

- $N_i, 1 \leq i \leq n-1$, if $0^t \in \mathcal{L}_i$.
- N_n, if $0^t \notin \bigcup_{i=1}^{i=n-1} \mathcal{L}_i$.

VCA For the VCA, both function and neighborhood depend of time. The neighborhood used to compute the arguments of the function by the CA is :

- $N_i, 1 \leq i \leq n-1$, if $0^t \in \mathcal{L}_i$.
- N_n, if $0^t \notin \bigcup_{i=1}^{i=n-1} \mathcal{L}_i$.

The function applied to the cells is :

- $\delta_i, 1 \leq i \leq n-1$, if $0^t \in \mathcal{L}_i$.
- δ_n, if $0^t \notin \bigcup_{i=1}^{i=n-1} \mathcal{L}_i$.

[1] It is possible to give a definition where the sizes of the neighborhoods are not the same, but it is easier to understand with fixed-size neighborhoods.

2.5 Language recognition

The power of the different models is studied through the languages that can be accepted by the CA. For a better definition of language recognition by CA, see [3].

A given CA accepts a word by having a special cell, called the *distinguished cell*, going in a special *acceptance* state[2], being given the word as input (*i.e.* embedded in the initial configuration). In a similar way, it rejects a word by having its distinguished cell going in a *refusal* state. The input is given as the initial state of the cells[3].

A given CA is said to accept the language $\mathcal{L}$ iff it accepts any word of $\mathcal{L}$ and rejects any word not in $\mathcal{L}$.

Note that the set of states of the CA must include the alphabet over which $\mathcal{L}$ is built, but can include supplementary states.

3 Equivalence results

We are going to show that all the different classes of varying cellular automata have the same effective power of computation.

Lemma 1 *Given any set of tally languages of empty intersection $\mathcal{O} = \mathcal{L}_1, \ldots, \mathcal{L}_{n-1}$, and any TVCA $\mathcal{T}$ with oracle $\mathcal{O}$, then there exists a NVCA $\mathcal{N}$ with oracle $\mathcal{O}$ such that there exists an injection $\iota : C_\mathcal{T} \to C_\mathcal{N}$ and that $\iota \circ F_\mathcal{T} = F_\mathcal{N} \circ \iota$[4].*

Proof. Let a TVCA $\mathcal{T} = (1, \sigma, N, \mathcal{O}, \delta_1, \ldots, \delta_n)$. Let's take the following :

- $\Sigma' = \Sigma \times \{1, \ldots, n\}$, n copies of Σ.
- $N_i = (a_1, \ldots, a_k, +i, 0)$ where $N = (a_1, \ldots, a_k)$. It means that N_i is the concatenation of N with the cell located i cells to the right (or to the left, depends on convention) and the center cell.
- $\delta((q_1, j_1), \ldots, (q_{k+2}, j_{k+2})) = (\delta_{(j_{k+1} - j_{k+2} + 1) \bmod n}(q_1, \ldots, q_k), j_{k+2})$.
- $\mathcal{N}$ is a NVCA given by $(1, \sigma', N_1, \ldots, N_n, \mathcal{O}, \delta)$
- $\iota(C)$ is such that if the m^{th} cell of C in state q then the m^{th} cell of $\iota(C)$ is in state $(q, m \bmod n)$. As we will see it later, this transformation can be done in linear time by any type of CA. This is important for the section about linear-time results. See figure 2 for an example.

Note that this definition is correct for one-dimensionnal CA. But the construction can be extended easily to any greater dimension, you just have to consider the additional neighbor as being along the first coordinate axis, at distance i. For example, for a two-dimensional CA, you just consider the two additionnal neighbors as being along the x axis, and ι adds marks only in that direction.

Now let's prove that $\iota \circ F_\mathcal{T} = F_\mathcal{N} \circ \iota$. We already have that the last two neighbors provide a measure of the function that has to be used : one has just to compute the difference of the second fields modulo n, it will be the same for all cells and will depend only on the chosen neighborhood. This is because the cells are given an identification, and they can compute how far is a pointed

[2] Some other models use the fact that all cells go in an acceptance state, but it won't be used here.

[3] For example, 1101 would be encoded with four cells, each of these having either 1 or 0 as initial state

[4] ι is the function that converts a configuration of $\mathcal{T}$ in a configuration of $\mathcal{N}$. The equality means that the two automata compute similar global functions over their respective sets of configurations.

$\dots$	p_{-2}	p_{-1}	p_0	p_1	p_2	$\dots$
$\Downarrow$ Action of δ_1						
$\dots$	q_{-2}	q_{-1}	q_0	q_1	q_2	$\dots$
$\Downarrow$ Action of δ_2						
$\dots$	r_{-2}	r_{-1}	r_0	r_1	r_2	$\dots$
$\Downarrow$ Action of δ_3						
$\dots$	s_{-2}	s_{-1}	s_0	s_1	s_2	$\dots$

Figure 1: Action of $\mathcal{T}$ over a configuration.

	$\dots$	p_{-2}	p_{-1}	p_0	p_1	p_2	$\dots$
$\Downarrow$ Action of ι							
$\dots$	$p_{-2}, 1$	$p_{-1}, 2$	$p_0, 0$	$p_1, 1$	$p_2, 2$	$\dots$

Figure 2: Transfomation of configurations.

cell without knowing what the effective neighborhood is. This computation is modulo n but that is just what we need to make a difference between all the parts of the oracle.

Now let's have a look at the construction of δ : δ first computes which function it has to use, this is the $(j_{k+1} - j_{k+2} + 1) \bmod n$, and then applies the right function to the other neighbors as would have done $\mathcal{T}$. The other part of the state remains quiescent. It comes from there that the second field of the image of a cell is the same, whether you apply $\iota \circ F_{\mathcal{T}}$ or $F_{\mathcal{N}} \circ \iota$. For the first field, the equality comes from the fact that the final state is whether $\delta((q_1, j_1), \dots, (q_{k+2}, j_{k+2})$ or $\delta_l(q_1, \dots, q_k)$, where l is the number of the function to be applied, i.e. $(j_{k+1} - j_{k+2} + 1) \bmod n$ as explained above. Hence, the first field is the same for both functions. Hence we have the equality

$$\iota \circ F_{\mathcal{T}} = F_{\mathcal{N}} \circ \iota$$

$\diamondsuit$

Example Let's see how the simulation works. Let's suppose we have the transformations in figure 1.

Now observe, how the configurations are transformed in figure 2. The difference of the second numbers of any cell, and a cell in some fixed relative position (for example, two cells to the right) computed modulo 3, is a constant throughout the whole line of cells, and will be used to compute which neighborhood is being used, and so which "function" should be applied.

This is shown in figure 3. Each cell is able to compute which neighborhood is being used ; and the function δ includes both δ_1 and δ_2, which means each cell can choose the function to apply. As the neighborhood is the same for all cells, the number of the function will be computed to be the same for all cells correctly.

Corollary 1 *Any language accepted by a TVCA in linear-time (resp. with no boundary on the time) can be accepted by a NVCA with an equivalent oracle in linear-time (resp. with no boundary*

$\ldots$	$p_{-2},1$	$p_{-1},2$	$p_0,0$	$p_1,1$	$p_2,2$	$\ldots$
$\Downarrow$ Action of $\delta(\ldots,i,i+1)=\delta_1(\ldots)$						
$\ldots$	$q_{-2},1$	$q_{-1},2$	$q_0,0$	$q_1,1$	$q_2,2$	$\ldots$
$\Downarrow$ Action of $\delta(\ldots,i,i+2)=\delta_2(\ldots)$						
$\ldots$	$r_{-2},1$	$r_{-1},2$	$r_0,0$	$r_1,1$	$r_2,2$	$\ldots$
$\Downarrow$ Action of $\delta(\ldots,i,i+3)=\delta_3(\ldots)$						
$\ldots$	$s_{-2},1$	$s_{-1},2$	$s_0,0$	$s_1,1$	$s_2,2$	$\ldots$

Figure 3: Action of $\mathcal{N}$ over a configuration.

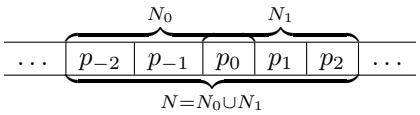

Figure 4: Merging many neighborhoods into a bigger one

on the time)[5].

Proof. This can be deduced from the preceding. As there already exists the possibility of computing in the same way with a NVCA, which is what is proved by lemma 1, we just have to make a function that computes the configuration with all the second fields correctly set. This is very easy to do in time $O(n)$, with a signal traveling at unit speed from the distinguished cell and when reaching the end of data, coming back to begin the computation by generating a "firesquad synchronisation", i.e. the effective computation will begin at time $t = 3n + 1$. So the oracle of the NVCA has to be shifted in order to get exactly the same computation. ♢

Lemma 2 *Given any VCA $\mathcal{V}$ with oracle $\mathcal{O}$, there exists a TVCA $\mathcal{T}$ with oracle $\mathcal{O}$ that simulates $\mathcal{V}$ with no loss of time.*

Proof. Let $\mathcal{V}$ be defined as $(d, \Sigma, N_1, \ldots, N_n, \mathcal{O}, \delta_1, \ldots, \delta_n)$. Then let's define $\mathcal{T}$ as $(d, \Sigma, N, \mathcal{O}, \Delta_1, \ldots, \Delta_n)$, where :

- $N = \bigcup_{i\in\{1,\ldots,n\}} N_i$, ordered in such a way that there exists $\nu_1, \ldots, \nu_n$ n functions defined like follows :
$$\nu_i : \{1, \ldots, \sharp N_i\} \to \{1, \ldots, \sharp N\}$$
The x^{th} element of N is the $\nu_i(x)^{th}$ element of N_i or $\star$ if it is not in N_i.
- Δ_i is the extension to N of δ_i, *i.e.*
$$\Delta_i(a_1, \ldots, a_k) = \delta_i(a_{\nu_i(1)}, \ldots, a_{\nu_i(k)})$$
where $a_\star$ are ignored.

In fact, it is very easy, the function Δ_i is just the projection of δ_i on a biggest neighborhood N, that contains all the neighborhoods $N_1, \ldots, N_n$ like in figure 4 and that doesn't need to change. Hence, the computation acts exactly in the same way. ♢

[5]If we allow arbitrarily large neighborhoods, then linear-time is equivalent to real-time. This is why the results about real-time are not discussed here.

Lemma 3 *Given any NVCA $\mathcal{N}$ with oracle $\mathcal{O}$, there exists a VCA $\mathcal{V}$ with oracle $\mathcal{O}$ that simulates $\mathcal{N}$ with no loss of time.*

This is the easy part. It is enough to give the same definition to $\mathcal{N}$ and $\mathcal{V}$, but the different functions of the VCA will all be equal to the function of $\mathcal{N}$.

Theorem 1 *The class of languages accepted by NVCA is exactly the class of languages accepted by VCA and exactly the class of languages accepted by TVCA with the same oracle. If a language is accepted in linear-time by one of these CA, then it is accepted in linear-time by any of the three kind of CA.*

This is just a consequence of lemma 1,2 and 3.

4 Dynamic Reconfiguration

Here, we shall try to study some properties of Cellular Automata that could, as part of their transition function, change their neighbors for the neighbors of their neighbors. The first thing that could be studied is the different kind of networks that could be obtained in such ways (see [8]). The second part would be to know whether this gives a greater power of computation to these automata (that still have a deterministic table of transition).

4.1 Formal definition

We can describe a dynamically reconfigurable cellular automata (DRCA) as $\mathcal{A} = (d, \Sigma, \mathcal{N}_I, \delta, \nu)$ where:

- $d \in \mathbb{N}$ is the *dimension* of the DRCA.
- Σ is the *set of states* of the DRCA.
- $\mathcal{N}_I$ is a finite subset of $\mathbb{Z}^d$, $\{\vec{r_1}, \ldots, \vec{r_k}\}$. It is the *initial neighborhood* of the DRCA. We shall denote by $k = \sharp\mathcal{N}_I$ the *arity* of $\mathcal{A}$.
- δ is a function from $\Sigma^k \to \Sigma$. This is the *transition function* of the DRCA.
- ν is a function from $\Sigma^k \times \{1, \ldots, k\} \to \{1, \ldots, k\} \times \{1, \ldots, k\}$. This is the *Neighborhood-transition function* of the DRCA[6].

Then we shall define a configuration of the DRCA as an application

$$C : \mathbb{Z}^d \to \Sigma \times (\mathbb{Z}^d)^k$$

which means that each cell contains both its internal state and the relative position of its neighbors. The notation can be made more explicit by having more fonctions :

- Each cell is a vector $\vec{c}$ of $\mathbb{Z}^d$
- The state of $\vec{c}$ in configuration C is $S_C(\vec{c})$.
- The i^{th} neighbor of $\vec{v}$ is given by a vector $\vec{v_i} = N_C^i(\vec{c})$ and is the cell $\vec{c} + \vec{v_i}$ (neighborhood is given in relative coordinates).

The set of all configurations of $\mathcal{A}$ will be denoted by $C_{\mathcal{A}}$.
An *initial configuration* is a configuration such that

$$\forall \vec{c} \in \mathbb{Z}^d, \forall i \in \{1, \ldots, k\}, N_C^i(\vec{c}) = \vec{r_i}$$

This means that the initial neighborhood is the same for all cell in the initial state, and is equal to $\mathcal{N}_I$. We shall denote by $N_C(\vec{c}) = (N_C^1(\vec{c}), \ldots, N_C^k(\vec{c}))$ the neighborhood of cell $\vec{c}$.

[6] see further examples on page 10 for the use of this function.

4.2 Computation with DRCA

The global function of a DRCA is a mapping from the set of configurations into itself usually denoted by the same letter as the automata, defined as follows:

$$\begin{array}{rcl} \mathcal{A} : C_{\mathcal{A}} & \longrightarrow & C_{\mathcal{A}} \\ C & \longmapsto & C' \\ S_{C'}(\vec{c}) & = & \delta(S_C(\vec{c}+N_C^1(\vec{c})),\ldots,S_C(\vec{c}+N_C^k(\vec{c}))) \\ N_{C'}^i(\vec{c}) & = & N_C^{b_i}(\vec{c}+N_C^{a_i}(\vec{c}))+N_C^{a_i}(\vec{c}) \\ (a_i,b_i) & = & \nu(S_C(\vec{c}+N_C^1(\vec{c})),\ldots,S_C(\vec{c}+N_C^k(\vec{c})),i) \end{array}$$

Hence, the computation of the new state occurs for each cell as in a normal CA, and the new neighborhood is computed in the following way :

- The neighbor from which the new i^{th} neighbor is to be taken is the first part of $\nu(\ldots)(i)$.
- The neighbor is set relatively to the preceding computed neighbor by the second part of $\nu(\ldots)(i)$ (this is the reason of the + sign, because neighborhoods are stored as relative neighborhoods).

The new neighborhood is obtained through an *indirection*.

Example If we take ν defined by $\nu(a_1,\ldots,a_k,0)=(0,0)$ and $\nu(a_1,\ldots,a_k,1)=(0,1)$, with neighbor 0 being the cell itself (the center cell), then the neighborhood will not change, as neighbor 0 becomes itself and so does neighbor 1. If we take ν defined by $\nu(a_1,\ldots,a_k,0)=(0,0)$ and $\nu(a_1,\ldots,a_k,1)=(1,0)$, with neighbor 0 being the cell itself (the center cell), then the neighborhood will not change, as neighbor 0 becomes itself and neighbor 1 becomes neighbor 0 of itself, hence nothing changes.

If we take ν defined by $\nu(a_1,\ldots,a_k,0)=(0,0)$ and $\nu(a_1,\ldots,a_k,1)=(1,1)$, with neighbor 0 being the cell itself (the center cell), Then the neighborhood will change, each cell pointing on itself and on the neighbor's neighbor. Hence, if each cell points at the beginning on the cell to the right, then next iteration, it will point two cells away to the right, then next iteration, it will point four cells away to the right, and so on. This is an important construction in the following.

If we take three neighbors, with neighbor 0 and 1 quiescent, and $\nu(a_1,\ldots,a_k,2)=(2,1)$ then neighbor 2 will "scroll" along the cells. It means each iteration, the second neighbor will point on a cell that is one cell further. This is because each cell is able to remember its initial neighborhood.

4.3 Why these definitions ?

The purpose of building such automata is to keep certain properties of CA, but giving the new model higher power, through the possibility of reconfiguration.

The first thing that is to be kept, is the local aspect of CA. That's why everything is given in relative position, and nothing is absolute. Even if the connections are given as reference to absolute numbers, the knowledge of these numbers acts only through local communications. Hence, the shift-invariance of CA is kept.

The second thing that has to be kept, is uniformity. Except for the distinguished accepting cell, and the embedding of the initial configuration in the states of the cell, the initial neighborhood is the same for all cells. It is not possible, so, to include a special graph structure in the

neighborhoods (which would certainly give much more possibilities). So the initial neighborhood has to remain the same.

Another thing is to suppress external control. The CA works with no external control, opposite to TVCA. But the power of the TVCA is strongly related to the complexity of the oracle involved. Instead, we consider the set of neighbors as part of each cell, and as such, give it a deterministic table of transition. This is the important fact : a configuration and the definition of the automata contains all the information that is necessary to compute the state after any number of iterations.

The other details come from the first three points : the constant number of neighbors, the way of referencing the new neighbors, and the way to compute the next state is made to look like CA, but including this special transition table for the neighbors. The definition becomes a bit more complicated, but the idea is still easy to grasp.

5 Speed-up with DRCA

Theorem 2 *The language* $\mathcal{L} = \{1^n, n \geq 1\}$ *can be recognized in time* $t = \lceil \log(n) \rceil + 1$.

Proof. We give an exemple of a DRCA that recognizes this language. Let $\mathcal{D}$ be $(1, \{\sharp, 0, 1, y, n\}, \{0, 1\}, \delta, \nu)$ with the following defintions for δ and ν :

- δ is defined by the following array :

Neighbor 1 $\rightarrow$	0	1	y	$\sharp$
0	0	0	0	0
1	0	1	1	y
y	is the (quiescent) acceptation state			
$\sharp$	is the quiescent state			

- ν is defined by the following rule :
  - $\forall a, b, \nu(a, b, 0) = (0, 0)$
  - $\forall a, \forall b \in \{0, 1, y\}, \nu(a, b, 1) = (1, 1)$
  - $\forall a, \nu(a, \sharp, 1) = (0, 1)$

This DRCA checks whether the pattern that is given as ${}^{\omega}\sharp(0|1)^n\sharp^{\omega}$ is a pattern made only of 1. It accepts the given pattern by the leftmost cell going to state y and it rejects the specified pattern by the leftmost cell going to state 0.

We can suppose that the leftmost cell is numbered 0, and hence, the first cell containing $\sharp$ is cell number n.

In fact, the following sentence is always checked : *at time t for cell c, neighbor* 0 *points always on c, neighbor* 1 *points on cell max*$(c + 2^t, n)$ *; the content of cell c is either* 1 *if all cells from c to max*$(c + 2^t, n) - 1$ *contained at the beginning only the symbol* 1*, y if the same range contains only* 1 *and that the cell points on cell n before the computation, and is* 0 *otherwise (except if it is* $\sharp$ *which is the quiescent state).*

First, let's check that it allows the acceptance of $\mathcal{L}$ in time $\lceil \log(n) \rceil + 1$. If the word is 1^n, then according to the preceding sentence, at iteration number $\lceil \log(n) \rceil$, the leftmost cell will point on the cell n as $2^{\lceil \log(n) \rceil} \geq n$. So at iteration $\lceil \log(n) \rceil$, the leftmost cell will be in state y.

In a second part, we have to check that the word is rejected if there is any 0 in the pattern. It is quite easy to see that the leftmost cell will eventually cover the whole pattern. Hence, if

Time	Leftmost cell	Other cells	Rightmost cell
Before k	\$	\$	\$
$\vdots$	f	q	l
Fire - 2	r	q	l
Fire - 1	r	q	s
Fire	F	F	F

Table 1: Evolution of the cells during the synchronisation

there is a zero, according to the second part of the sentence, the state of the leftmost cell will be 0. Remark that rejection of a word is made in time $\lceil \log(n) \rceil$.

Now we have to prove the assertion. It is true after the first iteration. because of the form of ν, the two rightmost cell will both point on themselves and the first $\sharp$ cell, and the other will point two cells to their right. And the other conditions are true, as δ is mainly a *and* operation between the two cells.

If we suppose it true at iteration t, it remains true at iteration $t+1$. The first part because the state can be 1 only if $c+2^t < n$ and the state of the two cells is 1 ; hence the state of all cells from c to $c+2^t+2^t-1 = c+2^{t+1}-1$ was 1 at the beginning. The y state can obviously been reached only if the good conditoins are assumed, as it needs to come from a 1 and a $\sharp$ symbols. And all other possibilities lead to state 0, so the third part of the sentence is also true.

Hence the sentence is true for all n. This concludes the proof. See also figure 5 for an example of acceptance and figure 6 for an example of refusal. $\diamondsuit$

Theorem 3 (Synchronisation Theorem) *Given a DRCA $\mathcal{D}$ with initial neighborhood containing {-1,0,1} such that before time k, all the cells reach a special state \$, then there exists another DRCA that computes the same thing as $\mathcal{D}$ and for which all cells reach a special state F simultaneously and for the first time at time k', and $k' \leq k + \lceil \log(n) \rceil + 1$.*

The preceding theorem states that if each cell knows after its computation that it has to synchronize with other cells (this is what is meant by all the cell reaching \$ before time k) then the whole line can be synchronized in time $\lceil \log(n) \rceil + 1$ since the moment where the last cell reaches the state \$.

This allows to make several computations one after another, with cells having non-quiescent states. This result is important. Even if it is still not possible to have cells in quiescent state synchronized in not-linear time (only in linear time, this is the well-known "firing-squad" theorem), it is possible to "merge" two DRCA and having a transition time of only $O(\log(n))$ between the two automata.

The principle of the proof is that each cell will try to point on a common cell that will give the signal of fire. But as the firing cell has to know if everyone is pointing on it, it will be chosen to be the leftmost cell and will try to point on the rightmost cell. When the rightmost cell gives the information that it is pointing on the leftmost one, then every cell is pointing on it, as it is the furthest possible. Hence we will call the state of the leftmost cell f (as in *first*), then r (as in *ready*), the state of the rightmost cell will evolve from l (as in *last*) to s (as in *set*) and all the other cells will remain in a q state (as in *query*). Note that in fact, the l and f states are equivalent to the query-state (see table 5)

Proof. Let $\mathcal{D}$ be a DRCA. We shall build $\mathcal{D}'$ which does the same computation than $\mathcal{D}$ except that all cell will finally reach a special state F. In fact, $\mathcal{D}'$ is the combination of two DRCA, one

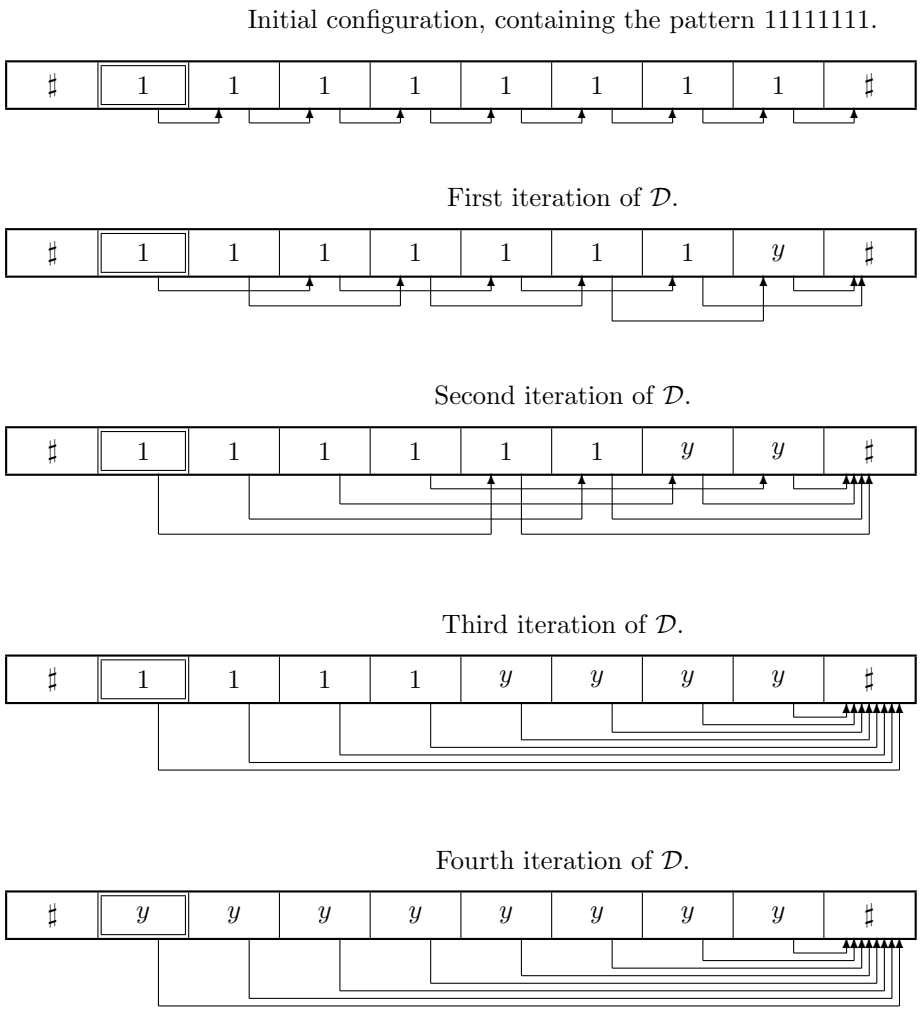

Figure 5: An exemple of DRCA

.

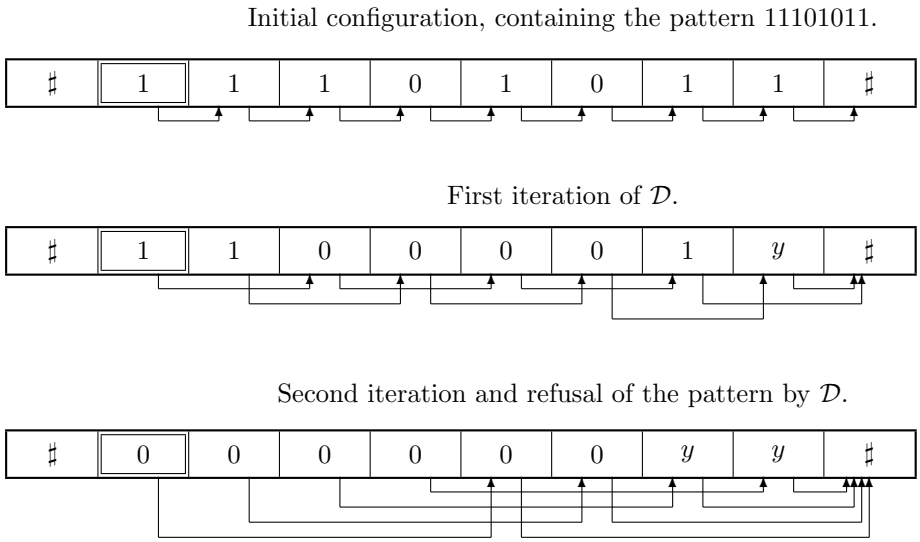

Figure 6: An exemple of refusal by the same DRCA

.

that does exactly the same work as $\mathcal{D}$, and then puts each cell in state \$ and one that begins from the state \$ as an initial state. Only this one will be described. We assume that it uses additional neighbors that remain quiescent during the work of the first automata. We shall call σ any state that is not one of the states of the synchronization automata.

We shall assume that $\mathcal{D}$ is of dimension one, but it is very easy to extend this synchronisation routine for higher dimensions.

Let's define $\mathcal{D}_S$ as $(1, \{\sharp, \sigma, \$, q, f, l, r, s, F\}, (0, -1, 1), \delta, \nu)$, with the following tables for δ and ν :

- δ is defined by following tables. We shall denote by a the state of the cell (neighbor 0), columns are for identical left neighbor (neighbor 1), rows are for identical right neighbor (neighbor 2).
- If $a = \sharp$, then the state remains $\sharp$.
- If $a = \sigma$, then it may either go in state σ or state \$ (computation phase).
- If $a = \$$:

	$\sharp$	$\$, q, \sigma$	f
$\sharp$	F	l	s
$\$, \sigma, q$	f	q	q
l	r	q	q

- If $a = q$:

	$\$, \sigma$	f, r, q
$\$, \sigma$	q	q
l, q	q	q
s	F	F

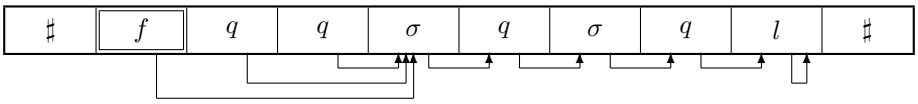

Figure 7: Sample scheme for synchronisation theorem
Only right neighbors are shown.
See how the leftmost cell has the leftmost right neighbor.
σ is any computational state.

- If $a = l$, then the state remains l, unless the left neighbor is in state f, in which case the state become s.
- If $a = f$, then the state remains f, unless the right neighbor becomes either s or l. In the first case, the state becomes F, in the second case, the state becomes r.
- If $a = r$ then it remains r, unless the left neighbor becomes s, in which case next state is F.
- If $a = s$, then the state becomes F.
- ν is defined very easily :
  - Neighbor 0 never changes, and is always pointing to the cell itself.
  - Neighbor 1 (left neighbor) either changes to itself if the state of the pointed cell is σ or \$, changes to the cell itself if the state of the pointed cell is ♯ or changes to the left neighbor's left neighbor if the pointed cell is in state q, f, r. In fact, as the left neighbor of the first cell is itself, it doesn't matter whether the neighbor keeps the same value or keeps going to the left, as it points to the same cell.
  - Neighbor 2 (right neighbor) acts symetrically. It changes to itself if the pointed cell is σ or \$, to the cell itself if it is the rightmost cell (i.e. if the pointed cell is in state ♯), and to the right neighbor's right neighbor in all other cases.

Correctness We will first prove that all the cells reach simultaneaously and for the first time the state F.

There are two special cases : when there is only one cell, it reaches the state F just after the state \$. And if there are only two cells, it works also.

From now on we will suppose that there are at least three cells. Every cell will expand its neighborhood each iteration, till it reaches either one of the border or a σ or \$ cell. As before time k, all cell will be in state \$, after time k, all cells will let their neighborhood grow till it reaches both ends.

The next step of the proof is that all the cells will always point further than the first cell. For example, all cells will always have a right neighbor more to the right than the one of the leftmost cell (see also figure 7). The same thing occurs for the other direction. We shall denote by $r_t(x)$ the right neighbor of cell x at time t.

Given a and b two cells, with $a < b$, either b is pointing on a q cell and in this case, $r_t + 1(a) \leq r_t(r_t(a))$, $r_t + 1(b) = r_t(r_t(b))$ and the order is preserved. If b is not pointing on a q cell, then either it is pointing on the last cell, either it is pointing on a σ or a \$ cell, and none of the cells

to the left of that one could possibly point further. Hence, $r_{t+1}(a) \leq r_t(b) = r_{t+1}(b)$. The proof is symetric for the left neighbor.

So, after a certain time (which will be proved to be lesser than $k + \lceil \log(n) \rceil$) the first cell points on the rightmost one. So it reaches a state r. Only after that, the rightmost cell can reach the state s. At that time anyway, all the cells are pointing on the rightmost cell. So when the state of the rightmost cell changes to s, everyone will at next turn go into state F.

Running-Time It can be easily proved by induction, in the same way than for theorem 2. After time k, we will prove that $r_{t+k}(c) \geq \max(n-1, c+2^t)$. This is true at time k. And from there on, $r_{t+1}(c) = r_t(r_t(c)) \geq \max(n-1, r_t(c)+2^t) \geq \max(n-1, c+2^t+2^t)$, and this proves the induction. The symmetrical argument stands for the left neighbor. Hence, at time $k + \lceil \log(n) \rceil + 1$, the first cell will be in state r and the fire will take place at time $k' = k + \lceil \log(n) \rceil + 3$.

$\diamondsuit$

6 Applications

6.1 Applications of VCA

The application of VCA is to have a small number of neighbor and a smallest function for some kind of computation. This works mainly in dimension 2, where the neighborhood can sometimes be reduced as far as the square root of the initial neighborhood. This is especially useful for image processing, which is a large area of applications for all systolic systems.

Theorem 4 *Given a cellular automaton $\mathcal{A}$ of dimension 2 under its totalistic form, with weight array $\mathcal{W}$ of size $n \times n$, it is possible to reduce the number of neighbors to $2n-1$ with only doubling the computation time.*

Proof. We will first suppose that the rank of $\mathcal{W}$ is 1, *i.e.*

$$\mathcal{W} = \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} (b_0 \dots b_{n-1})$$

The set of neighbors of the original automata is a $n \times n$ array, and the state $s_{t+1}(c)$ of each cell at time $t+1$ is equal to :

$$s_t(c) = \Phi \left(\sum_{\substack{i=k_0 \\ j=k_1}}^{\substack{i=k_0+n-1 \\ j=k_1+n-1}} \mathcal{W}_{i,j} s_t(c + (i,j)) \right)$$

Usually, $n = 2m+1$, and $k_0 = k_1 = -m$ (array centered on the cell itself).

We shall compute the sum first across the columns, and second across the lines and apply Φ to the result. Hence, the intermediary state in each cell will be $\sum_{i=0}^{i=n-1} a_i s_t(c + (i + k_0, 0))$ and the final state of the cell will be $\Phi\left(\sum_{j=0}^{j=n-1} \sum_{i=0}^{i=n-1} a_i b_j s_t(c + (i + k_0, j + k_1))\right)$. As $a_i b_j = \mathcal{W}_{i,j}$, the simulation is correct. And the number of neighbors is reduced from n^2 to $2n-1$.

If the rank of $\mathcal{W}$ is higher than 1, then it can be seen as the sum of arrays of rank 1. Then all the computations happen at same time for all the arrays of rank 1, and the final application of Φ is on the sum of all intermediary results. The number of neighbor is still $2n-1$, but the number of states has to be increased.

$\diamondsuit$

Example The typical applications of such totalistic automata are the applications in image processing : bluring, differential operators (Prewitt, laplacian operator). For exemple, a typical bluring operator of size 5×5 can be given by the following array :

$$\mathcal{W} = \begin{pmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 4 & 8 & 16 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \\ 2 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 4 & 2 & 1 \end{pmatrix}$$

6.2 Applications of DRCA

The set of languages (or patterns if we consider the two-dimensional case) which can be recognized by DRCA in time $O(\lceil \log(n) \rceil)$ is quite interesting because the same languages require time $O(n)$ for normal CA.

Theorem 5 *The* fractal patterns *can be recognized in time* $O(\lceil \log(n) \rceil)$ *by DRCA.*

First, let's define what is a *fractal pattern*. A fractal pattern may be defined on an array of dimension 1 or 2, or even greater, and is given by a set of states, an initial state and rewriting rules. Each instantiation of a fractal pattern is given by its *depth* : It is the number of times each symbol will be transformed using the rewriting rules.

Example The Thue-Morse pattern is given by two states $\{a, b\}$, and the following rules :

$$\begin{aligned} a &\longrightarrow ab \\ b &\longrightarrow ba \end{aligned}$$

The initial configuration beeing a, the following words are the configuration of depth 1,2,3 and 4 : *ab, abba, abbabaab, abbabaabbaababba.*

The equivalent in dimension 2 is splitting a square in four subsquares. The patterns that can be obtained through that way are well known : Sierpinski carpet, Thue-Morse pattern, chess board.

For more about the fractal patterns, see for example [1].

Proof. The DRCA that will recognize the language has the "basic unit" as initial neigborhood. For example, in dimension 1, the initial neighborhood will be $\{0, 1\}$, in dimension 2, it will be $\{(0,0), (0,1), (1,0), (1,1)\}$. As in previous DRCA, the rule for changing neighbors will be very simple : if the i^{th} neighbor is in a quiescent state, then don't change anything. Else the i^{th} neighbor becomes the i^{th} neighbor's i^{th} neighbor. This creates a tree for the cells, and in dimension 2, builds a *quadtree* instead of a tree.

As the pattern is built along a quadtree, it is very simple to recognize a specific pattern. The set of states is of size $2^k + 1$, where k is the number of internal states for the pattern, and a state either means "My subtree can be generated with the following states as initial states" (this gives exactly 2^k different states), and a special state that means "My subtree doesn't belong to the pattern". The state-transition table is built according to the rewriting rules. For the Thue-Morse code, the state-transition table would be :

Cell	Right neighbor	resulting state
a	b	a
b	a	b
a	a	$\perp$
b	b	$\perp$

Note that this is a special case, because none of the rules are ambiguous, so that no ab state is needed. All the lines of the table containing $\perp$ have obviously the result of $\perp$ (if a subtree can not be obtained through any initial state, then the tree can not be obtained too).

This DRCA admits the patterns that are part of the language by having its leftmost (and topmost for dimension 2) cell going in the initial state of the tree, and by finding it explore the whole pattern (when its neighbor is a quiescent cell). It rejects a tree by having the same cell going in the $\perp$ state. The set of patterns accepted comes from the definition of the DRCA, and is the requested fractal pattern. And any pattern that is an instanciation of the fractal pattern will be recognized, as the DRCA will try to reduce the depth of the given pattern.

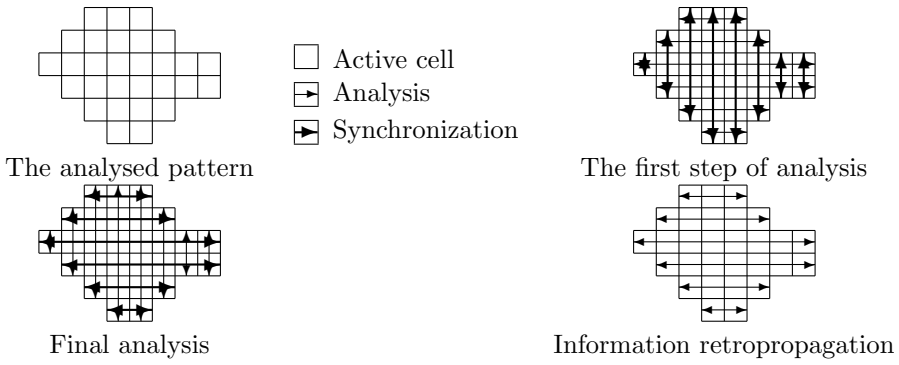

Figure 8: Example of pattern analysis

Remark The final accepting symbol has to be different from the "possible subtree" symbols. Else the right branch of each tree could not be of the same depth than the left one. If the accepting symbol is encountered in a further evolution, then it must be taken as beeing a refusal symbol.

If the accepting symbol is considered as a "possible subtree" symbol, then the right branch can be smaller than the left one. This is what is done when recognizing the language $\{1^n\}$ in theorem 2. In fact, this is the pattern given by the rule $1 \longrightarrow 1\ 1$ with initial state 1, but as a branch can be shorter than another, the tree can count as many 1s as necessary. $\diamondsuit$

6.3 Application of the synchronisation theorem

This theorem is meant to merge two DRCA easily. Merging two automatas is useful for some computations. For exemple, we described an automata that could identify whether its input pattern was uniform or not. In a later section, we described how some fractal patterns could be identified. But It is possible to want to identify a certain pattern across the lines, and then identifying the resulting pattern across the columns.

For example, if we have an image of different colours, indexed by numbers (e.g. 0 for background, 1 for blue and 2 for red), we might be interested in knowing if the connexe component is of a uniform color. This can be done by a simple method described below (see also figure 8) if the components are considered simple enough[7] :

1. Each cell looks at its line, checking if there are only cells of its color. Then it waits to be synchronized with the cells of its columns.

2. After synchronization, each cell repeats the same operation on the columns. If there is an error, at least one column of cells is in refusal state.

3. All the cells begin a new synchronisation with the cell of its lines, and checks whether there is at least one cell on its line in refusal state, as in step 1.

4. After that, all the cells know whether their component is uniform or not.

The synchronisation is important there. It allows all the cells to restart the analysis only when all the cells have already finished their previous computation. As the time of the synchronisation is equivalent to the time of the effective computation, the $O(\lceil \log(n) \rceil)$ running-time is kept.

[7] In fact, step 1 and 2 have to be repeated a sufficient number of times for more complicated components, as well as step 3 and step 2 to convey the information back.

7 Conclusion

This report presents the results about a new variation of CA.

It has been shown in this report that the concept of CA having varying neighborhood is interesting, especially if we consider the possibility of a dynamic reconfiguration, as in the case of DRCA.

There remains many open problems. For example, the Cellular Graph Automaton are capable of analysing their own graph structure as in [9, 10]. The question whether it is possible or not to use DRCA to solve such problems, or at least to configure themselves in a specifed graph structure (see also [8]) may be very interesting.

Another problem is to know if the DRCA are always more powerful than the corresponding CA. We saw in theorem 2 that the running-time could be decreased from $O(n)$ to $O(\lceil \log(n) \rceil)$. But is that speed-up always possible is still an unsolved problem.

The initial purpose of these automata was in fact to generate fault-tolerant algorithms, as in [7, 2]. The synchronisation theorem perhaps gives an answer to that question as giving the possibility to check the integrity of a cellular space in time $O(\lceil \log(n) \rceil)$ instead of linear-time. But more specific algorithms could be developed. Moreover, the synchronisation theorem itself is "tolerant", because it doesn't need to have all the cells finishing their computations at the same time.

The power of these automata should be carefully studied ; and the possibility of physical implementations too. Whether these variations of CA can model existing computers, like the Connection Machine is a matter which can be probed into. These automata have a better use of the paralellism of SIMD machines ; this could lead to interesting algorithms, in image processing for example, because of the intrinsec presence of a quadtree in the building of these automata.

Acknowledgement This work is carried out during my visit to Department of Computer Science ans Engineering, Indian Institute of Tehcnology, Madras, India on my internship. I thank Professor Kamala Krithivasan and Meena Mahajan for several helpful technical discussions at various stages of the development of the report. Last but not least, I thank all the Theoretical Computer Science Lab students for their warmth and help throughout my stay.

References

[1] J. Berstel and M. Morcrette. Compact representation of patterns by finite automata. In *Proceedings of PIXIM 89*, 1989.

[2] M. Harao and S. Noguchi. Fault tolerant cellular automata. *Journal of Computer and System Sciences*, 11:171–170, 1975.

[3] Alvy Ray Smith III. Real-time language recognition by one-dimensional cellular automata. *Journal of Computer and System Sciences*, 6:233–253, 1972.

[4] M. Mahajan. *Studies in the Language Classes Defined by Different Types of Time-Varying Cellular Automata*. PhD thesis, Indian Institute of Technology, Madras (India), 1993.

[5] M. Mahajan and K. Krithivasan. Some results on time-varying and relativised cellular automata. *Intern. J. Computer Math.*, 43:21–38, 1992.

[6] M. Mahajan and K. Krithivasan. Language classes defined by time-bounded relativised cellular automata. *Informatique Théorique et Applications*, 27:403–432, 1993.

[7] H. Nishio and Y. Kobuchi. Fault tolerant cellular spaces. *Journal of Computer and System Sciences*, 11:150–170, 1975.

[8] A. Rosenfeld and A. Wu. Reconfigurable cellular computers. *Information and Control*, 50, July 1972.

[9] A. Wu and A. Rosenfeld. Cellular graph automata i. basic concepts, graph property measurement, closure properties. *Inforamation and Control*, 42:305–329, September 1979.

[10] A. Wu and A. Rosenfeld. Cellular graph automata ii. graph and subgraph isomorphism, graph structure recognition. *Inforamation and Control*, 42:330–353, September 1979.