HEXAGONAL CELLULAR AUTOMATA

MEENA MAHAJAN

and

KAMALA KRITHIVASAN

Department of Computer Science and Engineering

I.I.T., MADRAS, 600036, INDIA.

ABSTRACT

Hexagonal Cellular Automata (HCA) are defined as a variation of the rectangular 2-D cellular automata (RCA) studied in the literature. Equivalence of HCA and RCA is shown. Some algorithms for manipulation of patterns on the HCA are presented. The application of HCA to generating and parsing languages over hexagonal arrays, as described by the grammatical models proposed by Siromoney and Siromoney[15], is discussed.

1. INTRODUCTION

Hexagonal arrays and hexagonal patterns are found in the literature on picture processing and scene analysis[7,8,9]. These arrays have been studied as formal models[15], where the authors introduce array grammars to generate languages of hexagonal arrays. They also indicate the application of such models to 2-D representations of 3-D scenes composed of rectangular parallelepipeds, and to the problem of recognition of perceptual twins[10].

Cellular automata as acceptors for string languages have been studied extensively[17,18,19]. Systolic pyramid automata[4], which are in some sense equivalent to 2-D cellular automata, have been studied with respect to recognition of matrix and array grammars[11,12,13]. In this paper we introduce hexagonal cellular automata (cells placed on a triangular grid), and study their relation to the formal models for hexagonal arrays[15].

Section 2 introduces hexagonal cellular automata, gives definitions, shows equivalence to Turing machines, and gives two examples to illustrate basic algorithms for pattern manipulation. Sections 3 and 4 study the relation of these automata to hexagonal kolam array languages and controlled table L-array languages respectively, the two models proposed in (15). Generators and acceptors for these languages are described.

2. HEXAGONAL CELLULAR AUTOMATA

In the 2-D cellular automata discussed in the literature[1], cells are arranged on a square lattice, i.e., in a one- to-one correspondence with I×I (I is the set of integers.). Each cell is directly connected to its two adjacent vertical neighbours and two horizontal neighbours. The underlying interconnection structure is a rectangular grid; so henceforth we shall refer to these cellular automata as RCA (rectangular CA). Hexagonal cellular automata (HCA) are defined as a direct extension of RCA. In these CA, the underlying interconnection structure is the equiangular triangular grid. Thus each cell has six neighbours, arranged in a regular hexagon with itself at the centre (fig. 1); hence the name.

The operation of the HCA is exactly like the operation of the RCA. There is a finite set of states Q; each cell is in some state $q \in Q$ at every time instant; at successive time instants all cells synchronously change state depending on their current state and the current states of their six neighbours. The transition function at each cell is identical; it is as if multiple copies of the same finite state automaton are placed at each cell. We assume the existence of a special state, q_0, called the quiescent state, and, as in the case of RCA, stipulate that at any time instant only a finite number of cells are non-quiescent. Further, a cell with a quiescent neighbourhood remains quiescent (neighbourhood of a cell = itself and its six neighbours).

Formally, an HCA is a 3-tuple $H = (Q, q_0, \delta)$ where Q is the set of states, $q_0 \in Q$ is the quiescent state, and $\delta : Q \times Q^6 \rightarrow Q$ is the transition function subject to $\delta(q_0, q_0, q_0, q_0, q_0, q_0, q_0) = q_0$. The arguments to δ are as follows: the first argument is the state of the cell under consideration, and the next six arguments are the states of its six neighbours read in clockwise order starting from the top. The function value gives the new state of the cell under consideration. Rather than write $\delta(x,a,b,c,d,e,f) = y$, we write this pictorially as

$$\delta\left(\begin{array}{ccc} & a & \\ f & & b \\ & x & \\ e & & c \\ & d & \end{array}\right) = y.$$

A configuration of the HCA is an assignment of states to cells on the triangular grid, such that at most a finite number of cells are non-quiescent. At each time instant, each cell simultaneously changes state as per the transition function, giving rise to a new configuration.

In sections 3 and 4 we shall consider languages of hexagonal arrays(15), and see how HCA relate to these classes of languages. Two models will be considered: a) the generator model, and b) the recogniser model. In the generator model, the HCA starts from an initial configuration and generates all the arrays belonging to a language L. Clearly, it may not be possible to generate these arrays at successive time steps; intermediate transitions may be required. Let $H_1, H_2, .. , H_i, ...$ be an enumeration of all arrays in L. Then whenever the HCA enters a configuration equal to H_i, it should indicate this. For this we adopt the convention that whenever an H_i is generated, the bottommost cell of H_i, cell 'D', will enter a special state. Thus intermediate configurations of the HCA are either not convex hexagons at all, or ones in which the cell 'D' is not in a special state.

For the recogniser model, the initial configuration is the input array, i.e., a convex hexagon. Its bottommost cell is now considered to be the distinguished cell 'D'. At any time t, if cell 'D' goes into an 'accept' state, then the input array belongs to the language L. Thus the notion of acceptance by an HCA can be defined for hexagonal array languages. If

the input hexagon H does not belong to the language, the behaviour of the acceptor can be anything. However, if we can define an acceptor which also goes to a prespecified reject state when H is not in the language, then we have a stronger notion of acceptance, termed recognition(18).

In an HCA, the interconnection graph is an equiangular triangular grid. It consists of 3 families of parallel lines, with orientations |, /, and \ respectively. Removing any one family leaves a graph isomorphic to the rectangular grid, as shown in fig. 2. Clearly, an HCA has all the connections of the RCA, and more. A mapping between the two CA may be defined as follows:

In an HCA, each cell x has 6 neighbours, whose positions are as indicated. In an RCA, a cell X has 4 neighbours - left(L), up(U), right(R), and down(D) as shown below.

$$\begin{array}{ccc} & a & \\ f & & b \\ & x & \\ e & & c \\ & d & \end{array} \qquad\qquad \begin{array}{ccc} & U & \\ L & X & R \\ & D & \end{array}$$

If a cell x of an HCA is to be mapped onto cell X of an RCA, consider the 4 neighbours resulting when the family of vertical lines of the HCA is deleted. i.e.,

$$\begin{array}{ccc} f & & b \\ & x & \\ e & & c \end{array}$$

These may be mapped to the 4 neighbours of X as follows:

$$f(x) = L(X), \quad b(x) = U(X),$$
$$c(x) = R(X), \quad e(x) = D(X). \qquad (1)$$

Doing a similar mapping for the neighbours of b and e, we get

$$a(x) = L(U(X)), \; d(x) = D(R(X)). \qquad (2)$$

The mapping is shown in fig. 3. The neighbourhood of cell x, as represented in the RCA, is highlighted. The missing vertical lines of the HCA are the dotted left-to-right diagonals of the RCA. It is also clear how an RCA can be mapped onto an HCA; for cell X of the RCA mapped onto cell x of the HCA, equation 1 gives the relationship between the neighbours of X and the neighbours of x.

The transitions of the RCA (Q, q_0, δ_R) can be simulated by the HCA (Q, q_0, δ_H) in one step as follows:

$$\delta_H\left(\begin{array}{ccc} & a & \\ f & & b \\ & x & \\ e & & c \\ & d & \end{array}\right) = \delta_R\left(\begin{array}{ccc} & b & \\ f & x & c \\ & e & \end{array}\right)$$

The transitions of the HCA can be simulated by the RCA at half-speed; i.e., for each transition of the HCA, the RCA goes through 2 transitions as follows:

$$\delta_R\left(\begin{array}{ccc} & c & \\ g & h & i \\ & m & \end{array}\right) = ghi,$$

```
a -- b -- c -- d -- e
f -- g -- h -- i -- j
k -- l -- m -- n -- o
```

giving rise to the intermediate configuration, and

```
abc  bcd  cde
fgh  ghi  hij
klm  lmn  mno
```

$$\delta_R\left(\begin{array}{ccc} & bcd & \\ fgh & ghi & hij \\ & lmn & \end{array}\right) = \delta_H\left(\begin{array}{ccc} & b & \\ g & & c \\ & h & \\ m & & i \\ & n & \end{array}\right)$$

Thus we have the following theorem:

Theorem 2.1 : Every HCA can be simulated by an RCA and vice versa; i.e., HCA and RCA are equivalent in power.

From this it follows that the sequential machine characterisation of an RCA can also be used for the HCA.

An RCA R can be simulated by a Turing machine M with a 2-D worktape T which stores in its cells the configuration of the RCA. For a single transition of the RCA, M will go through a series of transitions, sequentially updating the states of all cells with a non-quiescent neighbourhood. Thus a Turing machine can simulate any RCA and hence any HCA.Conversely, a HCA or RCA can simulate any Turing machine, because it contains an unbounded 1-D CA within it, and 1-D CAs have been shown to simulate Turing machines(1). Thus we have the following result:

Theorem 2.2 : Hexagonal Cellular Automata and Turing machines are exactly equivalent in computing power.

Similar to the one-way 1-D and 2-D CA, we can define one-way HCA, OHCA, where a cell gets input only from neighbours in positions e, d, c, and gives output only to neighbours in positions f, a, b. The OHCA can be shown to be equivalent to the one-way RCA and thus also equivalent to the trellis defined in (3).

If the states of a CA are mapped onto integers, then the CA is said to be totalistic if the transition function depends only on the sum of the states of all cells in the neighbourhood of a given cell. For every HCA, an equivalent totalistic HCA which simulates it in real time can be constructed. The construction is based on giving a suitable colouring for the underlying graph and then applying the technique described in (2).

The interconnection structure for the HCA is the equiangular triangular grid. Thus any convex hexagon represented on the HCA is equiangular and has opposite sides parallel. It can be shown that if the lengths of the sides of the hexagon, taken in clockwise order, are p, q, r, s, t, u, (see fig. 4) then s = p + k, t = q - k and u = r + k, for some k. If k = 0, then the opposite sides of the hexagon are parallel and equal;

such hexagons are referred to as b-hexagons(15). If k = 0, we have a non-b hexagon. A b-hexagon with p = q = r has opposite sides parallel and all sides equal; it is a regular hexagon.

Several pattern transforms can be defined for convex hexagons; typical examples are rotation through multiples of 60° and mirror reflection about an edge. Similarly, property checking for convex hexagons is of interest for properties like regularity, 60° rotation symmetry, 120° rotation symmetry and so on. Algorithms on the HCA for some of the above problems have been presented in (5). For illustration, two of them are given below. The convex hexagon is represented on the HCA by having cells within the hexagon in some non-quiescent state and all other cells in quiescent state. Some observations which are often used in the algorithms are given below.

(i) In a convex hexagon, cells along the border 'know' that they are on the border. This is so because cells on vertices of the hexagon have exactly 3 quiescent neighbours, while cells along an edge have exactly 2 quiescent neighbours.

(ii) If cells along a continuous non-self-intersecting path on the HCA are marked, then they can operate in conjunction as a 1-D CA, even though they may not be in a straight line. This follows from the fact that along the path, each cell knows its predecessor and successor (this information must be imparted at the time of marking the path). See fig. 5.

(iii) Each cell affects the change of state of every cell in its neighbourhood. This can be viewed as the sending of signals by a cell to all its neighbours. The algorithms presented below will thus talk of signals being sent across the HCA, rather than state changes at cells.

Algorithms:

(1) Firing Squad problem for a regular hexagon :

Given a regular hexagon as the initial configuration, with the

distinguished cell D (bottommost vertex) in state \$ and all other cells in state @, we wish to achieve, at some time t, a configuration where all cells of the hexagon are in state #, such that no cell of the hexagon enters state # prior to t.

The firing squad problem for linear arrays (initial configuration $\$@^{n-1}$, final configuration $\#^{n}$) has been solved in various ways(4). The time taken for this algorithm, L(n), has been shown to be at least 2n-2. This algorithm can be easily extended to RCA as follows. Let the topleft cell of the mxn rectangle be in state \$. In time L(n), all cells in the topmost row can be brought simultaneously into an intermediate state #'. At this time, each cell c_{1j} of the topmost row can initiate a firing squad algorithm on the jth column; so in another L(m) time units all cell in each column go to state # (denoted as 'fired'), i.e., all cells in the entire rectangle fire at time L(n) + L(m).

For a regular hexagon H, the trick lies in dividing the hexagon into rectangles which can fire simultaneously. To do this, D sends signals S_F, S_B and S_E to cells F, B, E respectively. See fig. 6. S_E returns to D from E. Thus S_F, S_B and S_E reach F, B and D at the same time instant 2n (where n is the length of a side). Cells B, D, F are the corner vertices of rectangles BCOA, DEOC and FAOE respectively. They can run RCA firing squad algorithms on these rectangles, firing all cells of H after a total time of 2n + 2L(n). Note that the cells along the radii OA, OC and OE must be able to keep track of signals for 2 distinct RCA algorithms, since they participate in the firing squad algorithms of both rectangles adjacent to them.

(2) To move a convex hexagon parallel to an edge until the edge reaches a prespecified boundary.

Without loss of generality, assume that the edge is vertical (edge BC of fig. 4) and the desired movement is to the right. A boundary GG' to the right of edge BC and parallel to it is premarked as the edge along which edge BC of the shifted hexagon must finally reside. Note that there is no straight line path for right movement. So a q-pulse must travel right in a zigzag fashion, moving to the neighbour at position b at odd time instants and to the neighbour at position c at even time instants. So the pulse travels as a tuple (q, up) or (q, down) indicating whether it will move to neighbour b next (and become (q, down) there) or to neighbour c (and change to (q, up)). Positions of neighbours of a cell are shown in figure 1.

The rightmost column of the hexagon H sends pulses, travelling right at unit speed, at time $t = 1$. The next column sends out pulses at $t = 2$, the third rightmost column at $t = 3$ and so on. Effectively, a column sends pulses right one time unit after the column to its right has done so. When the rightmost column pulses reach GG′, they stop moving and 'stabilise'; i.e., they revert from tuples (q,up) or (q,down) to singletons q. The pulses from other columns stabilise when they find that the pulses to their right have stabilised; i.e., they halt one column before the previous column. Thus if GG′ is m columns away from BC, then the rightmost column stabilises at time $t = m$, the next at $t = m+1$, and so on. See fig. 7. If the width of H along the direction normal to GG′ is w, then this algorithm requires $m+w-1$ units of time.

3. GENERATING AND RECOGNISING HEXAGONAL KOLAM ARRAY LANGUAGES

The main motivation for studying HCA has been to study its relationship to the languages of hexagonal arrays(15). In (15), formal grammatical models for generating such arrays have been discussed. The main point to be noted is that unlike strings, hexagons cannot be concatenated to give hexagons. So the grammatical models use arrowhead concatenation, which is illustrated in fig. 8.

A hexagonal kolam array grammar has 3 disjoint sets of alphabets - nonterminals, intermediates and terminals. There are 3 types of rewriting rules : (i) In the first type, called non-terminal rules, the left hand side (l.h.s) has one non-terminal and the right hand side (r.h.s) has exactly one non-terminal along with zero or more intermediates. Treating the

intermediates as terminals, these rules are either left linear or linear. (ii) The second type has a single terminal rule with a non-terminal on the l.h.s and a hexagonal array of terminals on the r.h.s.. (iii) In the third type, comprising of intermediate rules, usually the intermediate languages are specified rather than enumerating the intermediate rules. These languages consist of arrowheads made up of terminal symbols.

Derivation proceeds as follows:

Initially, non-terminal rules are applied in sequence, with arrows indicating direction of concatenation, and parentheses inserted at each step. This results in a string with intermediates and one non-terminal. In the second phase, the non-terminal is replaced by a hexagonal array of terminals, using the terminal rule. In the last phase, starting from the innermost parenthesis, each intermediate is replaced by an arrowhead of appropriate size (appropriate with respect to concatenation) from the corresponding language. This arrowhead is concatenated to the existing hexagon to get a new hexagon.

The grammar is called regular (R) or linear (L) according as the first phase (applying non-terminal rules) is left linear or linear. (Note : In this context, a linear rule is one which involves arrowheads of opposite directions, like $S \rightarrow ((S' \uparrow a) \downarrow b)$. This parallels the linear rule $S \rightarrow aS'b$ of string grammars.) Further, a regular grammar may be R:R or R:CF or R:CS according as all intermediate languages are regular, or all are context-free and at least one is not regular, or all are context-sensitive and at least one is not context-free. Similarly, we have (L:R), (L:CF) and (L:CS) grammars.

example 3.1 G_1 is an R:R grammar.

$G_1 = (V, I, P, S, \mathcal{L})$	
$V = V_1 \cup V_2$	
$V_1 = \{S, S_1\}$	non-terminals
$V_2 = \{a,b,c,x,y,z\}$	intermediates
$I = \{G, Y\}$	terminals
$P = P_1 \cup P_2$	

$P_1 = \{S \rightarrow (((S_1 \nearrow a) \nwarrow b) \downarrow c),\ S_1 \rightarrow (((S \nearrow x) \nwarrow y) \downarrow z)\}$

non-terminal rules

```
              G
           G     G
P2 = {S →     G      }        terminal rule
           G     G
              G
```

$\mathcal{L} = \{L_a, L_b, L_c, L_x, L_y, L_z\}$ set of intermediate languages

$L_a = L_c = \{G^n\langle G\rangle G^n\}$ where the letters of the arrowhead are written clockwise and the letter within $\langle\ \rangle$ appears at the vertex of the arrowhead.

$L_b = \{G^n\langle G\rangle G^{n-1}\}$, $L_y = \{Y^n\langle Y\rangle Y^{n-1}\}$

$L_x = L_z = \{Y^n\langle Y\rangle Y^n\}$

A typical derivation is shown below:

<u>Phase 1</u> $S \Rightarrow (((S_1 \nearrow a) \nwarrow b) \downarrow c)$

$\Rightarrow ((([[[S \nearrow x] \nwarrow y] \downarrow z] \nearrow a) \nwarrow b) \downarrow c)$

```
                  G
               G     G
Phase 2  ⇒ ((([[[     G     ↗ x] ↖ y] ↓ z] ↗ a) ↖ b) ↓ c)
               G     G
                  G
```

```
Phase 3                Y
                    G     Y
                 G     G
        ⇒ ((([[     G     Y  ↖ y] ↓ z] ↗ a) ↖ b) ↓ c)
                 G     G
                    G
```

```
                 Y
              Y     Y
           Y     G     Y
              G     G
⇒ ((([   Y       G     Y   ↓ z] ↗ a) ↖ b) ↓ c)
              G     G
                 G
```

```
                 Y
              Y     Y
           Y     G     Y
              G     G
⇒ (((      Y     G     Y   ↗ a) ↖ b) ↓ c)
              G     G
           Y     G     Y
              Y     Y
                 Y
```

```
                 G
              G     G
           G     Y     G
        G     Y     Y     G
           Y     G     Y
        G     G     G     G
*
⇒          Y     G     Y
        G     G     G     G
           Y     G     Y
        G     Y     Y     G
           G     Y     G
              G     G
                 G
```

Consider the generator model HCA for $L_1 = L(G_1)$. The HCA should start off in some specified initial configuration H_0, and generate all hexagonal arrays belonging to L_1. A natural choice for H_0 is the r.h.s of the terminal rule. The arrays of L_1, say H_0, H_1, ..., H_i, ... are such that H_i has i concentric hexagons of Y's. So the HCA can generate H_{i+1} from H_i in a very natural way, in 2 steps, as described by the transition function below.

$\delta(Y) = \overline{Y}$, $\delta(G) = \overline{G}$, $\delta(\overline{Y}) = Y$, $\delta(\overline{G}) = G$
(where $\delta(x) = y$ means that a cell in state x goes to state y independent of the states of its neighbours.)

$$\delta\left(\begin{array}{ccc} & a & \\ f & & b \\ & q_0 & \\ e & & c \\ & d & \end{array}\right) = q_0 \text{ if } a,b,c,d,e,f = q_0$$

$= \overline{Y}$ if exactly 1, or 2 adjacent neighbours, are in state G (i.e., the cell borders an H_i.)

$= G$ if exactly 1, or 2 adjacent neighbours, are in state $\overline{Y}$, (i.e., the cell is on an edge of H_{i+1}.)

Clearly, this HCA goes through a series of configurations C_0, C_1, ..., C_i, ... where every C_{2i} belongs to the language and every C_{2i+1} does not. As per the convention adopted in section 2, if C_i is a hexagon belonging to the language, then the bottommost cell D of C_i should go into a special state to indicate this. So we can modify the transition function such that

$$\delta\left(\begin{array}{ccc} & a & \\ f & & b \\ & q_0 & \\ e & & c \\ & d & \end{array}\right) = (G,\#) \text{ if } a = \overline{Y} \text{ and } b,c,d,e,f = q_0,$$

and keep the rest of the function unaltered. Also, the initial configuration is changed to

$$\begin{array}{ccc} & G & \\ G & & G \\ & G & \\ G & & G \\ & (G,\#) & \end{array}$$

The first 3 configurations this HCA goes through are shown in fig. 9.

Now, it is clear why the distinguished cell is allowed to change with time; it is visually more appealing. This is merely a matter of convenience; it is possible to fix the distinguished cell a priori and to send signals to it when it must enter a special state.

Another point to note is that in the above example, the concatenation of three arrowheads (a,b,c or x,y,z) is simulated simultaneously in one transition by the HCA; all boundaries of the hexagon expand simultaneously. This is possible because of symmetry in the resulting hexagons.

<u>example 3.2</u> G_2 is a R:CF grammar

$G_2 = (V, I, P, S, \mathcal{L})$
$V = V_1 \cup V_2$, $V_1 = \{S\}$, $V_2 = \{x,y,z\}$
$I = \{O, B\}$,
$P = P_1 \cup P_2$, $P_1 = \{S \rightarrow (((S \nearrow x) \nwarrow y) \downarrow z)$

$$P_2 = \{S \rightarrow \begin{array}{ccc} & B & \\ B & & B \\ & B & \\ B & & B \\ & B & \end{array}\}$$

$\mathcal{L} = \{L_x, L_y, L_z\}$, $L_x = \{O^n\langle B\rangle O^n\}$

$L_y = \{O^n\langle B\rangle O^n B\}$, $L_z = \{BO^n\langle B\rangle O^n B\}$

The first three members of $L(G_2)$ are shown in fig. 10.

In this grammar, again, successive hexagons $H_1, \ldots, H_i, \ldots$ can be generated by padding the previous hexagon with an outer border. Thus a generator model HCA for $L_2 = L(G_2)$ starts with initial configuration

```
      B
  B       B
      B
  B       B
    (B,#)
```

Let $H_0, H_1, \ldots, H_i, \ldots$ be an enumeration of the hexagons in L_2. The HCA goes through a sequence of transitions such that at time $t = i$, the configuration of the HCA is H_i, with node D indicating membership in L_2 by symbol #. The transition function can be given as follows:

$\delta(B) = B, \quad \delta(O) = O, \quad \delta([B,\#]) = B,$
$\delta(q_0) = O$ if q_0 is on the border but not a vertex,
$\quad = B$ if q_0 is a new vertex adjacent to B,
$\quad = [B,\#]$ if q_0 is a new vertex adjacent to $[B,\#]$.
$\quad = q_0$ otherwise

In general, for any hexagonal kolam array grammar (HKAG), a generator model HCA can be designed by examining the production rules present in the grammar. Let G be an HKAG $G = (V, I, P, S, \mathcal{L})$

$V = V_1 \cup V_2, \qquad V_1$ = non-terminals
$\qquad\qquad\qquad V_2$ = intermediates
$P = P_1 \cup P_2$
$\mathcal{L}$ = set of intermediate languages
I = set of terminals
P_1 has rules of the type $S_1 \to S_2 \nearrow a \mid S_2 \nwarrow b \mid S_2 \downarrow c$
where $S_1, S_2 \in V_1$, $a,b,c \in V_2$,

P_2 has a terminal rule $S_1 \to H$,
where $S_1 \in V_1$, H is a hexagonal array.

Following the pattern along which derivation proceeds, the HCA should generate strings over $S_1V_2^*$ using rules from P_1, where S_1 occurs on the l.h.s of the rule in P_2. It should then construct H, and concatenate arrowheads according to the string generated. A deterministic HCA would thus work as follows: The initial configuration is H, with cell D marked as the distinguished node. D can be viewed as the topmost cell of a 1-D CA with cells arranged vertically rather than horizontally. This CA can simulate a Turing machine which enumerates all strings generated using P_1. Such an enumeration is possible because these strings belong to a regular or linear language and hence form a recursive set. Whenever the CA enumerates a new string $S_1\alpha$, the enumeration process is suspended. Initially H exists in the configuration; at later stages it will have to be reconstructed. Since H is finite, its description can be included within the state of D, and H can be constructed. α is a string of intermediates, say, $a_1a_2\ldots a_n$. After H has been constructed, D sequentially examines a_i, i = 1 to n. Each a_i has, associated with it, a direction of concatenation, $\nearrow$ or $\nwarrow$ or $\downarrow$. Let $\nearrow$ be the direction of concatenation. Then D sends a signal to vertex B, signifying that an arrowhead from language L_{a_i} is to be appended to the two edges adjacent at B. To make all cells on these edges react simultaneously, B must initiate a firing squad algorithm on the edges. Then the cells again participate in simulating a Turing machine to enumerate the strings of L_{a_i}. When a string of appropriate length (with respect to concatenation) is generated, enumeration stops and the string is concatenated to the arrowhead. B now sends a signal to D to indicate that the concatenation is complete and the next symbol a_{i+1} may be scanned. Note that if the arrowhead is to be appended at D itself, then after concatenation, the new distinguished cell as well as the 1-D vertical CA below it are shifted one unit downwards. When D has finished processing all a_i, the resulting hexagon above D (i.e., ignoring the vertical string below D) is one which belongs to the language. So for one time unit, D enters a special state to indicate membership in L. It then erases the hexagon (to avoid overlap with the next one) and resumes the suspended process of enumerating strings over $S_1V_2^*$ as per P_1. Thus the HCA systematically generates all hexagons in L(G).

For accepting the language L = L(G) where G is an HKAG, the initial configuration of the HCA will be some convex hexagon H′, with the bottommost cell D marked as the distinguished node. The HCA has to decide whether $H' \in L(G)$. First consider some simple examples.

example 3.3 Consider G_1,the grammar defined in example 3.1. H′ is the input (initial configuration) of the acceptor HCA. The acceptor can merely reverse the steps of the generator, and, after every alternate step, check whether the hexagon remaining is H_0. This can be done as follows: All border cells in state G (or Y) with non-boundary adjacent cells in state Y (or G) go to state BG (or BY). For testing whether a cell is on the boundary, states BG and BY are considered equivalent to q_0. When a boundary cell is unable to make a transition, it sends a signal to D. (Signal travels vertically down and then along the edge to D.) When D gets such a signal, it initiates a checking algorithm which determines (i) whether the non-quiescent non-BG/BY cells form a convex hexagon, and, if yes, (ii) whether this convex hexagon is equal to H_0. If the answer to either (i) or (ii) is No, $H' \notin L(G_1)$, else $H' \in L(G_1)$. Since H′ is finite, the checks in (i) and (ii) run in finite time. Thus $L(G_1)$ can be accepted. In fact, rejection is also done, so the HCA described above is a recogniser for L_1.

In the general case, acceptance becomes more difficult. The technique used for the generator HCA can be used here also; i.e., generate strings $S_1\alpha$, $\alpha \in V_2^*$, using P_1, and now delete rather than append arrowheads according to α. If at any stage deletion fails because the arrowhead present is not a member of the intermediate language, then string $S_1\alpha$ is discarded and a new string $S_1\alpha'$ is generated. Also, if after all arrowheads corresponding to α have been deleted, the hexagon left behind is not H, the r.h.s. of the terminal rule in G, then $S_1\alpha$ is discarded and a new $S_1\alpha$ is generated. The input must be available for testing the new string, so the preceding deletes should be only logical, i.e., it must be possible to recover the input configuration. However, if for a $S_1\alpha$ generated using P_1, deleting arrowheads corresponding to α leaves behind H, then clearly the input H' belongs to L(G). So D goes into an accept state. This means that through this mode of acceptance, rejection may take infinite time; i.e., recognition may not always be possible.

Another method of acceptance is to use the generator model HCA as a subroutine to generate hexagons belonging to L(G). Every hexagon so generated can be shifted so that its edges AB and BC are aligned with those of the input H' (using algorithm 2). Then at each cell the two hexagons are compared. If no cell sends a reject value to D, then D accepts the input. Else the generator is resumed to generate further hexagons.

A parser for the language will not only determine membership of the input, but further, if the input belongs to the language, it will produce a sequence of rules of G, which, when applied in that order starting from the start symbol, derive the input. The acceptors described above can be modified to restricted parsers in the sense that recognition may take infinite time, but if the input is accepted, the corresponding sequence of rules is also generated. To do this, consider the first model. Strings $S_1\alpha$ are generated and tested. Along with the string, the rules used to generate it can also be stored. Then, if a string leads to success, the set of rules is already stored within the cellular array. In the second model, the generator HCA is used. This HCA can store, in the vertical 1-D array below its distinguished node, the rules used to generate the hexagonal array. Then in case the two hexagons match, the parsing sequence is again stored within the cellular array.

Both the accepting methods discussed above are slow and inefficient. However for most specific examples, efficient acceptors can be designed after an ad hoc examination of the production rules present.

4. GENERATING AND RECOGNISING CONTROLLED TABLE HEXAGONAL L-ARRAY LANGUAGES

L-systems were originally defined for string languages to describe the growth of biological systems. These systems have been extended to rectangular arrays[16], circular patterns[14], and hexagonal arrays[15]. In this section we will study the latter and their relation to HCA.

The controlled table hexagonal arrays[15] are either 0-L (context independent) or 1-L (context dependent, dependence on one neighbour). The starting point (axiom) is a hexagon, and there are three types of tables corresponding to the three directions of concatenation $\nearrow$, $\nwarrow$, and $\downarrow$. (6 directions may also be considered.) Each table consists of 0-L rules in normal form $a \rightarrow bc$ or 1-L rules in normal form $ab \rightarrow acd$ with neighbourhood context. Application of a table means that the set of rules in the table acts in parallel along the entire edge defined by the arrowhead. The sequence of application of tables in the arrowhead directions is controlled by regular, context-free or context-sensitive languages. Consider the example given below.

example 4.1 $G_3 = (V, H_0, \rho, C)$

$V = \{O, X\}$ is the set of terminals,

```
          X
       X     X
    X     O
H =    O     X    is the axiom,
    X     O
       X     X
          X
```

$\rho = \{T_1, T_2, T_3\}$ is the set of tables, and

$C = \{(T_1T_2T_3)^n \mid n \geqslant 0\}$ is the control language regulating the application of tables.

$T_1 = \{ OX \nearrow OOX, XX \nearrow XXX \}$

$T_2 = \{ XO \nwarrow XOO, XX \nwarrow XXX \}$

```
       O ↓ O      X ↓ X
T3 = {  X   O  ,  X   X }
          X          X
```

The first three members of this language are shown in fig. 11. The sequence of application of tables for generating H_1 is shown in fig. 12.

To give a generator HCA for this language, note that an HCA can incorporate the effect of a control string $T_1T_2T_3$ in one step, whereby all border cells change state from X to O, and all quiescent cells change state to X if they have a neighbour in state X. Thus, starting with the axiom H_0 as the initial configuration (bottommost cell in state [X,#] rather than X), the HCA generates all hexagons belonging to the language through the following transitions :

$\delta(O) = O, \quad \delta(X) = O, \quad \delta([X,\#]) = O,$

$\delta(q_0)$ = X if all non-quiescent neighbours are in state X,

= [X,#] if there is only one non-quiescent neighbour, in state [X,#]

= q_0 otherwise.

example 4.2 $G_4 = (V, H_0, \rho, C)$ where

$V = \{O, B, P, X\}$

```
          X
       X     X
H0 =      X
       X     X
          X
```

$\rho = \{T_1, T_2, T_3\}$,

$T_1 = \{B \nearrow BO,\ O \nearrow OO,\ P \nearrow PO,\ X \nearrow XO\}$
$T_2 = \{B \nwarrow BB,\ O \nwarrow BO,\ P \nwarrow BP,\ X \nwarrow BX\}$
$T_3 = \{B \underset{P}{\downarrow} B,\ O \underset{P}{\downarrow} O,\ P \underset{P}{\downarrow} P,\ X \underset{P}{\downarrow} X\}$

Here T_1 has the effect of appending an arrowhead with all Os, T_2 appends arrowheads with all Bs, and T_3 appends arrowheads with all Ps. This is an example of a grammar where control $\{T_1^nT_2^nT_3^n \mid n \geqslant 0\}$ gives a language different from that obtained with control $\{(T_1T_2T_3)^n \mid n \geqslant 0\}$. Let C be the former control, i.e., a context-sensitive control. To generate $L_4 = L(G_4)$, an HCA must start with initial configuration H_0, and revert to it after generating each H_i, because H_{i+1} cannot be directly generated from H_i. To achieve this, let the configuration be H_i, $i > 0$ (obtained through control $T_1^iT_2^iT_3^i$), and let D_i (the distinguished cell of H_i) be in state [P,#]. D_i sends a signal to all quiescent cells bordering H_i, to change to a special state \$. D_i then fires all edges which delete symbols P, B, O and then pass the delete signal inwards. So in i time steps the configuration changes to one with axiom H_0, markers \$ along the boundary positions of H_{i+1} (to be constructed), and special state S at cell D_i of H_i. See fig. 13. D_i now sends a signal to B which fires off edge ABC for applications of table T_1. With each application of T_1, edge ABC advances towards the \$s. When it reaches the \$ edge, application of T_1 is stopped and a signal is sent to F. The process is repeated on edge EFA, applying T_2, until \$s are reached, and then on edge CDE, with table T_3. At this stage, D_i reverts to state P, D_{i+1} enters state [P,#], and the process of generating H_{i+2} begins. (H_1 is generated from H_0 by a slight modification of the above.) Thus all hexagons belonging to L_4 are generated.

For the general case, let $G = (V, H_0, \rho, C)$ be a controlled table hexagonal L-array grammar. To generate L(G), the HCA begins in the initial configuration H_0. As in the case of HKAG, the 1-D CA vertically below D systematically generates all strings in C. (C may be regular, CF or CS.) For every such string α generated, the HCA constructs axiom H_0 (which may have been overwritten for the previous string), and, by sending signals to appropriate vertices, applies tables in the sequence defined by α.

The process is similar to that of appending arrowheads in an HKAG generator. One point of difference is that in a 1-L system, the quiescent boundary cells have to change state depending on not only the boundary cells but also cells one unit inside the boundary. This requires an intermediate step. For instance, along the $\nearrow$ direction, to implement a rule ex $\rightarrow$ egh, the quiescent cell does not know state e directly. So the intermediate step brings information outwards as follows:

$$\delta\left(\begin{array}{ccc} & a & \\ f & & b \\ & x & \\ e & & c \\ & d & \end{array}\right) = [ex].$$

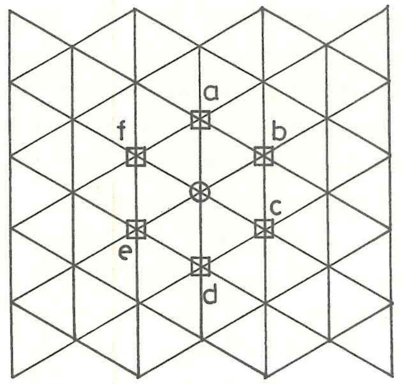
In the next step, the e and x cells are in states ye, ex for some y. So the quiescent cell (in position b) now knows both e and x directly from its neighbour and can accordingly change state. Such intermediate transitions must be defined for all directions.

Recognising such languages follows the method described in section 3: generate-and-compare. So it is not discussed here. In both types of languages, this recognition method will be inefficient, and it will be of interest to investigate efficient recognition algorithms for the HCA.

5. CONCLUSION

In this paper, hexagonal cellular automata have been introduced as a variation of the 2-D RCA, and their relationship to hexagonal array languages has been investigated. One of the applications of hexagonal arrays is to the 2-D representation of 3-D scenes composed of rectangular parallelepipeds. Manipulating (rotation, translation and scaling) such scenes is an important part of computer graphics software. It would be very useful if efficient parallel algorithms for these, running on 2-D or even 3-D cellular processors, could be devised. Also, it is of interest to find more efficient ways of accepting hexagonal array languages. This could assist in pattern recognition where the patterns are types of blocks.

FIGURES



○ : cell in HCA

□ : its neighbour

Figure. 1.
Neighbourhoods in HCA

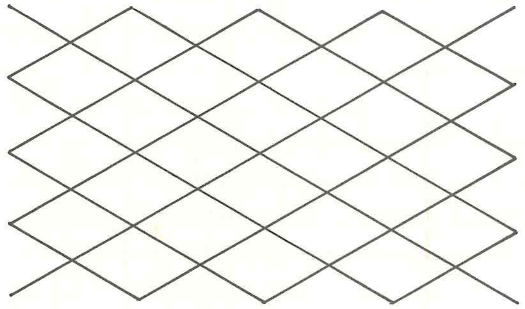


Figure. 2.
HCA with family of vertical lines missing.

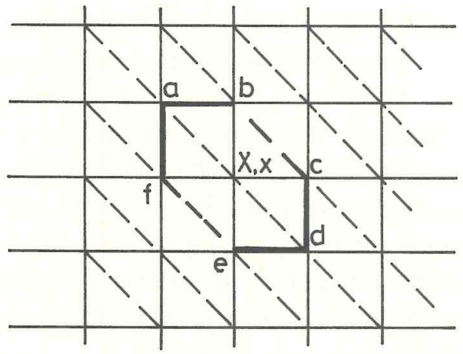


Figure. 3.
HCA embedded on RCA

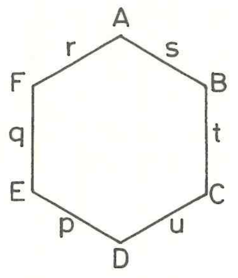


Figure. 4.
Lengths of sides of a convex hexagon.

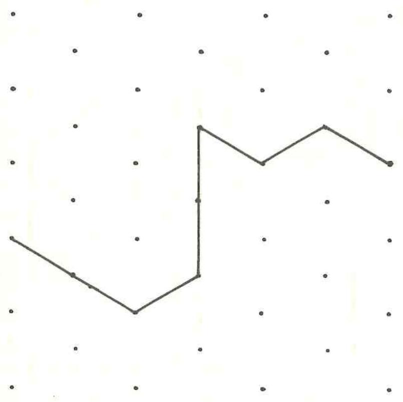


Figure. 5.
1-D CA embedded in HCA configuration

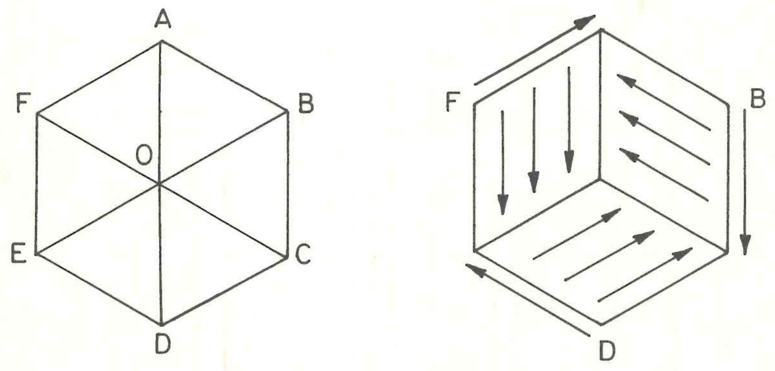


Figure. 6. Firing Squad Algorithm

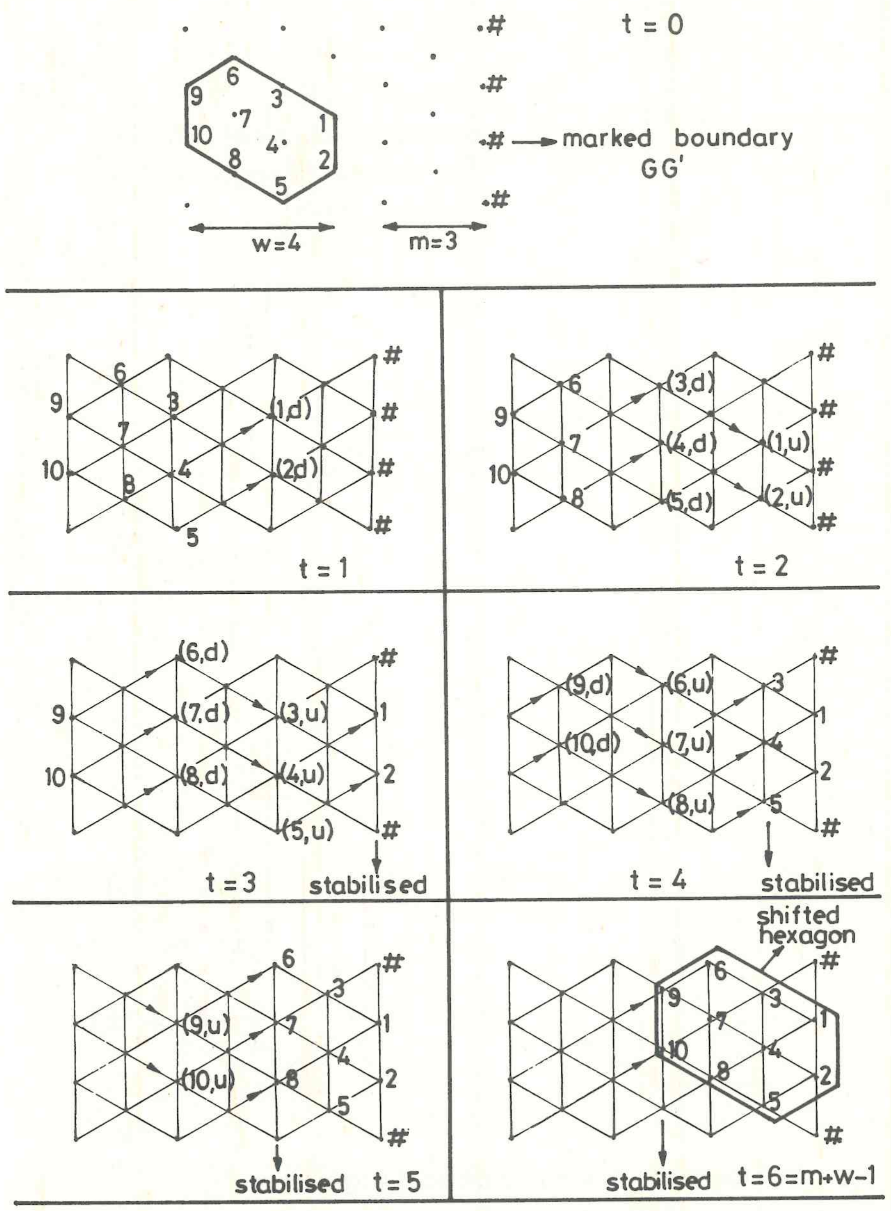


Figure. 7. Algorithm 6 illustrated

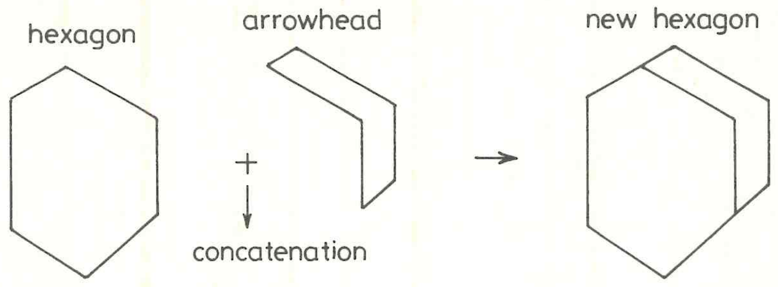


Figure. 8. Arrowhead concatenation

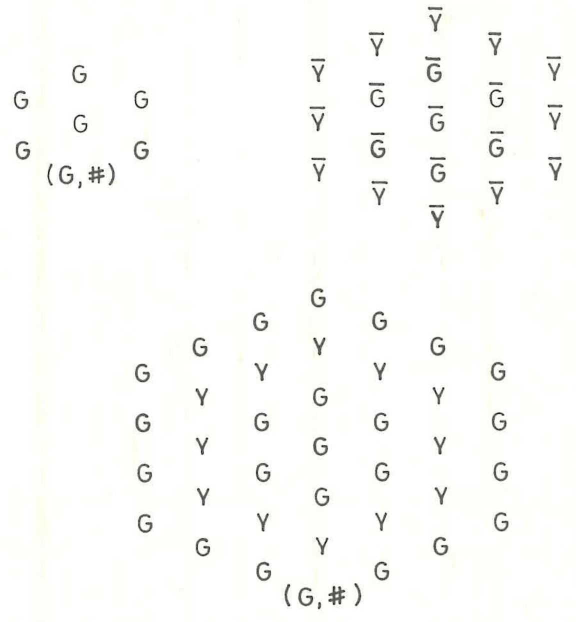


Figure. 9. The first 3 configurations of generator HCA.

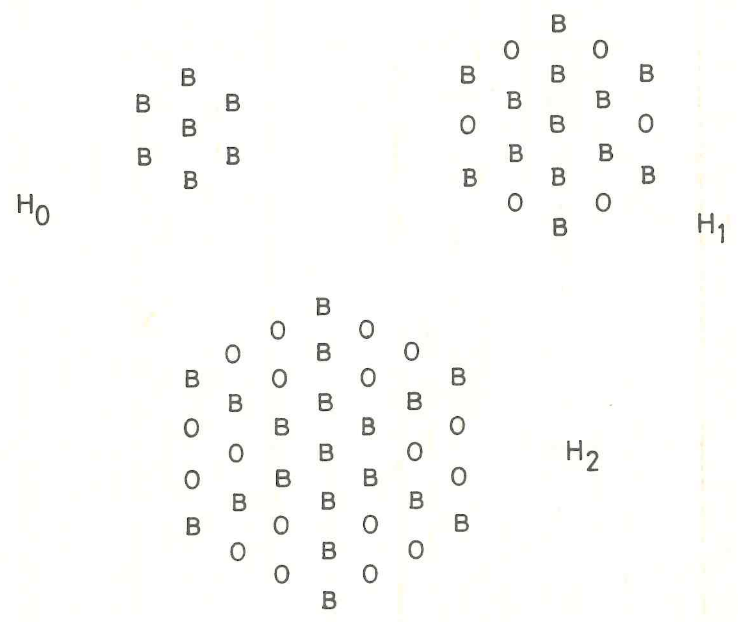


Figure. 10. The first three members of $L(G_1)$

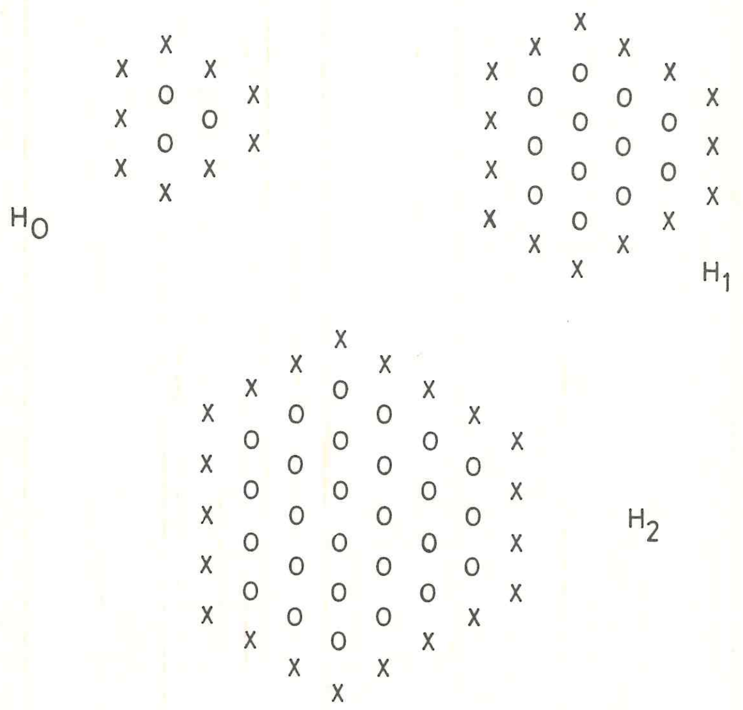
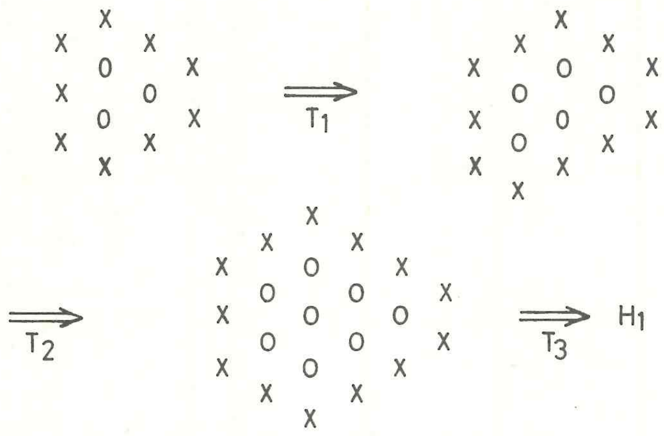


Figure. 11. The first three members of $L(G_3)$.



Control string = $T_1 T_2 T_3$

Figure. 12 Derivation of H_1 in $L(G_3)$

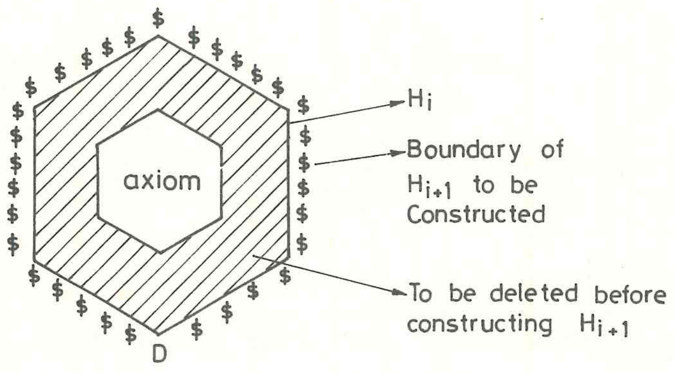

axiom
Hi
Boundary of Hi+1 to be Constructed
To be deleted before constructing Hi+1
D

Figure. 13. Generator HCA for $L(G_4)$

References

1. Codd, E. F., "Cellular Automata", ACM Monograph Series(1968).

2. Culik, K., III, and Karhumaki, J., "On totalistic systolic networks", Information Processing Letters 26, 231-236 (1987/88).

3. Ibarra, O. H., Kim, S. M., and Moran, S., "Sequential Machine characterizations of trellis and cellular automata and applications" SIAM Journal of Computing 14, 426-447 (1985).

4. Krithivasan, K., Thulasiraman, K., and Swamy, M. N. S., "Systolic pyramid automata - properties and characterizations", submitted for publication.

5. Krithivasan, K., and Mahajan, M., "Hexagonal cellular automata", Technical Report, Dept. of Computer Science and Engg., I.I.T., Madras (November 1988).

6. Minsky, M., "Computation : Finite and Infinite Machines", Englewood Cliffs, N.J., Prentice Hall (1967).

7. Narasimhan, R., and Reddy, V. S. N., in "Some experiments in scene generation using COMPAX in Graphics Languages" (F. Nake and A. Rosenfeld, Eds.), pp. 111-120, Amsterdam, North-Holland (1972).

8. Preston, K., Jr., "Applications of cellular automata in biomedical image processing", in 'Computer Techniques in Biomedicine and Medicine' (Enoch Haga, Ed.), Auerbach, Philadelphia (1973).

9. Rosenfeld, A., and Pfaltz, J. L., "Distance functions on digital pictures", Pattern Recognition 1, 33-61 (1968).

10. Sankar, P. V., "A vertex coding scheme for interpreting ambiguous

trihedral solids", TR 127, Tata Institute of Fundamental Research, Bombay (November 1974).

11. Siromoney, G., Siromoney, R., and Krithivasan, K., "Abstract families of matrices and picture languages", Computer Graphics and Image Processing 1, 284-307 (1972).

12. Siromoney, G., Siromoney, R., and Krithivasan, K., "Picture languages with array rewriting rules", Information Control 22, 447-470 (1973).

13. Siromoney, G., Siromoney, R., and Krithivasan, K., "Array grammars and kolam", Computer Graphics and Image Processing 3, 63-82 (1974).

14. Siromoney, G., and Siromoney, R., "Radial grammars and radial L-systems", Computer Graphics and Image Processing 4, 361-374 (1975).

15. Siromoney, G., and Siromoney, R., "Hexagonal arrays and rectangular blocks", Computer Graphics and Image Processing 5, 353-381 (1976).

16. Siromoney, R., and Siromoney, G., "Extended controlled table arrays", TR-304, Computer Science Centre, University of Maryland (May 1974).

17. Smith, A. R., III, "Cellular automata and formal languages", in Proceedings, 11th SWAT, pp. 216-224 (1970).

18. Smith, A. R., III, "Real-time language recognition by one-dimensional cellular automata", JCSS 6, 233-253 (1972).

19. Sommerhalder, R., and van Westrhenen, S. C., "Parallel language recognition in constant time by cellular automata", Acta Informatica 19, 397-407 (1983).