Theoretical Computer Science 123 (1994) 199-237
Elsevier

A universal cellular automaton in quasi-linear time and its S–m–n form*

Bruno Martin

Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France

Communicated by M. Nivat
Received December 1991
Revised July 1992

Abstract

Martin, B., A universal cellular automaton in quasi-linear time and its S–m–n form, Theoretical Computer Science 123 (1994) 199-237.

In this paper, we describe a quasi-linear time universal cellular automaton. This cellular automaton is not only computation universal (in the sense of simulating any Turing machine), but also *intrinsically* universal (it is capable of simulating arbitrary one-dimensional cellular automata, even two-way). The simulation is based on a novel programming language (the *brick language*), which simplifies the recursive specifications of transition functions.

Moreover, we prove that cellular automata form an *acceptable programming system* for parallel computation, thus providing an *S–m–n* theorem for cellular automata. This allows us to apply well-known results of the general theory of computation to cellular automata and might give a practical framework for studying the structural complexity of cellular automata computations.

Introduction

It is a well-known result that cellular automata are capable of universal computation. To get the computation-universality, one has just to simulate a universal Turing machine. This simulation has been first exhibited by Smith [12]. He showed that cellular automata in one spatial dimension are capable of universal computation requiring only eighteen states per cell. More recently, Lindgren and Nordhal [6] have

Correspondence to: B. Martin, LIP-IMAG, CNRS, Ecole Norm. Supérieure de Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France. Email: bmartin@lip.ens-lyon.fr.

* This work was supported by Esprit Basic Research Action Working Group "*Algebraic and Syntactic Methods In Computer Science*".

SSDI 0304-3975(92)00076-U

stated more precisely the simulation given by Smith. They prove the existence of a computation-universal unidimensional cellular automaton with seven states per cell in the nearest-neighborhood and decrease to four states per cell if the neighborhood is extended. To that end they simply simulate a small universal Turing machine. If such simulations are interesting in terms of machine state-complexity, it is not so in terms of parallel computation. The great drawback is that the simulation's speed is bounded by the speed of the Turing machine.

Wolfram [14] was the first to make an incursion into the world of universal computation of unidimensional cellular automata without referring to the Turing machines. He defined four classes of behavior of cellular automata and conjectured that class four contains cellular automata capable of universal computation. The first authors who solved the problem of the self-referring (*intrinsic*) universal computation by a one-dimensional cellular automata were Albert and Culik [1]. Their cellular automaton can simulate any one-way and "totalistic" cellular automaton with only eighteen states per cell. *One-way* means that, instead of sending a flow of signals resulting from each cell in both directions, the flow is restricted to being sent only in one direction (the two models have been proved to be equivalent [4]). *Totalistic* cellular automata correspond to a normal form for cellular automata with the help of a numbering that gives, in a certain sense, a Gödel numbering of cellular automata.

We present here a new intrinsic universal unidimensional cellular automaton which improves the complexities in space and in time of the cellular automaton proposed by Albert and Culik. Moreover, it can also simulate two-way cellular automata, requiring only that the transition function be described in totalistic form.

Furthermore, our cellular automaton can also be adapted to get the composition of programs leading to the definition of an *acceptable programming system* proved equivalent to the formalism of Blum [3] by Machtey and Young [7]. Then, as a consequence, it also supports a form of the well-known S–m–n theorem. This allows us to apply well-known results of the general theory of algorithms to cellular automata.

In order to describe the evolution, we introduce the notion of a *brick*. A brick can be interpreted as a procedure in a high-level programming language. Such a formalism leads to a simple description of the behavior of a cellular automaton.

1. Preliminaries

1.1. Definitions

Definition 1.1. A *cellular automaton* is a doubly infinite array of identical cells indexed by $\mathbb{Z}$, the set of integers. Each cell is a finite state machine $C=(Q, \delta)$, where

- Q is a finite set, the set of states,
- δ is a mapping $\delta: Q \times Q \times Q \to Q$.

The mapping δ, often named *local transition function*, has the following meaning. The next state of the ith cell at time t is a function of the following states: its left neighbor (the cell $i-1$) at time $t-1$, its own state at time $t-1$ and the state of its right neighbor (the cell $i+1$) at time $t-1$. In other words, if $c(i, t)$ denotes the state of the ith cell at time t, then the following equality holds:

$$c(i, t)=\delta(c(i-1, t-1), c(i, t-1), c(i+1, t-1)).$$

We have a special state for cellular automata, namely the *quiescent state*, often given as q. Its particularity is $\delta(q, q, q)=q$.

We observe that, in general, the set of states of the cellular automaton has no structure. Below, we focus on a particular case with structured set of states. In order to structure the set of states, we identify its letters with an initial segment of the nonnegative integers. This numbering allows us to define the local transition function δ simply by a mapping of a subset X of $\mathbb{N}$ into itself.

Definition 1.2. A cellular automaton with set of states Q and transition function $\delta: Q\times Q\times Q\to Q$ is called *totalistic*, if $Q\subset\mathbb{N}$ and there exists a function $f:\mathbb{N}\to Q$ such that

$$\forall a, b, c\in Q, \quad \delta(a, b, c)=f(a+b+c).$$

The local transition function is not a function of a triple but a function of the sum of the states of the three cells. For any $i\in\mathbb{Z}$ and $t\in\mathbb{N}$,

$$\begin{aligned}&\delta(c(i-1, t-1), c(i, t-1), c(i+1, t-1))\\&\quad=f(c(i-1, t-1)+c(i, t-1)+c(i+1, t-1)),\end{aligned}$$

where the $c(i, t)$ are numbers in $Q\subset\mathbb{N}$.

We observe that totalistic cellular automata can be made to differentiate their own state from the states of their left and right neighbors. This comes from the following lemma proved by Albert and Culik [1].

Lemma 1.3. *For every cellular automaton, there exists a totalistic one which simulates the usual one without loss of time and has at most four times as many states.*

We have defined what cellular automata are and the way they evolve locally but not how to represent their global evolution.

Definition 1.4. There are two types of configurations:

(1) We call configuration of a cellular automaton any mapping $c:\mathbb{Z}\to Q$ which assigns a state of Q to each cell of the cellular automaton;

(2) we call finite configuration of a cellular automaton any mapping $c:\mathbb{Z}\to Q$ which assigns a state of Q to each cell of the cellular automaton in such a way that the nonquiescent part is finite and connected. In that case the quiescent state becomes a delimiter.

We will write the infinite repetition of the letter a of Q to the right by a^ω. Similarly, an infinite repetition of the letter a of Q to the left is denoted by ${}^\omega a$. A configuration is then denoted by a doubly infinite word of the form ${}^\omega qyq^\omega$, where letter q (the quiescent state) does not occur in word $y \in Q^*$. Similarly, a finite configuration will be written as a finite word over Q. A sequence of (finite) configurations will be called a *time–space diagram*. Time–space diagrams are also helpful as a heuristic aid for the proofs. The device is not original, it has been introduced by Minsky [9] and used, for instance, by Waksman [13], Fischer [5] and Smith [11]. With the time–space diagrams it is also useful to define some types of cellular automata in the way the number of nonquiescent cells increases.

In the general case, between two time-steps, the length of the nonquiescent part of a configuration may increase by two, by the "birth" of nonquiescent cells at the leftmost and rightmost ends. If the nonquiescent part of a configuration is restricted to grow only in one direction, the cellular automaton is called a *half-line of automata*. We can also remark that it is an easy trick to simulate the evolution of a cellular automaton by a half-line. If one would like to do that, just notice the following idea, often useful with Turing machines: the doubly infinite array of identical cells can be folded, defining in this way a central cell which can be viewed as the leftmost cell of a half-line of automata and any other cell carries its previous value plus the value of the cell which was symmetrical with respect to the central cell. For convenience we always assume that the leftmost end is numbered 0. If we do not want to let the length of the nonquiescent part of the configuration to be increased at all, we particularize the two "border" cells. In this situation we define a *segment of automata*.

1.2. Different representations of cellular automata

As for finite automata, it is possible to define a product of finitely many cellular automata (CA).

Definition 1.5. A *product of cellular automata* is a cellular automaton $\mathscr{C} = (Q, \delta)$ whose set of states is the cartesian product of finitely many sets of states $Q = \prod_{i=1}^{n} Q_i$ and whose local transition function δ is a mapping $\delta : Q \times Q \times Q \to Q$ with following definition:

$$\delta(\underline{x}, \underline{y}, \underline{z}) = (\delta_1(x_1, y_1, z_1); \delta_2(x_2, y_2, z_2); \ldots; \delta_n(x_n, y_n, z_n);)$$

for $x_i \in Q_i$ $(1 \leqslant i \leqslant n)$ and $\underline{x} = (x_1, \ldots, x_n)$, $\underline{y} = (y_1, \ldots, y_n)$, $\underline{z} = (z_1, \ldots, z_n) \in Q$.

Remark 1.6. In a CA product the transititions are in general componentwise independent. We say that the behaviors of the cellular automata which form the CA product are superimposed.

There are several ways to represent a cellular automaton. We have given in Definition 1.1 the most usual representation; however, it is also possible to define cellular automata not as functions of three elements of the set of states but as functions of three n-tuples, elements of a set cartesian product of finitely many states. The following definition may be compared with the one of a CA product given in Definition 1.5, where the behavior of the cellular automaton depends upon all the previous products of states.

Definition 1.7. We call *tuple-cellular automaton* $D=(Q, \delta)$ any cellular automaton with set of states equal to a cartesian product of finitely many finite sets and with local transition function $\delta: Q \times Q \times Q \to Q$ such that

$$\underline{a}=\delta(\underline{x}, \underline{y}, \underline{z})$$

where $\underline{x}$, $\underline{y}$, $\underline{z}$ and $\underline{a}$ are n-tuples.

We will represent such a cellular automaton as depicted in Fig. 1, where the main state corresponds to the usual state of a cellular automaton and the other "layers" to the memory of the messages which transit in each cell. We will use in the next sections a 3-tuple cellular automaton on which we superimpose, by cartesian product, another cellular automaton which synchronizes the universal behavior of the 3-tuple CA. Let us denote the first part as the *main-state* part, the second one as the *main-signal* part, the third one as the *second-signal* part, and the fourth one as the *synchronization part.*

1.3. Firing squad synchronization

The *firing squad synchronization problem* is due to Myhill (1957). It can be expressed as follows:

Given an initial line of soldiers, how can they fire at the same time knowing that the order to fire, coming from a general located at one end of the line, needs a certain constant time to propagate?

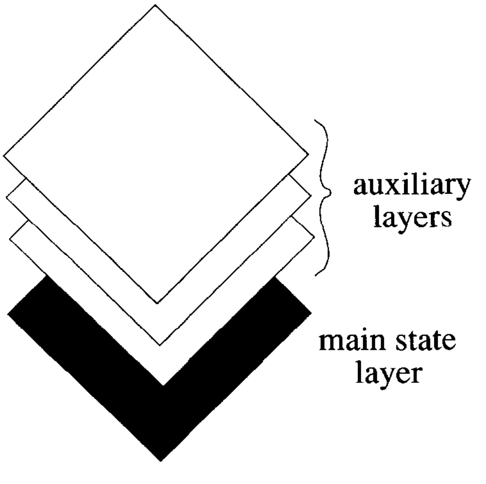

Fig. 1. A tuple cell.

Each soldier may be represented by a cell of a cellular automaton. The problem is to build a local transition function for a cellular automaton. The first answer was given in 1965 by Minsky and MacCarthy. First minimal solutions are due to Goto, Waksman and Balzer. After several years, a minimal time solution with 6 states was given by Mazoyer.

We will use the following results in order to synchronize a segment of automata [13, 2].

Lemma 1.8 (Firing squad lemma). *There exists a CA* $Z=(Q, \delta)$ *with special symbols* ç, $\$\in Q$ *and a quiescent state* q *such that*

$$\Delta^t({}^{\omega}q0^{n-1}\text{ç}q^{\omega})={}^{\omega}q\$^n q^{\omega}$$

for $t=2n-2$ *and* $\Delta({}^{\omega}q0^{n-1}\text{ç}q^{\omega})(i,t)\neq\$$ *for* $0\leqslant t<2n-2$, *where* Δ *denotes the global transition function corresponding to the local transition function* δ.

The states have the following meaning: the cell with the special symbol ç is the general which gives the order to fire. At the end of a certain process, all the "soldiers" are ready to fire (that is, the cells with special state \$).

It is also possible to improve the time of synchronization if the initial line has not only one general but two. In that case we have the following result [10].

Lemma 1.9 (Firing squad lemma with two generals). *There exists a CA* $Z=(Q, \delta)$ *with special symbols* ç, $\$\in Q$ *and a quiescent state* q *such that*:

$$\Delta^t({}^{\omega}q\text{ç}0^{n-2}\text{ç}q^{\omega})={}^{\omega}q\$^n q^{\omega}$$

for $t=n$ *and* $\Delta({}^{\omega}q\text{ç}0^{n-2}\text{ç}q^{\omega})(i,t)\neq\$$ *for* $0\leqslant t<n$, *where* Δ *denotes the global transition function corresponding to the local transition function* δ.

The two ç's on the initial configuration denote the two generals. This is an immediate consequence of Lemma 1.8 (firing squad lemma).

For the proofs of the previous lemmas, the interested readers can refer to [13, 2, 10, 8].

1.4. Two types of universality

Several definitions of universality occur in the literature. Informally, universal behavior for any machine is the ability to mimic the behavior of any other machine with respect to the Church–Turing thesis. The universality of cellular automata was first pointed out by Smith [7]. This process was an external process. It used the simulation of a universal Turing machine by a cellular automaton. Such a universal cellular automaton has about 10 states but it is not a notion of universality intrinsic to cellular automata theory.

In 1987, Albert and Culik [1] exhibited a universal cellular automaton which works in quadratic time with no more than 14 states capable of simulating any given totalistic one-way cellular automaton on any initial configuration. This cellular automaton solves the problem of universality intrinsic to the cellular automata theory.

Minsky [11] describes how a universal machine works in the following terms: "*The universal machine works by operating on the description of another machine. It interprets such a description step by step to imitate the behavior of the other machine*". Our universal automaton does not behave much differently from the machine described above.

We begin this section with the notion of computability used in the rest of the paper.

Definition 1.10. Let D be a finite set, f be any partial recursive function defined on S^* and A a cellular automaton with set of states $Q=\{q_0, q_1, \ldots, q_k\}$. A computes f if there exist two functions g from S into $\{1,2,\ldots,k\}$ and h from $\{1,2,\ldots,k\}$ into S such that for any word $s=s_1s_2\ldots s_n$ of S^*, the evolution of the half-line of automata A on the initial configuration $q_{g(s_1)}, q_{g(s_2)}\ldots, q_{g(s_n)}q^{\omega}$ leads to a 1-periodical configuration $q_{i_1}, q_{i_2}, \ldots, q_{i_m}, q^{\omega}$ with $f(s)=h(q_{i_1})h(q_{i_2})\ldots h(q_{i_m})$.

Definition 1.11. A cellular automaton is *computation-universal* if it can compute any partial recursive function. More precisely, if f is any recursive function and ϕ its encoding for cellular automata and x an integer in the domain of f, we say that a cellular automaton computes the function f on the entry x if, at a certain time the result code $[[\phi, f(x)]]$ appears as the configuration on the cellular automaton. Its evolution has stabilized and the result remains as the unchanging configuration.

Remark 1.12. In particular, a cellular automaton is computation-universal if it simulates a universal Turing machine.

The above notion of universality is somewhat unsatisfactory, as it gives a notion of universality of cellular automata in terms of Turing machines. We prefer an internal notion of universality.

Definition 1.13. Let A be the set of cellular automata and $\mathscr{B}=\{(A, C)$: $A\in A$ and C is a finite configuration of $A\}$. A cellular automaton U is said to be *intrinsically universal* if there exists an injective mapping $\phi: X\subset\Gamma\to\mathscr{B}$ where Γ denotes the set of the configurations of U. The mapping ϕ is such that

- it sets up a correspondence between the configurations of U and the configurations of A at any time of the simulation of A by U, i.e.

  $$\phi(\tilde{\mathscr{C}}_0)=(A, C) \;\Rightarrow\; \forall t\in\mathbb{N}, \phi(\tilde{\mathscr{C}}_t)=(A, C'),$$

  where C' is obtained from C in t units of time and $\tilde{\mathscr{C}}_t$ is the configuration of U obtained from $\tilde{\mathscr{C}}_0$ by simulating t steps of A;

- for each configuration of A, there exists a corresponding configuration of U.

$$\forall (A, C) \in \mathscr{B},\ \exists \phi(\tilde{\mathscr{C}}) = (A, C)$$

- both ϕ and ϕ^{-1} are recursive.

Remark 1.14. In the case of infinite configurations, ϕ becomes a function which to any function g assigns a recursive initial configuration $\tilde{\mathscr{C}}$. We then get a pair (A, C).

Remark 1.15. It is clear that a cellular automaton which is CA-universal is also computation-universal. To get computation-universality, a CA-universal cellular automaton has just to simulate any computation-universal cellular automaton.

We aim to present another CA-universal cellular automaton more efficient than the one of Albert and Culik. This efficiency is obtained by another coding of the given cellular automaton to be simulated. In the following sections we develop some material necessary for the universal cellular automaton before describing the simulation of a given cellular automaton on a given initial configuration.

2. Examples of bricks

We present in this section a new way to define the local transition function by defining the notion of *bricks*. It is an algorithmic idea for decomposing the functioning of a cellular automaton. The notion of a brick is tied with the notion of tuple cellular automaton in which the set of states is a cartesian product of sets of states, each of them having a special semantics. Bricks are put together to describe the functioning of the cellular automaton. To do that we need the notion of parallel actions and some other notions which will be detailed.

In the next section, we give the first brick which, in fact, will be useful in describing the evolution of the universal cellular automaton. Its role is to find the first occurrence of a given symbol.

2.1. The `Find_Symbol` *brick*

We can build a local transition function whose role is to find the first occurrence of a given symbol on the right side of the cell which starts the brick. The `Find_Symbol` brick receives as parameter the special symbol to be found and is initiated at any place of the half-line. The time required to find the symbol is the distance between the cell which sends the order `Find_Symbol`(a) (where a denotes the special symbol to be found) and the cell containing the first occurrence of the symbol. A two-tuple cellular automaton is required as the "search signal" emitted runs from the left to the right on the signal part of the tuple. It is initiated by the special symbol $\star$ which appears on the signal part. Its strategy is to move as quickly as possible to the right when the symbol searched is not on the main state of the tuple. Thus, its transition function is the following:

$$FS_a : Q \times Q \times Q \to Q, \quad \text{where } Q = Q_1 \times Q_2,$$

$$FS_a((b, q), (c, \star), (d, q)) = (c, q),$$

$$FS_a((b, q), (c, q), (d, \star)) = (c, q),$$

$$FS_a((b, \star), (c, q), d, q)) = \begin{cases} (c, \star) & \text{if } c \neq a, \\ (a, \surd) & \text{if } c = a. \end{cases}$$

It is clear that, when the star meets the symbol a, the move of the signal to the right stops and the cell become marked by the special symbol $\surd$. The time needed for this action is obviously the distance between the ordering cell and the cell with symbol a. A corresponding time–space diagram illustrates the `Find_Symbol` brick (see Fig. 6).

Lemma 2.1. *If, at any time, the configuration of a cellular automaton is of the form given by the doubly infinite word in the set* ${}^{\omega}q(A^+)q^{\omega}$ *where* A *denotes any finite alphabet product of* Q_1 *and* Q_2 *and where* $Q_1 \cap \{\star, \surd\} = \emptyset$ *and* $Q_2 \cap \{\star, \surd\} \neq \emptyset$, *then, the brick* FS_a *(namely,* `Find_Symbol`(a)*) marks by* $(a, \surd)$ *the first occurrence of the symbol* (a, q) *from* A^+ *on the right side of the unique occurrence of the symbol* $(b, \star)$ *with* $b \neq a$.

In the same way, it is possible to define the symmetrical operation, which is to find a given symbol located to the left of the requiring cell.

2.2. *The* `Shift_Left` *brick*

We describe in this section the way to move a \$-block without using any other property of a tuple cellular automaton than the arrival of a special signal at the beginning of the \$-block. That means that the information contained on the cell is simultaneously preserved and shifted to the right while the \$-block moves to the left. When the special signal reaches the first $\underline{\$}$ at the beginning of the block, it orders it to become a \$ and to exchange its state with the symbol which was previously at its left. Then, the \$ follows the strategy: "if my left cell does not contain a \$ or a quiescent state, I exchange my state with the content of the left cell". This ensures us that the \$-block is shifted to the left while the segment it goes through is shifted to the right. We give below the transition function of the brick assuming that the pairs (a, b) occur only at the start of the process and nowhere else:

$$SL((a, q), (\underline{\$}, \star), (\$, q)) = (a, q) = a, \qquad SL((b, q), (a, q), (\underline{\$}, *)) = (\$, q) = \$,$$

$$SL(\underline{\$}, \$, \$) = \$, \qquad SL(\$, \$, \$) = \$,$$

$$SL(\$, \$, a) = \$, \qquad SL(a, \$, b) = a, \qquad SL(c, a, \$) = \$, \qquad SL(q, \$, ?) = \$,$$

where a, b, c denote any symbol of the set of the state except the symbols q, \$ and $\underline{\$}$, q denotes the quiescent state and ? any symbol other than the quiescent state.

Clearly, the Shift_left brick works in real time plus a small little constant time. That is, to swap two segments, one of length l and the other of length t, we need $l+t+o(1)$ units of time.

Lemma 2.2. *The brick SL (namely,* Shift_Left*) swaps the finite word A^+ and the finite word $\underline{\$}\$^\star$ in real time whenever the configuration of the cellular automaton is of the form given by the doubly infinite word in the set ${}^{\omega}q(A^+)\underline{\$}\$^\star q^{\omega}$ where A denotes any finite alphabet without* \$, $\underline{\$}$ *and q.*

Its behavior is illustrated in Fig. 6, where the first diagonal represents the departure times of the \$ word and the second one their arrival times.

It is also interesting to be able to move a segment to the left (resp. to the right) and leave the place free. This can be done by the Special_Send_Left brick.

2.3. *The* Special_Send_Left *brick*

Given a certain configuration with a segment identified, we aim to send it to the left end of the half-line. In that case, all the cells which are "ready to fire" are emitted on the main signal of the cell and replaced by the $\underline{\$}\$^\star$ configuration described in Section 2.2. They move themselves to the left as long as their left neighbor on the main signal is a quiescent state and halt as soon as the main state of their left neighbor is the special marker # and the main state of the current cell is also a # (i.e. the beginning of the code segment). We give the local transition function in terms of bricks and, furthermore, as three bricks superimposed on each other with interactions.

The strategy of the Special_Send brick is the following: its cell to be emitted (which is a special symbol on the alphabet) receives a Send signal and emits its main state to the main signal of its left neighbor and substitutes the content of the main state with the special state \$. Simultaneously, it sends a special signal to the right moving at unit speed to order the other cells of the block to emit their main state to the left and replace it with the special marker \$.

The condition for emitting is that the main signal of the left neighbor of a cell has become quiescent after the passage of the message and that the cell has received a special signal. The cell can emit as soon as the main signal of its left neighbor has become quiescent. At the same time, the emission signal must be sent to the right. Thus, when an emitting cell sends something, it sends its state to the main signal of its left neighbor and the emitting signal to the main signal of its right neighbor.

With the above remarks, we can give the beginning of the local transition function of the Special_Send brick which describes the initiation of the process:

$$SS_1((a, q), (d_1, \star), (d_2, q)) = (\underline{\$}, q),$$

$$SS_1((a_{n-1}, q), (a, q), (d_1, \star)) = (a, d_1),$$

$$SS_1((d_1, \star), (d_2, q), (d_3, q)) = (d_2, \sim),$$

$$SS_1((\underline{\$}, q), (d_2, \sim), (d_3, q)) = (\$, q),$$

$$SS_1((a, d_1), (\underline{\$}, q), (d_2, \sim)) = (\$, d_2),$$

$$SS_1((d_2, \sim), (d_3, q), (d_4, q)) = (d_3, \sim).$$

Then, we describe the emission of the rest of the data block, substituted at the end by the $\underline{\$}\$^\star$ word.

$$SS_2((d_{k-1}, \sim), (d_k, q), (d_{k+1}, q)) = (d_k, \sim),$$

$$SS_2((\$, q), (d_{k-1}, \sim), (d_k, q)) = (\$, q),$$

$$SS_2((\$, d_{k-1}), (\$, q), (d_k, \sim)) = (\$, d_k),$$

$$SS_2((\underline{\$}, d_{k-1}), (\$, q), (d_k, \sim)) = (\$, d_k).$$

SS_2 is used for the rest of the data block except the special marker @, which denotes the end of the data block. For this special marker, the emitting process must stop after its emission, which implies the following special strategy:

$$SS_3((\$, q), (d_k, \sim), (@, q)) = (\$, q),$$

$$SS_3((d_k, \sim), (@, q), (h, q)) = (@, \sim),$$

$$SS_3((@, \sim), (h, q), (h_1, q)) = (h, q),$$

$$SS_3((\$, q), (@, \sim), (h, q)) = (\$, q),$$

$$SS_3((\$, d_k), (\$, q), (@, \sim)) = (\$, @).$$

When the whole data block (of length $k+2$, i.e. $dd_1d_2 \ldots d_k@$, with respect to the coding of a data block) has been emitted, we must indicate that this message runs from right to left on the first signal part of the cellular automaton until the left neighbor of the first digit of the message meets the # # part and stops above the second #. We describe now the message-transmission part:

$$SS_4((c_{j-1}, q), (c_j, q), (c_{j+1}, q)) = (c_j, q),$$

$$SS_4((c_{j-1}, m_{k+2}), (c_j, q), (c_{j+1}, q)) = (c_j, q),$$

$$SS_4((c_{j-1}, q), (c_j, q), (c_{j+1}, m_1)) = (c_j, m_1),$$

$$SS_4((c_{j-1}, q), (c_j, m_1), (c_{j+1}, q)) = (c_j, q),$$

$$SS_4((c_{j-1}, m_1), (c_j, q), (c_{j+1}, m_2)) = (c_j, m_2)$$

for $c_{j-1}c_j \neq \# \#$ and $c_jc_{j+1} \neq \# \#$. For these two special cases, the message must stop its movement to the left. Note that this does not include the case where one of them equals $\#$, this case is described in SS_3. Note also that the message is of the form $m_1qm_2q\ldots m_{k+2}q$. We describe now the case where $c_{j-1}c_j = \# \#$ or $c_jc_{j+1} = \# \#$:

$$SS_5((\#, m_1, q), (c_1, q, q), (c_2, m_2, q)) = (c_j, m_2, q),$$

$$SS_5((\#, q, q), (\#, m_1, q), (c_1, q, q)) = (\#, m_1, \circ),$$

$$SS_5((\#, q, q), (\#, m_1, \circ), (c_1, m_2, q)) = (\#, m_1, q),$$

$$SS_5((\#, m_1, \circ), (c_1, m_2, q), ((c_2, q, q)) = (c_1, m_2, \circ),$$

$$SS_5((\#, m_1, q), (c_1, m_2, q), (c_2, m_2, q)) = (c_1, m_2, q),$$

$$SS_5((\#, m_1, q), (c_1, m_2, \circ), ((c_2, m_3, q)) = (c_1, m_2, q),$$

$$SS_5((c_1, m_2, \circ), ((c_2, m_3, q), (c_3, q, q)) = (c_2, m_3, \circ).$$

We can then give the new lemma which describes more precisely the conditions for `Special_Send` brick.

Lemma 2.3. *If, at any time, the cell containing the first d which marks the beginning of the first data block receives a message* $\star$, *it initiates a process which sends to the left the data block starting from the beginning. The content of the main state of the cell is replaced by a* \$. *When the emitting signal* $\sim$ *leaves any cell it indicates that the content of its main state must be sent to the left at the next step and that the signal itself must be transmitted to the right. A cell becomes inactive, entering the special state* \$, *when it has emitted the content of its main state and transmitted the signal* $\circ$. *Then the emitted message is shifted from right to left until the first bit of the message arrives on the cell marked* $\#$ *has a left neighbor which is also the state* $\#$. *The first bit of the message stops and sends a stop signal to the right, telling the other to stop.*

The work of the `Special_Send` brick is described in Fig. 6, where the first diagonal represents the departure times of the data, the second one their arrival times at the leftmost end of the half-line of automata. We recall that this brick works together with the receiving brick which gives the stopping condition for that process.

2.4. Composition of bricks

We can now begin to build something like a "wall" with the bricks. This example gives the manner to compose the bricks together. We use the composition of bricks and all the operators on the bricks as the "cement" of the wall. We will identify a data block concatenated after the code block of the initial configuration of the universal cellular automaton and send that data block to the left. Then, we swap the \$-block which appears with the code segment.

Initially, we have a configuration given by the doubly infinite word in the set ${}^{\omega}qA^+A^+q^{\omega}$, where the first word on A^+ denotes the code segment, the second one the data segment, and q the quiescent state. We first let the `Find_Symbol_right` (d) work starting from the far nonquiescent of the word. It finishes its work in real time (i.e. length(code segment)) when it meets the first occurrence of the special symbol 'd' which denotes the beginning of the data block. Then, we enter the initial configuration of the `Special_Send` brick which sends the data block to the far left of the half-line. It finishes its movement after length(code segment)+length(data block)+1 units of time. The place it left is filled with a \$-block.

The stop signal which finishes the `Special_Send_Left` brick is followed by a `Find_Symbol` which begins as the stop signal finishes. It looks after the first $\underline{\$}$ it meets, that is the beginning of the \$-block. It is a signal initiating the `shift_Left` brick.

When the `shift_Left` brick starts to work, the code segment and the \$-block are swapped in real time. At the end, the data block remains on the main signal of the very first cells of the automaton.

All that remains to be done is to push the content of the main signal of the configuration into the \$-block which is at the beginning of the configuration. That can be done by a new simple brick which we call the `fall_Into` brick whose behavior is trivial. The operation above can be described briefly by the meta brick below:

```
Start (configuration 0)
  Find_Symbol_right (d);
Special_Send_left chained with Find_Symbol_right($);
Shift_Left;
```

We have introduced in the example the meta instruction Brick 1 *chained with* Brick 2, which says that the beginning of the second brick can be chained with the end of the first brick. The operation of this meta instruction is illustrated in Fig. 6.

When the `Special_Send_Left` brick finishes its work, it sends a signal which can be interpreted as a stop signal. It can initiate a `Find_Symbol` signal. In that case, we would have to define a simple local transition function which transforms the stop signal into a find signal when it passes the cell corresponding to the end of the work of the first brick. Such a transition rule is quite simple and we will not detail it.

2.5. *The* `Asynchronous_Send` *brick*

The strategy of the `Asynchronous_Send` brick is the following: (1) the first cell (which is a special symbol on the alphabet) receives a Send signal and emits its main state to its right neighbor main signal; (2) it underlines the content of its right neighbor main state. The underlining corresponds to the instruction "*be ready to be emitted*". This transition takes place after the state of the left neighbor has been emitted to the right.

The condition of emission is that the main signal of the right neighbor of an "underlined" cell has become quiescent after the passage of the message. The local transition function corresponding to the `Asynchronous_Send` brick is the following:

$$AS(b,q),(\#,>),(a,q))=(\#,q),$$

$$AS((\#,>),(a,q),(b,q))=(\underline{a},\#),$$

$$AS((a,q),(b,q),(\#,>))=(b,q),$$

$$AS((\underline{a}_1,m),(a_2,q),(a_3,q))=(\underline{a}_2,m),$$

$$AS((\underline{a}_1,m_1),(a_2,m_2),(a_3,m_3))=(\underline{a}_2,m_1),$$

$$AS((\underline{a}_1,q),(\underline{a}_2,m_1),(\underline{a}_3,m_2))=(\underline{a}_2,a_1),$$

$$AS((a_1,q),(a_2,m_1),(\underline{a}_3,m_2))=(a_2,q),$$

$$AS((a_1,q),(\underline{a}_2,m_1),(\underline{a}_3,m_2))=(a_2,a_1),$$

$$AS((a_1,q),(a_2,m_1),(a_3,m_2))=(\underline{a}_2,q),$$

$$AS((a_1,q),(\underline{a}_2,q),(\underline{a}_3,m_1))=(a_2,q),$$

$$AS((a_1,m_1),(a_2,m_2),(a_3,m_3))=(a_2,m_1),$$

$$AS((\underline{a}_1,m_1),(a_2,m_2),(a_3,q))=(a_2,m_1),$$

$$AS((a_1,q),(a_2,q),(\underline{a}_3,m_1))=(a_2,q),$$

$$AS((a_1,q),(a_2,q),(a_3,q))=(a_2,q),$$

$$AS((a_1,q),(a_2,q),(a_3,m_1))=(a_2,q).$$

We can then give the new lemma which describes more precisely the conditions for `Asynchronous_Send` brick.

Lemma 2.4. *If at any time the cell containing the first # which marks the beginning of the code segment receives a message "$>$" it initiates a process which sends to the right the code segment starting from the beginning. The content of the main state of the cell is not destroyed. When the # passes above any cell it underlines the content of its main state. A marked cell becomes unmarked when it has emitted the content of its main state. This can be done if the main signal of its right neighbor has just become quiescent.*

The beginning of the work of the `Asynchronous_Send` brick is described by Fig. 2. We recall that this brick works together with the receiving brick which gives the stopping condition for that process.

2.6. *The* `Receive` *brick*

The `Receive` brick works together with the `Asynchronous_Send` brick. We recall quickly that the `Asynchronous_Send` allows the message to be sent along the main signal of the cells.

The message moves as quickly as possible to the right until the first symbol of the message has arrived just at the beginning of the data block. The rest of the message must then get over the well-placed segment of the message. As it arrives in the reverse order, it must jump over the message which is at its right place and move to the right on the second signal of the cells until the main signal of its right neighbor is a quiescent state.

It can then "fall" into the main signal of the cell and take a special state.

Notice that the message jumps over the well-placed duplicated code segment and then falls at its right place. Clearly, the message which was previously reversed is placed in the right order at its place. The Receive_brick finishes its operation after the reception of the entire message. The stopping condition of the Asynchronous_Send is that the main state of the right neighbor is a special marker, for instance, denoting the end of the data block (cf. Fig. 3).

We can thus give the local transition function of the Receive brick and we will give a third brick to stop the emission. Those three bricks work together with a special meta brick which denotes the parallelization of the process. In order to give the local transition function we use the definition of the triple cellular automaton. We will describe precisely the local transition functions for each value of the triple and give the conditions necessary to evolve together.

The transitions of the main state is obvious. It is an invariant of this brick. It is necessary only to give the stopping conditions for the processes. The main signal has a behavior which is more complex. We assume with respect to Lemma 2.4 that another brick emits what is to be emitted. Thus, the nonquiescent cells on the main signal of the cells move as quickly as possible until one meets the first occurrence of

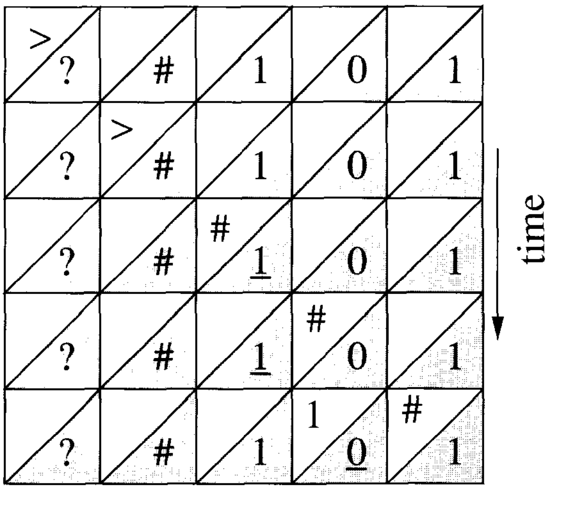

Fig. 2. The behavior of the Asynchronous_send brick.

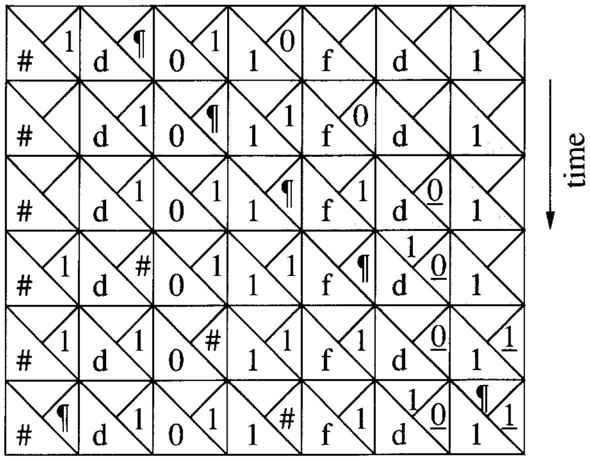

Fig. 3. The behavior of the Receive brick.

a given symbol (say f, for instance). Its local transition function is then

$$R_1(a, b, c)=a, \qquad R_1(q, b, c)=q, \qquad R_1(a, b, q)=a,$$

$$R_1(a, q, q)=a, \qquad R_1(q, q, q)=q, \qquad R_1(q, q, a)=q.$$

To give the rules of moving to the right as quickly as possible as long as they can, we give here the stopping condition for the first symbol emitted. The first element of the pair denotes the main state of the cell and the second the main signal of the cell. Thus,

$$R_2((x_1, a), (x_2, q), (x_3, q))=(x_2, a),$$

$$R_2((x_1, a), (x_2, q), (f, q))=(x_2, a),$$

$$R_2((x_1, a), (f, q), (x_3, q))=(f, a),$$

$$R_2((f, a_1), (x_1, q), (x_2, q))=(x_1, \underline{a}_1),$$

where x_1, x_2 and x_3 are different from the special marker f.

To complete the description of the behavior of the evolution of the main signal of the cells, we must give the condition where the messages continue their movement on the second signal of the cells. To that end, we need to consider triples where the two first elements are as before and the third one denotes the second signal of the cells. Thus, the local transition function can be written as

$$R_2((x_1, a, q), (x_2, \underline{b}, q), (f, q, q))=(x_2, \underline{b}, a),$$

$$R_2((x_0, a_2, q), (x_1, a_1, q), (x_2, \underline{b}, q))=(x_1, a_2, q).$$

Then, as the message is on the second signal of the cells it continues its movement to the right as quickly as possible. The local transition function is then given by the following rules where the elements are on the second signal of the cells:

$$R_3(a, b, c) = a, \qquad R_3(q, b, c) = q, \qquad R_3(a, b, q) = a,$$

$$R_3(a, q, q) = a, \qquad R_3(q, q, q) = q, \qquad R_3(q, q, a) = q.$$

The message runs on the second signal of the cells until its right neighbor contains on the main signal a quiescent state which indicates that the message can "fall" in the main signal line. The local transition function which describes that process is given by the following rules where the first element of the pair denotes the state of the main signal of a cell and the second the state of the second signal of the cell. Thus,

$$R_3((\underline{x}_0, a_2), (\underline{x}_1, a_1), (\underline{x}_2, q)) = (\underline{x}_1, a_2),$$

$$R_3((\underline{x}_0, a_1), (q, q), (q, q)) = (\underline{a}_1, q),$$

$$R_3((\underline{x}_0, a_1), (\underline{x}_1, q), (q, q)) = (\underline{x}_1, a_1),$$

$$R_3((\underline{x}_0, a_3), (\underline{x}_1, a_2), (\underline{x}_2, a_1)) = (\underline{x}_1, a_3).$$

By combining the different rules we have given, it is possible to define the local transition function. With the local transition function above, we can give the new brick.

Lemma 2.5. *If, at any time, the triple cellular automaton has on its main signal a nonquiescent segment identified with a reverse message according to Lemma* 2.4, *then if the cell containing the first symbol of the reversed message has a left neighbor containing on its main state a special symbol (for instance f, the special marker which indicates the end of a data block) the content of its main signal becomes marked and does not move to the right anymore. When the main signal of the following cells meets a right neighbor with an underlined main signal, they both continue their movement to the right by jumping over the underlined main signal on the second signal of the cell. They become marked when the main signal of their right neighbor is a quiescent state and then fall into the main signal of their right neighbor.*

We have given the strategies of the asynchronous emission of data to the right in the reverse order and how they are received and replaced in the correct order. We must now give the condition to stop the emitting process. The receiving process is trivial to end. It stops when the signal on the main signal of the cells disappears.

2.7. *The* Stop *brick*

In the previous section we have seen the importance of the Stop brick. The emission of code can be stopped when the last symbol is to be emitted. In our case, such a symbol is given by the marker of the beginning of the data segment: the '*d*' marker. When the left neighbor of the cell which contains in its main state the *d* symbol is emitted, the emission stops. This terminates the process of the `Asynchronous_Send` brick.

Lemma 2.6. *In order to stop the emission process the half-line must contain in a cell a special symbol (the d marker for instance) on the main state of the cells. When the emission process arrives at the cell located before the one which contains the special symbol, it stops the emission and ends the two bricks* `Asynchronous_Send` *and the* `Stop` *brick.*

We now give a more detailed explanation of the meta brick which allows to run different bricks and stop at least one of them at different times.

2.8. *The* `In_parallel` *meta brick*

Recalling the bricks corresponding to Lemmas 2.4–.2.6, we initially assumed the existence of a meta brick `In_Parallel` which allowed us to compose the brick previously described and let them work together. Effectively, the last three bricks cannot be realized sequentially. The `Asynchronous_Send` bricks need the `Stop` brick to end the emission of data and this implies that the `Receive` brick has finished. This combination of the three bricks is quite difficult to detail in the local transition function: the number of transitions increases so quickly that their description would become not understandable. All the cases which may occur in the action of the configuration are treated by the bricks. The bricks sort the impossible configurations and work as soon as they can.

Clearly, the `Asynchronous_Send` will not apply on a cell which has a second signal which is nonquiescent. The `Receive` brick will not work on a cell with a quiescent signal as left neighbor. These two examples show that the meta brick is capable of telling which brick is going to be applicable for any possible neighbors. One can convince himself that it is possible to write the corresponding local transition functions which are the same as the following instructions in our "brick language":

```
In_Parallel
  Asynchronous_Send//
  Stop//Receive.
```

2.9. *The* `Sum_up` *brick*

From the usual finite automaton which computes addition, it is possible to design a brick for the same purpose. To that end, we need a cellular automaton with three layers; the main state, the main signal, plus the synchronization. The main state and the main signal contain symbols in the set $\{0, 1, \#, r\}$ and the synchronization is superimposed on the sum process. Thus, we have the following result.

Lemma 2.7. *The addition of two numbers written in binary with the most significant bit first with two border cells can be done in real time by the* `Sum_up` brick.

Proof. We give the local transition function corresponding to the brick:

cell	$(0, r)$	$(1, r)$	$(0, 0)$	$(0, 1)$	$(1, 0)$	$(1, 1)$
$((-, 0), -)$	$(0, 0)$	$(0, 0)$	$(0, 0)$	$(1, 0)$	$(1, 0)$	$(0, r)$
$((-, 1), -)$	$(0, 0)$	$(1, 0)$	$(0, 0)$	$(1, 0)$	$(1, 0)$	$(0, r)$
$((-, r), -)$	$(1, 0)$	$(0, r)$	$(1, 0)$	$(0, r)$	$(0, r)$	$(1, r)$

The $-$ state corresponds to any symbol. It is easy to see that the worst time for computing the sum is equal to the length of the numbers, that is, the time to propagate the carry r from the leftmost end to the rightmost end. Thus, we synchronize the brick with the help of the firing squad lemma with the two border cells as generals. □

3. Coding the transitions and the configurations

As in [1] we will encode the transition function and the initial configuration of the cellular automaton to be simulated. Indeed, if we refer to Definition 1.13, CA-universality is obtained by simulation of the transition function starting on an initial configuration. The universal cellular automaton must then contain a description of those two parameters. It is the same idea as used for constructing a universal Turing machine.

In the present section we give the binary coding of the transition function of the cellular automaton to be simulated by the universal one.

Initially, we assume that the cellular automaton A to be simulated is given by its transition function given in a totalistic form with respect to Definition 1.2 and by its initial configuration (that is the nonquiescent part of its cells) given in nontotalistic form.

There is no loss of generality in assuming that the transition function of the cellular automata to be simulated is given in additive form. Lemma 1.3 ensures that it is always possible to transform any cellular automaton into a totalistic one.

3.1. Coding the transition function

The transition function of the cellular automaton A to be simulated is supposed to be in totalistic form. That is, the transition function is given as $f: \mathbb{N} \to \mathbb{N}$. We have then some pairs of data $(i, f(i))$ according to the natural definition of a function with the i's ordered as usual. The local transition function is then defined as the set:

$$f := \{(i, f(i)) \colon i \in \{1, 2, \ldots, 3 \times n\}\}$$

where n denotes the number of states of the cellular automaton.

A transition of the cellular automaton can be divided into two steps: first, the cellular automaton computes the sum of its own state with the state of its left neighbor and with the state of its right neighbor for each nonquiescent cell and for the two quiescent bordering cells of the configuration.

This is the reason why we need to have numbers of the interval of $\mathbb{N}$ comprised between 1 and $3n$ in the description of the local transition function of the cellular automaton to be simulated. The result of this sum can be interpreted as an i of the set f. The cellular automaton then reads the contents of $f(i)$ and replaces the old state by the image of the sum by the transition function.

We will first describe the set f given previously by a word of the form

$$\#(\# x \P y)^{3n} \#$$

such that

$$x = \{Bin(i) \text{ written with } \lceil \log_2 3n \rceil \text{ digits: } i \in Q\} \textit{ and}$$

$$y = \{Bin(f(i)) \text{ written with } \lceil \log_2 3n \rceil \text{ digits: } f(i) \in Q\}$$

such that i corresponds to the first element of f and $f(i)$ to the second one. The first two $\#$ introduce the code segment, one $\#$ separates two blocks of code and the $\P$ is a special symbol to separate the two numbers in a block of code.

Clearly, the domain and the image of the function f can be considered as words over the alphabet $\{0, 1, \ldots, 9\}$ in decimal format. This alphabet is not convenient for a cellular automaton.

Hence, as usual, we will take the binary representation of the numbers given as words. To make the representation easier, we want all words to be of the same length. To do that, we take the smallest size necessary to represent the biggest number in binary representation. In other terms, $\lceil \log_2 3n \rceil$ is the number of bits necessary to represent the maximum number which can be obtained by summing up the states. Moreover, the most significant bit is at the right. For instance, if the local transition function f contains the pair (2, 6) and the cardinality of the set of the states of the cellular automaton is two, the pair will be coded in the universal cellular automaton by the word $010\P\,011\,\#$ called *code block*. The concatenation of all the code blocks ordered by the usual order is called the *code segment*. The two symbols $\P$ and $\#$ are special symbols of the alphabet. The role of the $\#$'s is to separate the code blocks and the role of the $\P$'s to separate the binary representation of the two numbers of the pairs.

Example 3.1. Let $A = (Q, f)$ be an additive cellular automaton with $Q = \{1, 2\}$ and f defined as follows:

i	1	2	3	4	5	6
$f(i)$	–	–	2	2	1	2

The transition function f will be coded in the following way: the two –'s corresponding to any state, are replaced by the special sequence 111:

the # introduces the code segment	maximal length	separates two blocks	separates two numbers

$$\underbrace{\#\ \#\ 100\P 111\ \#\ 010\P 111\ \#\ \underbrace{110\P 010}_{\text{a code block}}\ \#\ 001\P 010\ \#\ 101\P 100\ \#\ 011\P 010\ \#}_{\text{the code segment}}$$

The coding of the transition function of the cellular automaton A needs $6n(\lceil \log_2 3n \rceil + 2) + 2$ cells of the universal cellular automaton. Each cell of U contains a symbol in $\{0, 1, \#, \P\}$.

In this coding, we can remark that the first element of the pairs of the transition function could have been omitted. It would seem to be very economical. But it is an easy trick to help our universal cellular automaton when it is searching the datum corresponding to the sum it has just computed. If we do not, the universal cellular automaton has to count the number of #'s it has crossed. Such a mechanism would be expensive in states and too long for the universal computation. Indeed, the universal cellular automaton not only has to compare the digits of the signal emitted with the cells corresponding to the transition function, but also memorizes the number of #'s it has already seen.

3.2. Coding the initial configuration

The initial configuration of the cellular automaton to be simulated by the universal one is supposed to be a snapshot. This snapshot is coded into the universal cellular automaton.

There are two techniques to encode the initial configuration. They both use the binary representations of the states to be described. The difference between the two techniques is based on the performance of the universal computation which can be cut in two principal steps. As first "step", the cellular automaton computes the sum of its own state with the state of its left neighbor and with the state of its right neighbor for each nonquiescent cell and for the two quiescent bordering cells of the configuration. That is the reason why we need to have numbers of the interval of $\mathbb{N}$ comprised between 1 and $3n$ in the description of the initial configuration of the cellular automaton to simulate. The result of this sum can be interpreted as an i of the set f. The cellular automaton reads then the contents of $f(i)$ and replaces the old state by the image of the sum by the transition function. The first possibility is to code the initial configuration as it is on the cellular automaton to be simulated. So, the first step of the universal computation will be to sum up the states in groups of three neighbors. The second is to compute the sum first and code the result in the universal cellular automaton. The first coding will be called *instantaneous description* of the initial configuration and the second one the *totalistic form.*

We will prefer the totalistic form for many reasons. The first one is that computing the sums first before the universal computation is quicker than to begin the universal computation with the computation of the sums. Indeed, the two sums described above are, by assumption, done in real time. That is, the data are nearer before the universal cellular automaton starts. Secondly, the very first steps of the universal computation are more understandable if we begin with the search of the image of the additive state than by sums and communications. Third, we thus restrict the expansion of the cellular automaton to the right in the case where we simulate a half-line rather than a segment.

For instance, if the initial configuration of the cellular automaton to be simulated is 112122, it will first be summed up (more precisely, the binary representation of the initial states will be summed up three by three) in:

- 4455 if the cellular automaton to be simulated is a segment;
- 44554 if the cellular automaton to be simulated is a half-line.

The word representing the coding of the configuration of the cellular automaton to be simulated in the universal cellular automaton is such that:

the beginning and the end of the segment are marked by #'s

each @ separates the coding of the cells of A

$$\underbrace{\# @ 100 @ 100 @ 010 @ \underbrace{100}_{\text{data block}} @ 010 @ 010 @ \#}_{\text{data segment}}$$

Each data segment can be written as a word of the following form:

$$\# (x@)^k \#$$

with

$$x = \{Bin(i) \text{ written with } \lceil \log_2 3n \rceil \text{ digits}, i \in Q\},$$

such that x corresponds to the binary representation of the additive form of the initial configuration of A and k denotes the number of nonquiescent cells on the additive form of the decimal representation of the additive form of the initial configuration of A. We can then evaluate the space each data block needs:

$$\lceil \log_2 3n \rceil + 1.$$

Thus, the number of cells needed to encode a data segment containing k additive data is $(k(\lceil \log_2 3n \rceil + 1)) + 2$.

4. Distribution of the code

In this section, we present the manner in which the code is distributed between the given totalistic data blocks of the universal cellular automaton's initial configuration.

We assume first that the concatenation of the code segment and of the data segment of the cellular automaton we want to simulate is now the initial configuration of the universal cellular automaton U. Thus, each cell of U represents one of the following symbols: $\{0, 1, \#, @, \P\}$ and $\{q\}$ where q denotes the quiescent state.

The code segment represents exactly the coding of the totalistic transition function of the cellular automaton to be simulated. After the code segment we concatenate the data segment. We can assume, without loss of generality, that the data segment is composed of the sums by triples of the initial configuration of the cellular automaton to be simulated. We have a data segment made up with the right infinite word:

$$\# d_1 + d_2 + d_3 @ \ldots @ d_{n-2} + d_{n-1} + d_n \# q^\omega$$

Lemma 4.1. *It is always possible to get the totalistic initial configuration from the nontotalistic one in time* $3\lceil \log_2 3n \rceil + o(1)$, *where n denotes the number of the states.*

Proof. Assume the initial configuration of the cellular automaton to be simulated is coded in nontotalistic form as described in Section 3.2. That is, we have a configuration of the form

$$\# d_1 @ d_2 @ d_3 @ \ldots @ d_{n-2} @ d_{n-1} @ d_n @ \# q^\omega.$$

We also assume that the cellular automaton to be simulated is a segment rather then a half-line of automata or any other cellular automaton. It is not much harder to modify the process for the other types of cellular automata. The final configuration we aim to obtain is of the following form:

$$d_1 + d_2 + d_3 @ d_2 + d_3 + d_4 @ \ldots @ d_{n-2} + d_{n-1} + d_n @ \# q^\omega.$$

To get the initial configuration in totalistic form we twice use the cellular automaton which computes the sum of two numbers of the same length in *real time* (cf. Lemma 2.7). By real time, we mean that the number of units of time to process the sum is exactly the same as the length of its initial configuration (the two numbers). The primitive idea is to cut the additions into two phases:

(1) summing up the state with its left neighbor;

(2) summing up the partial result with the right neighbor.

The data segment is entirely synchronized by means of the firing squad lemma with one general. Then, the coded states are emitted synchronously to the right as quickly as possible. They arrive above the nearest right neighbor after length(data block) units of time as illustrated in Fig. 4.

Then, the cellular automaton which computes the sums, adds up the two data blocks, memorizes the partial result, and synchronizes the data segment block by block. Then, to sum up the triples, the cellular automaton does the symmetrical operation. It corresponds to the second part of the sum. To that end, it sends the data blocks to the left as quickly as possible. The data blocks emitted arrive above the nearest left neighbor after length(data block) units of time. Then, the cellular automaton that computes the sums, adds up the data block received and the partial sum and memorizes the final result.

Clearly, thanks to the cellular automaton which computes the sums, the cellular automaton which transforms a nontotalistic initial configuration into a totalistic one can easily be constructed. □

As we have just seen, we can consider, without loss of generality, that the initial configuration of the universal cellular automaton is of the form $(C \times D)$, where D denotes the concatenation of all the data blocks or, in other words, the data segment.

We must transform the initial configuration into a configuration of the form

$$d_1 C d_2 C d_3 C \ldots d_k C$$

with may be some special markers between the d_i's and C's and where k denotes the number of totalistic data blocks; we call this operation the *distribution of code*.

One may ask the reasons of the transformation of $(C \times D)$ into $(d \times C)^k$. The goal of this transformation is to speed up the search for the image of a data by the coded transition function. If the code were too far, we would have needed plenty of time to get the same result. We can also imagine other possibilities for distributing the code between the data. The dual solution of the one we gave before is to have an initial

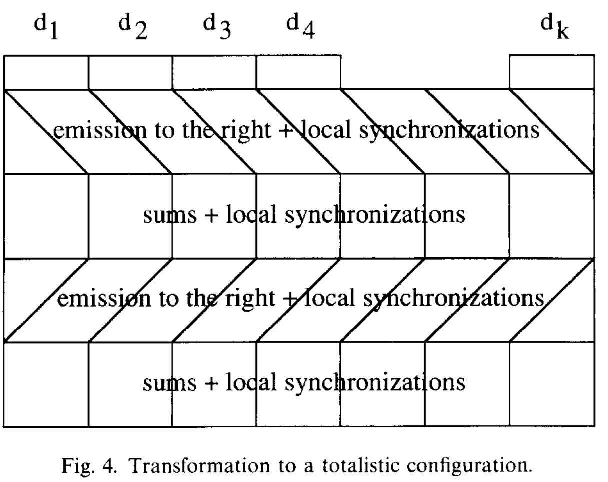

Fig. 4. Transformation to a totalistic configuration.

configuration of the form $(C \times d)^k$ which is of the same type as ours. Another interesting alternative comes from the fact that the code must not be far from the data. Thus, to make a more economical solution, we could have distributed the code between each two data blocks and got another initial configuration of the form $d_1 d_2 C d_3 d_4 C \dots d_{k-1} d_k C$ but, in that case, we would have had plenty of crossing signals when the data are emitted. To avoid the collisions we would have been led to increase the size of the tuple cellular automaton, which would have multiplied the number of states of each cell. We will not go further into the description of possible initial configurations. There are plenty of them. We retain the first one.

The distribution of the code is made in the following way: the first operation of the cellular automaton is to send a quick signal to the right to find the special symbol which delimits the end of the first data block. This one is then emitted to the left and its place is left free. It thus allows the code to be shifted to the right. The following operations are to duplicate the code segment and to shift it to the right after the next data block if some data blocks still remain. Such a process is depicted by Fig. 5. To detail these operations we introduce the notion of brick which describes a routine of the distribution of code.

In the following sections, we will not give the details of the synchronizations which occur in the behavior of the cellular automaton. We assume that every operation is able to be synchronized and we will give later the synchronization strategy.

4.1. General script of the code distribution

We recall briefly the initialization of the process (see Fig. 6) by the use of the bricks and chain with the following:

```
In_Parallel
  Start(configuration 0)
  Find_Symbol (d);
  Special_Send_Left chained with Find_Symbol($);
  Shift_Left;
  Fall_into (main signal);
  Find-Symbol (d);
Iterate
  In_Parallel
    Asynchronous_Send//
    Stop//
    Receive;
  Shift_Right (data segment under the duplicated code);
  Fall_into (main signal);
  Find_Symbol_Left(end of a data block symbol);
  Send_Signal_Right(send_signal)
Until no more data blocks can be shifted to the right.
```

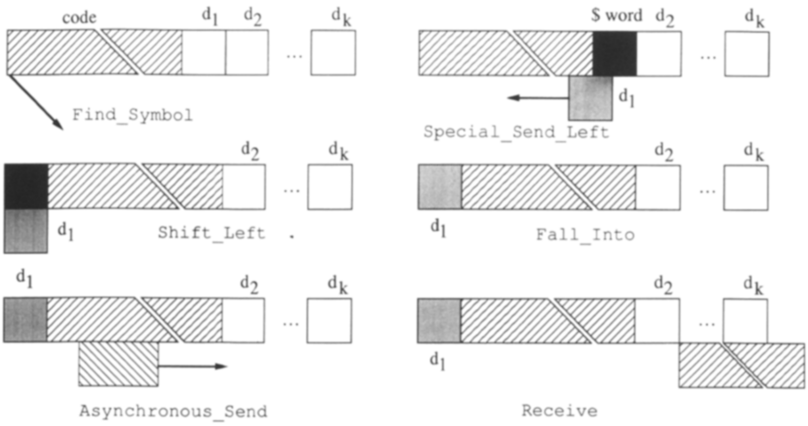

Fig. 5. The code distribution.

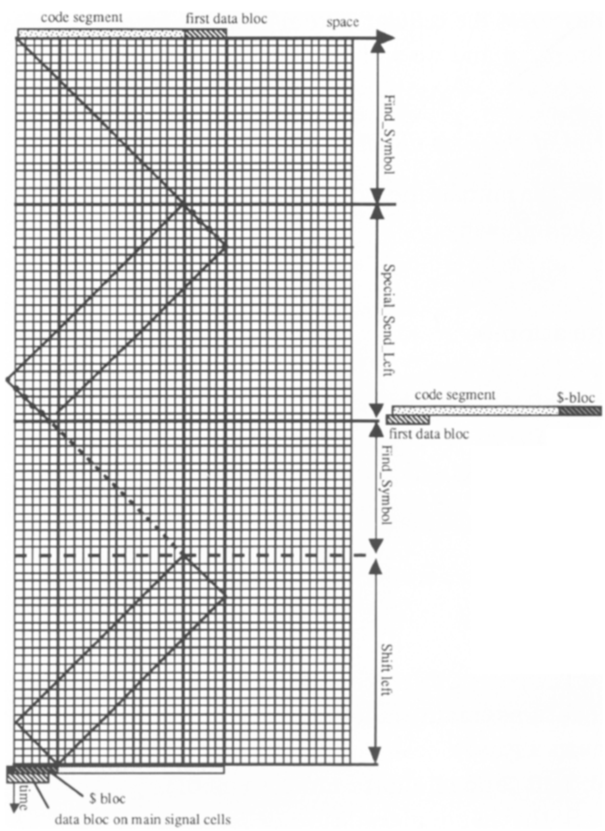

Fig. 6. Beginning of the code distribution.

A new meta brick has been introduced in order to iterate the process of the code duplication. The condition "no more data blocks can be shifted to the right" is obtained by the occurrence of the special marker # after the end of the last data block. It says that the duplication of the code segment has only to be done once again. The operation of the `Fall_Into` brick introduced above is to replace the \$-word by the data block.

4.2. Time complexity of the code distribution

We give the time complexity of the code distribution. We describe step by step the work of the code distribution given in *brick language*: First, the cellular automaton sends a signal moving to the right as quickly as possible to find the end of the first data block. This process is identified with the brick `Find_Symbol`(d). The signal reaches the symbol d after: $t_1 = \text{length(code)} = 6n\lceil \log_2 3n \rceil + o(1)$ units of time.

When the `Find_Symbol` brick has identified the position of the beginning of the first data block it sends it to the left and leaves its place free. This is the action of the `Special_Send_Left` brick. The entire first data block is gone after: $t_2 = 2\,\text{length(data)} = 2\lceil \log_2 3n \rceil + o(1)$ units of time and finishes its movement to the left end of the half-line after $t_3 = \text{length(code)} = 6n\lceil \log_2 3n \rceil + o(1)$ units of time. The next step is to identify the first $\underline{\$}$ and shift the \$-block to the beginning of the half-line. This is done in time: $t_4 = 2\,\text{length(code)} + o(1)$.

Then, we order the first data block, which is on the main signal to the main state instead of the "free" markers. This is done by the `Fall_Into` brick. One can see easily that it works in real time. Thus, the time to do the `Fall_Into` brick is: $t_5 = \text{length(data)} = \lceil \log_2 3n \rceil + o(1)$.

The role of the `Fall_Into` is to replace the \$-word by the data block. Its behavior is intuitive enough and one can convince himself that it is possible to write its local transition function. It works in an asynchronous manner and uses a signal which dies at the left end of the data segment.

Then a new `Fall_Symbol` brick is made to find the beginning of the code segment in order to enter the *iterate* meta brick. From now on we enter a more general loop which is iterated while some data blocks still remain.

The general strategy of the code distribution can be divided in two main phases:

(1) the code segment is duplicated and shifted;

(2) the rest of the data segment on the right side of one data block is shifted to the right, at the end of the duplicated code segment.

The time–space diagram shows clearly that the time is the following: $t_6 = 3\,\text{length(code)} + \text{length(data)} = (18n+1)\lceil \log_2 3n \rceil + o(1)$ for the entire duplication of the code, which is asymptotically of order $O(n \log_2 3n)$.

The rest of the data segment must be shifted to the right and leave the place free for the code segment which must be integrated into the main state of the half-line. Notice that the `Shift_Right` corresponding to the process in brick language works analogously to the code emission and uses the second signal of the cells instead of the main signal and respectively. The brick needs the following time to be executed: $t_7 = 2(\text{length code}) + k\,\text{length}(\text{data})) = 12n(\lceil \log_2 3n \rceil + 1) + 2k\lceil \log_2 3n \rceil + 2)$, where k denotes the number of data blocks remaining.

Note that we have compressed the Shift_Right and Fall_Into bricks into one. It is possible to find one transition function which can do the two processes together. We will not give the details. Its construction is similar to the constructions we have made in the previous sections. The important thing is that, at the end of the process, the active cell is the last one. That is the reason why we send a Find_Symbol_Left to seek for the beginning of the last code segment which has been duplicated, to go further in the iteration of the process if some data blocks still remain.

Henceforth we are able to give the entire time complexity of the code distribution. To that end, let us assume that the number of data blocks in totalistic form is denoted by the letter k. Moreover, let us assume that the length of a data block is denoted by d and the length of the code segment by the letter c. Clearly, we have $c = O(6nd)$, where n is the number of states of the cellular automaton. If we take brick by brick the time complexities given before, we get for the following:

```
Find_Symbol (d);
Special_Send_Left chained with Find_Symbol ($);
Shift_Left;
Fall_Into (main signal);
Find_Symbol (d);
```

a time complexity of $4d + 4c$ and for

```
Iterate
    In_Parallel:
    Asynchronous_Send//
    Stop//
    Receive;
  Shift_Right (data segment under the duplicated code);
  Fall_Into (main signal);
  Find_Symbol_Left(end of data block symbol);
  Send_Signal_Right(send_signal)
Until no more data blocks can be shifted to the right.
```

which is made $(k-1)$ times – as long as there are some data left on the half-line after the first part of the work of the process – we get a time complexity of: $6(k-1)c + 3(O(k^2)d)$. Thus, by adding the two time complexities, we get a time complexity of the process of the code distribution of the order $O((k \log n)(n+k))$.

5. Computing universality

In this section, we describe how the universal computation on a segment of automata is done. It can easily be transformed for a half-line of automata, but this will not be presented in this section.

Recalling the results of the previous section, we now dispose of an initial configuration of the cellular automaton of the form $(d \times C)^k$. This configuration allows us to compute the simulation of an arbitrary cellular automaton, say A. We will distinguish the following steps in this process:

(1) send the data block numbered j to the next data block at the right;
(2) sum up the data numbered j and the data $j+1$, keep the data $j+1$;
(3) send the data block numbered j to the previous data block at the left;
(4) sum up the data numbered j and the final sum of $(j-1)+(j-2)$;
(5) send the result of the sum to the corresponding number of the transition table;
(6) take the result of the application of the transition function to the number and return it to the previous data block. Erase the content of the old value;
(7) return to step 1.

All the data blocks are emitted to their right neighbor. When the data blocks arrive on the main signal of the cells containing the next data on their main state, the contents of the two cells are added and the partial result obtained is written in the second signal of the cells. Then step 3 is carried out and the contents of the data blocks are sent to their left neighbor.

When the data blocks arrive on the main signal of the cells containing the final result computed before on their second signal, the contents of the two blocks are added and the final result obtained is written in the main state of the cells and the other parts return to the quiescent state. Figure 7 illustrates the first steps of the simulation.

The first four steps may be described in our brick language in the following way:

```
Start(configuration "code distributed")
  Firing_Squad_1(left) on nonquiescent part;
  Special_Send_Right(main state);
  Superimpose
    Sum_up(main state+main signal-> second signal)
  with
    Firing_Squad_2 (on data blocks);
  Special_Send_Left(main state);
  Superimpose
    Sum_up(second signal+main signal -> main state)
  with
    Firing_Squad_2(on data blocks);
final configuration ("end of sums")
```

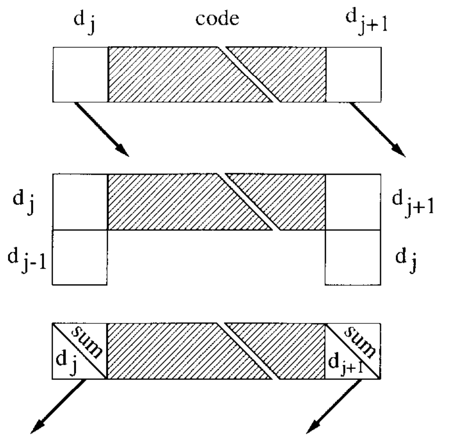

Fig. 7. The first steps of the simulation.

The new bricks can easily be deduced from the bricks detailed in the previous sections and will not be described. The meta brick `Superimpose with` indicates that two processes are superimposed, the send process or the summing process with a firing squad which runs on the synchronization part of the cells. It can be seen as a CA-product with respect to Definition 1.5.

We can then give the time complexity of the first four steps of the process which is clearly given by the synchronization time. The entire segment of automata is synchronized with the help of the firing squad lemma with one general. Thus, we get the time complexity: $t_1 = 2(k(c+d)) - 2 + 2c + 2d$.

We can now describe the other part of the process, namely the `Find_Image` meta brick. The result of the sum is emitted to the right in order to be compared with the first part of each block of code corresponding to the data emitted. When the entry corresponding to the data emitted has been found, it is emitted to the left in order to replace the old value of the data block. Arriving left, the image of the old value takes the place in the data block. This simulates one transition of the given totalistic cellular automaton A. Figure 8 illustrates the `Find_Image` brick. Those steps can be described in our brick language to form the meta brick `Find_Image` which simulates one transition of the cellular automaton A.

```
Find_Image:= Start(configuration "end of sum");
Superimpose
  In_parallel
    Special_Send_Right (data)//
    Compare_Stop;
with
Firing_Squad_1 (left) on data block+ code segment;
end;
final configuration ("end of transition");
```

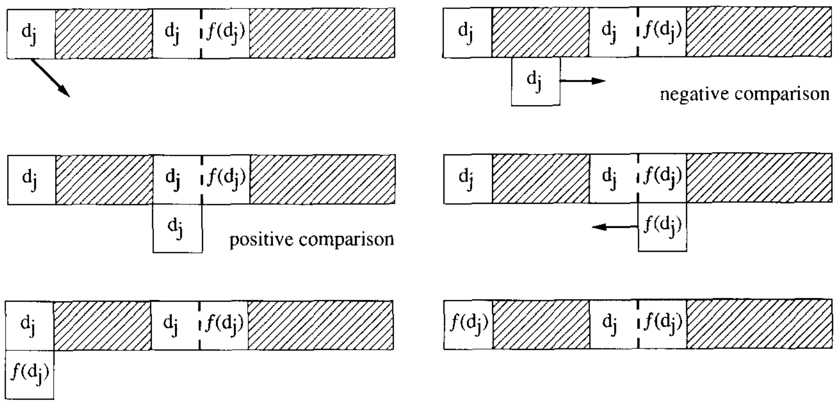

Fig. 8. The behavior of the find image brick.

In the above description, Firing_Squad_1 (left) stands for Minsky's solution to the firing squad synchronization problem which is in time 3 length(segment) − 2. The time complexity of the Find_Image brick is trivially given by the time of the Firing_Squad with one general. Anyway, we must wait for the worst case, which is as follows: the data emitted correspond to the last data of the code segment. Thus, the time complexity of the Find_Image brick is given by: $t_2 = 3(c+d) + o(1)$.

The simulation of one transition of A is thus defined by the concatenation of the previous bricks.

```
Simulate::= Start (configuration "code distributed")
  Firing_Squad_1(left) on nonquiescent part;
  Iterate
    Special_Send_Right (main state);
    Superimpose
    Sum_up(main state + main signal -> second signal)
  with
    Firing_Squad_2 (on data blocks);
  Special_Send_Left(main state);
  Superimpose
      Sum_up(second signal + main signal -> main state)
    with
    Firing_Squad_2 (on data blocks);
    Find_image;
  until any halting condition;
```

The time complexity of one iteration simulated is thus $t_1 + t_2 = 2(k+2)(c+d) + o(1)$. Figure 9 gives the general scenario of the superimposed synchronizations. The first Firing_Squad synchronizes once the whole half-line in order to emit the data blocks

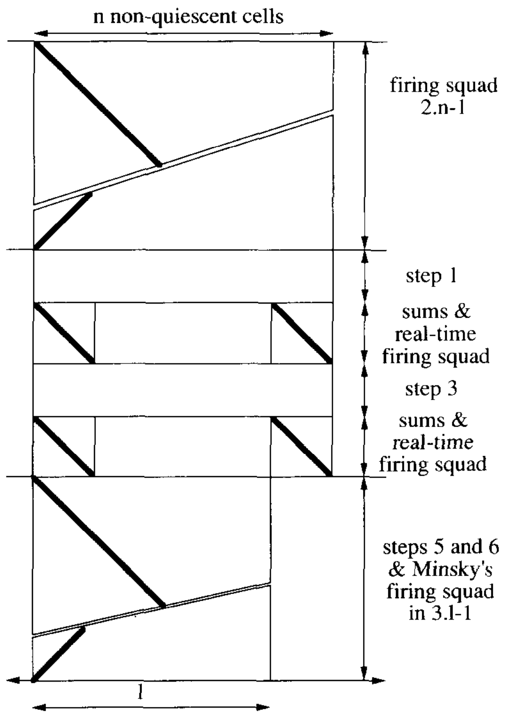

Fig. 9. The synchronizations.

to the right to be compared with the first part of each data block until the corresponding entry is found. During the search of the corresponding entry, k firing squads run to synchronize locally the half-line with the worst time possible. That is the corresponding entry is the last one.

Remark that it is strongly required that the cellular automaton to be simulated be in totalistic form. If it were not, by applying Lemma 1.3, the corresponding additive CA would have $N = n(n+1)^3$ states, where n is the number of states of the original CA. Since the universal CA U simulates additive CA, this would lead to a time complexity for simulating one step of the original CA of $O(N \log N) = O(n^4 \log n)$, and thus the simulation would not work in quasi-linear time anymore!

6. Final details

If we wish our CA U to be computation-universal without using Turing machines, we can consider that a CA A enters on its first cell in an accepting state and freezes its evolution – with the help of synchronizations, for instance – in order to keep the result. In such a case, the universal CA U must, to end its simulation, clean up its line.

If the CA A to be simulated has an accepting state, the CA U must leave only the data resulting from the computation of A on its entry. To that end, it must destroy all the parts of the CA containing the code segment and concentrate all the cells with data segments.

This can be done by the following steps:

(1) all the data blocks are emitted to the left at unit speed;
(2) the main line is destroyed;
(3) all the data blocks concentrated on the left of the line fall into the main line.

This type of cellular automata, able to compute and stop when the computation is completed, is called "computational-CA" or C-CA for short.

7. Application to theory of algorithms

We have shown in the previous parts of this paper that there exist a CA which is CA universal in quasi-linear time. This result allows us to apply the S–m–n theorem to the CA's.

We show here that cellular automata are really an acceptable programming system in the sense of Blum. We define first what is called an acceptable programming system.

Definition 7.1 A *programming system* is a listing $\varphi_0, \varphi_1, \ldots$ which includes all the partial recursive functions (of one argument over $\mathbb{N}$). A programming system $\varphi_0, \varphi_1, \ldots$ is *universal* if the partial function φ_{univ} such that $\varphi_{\text{univ}}(i, x) = \varphi_i(x)$ for all i and x is itself a partial recursive function; that is, if the system has a universal partial recursive function. A universal programming system $\varphi_0, \varphi_1, \ldots$ is *acceptable* if there is a total recursive function c for composition such that $\varphi_{c(i,j)} = \varphi_i \circ \varphi_j$ for all i and j.

Programming systems are often referred to as indexings of the partial recursive functions. An important and useful property of reasonable (i.e. satisfying the definition above) programming systems is the ability to modify programs so that some input parameters are held constant. We recall quickly the S–m–n theorem, which shows that this property holds for all acceptable programming systems.

Theorem 7.2. (S–m–n theorem). *For any acceptable programming system $\varphi_0, \varphi_1, \ldots$ there is a total recursive function S such that for all i, all $m \geqslant 1$ and $n \geqslant 1$, and for all $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$*

$$\phi_{S(i, m, x_1, \ldots, x_m)}(y_1, \ldots, y_n) = \phi_i(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

and $y_1, \ldots, y_n$.

That is, the function allows us to specify that the first m arguments for the ith program be held constant at $x_1, \ldots, x_m$. The proof will be omitted, since it can be found in any book on computability and there is no special internal interpretation of it for cellular automata.

As a particular case of this theorem we can apply it as S–1–1 theorem.

Theorem 7.3 (S–1–1 theorem). *For any acceptable programming system* $\varphi_0, \varphi_1, \ldots$ *there is a total recursive function S such that for all i and for all x and y,*

$$\varphi_{S(i,x)}(y) = \varphi_i(x, y).$$

That is, the function allows us to specify that the first argument for the ith program be held constant at x. We will not develop the proofs of those theorems in the present paper. They are well known and can be found in Machtey–Young [9] for instance. In this paper, we will show how the class of computational-CA is an acceptable programming system.

Proposition 7.4. *The class of C-CA contains all the partial recursive functions.*

The proof of this proposition is easy. It comes directly from the well known simulation of a Turing machine by a CA. The other fact necessary to end the proof is that the Turing machines form an acceptable programming system.

Proposition 7.5. *Any Turing computable function can be computed by a C-CA.*

It is well known that the computation of partial recursive functions is equivalent to Turing computation. We use a simulation of a Turing machine by a C-CA. This way, we get an indexing of partial recursive functions. We thus get the following result.

Proposition 7.6. *The C-CA form a programming system.*

This is the first step in the proof of the fact that C-CA are an acceptable programming system. We still have to prove, according to Definition 7.1, that C-CA form a universal and acceptable programming system.

The universality of the C-CA programming system comes from the universality of the Turing machines programming system and from the following proposition.

Proposition 7.7. *Every C-CA function can be computed by a Turing machine.*

We can then give our theorem.

Theorem 7.8. *The C-CA programming system is an acceptable programming system.*

Proof. We consider the following facts.

Fact 1. The previous propositions allow us to say that the C-CA are a universal programming system. One type of universality comes from the existence of a C-CA which simulates the evolution of a universal Turing machine.

Here, we consider the intrinsic universality. The existence of an intrinsic CA comes from [1] and from the CA described in this paper.

Fact 2. With the coding of [1] it is hard to obtain the composition as defined above. U receives as entry a segment composed by the code C of the CA A to be simulated followed by a segment containing the initial line of A. This last segment will be called x. Let C_i and C_j denote the codes of the two CA's ϕ_i and ϕ_j.

The C-CA V numbered $V_{c(i,j)}$ realizes the following operations on an initial line containing the successive segments C_iC_jx.

- The CA U simulates the work of the ith CA on the entry x.
- If the computation $\phi_i(x)$ ends, U cleans up its line.
- Then, a special CA transforms the segment C_jC_i into the segment C_iC_j. Such a CA will not be described there, but it is easy to obtain one which works in real time. (i.e. $\text{length}(C_j)+\text{length}(C_i)$).
- The CA U simulates the work of the jth CA on the entry $\phi_i(x)$.
- If the computation $\phi_j(\phi_i(x))$ ends, U cleans up its line.
- At last, the CA transforms C_iC_j into the segment C_jC_i.

It is clear that we can construct effectively the transitions of such a CA from the transitions of U and of the reversing CA. We have thus defined a function c which is total recursive.

The composition could also be obtained by using the fact that there is a recursive transformation from CA to Turing machines and vice versa, but in that case, we would not have proved that CA form intrinsically an acceptable programming system. □

We have completed the proof that the C-CA form an acceptable programming system. This leads us to the world of general theory of algorithms.

For instance, we have the general result which allows us to translate directly a programming system to another in an effective way.

7.1. Obtaining the S–m–n theorem

In the previous section we have proved that an intrinsic universal CA is an acceptable programming system and the way we get the composition. This gives us the S–m–n theorem. This is a consequence of the programming systems. To get this result we recall quickly the proof of the theorem. The coding of the function of several arguments into functions of one argument is another property of the function S. It does not need to have n arguments. We introduce the following functions:

Let $\langle\ \rangle_n$ be the coding of $\mathbb{N}^n \to \mathbb{N}$. This function is defined in a recursive way as below:

$$\langle x, y\rangle = 1/2\,[(x+y)^2 + x + 3y],$$

$$\langle x_1, \ldots, x_n\rangle = \langle x_1, \langle x_2, \ldots, x_n\rangle\rangle.$$

Let $\mathrm{Con}(m, \langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_n \rangle) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle$ be the concatenation function. Then, the S–m–n can be written as:

$$\phi_{S(i, m, \langle x_1, \ldots, x_m \rangle)}(\langle y_1, \ldots, y_n \rangle) = \phi_i(\langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle).$$

Using the primitive recursive function Con, we see that a total recursive function S such that for all i, all $m \geqslant 1$ and all x and y,

$$\phi_{S(i,m,x)}(y) = \phi_i(\mathrm{Con}(m, x, y))$$

will satisfy the conditions of the theorem.

To finish the proof we define three helpful total recursive functions P, Q and R: Since the C-CA form an acceptable programming system, there are CA's numbered p and q such that $\phi_p = P$ and $\phi_q = Q$.

$$P(y) = \langle 0, y \rangle,$$

$$Q(\langle x, y \rangle) = \langle x+1, y \rangle \quad \text{for all } x \text{ and } y,$$

$$R(0) = p \quad \text{and} \quad R(x+1) = c(q, R(x))$$

for all x where c is the total recursive function for composition. It is also easy to see, by an easy recurrence, that $\phi_{R(x)}(y) = \langle x, y \rangle$ for all x and y.

Recalling that, by definition of the function $\langle\ \rangle_n$, $\langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle$, we notice that

$$\phi_{R(x)} \circ \phi_{R(y)}(z) = \phi_{R(x)}(\langle y, z \rangle) = \langle x, y, z \rangle.$$

Finally, let k be such that $\phi_k(\langle m, x, y \rangle) = \mathrm{Con}(m, x, y)$.

If we define $S(i, m, x) = c(i, c(k, c(R(m), R(x))))$, we have

$$\phi_{S(i, m, x)(y)} = \phi_i \circ \phi_k \circ \phi_{R(m)} \circ \phi_{R(x)}(y) = \phi_i(\mathrm{Con}(m, x, y))$$

which completes the proof that the S–m–n theorem applies on the C-CA.

Theorem 7.9. *Let $\varphi_0, \varphi_1, \ldots$ be any universal programming system and let $\psi_0, \psi_1, \ldots$ be any programming system with a total recursive S–1–1 function: that is, there is a total recursive function S such that: for all i, x and y, there is a total recursive function t which translates the system $\varphi_0, \varphi_1, \ldots$ into the system $\psi_0, \psi_1, \ldots$, that is, $\varphi_i = \psi_{t(i)}$ for all i.*

The proof is omitted and can be found, for instance, in [9]. In other words that there exist an effective procedure to translate C-CA programming system to any other programming system. Thus, any acceptable programming system is equivalent to the C-CA programming system in a fairly strong sense.

7.2. Algorithmically unsolvable problems

We give now some examples coming from the general theory of algorithms which are directly applicable to cellular automata. Let $\Phi_0, \Phi_1, \ldots$ be an acceptable programming system.

If ψ is a partial recursive function, we say that $\psi(x)$ is convergent if $\psi(x)$ exists, or equivalently $x \in D\psi$ where $D\psi$ denotes the domain of the function ψ. Conversely, we say that $\psi(x)$ is divergent if $\psi(x)$ does not exist, or equivalently $x \notin D\psi$. This means that if the calculus of ψ on the input x is convergent then the calculation halts on x.

We show, in the present section, that some general problems are algorithmically unsolvable. We start with the halting problem which means that there does not exist a cellular automaton which decides if a given "program" stops (or does not) on a given input. This problem is independent of any programming system and thus applicable to cellular automata.

Theorem 7.10. *Let $\psi_0, \psi_1, \ldots$ be an acceptable programming system. The function f such that for all x and y, natural numbers,*

$$f(x, y) = \begin{cases} 1 & \text{if } \psi_x(y) \text{ is convergent,} \\ 0 & \text{otherwise} \end{cases}$$

is not a recursive function.

The proof of the theorem will be omitted. It is a direct consequence of the fact that CA form an acceptable programming system.

Remark 7.11. We find again the fact that, during the evolution of a cellular automaton on a given input, the occurrence of a given state cannot be predicted.

Similarly, we also obtain Rice's theorem which shows that in any acceptable programming system there are no nontrivial properties of the input–output behavior of programs, that is, properties of partial recursive functions, which can be decided by looking at the programs.

We define $\mathscr{P}_C$ the set of the partial recursive function with a given property by

$$\mathscr{P}_C = \{x \colon \phi_x \in C\}.$$

Then, we can give the following version of Rice's theorem.

Theorem 7.12. *$\mathscr{P}_C$ is recursive if and only if $\mathscr{P}_C = \emptyset$ or $\mathscr{P}_C = \mathbb{N}$.*

The proof of the Rice's theorem is well known and can be found in any book on theory of algorithms or complexity theory (e.g. [9]).

Remark 7.13. All the usual consequences of the Rice's theorem can be applied to the cellular automata. Particularly, it is not possible to test algorithmically if two cellular automata can do the same computation.

Rice's theorem is very important because it destroys any hope to test algorithmically the behavior of the input–output of programs.

8. Conclusion

The idea of the decomposition by bricks of our universal cellular automaton comes from programming the behavior of the universal simulation on a computer. The bricks correspond approximatively to the procedure which occurs in the pascal-program. Programming has also been helpful for building the details of the transition functions of the bricks.

The bricks are also helpful for establishing various properties of cellular automata. If not, we would have to make a recurrence with three levels and give cases, under-cases, and so on. Furthermore, it defines a new way to prove results for cellular automata, which is perhaps not so formal as a recurrence but good enough to convince the reader that the cellular automaton works as it is supposed to.

The other original idea of this paper is the proof that certain classes of cellular automata form an acceptable programming system. The definition we use is rather different from the original definition introduced by Blum [3] although it was proved to be equivalent in [9]. We hope that people using Blum's formalism would agree with this definition. Furthermore, it allows us to apply results which hold for any acceptable programming systems such as Turing machines and RAM machines. For example, a universal machine and an S–m–n theorem, yields a definition of abstract complexity, which leads to the difficult problem of defining good complexity measures for cellular automata.

This paper shows also that it is worth studying parallel abstract machines such as cellular automata from the point of view of the computability theory. Though it has been well known [7] that all the results applying to Turing machines can be applied to cellular automata, it is new to study the computability intrinsically from the cellular automata [1].

References

[1] J. Albert and K. Culik II, A simple universal cellular automaton and its one-way and totalistic version, *Complex Systems* **1** (1987) 1–16.

[2] R. Balzer, An 8 state minimal time solution to the firing squad synchronization problem, *Inform. and Control* **10** (1967) 22–42.

[3] M. Blum, A machine independent theory of the complexity of recursive functions, *J. ACM* **14** (1967) 322–336.

[4] C. Choffrut and K. Culik II, On real time cellular automata and trellis automata, *Acta Inform.* **10** (1984) 393–407.

[5] P.C. Fisher, Generation of primes by a one dimensional real-time iterative array, *J. ACM* **23** (1965) 388–394.

[6] K. Lindgren and M.G. Nordhal, Universal computation in simple one-dimensional cellular automata, *Complex Systems* **4** (1990) 299–318.

[7] M. Machtey and P. Young, *An Introducton to the General Theory of Algorithms*, Theory of computation series (North-Holland, Amsterdam, 1978).

[8] J. Mazoyer and N. Reimen, A linear speed-up theorem for cellular automata, Research Report 91.16, Laboratoire d'Informatique Théorique et Programmation, 1991; *Theoret. Comput. Sci.* **100**, to be published.

[9] M. Minsky, *Finite and Infinite Machines* (Prentice-Hall, Englewood Cliffs, NJ, 1967).
[10] E.F. Moore, *Sequential Machines, Selected Papers* (Addison-Wesley, Reading, MA, 1964).
[11] A.R. Smith III, Real-time language recognition by one-dimensional cellular automata, *J. Comput. System Sci.* **6** (1971) 233–253.
[12] A.R. Smith III, Simple computation-universal cellular spaces, *J. ACM* **18** (1971) 339–353.
[13] A. Waksman, An optimum solution to the firing squad synchronization problem, *Inform. and Control.* **9** (1966) 66–78.
[14] S. Wolfram, Universality and complexity in cellular automata, *Physica D* **10** (1984) 1–35.