Contents

List of Figures

Abstract

CA are an abstract model for parallel language recognition. Due to their inherent parallelism, they are intrinsically faster than many other language recognition devices. Thus a lot of interest has centred on how fast CAs can compute, and a fair amount of research has focussed on whether at all there is a difference between real-time, linear-time, and unbounded-time CA computation (rCA, lCA and CA respectively). It is not even known whether these classes are distinct when restricted to tally sets. Another longstanding open problem is whether one-way communication reduces the power of a CA. The motivation for this thesis has been to further refine these open problems.

In this thesis, we first define time-varying CA ($TVCA$) and then give different interpretations to the ensuing computations. A $TVCA$ is essentially a CA where the component cells act differently at different time steps. The variation is dictated by some prespecified control set. This corresponds to the notion of an external control, through global broadcast to all cells. Our interest is language-theoretic; we study the language recognition capabilities of $TVCA$ and related models with an approach towards characterising the computational power of these models.

We have studied how the complexity of the controlling languages affects the power of the resulting $TVCA$. At one extreme, ultimately periodic tally languages do not impart any additional power to $TVCA$ over CA; at the other extreme, CA control makes $TVCA$ strictly more powerful than CA. In between, using rCA or lCA control, we obtain languages which are contained in OCA, but which do not appear to be in lCA. This supports the conjecture that lCA are not as powerful as OCA or CA.

We have also examined the effect of varying the number of controlling languages. While for linear-time and unbounded-time computation this is not a crucial factor, it appears to be so for real-time computation.

The controlling language L in a 2-function $TVCA$ can be viewed as an oracle to which the $TVCA$ makes implicit queries. This is a restricted form of oracle access, since queries must be made at each step, and in a specific order. Nonetheless, we show that the mechanism

is powerful enough to exhibit separations as well as strong separations of the relativised classes rCA, lCA and CA. The equality $CA = DSPACE(n)$ becomes a containment from left to right if CA are relativised via $TVCA$ and the $DSPACE(n)$ machines via the unrestricted query model. On the other hand, if the $DSPACE(n)$ machines use the bounded query model, then the equality changes to a containment from right to left. However, the $lCA \subseteq DSPACE(n)$ relationship relativises in both cases.

Restricting the study to tally languages, we find that there exists a hierarchy of languages between the complexity classes rCA and OCA. The hierarchy is built with rCA and lCA at the base, and each level is obtained by relativising rCA and lCA with respect to languages at the previous level. Examining the structure of this hierarchy, we have found that rCA control does not increase the power of rCA beyond lCA, and it does not increase the power of lCA at all. Of course, if it should turn out that rCA and lCA (when restricted to tally languages) are equal, then all classes in this hierarchy are equal. However, the difficulty of showing any more equalities in this hierarchy offers further evidence that the classes of tally languages in rCA and lCA are indeed distinct.

We next consider an interpretation of $TVCA$ as nondeterministic CA; instead of having controlling languages, the $TVCA$ nondeterministically decides, at each time step, which transition function to use. With this interpretation, we can now study classes of time-bounded nondeterministic CA, which we refer to as $NTVCA$. This mode of nondeterminism differs significantly from the traditional notion, and many standard proof techniques fail to carry over. Despite this, we have obtained some non-trivial results relating the classes so defined. The main results are that given unbounded time, the two modes of nondeterminism do not differ, and that in the mode we define, one-way communication does not decrease the power of the unbounded-time CA. We have also shown that the membership problem for real-time $NTVCA$ is NP-complete, and that linear-time $NTVCA$ are no more powerful than OCA. We have also modelled restricted nondeterminism by allowing the $TVCA$ to choose transition functions only in sequences from a specified set.

Instead of checking whether at least one path of the $TVCA$ ends in an accepting configuration, we could check if more than half of them end in accepting configurations. This gives the operation of the $TVCA$ a probabilistic interpretation. A further generalisation of this concept gives us another interpretation of $TVCA$, as alternating CA. With this interpretation, fast checking of properties of substrings becomes possible. We have briefly considered such probabilistic and alternating CA.

Finally, we have considered the closure of CA classes under various language operations. This is mainly motivated by the fact that rCA are closed under reversal if and only if $rCA = lCA$. Some of the closure proofs involve interesting automata-theoretic constructions. We have also tried to relate the closure of the CA classes to the newly obtained nondeterministic classes defined in terms of $NTVCA$, and to the alternating CA classes.

CERTIFICATE

This is to certify that the thesis entitled "Studies in Language Classes Defined by Different Types of Time-Varying Cellular Automata", which is being submitted by Meena Bhaskar Mahajan to the Indian Institute of Technology, Madras, for the award of the degree of Doctor of Philosophy, is a record of bonafide research work carried out by her under my supervision and guidance. The contents of this thesis have not been submitted to any other Institute or University for the award of any degree or diploma.

Madras 600 036.
November 1992.

Dr. Kamala Krithivasan.
Professor,
Department of Computer Science
and Engineering,
I.I.T., Madras.

STUDIES IN LANGUAGE CLASSES DEFINED BY DIFFERENT TYPES OF TIME-VARYING CELLULAR AUTOMATA

A THESIS

submitted for the award of the Degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND ENGINEERING

by

MEENA BHASKAR MAHAJAN

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY

MADRAS 600 036, INDIA

NOVEMBER 1992

Chapter 1

Introduction

Cellular automata deal with large collections of interconnected finite state automata, each automaton being thought of as a cell. The pioneering work in this field was done by John von Neumann and reported by A. W. Burks [Neu66]; similar studies are also reported in [Cod68]. Von Neumann's study focussed on a particular cellular space and investigated conditions under which it is possible to exhibit (a) universal computing—the computation of all computable functions, and (b) universal constructibility—embedding in the cellular space an automaton A which, given the specifications of any constructible automaton B, builds B and sets it free to work independently of A. This is akin to asking whether a robot can mechanically, given the specifications, build another robot meeting those specifications.

von Neumann's interest in studying these questions was in the development of organisation in biological systems. What he proved through his cellular space construction is well summarised by A. R. Smith III [Smi76] as follows:

> "... that a very large array of not very powerful computers operating in parallel can be programmed to be quite powerful, and that the program to accomplish this can spread copies of itself throughout the array. So, not only is the array of relatively powerless computers capable of computing what one powerful computer can, but it is also capable of computing what a large number of powerful computers can compute simultaneously, with only one of the programs being provided by man."

Subsequently the study of CA continued under differing motivations. CA have been usefully applied to the study of a variety of complex systems — lattice gas dynamics, statistical mechanics, etc. A very good survey of this kind of work appears in Wolfram's book [Wol86]; also see [DGT85]. Several related and similar models have also been independently studied. A survey of such "polyautomata" appears in A. R. Smith III's article [Smi76], from which the above extract is taken.

To the computer engineer, the significance of CA lies in the fact that CA have enormous computing power obtained by interconnecting several fairly primitive cells in a regular fashion. Thus they possess homogeneity of components, facilitating mass production, easily incrementable power, and ease of reorganising these increments to suit various special needs. A major spurt in CA investigation in the last few years stems from the advances in devices for parallel computation. Special interest has arisen in systolic systems [Kun79, Kun80, Kun82] —arrays of uniformly interconnected identical processors, working in a synchronous manner, while data travels across the array in a rhythmic fashion. Such systems have homogeneity, modularity, simplicity—all factors making them easy to implement in VLSI circuitry. The design and study of systolic systems and algorithms has benefitted from the study of bounded CA systems. Notable examples are the work of Culik et al [BC84, CC84, CGS84a, CGS84b, CGS86] on systolic trellis automata (which are equivalent to bounded-space real-time one-way CA) and of Ibarra et al [IK84, IKP86, IPK85b] on linear systolic arrays modelled as 1- and 2-Dimensional CA, one-way CA (OCA), and a closely related model, the iterative arrays. Their work has resulted in the development of hosts of easy-to-implement systolic algorithms using the techniques developed for CA and equivalent sequential machines. Another instance of directly applying CA techniques is the work reported by Pries et al. and others [DP88, KA87, PTC86] on exploiting the group properties of additive CA to implement on-chip test-pattern-set-generators for VLSI circuits.

CA, due to their inherent parallelism, are intrinsically faster than many other computation models [Smi72]. It is but natural that some interest will centre on how fast CAs can compute. A lot of research has focussed on whether at all there is a difference between real-time, linear-time, and unbounded-time CA computation (rCA, lCA and CA respectively). Some of this work is reported in [BC84, CC84, IJ88, Smi71, Smi72]. (In real-time language recognition, if the input is of length n, the CA requires at most n steps to determine whether or not the input belongs to the language. In linear-time language recognition, there is a constant c such that on inputs of length n, at most cn time is required.) Another longstanding open problem is whether one-way communication reduces the power of a CA; related research results are reported in [CIV88, Dye80, IJ87, UMS82].

The motivation for this thesis has been to further refine these open problems, in an attempt to come closer towards solving them. With this end, we first define time-varying CA ($TVCA$) and then give different interpretations to the ensuing computations. A $TVCA$ is essentially a CA where the component cells act differently at different time steps. The variation is dictated by some prespecified control set. This corresponds to the notion of an external control, through global broadcast to all cells.

The idea of imposing external control in some fashion is not new. There are several instances in automata and language theory of modifying the generative/recognising power of a grammar or a language through some form of external control—regulated rewriting is a well-researched area in language theory [KD85, Sal73]. A practical application is in compiler design— while most programming languages are context-sensitive, efficient parsing is known only for context-free languages. So the programming languages are usually modelled as context-free languages where the context-sensitive properties are incorporated not through the form of the productions (which would make parsing difficult) but through regulating and controlling their usage. Similar uses can be envisaged for $TVCA$.

This thesis, however, is not directly concerned with such applications. Our interest is language-theoretic; we study the language recognition capabilities of $TVCA$ and related models with an approach towards characterising the computational power of these models. Time-variations have been shown to vastly increase the generative power of regular grammars [KD84, Sal73]. Analogously, time-varying finite-state automata and pushdown automata have been shown to be far more powerful language acceptors than their non-time-varying counterparts [KD86, KS88]. Our attempt has been to perform a similar investigation for time-varying cellular automata.

A $TVCA$ is similar to a CA except in one respect: the transition function which specifies the next state of a cell in terms of its current state and the current state of its neighbours is not fixed. Instead, there is a finite set of local transition functions, and at each time step, exactly one of them is used, depending on how many time steps have elapsed since the $TVCA$ operation began. This dependence on time can be expressed in terms of a set of control languages which are tally (ie. over a unary alphabet $\{0\}$): at the i^{th} time step, use the j^{th} function if and only if the i^{th} string over the alphabet belongs to the j^{th} controlling language.

We have studied how the complexity of the controlling languages affects the power of the resulting $TVCA$. At one extreme, ultimately periodic tally languages do not impart any additional power to $TVCA$ over CA; at the other extreme, CA control makes $TVCA$ strictly more powerful than CA. In between, using rCA or lCA control, we obtain languages which are contained in OCA, but which do not appear to be in lCA. This supports the conjecture that lCA are not as powerful as OCA or CA.

We have also examined the effect of varying the number of controlling languages. While for linear-time and unbounded-time computation this is not a crucial factor, it appears to be so for real-time computation.

The controlling language L in a 2-function $TVCA$ can be viewed as an oracle to which the $TVCA$ makes implicit queries; at time step i, it queries the oracle on membership of 0^i in L. This is a restricted form of oracle access, since queries must be made at each step, and in a specific order. Nonetheless, the mechanism is powerful enough to exhibit separations as well as strong separations of the relativised classes rCA, lCA and CA. The equality $CA = DSPACE(n)$ becomes a containment from left to right if CA are relativised via $TVCA$ and the $DSPACE(n)$ machines via the unrestricted query model. On the other hand, if the $DSPACE(n)$ machines use the bounded query model, then the equality changes to a containment from right to left. However, the $lCA \subseteq DSPACE(n)$ relationship relativises in both cases.

Restricting the study to tally languages, we find that there exists a hierarchy of languages between the complexity classes rCA and OCA. The hierarchy is built with rCA and lCA at the base, and each level is obtained by relativising rCA and lCA with respect to languages at the previous level. Examining the structure of this hierarchy, we have found that rCA control does not increase the power of rCA beyond lCA, and it does not increase the power of lCA at all. Of course, if it should turn out that rCA and lCA (when restricted to tally languages) are equal, then all classes in this hierarchy are equal. However, the difficulty of showing any more equalities in this hierarchy offers further evidence that the classes of tally languages in rCA and lCA are indeed distinct.

Instead of providing the $TVCA$ with a controlling language deciding which transition function is to be used at each step, we can allow the $TVCA$ to proceed in all possible ways simultaneously and check whether any one (or more) such way leads to an accepting configuration. This is a kind of nondeterministic computation; the $TVCA$ nondeterministically decides, at each time step, which transition function to use. With this interpretation, we can now study classes of time-bounded nondeterministic CA, which we shall refer to as $NTVCA$. This mode of nondeterminism differs significantly from the traditional notion, and many standard proof techniques fail to carry over. Despite this, we have obtained some non-trivial results relating the classes so defined. The main results are that given unbounded time, the two modes of nondeterminism do not differ, and that in the mode we define, one-way communication does not decrease the power of the unbounded-time CA. We have also shown that the membership problem for real-time $NTVCA$ is NP-complete, and that linear-time $NTVCA$ are no more powerful than OCA.

Restricted nondeterminism can be modelled by allowing the $TVCA$ to choose transition functions only in sequences from a specified set. This offers more choice than the relativised CA, where the controlling language specifies a single sequence of transition functions. At the same time, it is not as general as unrestricted $NTVCA$. We consider such restricted $NTVCA$ where the allowable sequences are specified by simple constraints — in a 2-function $TVCA$, the sequence is allowed to alternate between two transition functions, δ_1 and δ_2, at most k times, for some given constant k, or is allowed to use δ_2 at most k times. We show that the classes defined through this form of nondeterminism are likely to be strictly stronger than deterministic CA, and yet strictly weaker than the unrestricted nondeterminism classes.

Instead of checking whether at least one path of the $TVCA$ ends in an accepting configuration, we could check if more than half of them end in accepting configurations. This gives the operation of the $TVCA$ a probabilistic flavour. Again, restricted classes can be studied; of the paths possessing some specific property, more than half must be accepting paths. We have shown some results concerning such $TVCA$, the most notable being that linear-time probabilistic $TVCA$ are no more powerful than OCA.

A further generalisation of this concept brings us to alternating CA. States of the CA are of one of the following four types: universal, existential, accepting or rejecting. A configuration is accepting or rejecting if the leftmost cell is in such a state. If the leftmost cell is in an existential (universal) state, then at least one (all) of the subtrees of the computation tree at that point must be accepting. This allows fast checking of properties of substrings. Some of these alternating CA classes are used subsequently to locate closures of CA classes under various language operations.

Closure properties of language classes play an important part in our understanding of the structure of the classes; they also sometimes give more insight into the containments amongst the classes. For instance, it is not known whether rCA are closed under reversal, but it is known that this is the case if and only if $rCA = lCA$ [IJ88]. We have considered the closure of CA classes under various language operations. Some of the closure proofs involve interesting automata-theoretic constructions. The most notable results are that for L in rCA, given a string x, checking whether some or all or none or an odd number of prefixes of x are in L (ie. checking whether x is in $\exists PRE(L)$, $\forall PRE(L)$, $MIN(L)$, $\oplus PRE(L)$) can also be done in real time. Also, for L in rCA or lCA, the string obtained by interchanging, pairwise, letters from strings in L ($SHUFFLE(L)$), can also be recognised by an rCA or lCA.

We have also tried to relate the closure of the CA classes to the newly obtained non-deterministic classes defined in terms of $NTVCA$, and to the alternating CA classes. For instance, consider recognising the language $\exists PRE(L)$. For L in $rOCA$ and rCA, we have shown that $\exists PRE(L)$ is also in $rOCA$ or rCA respectively. For L in lCA, however, the fastest CA construction we have for recognising $\exists PRE(L)$ requires $O(n^2)$ time. But using a restricted $NTVCA$, we can accept the same language in linear time. Several other such closures have been expressed in terms of $NTVCA$ and restricted $NTVCA$.

The thesis is organised as follows.

In chapter 2 we describe the notation to be used in the rest of the thesis. We then give the definition of CA and the associated language classes, and survey some of the known results about these classes. In chapter 3 we introduce $TVCA$. We study the effect of two important parameters—the complexity of the controlling languages, and the number of controlling languages—on the power of the resulting $TVCA$. In the next two chapters, the operation of a $TVCA$ is interpreted as that of a relativised CA, with the controlling language acting as the oracle. Special attention is paid to tally language recognition. Chapter 6 considers an interpretation of $TVCA$ without controlling languages as a nondeterministic CA computation. In chapter 7 we briefly look at two other interpretations of $TVCA$ computation—as a probabilistic CA and as an alternating CA. Finally in chapter 8 we consider closure properties of CA language classes. Chapter 9 presents a discussion of the extent and significance of the work done and summarises the problems which remain open at the end of this work.

Chapter 2

Preliminaries and Basic Results

In this chapter the notation and definitions used in the rest of the thesis are presented, and basic results in the field are surveyed. The survey is not meant to be comprehensive, but it covers all the results used or cited in the rest of the thesis. Most of the results are only mentioned; however, in a few cases, where the proof techniques have been put to use in subsequent chapters, the proofs are sketched.

2.1 Notation

Throughout this thesis, the symbol Σ (or Σ', $\overline{\Sigma}$, ...) will be used to denote a finite alphabet. Small letters $a, b, c \ldots$, or bits 0, 1 will be letters in the alphabet; small letters $u, v, w, x, y \ldots$ will typically denote strings of finite length over the alphabet. Σ^* (respectively Σ^+) denotes the set of all finite (respectively, finite non-zero) length strings over Σ. If $x \in \Sigma^*$, $|x|$ denotes the length of x. x^R denotes the string obtained by writing the letters of x in reversed order. For a set S, the cardinality of the set is denoted by $\|S\|$. $\mathbf{N}$ (respectively $\mathbf{N}^+$) is the set of (positive) natural numbers. For any positive natural number n, let $1x$ be the unique binary representation of n with no leading zeroes. Then x, denoted $\langle n \rangle$, is the standard representation for the number n.

δ is used to denote transition functions, and Q is used to denote a finite set of states. States are usually represented by the letters $p, q, \ldots$; however frequently states will coincide with letters of an alphabet, in which case those letters will be used.

A language is a subset of Σ^*; languages will usually be denoted by the capital letter L. $\overline{L}$ is the complement of L, and contains all those strings from Σ^* which are not in L. L^R is the reversal of L, and contains exactly those strings x for which x^R is in L.

We assume a basic familiarity with formal language theory, finite state automata and Turing machines; for a general introduction to automata theory, see [HU79]. For language classes obtained through resource-bounded Turing machine computations, we use the standard notation from [BDG88, BDG90]. Thus $DSPACE(s(n))$ (respectively $NSPACE(s(n))$, $ASPACE(s(n))$) is the class of languages which can be accepted by deterministic (respectively nondeterministic, alternating) Turing machines using no more than $s(n)$ space on their worktapes for inputs of length n. (The input is on a separate read-only tape; thus $s(n)$ could well be less than n.) The classes $DTIME(t(n))$, $NTIME(t(n))$ and $ATIME(t(n))$ are similarly defined; now the corresponding machines are constrained to use only $t(n)$ time, on inputs of length n. The union of $DTIME(p(n))$, $NTIME(p(n))$ and $DSPACE(p(n))$ classes respectively, over all polynomial functions p, gives the classes P, NP and $PSPACE$ respectively.

2.2 Definitions

Cellular automata (CA) are a simple model for parallel recognition of languages. A CA consists of an array of identical finite-state machines (FSM), one for each letter of the input. (Our work is restricted to one-dimensional CA. For results on CA of higher dimensions, see [Col69, IPK85b].) The FSMs are called cells. Each cell initially contains one letter of the input. The cells change their states in a synchronised fashion, at discrete time steps. The state of a cell at a time instant is a function of the states of cells in its neighbourhood at the previous time instant; this function is the transition function of the CA and is the same for all cells. Throughout this work, we will consider the three cell or two cell neighbourhoods. In the former, the neighbourhood of a cell consists of itself, the cell to its immediate left, and the cell to its immediate right. In the latter, the right neighbour is excluded. See Figures 2.1 and 2.2.

Let $c(i,t)$ denote the state of the i^{th} cell at time t. Then, as mentioned above, the CA is

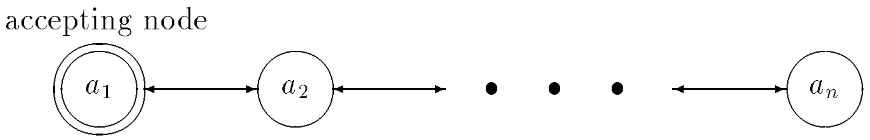


Figure 2.1: A cellular automaton

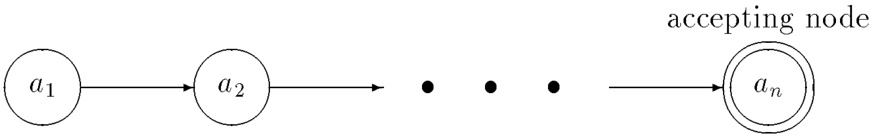


Figure 2.2: A one-way cellular automaton

initialised by setting $c(i,0)$ to a_i, where the input is $a_1a_2\ldots a_n$. Let the transition function of the CA be δ. Then for $1 \le i \le n$,

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

Thus the subsequent operation of the CA is autonomous.

(The cells at the boundary of the array have one neighbour missing. These cells take a special state # as the missing argument when applying the transition function.)

The leftmost cell of the CA is a special cell; it denotes whether the input is accepted by the CA. Let Q be the finite set of states each cell can take. A subset of this, A, is specified as the set of accepting states. Now, if the leftmost cell ever enters a state from A, then we say that the input has been accepted. Note that for nontrivial language recognition, acceptance requires at least n time steps on an input of length n.

If the two cell neighbourhood is considered, then

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

Now the rightmost cell is the special cell denoting acceptance. Such CA are called one-way CA (OCA), since information can only flow in one direction (from left to right).
Formally, CA and OCA are defined as follows.

Definition 2.1 *A cellular automaton is a 4-tuple* $C = (Q, \#, \delta, A)$ *where*

- Q *is a finite set of states*

- $\# \in Q$ *is the boundary state*
- $\delta : Q \times Q \times Q \rightarrow Q$ *is the local transition function satisfying*

$$\delta(a, b, c) = \# \qquad \text{if and only if } b = \#$$

- $A \subseteq Q$ *is the set of accepting states*

Definition 2.2 *A one-way cellular automaton is a 4-tuple* $C = (Q, \#, \delta, A)$ *where* Q, $\#$ *and* A *are as in Definition 2.1, and* $\delta : Q \times Q \rightarrow Q$ *is the local transition function satisfying*

$$\delta(a, b) = \# \qquad \text{if and only if } b = \#$$

The set of strings accepted by the CA C is denoted $L(C)$.

The operation of the CA (and OCA) is frequently represented using a time-space diagram. This is an array where the topmost row has the input configuration, and successive configurations appear in successive rows beneath it. Thus the i^{th} row gives the configuration of the CA after i time steps, and the j^{th} column gives the sequence of states entered by the j^{th} cell of the CA. Such diagrams give a visual representation of the CA computation and make it more easily comprehensible. Signals travelling across the array of FSMs are shown in such diagrams by lines of varying slopes, depending on the speed at which the signal is travelling.

A CA C is said to operate in $T(n)$ time if for each n, for each string w of length n, if w is accepted then it is accepted within $T(n)$ time steps. In other words, if $c(1,0)c(2,0)\ldots c(n,0) = w$ and $w \in L(C)$, then $\exists t \leq T(n)$ such that $c(1,t) \in A$. Here $T(n)$ could be any function $T : \mathbf{N}^+ \rightarrow \mathbf{N}^+$. Of special interest are the cases when $T(n) = n$, giving "real-time" CA (rCA and $rOCA$), and $T(n) = cn$ for some constant c, giving "linear-time" CA (lCA and $lOCA$). If we allow the CA or OCA to use an unbounded amount of time, the class of languages accepted is denoted merely by CA or OCA respectively. (Thus CA would denote the class of cellular automata as well as the class of langauges accepted by them; the meaning will be clear from the context.) Towards the end of this chapter we will give examples of some languages belonging to each of these classes. Though CA and OCA can use an unbounded amount of time in principle, in practice it is always possible to design equivalent CA or OCA

using at most 2^{cn} time for some constant c; this follows from the fact that since $\|Q\|$ is finite, there are only a finite number of configurations that a CA can go through without looping.

The definitions of CA and OCA can be generalised to the nondeterministic case. Now δ will map $Q \times Q \times Q$ (or $Q \times Q$, for OCA) to subsets of Q, and the input will be accepted if for some computation of the CA satisfying δ, the accepting cell enters a state from A.

Nondeterministic CA (NCA and $NOCA$) have also been studied in some detail in the past; some of the results can be found in [IK84, Smi72]. For a nondeterministic CA, for the same input there can be several time-space diagrams, corresponding to different nondeterministic choices.

2.3 Basic Results

The relationship between CA and Turing machines is easily expressed: since on inputs of length n the CA has exactly n cells, the computation can be simulated by a Turing machine which uses only linear space on its worktape. It will however require $O(n)$ steps to simulate one step of the CA, since the parallel action of all the cells has to be simulated sequentially. Thus a $T(n)$-time CA can be simulated by a Turing machine in $O(nT(n))$ time using $O(n)$ space. Conversely, a Turing machine using linear space (ie. a $DSPACE(n)$ machine. For this notation, see [BDG88]) can be simulated by a CA in the same amount of time. Each cell of the CA will hold the contents of one worktape cell of the CA; additionally, at each time step, exactly one CA cell will be in a special state indicating that the tape head is positioned here, and also indicating the state of the Turing machine. Clearly, a consistent updating of the states of all the CA cells is possible using only local neighbourhood information; thus a suitable transition function for the CA can be constructed to make the CA behave like the Turing machine. Thus

Lemma 2.3 *$CA = DSPACE(n)$.*

We now present some of the important well-known results about CA. The first result is the famous firing squad synchronisation (FSS) lemma. This lemma states that there is a CA which, when started off in a configuration with the leftmost cell (the general) in a special state b and all other cells (soldiers) in a "quiescent" state c, enters a configuration with all cells in a "fire" state \$ after $2n$ steps. Here n is the total number of cells (general + soldiers) in the firing line. Also, no cell enters the "fire" state before this time; all cells fire for the first time simultaneously. The transitions do not depend on n. There is also a real-time version if generals are placed at both ends of the firing line.

(When the general is at one end, such a synchronisation can be achieved by sending two signals, one thrice as fast as the other. The fast signal reflects off a boundary and meets the slower signal at the midpoint of the array. From this point, the algorithm is repeated recursively on both halves. This algorithm will require $3n$ time steps, as shown in Figure 2.3. The $2n$-time algorithm is too complex to be described here.)

Lemma 2.4 *There is a CA which, starting with configuration bc^{n-1}, reaches the configuration $\$^n$ in exactly $2n$ steps, with no cell entering the state \$ before time $2n$ [LM68, Wak66]. Also, there is a CA which, starting with configuration $bc^{n-2}b$, reaches the configuration $\$^n$ in exactly n steps, with no cell entering the state \$ before time n [BC84].*

A closely related concept is the concept of CA-time-constructibility, as defined below.

Definition 2.5 *A function $T(n) : \Sigma^+ \to \mathbf{N}^+$ is said to be CA-time-constructible if there is a CA which, on any input of length n, puts its accepting cell into a special state after exactly $T(n)$ time steps. The function is said to be strongly CA-time-constructible if the CA puts every cell into a special state, for the first time, after exactly $T(n)$ steps. In other words, after $T(n)$ steps, all cells simultaneously "fire" for the first time.*

Thus Lemma 2.4 essentially says that the function mapping strings of length n to the number $2n$ (or n) is strongly CA-time-constructible.

The next lemma states that lCA and OCA are equivalent to a restricted form of an online single-tape Turing machine, called a Sweeping Automaton (SA). An SA consists of a semi-infinite worktape (bounded at the left by a special boundary marker ¢) and a finite-state control with an input terminal at which it receives the serial input $a_1a_2\ldots a_n$. The symbol \$ is used as endmarker. The SA operates in left-to-right sweeps as follows:

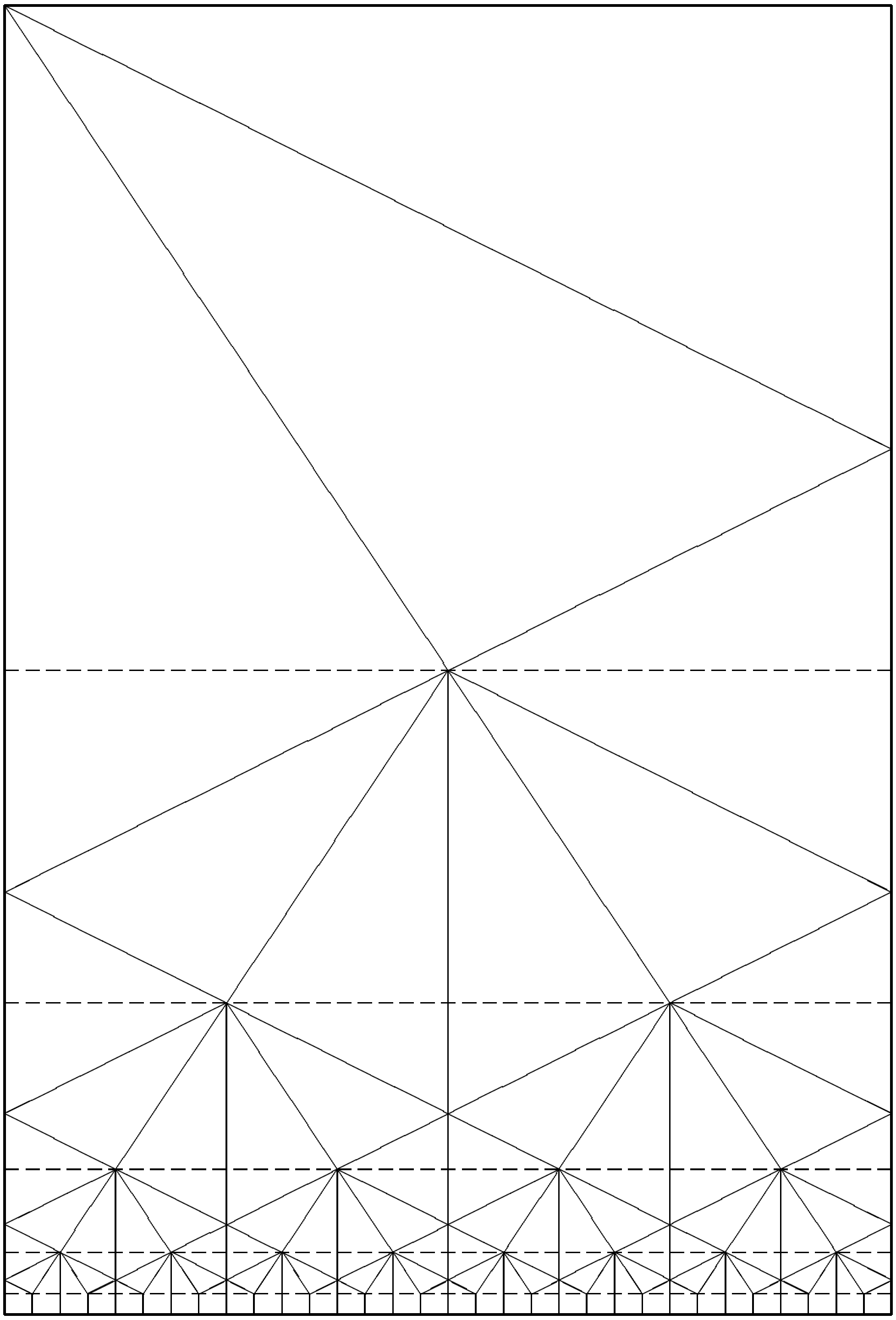


Figure 2.3: Firing squad synchronisation

Initially, all cells of the worktape to the right of ¢ contain the blank symbol λ. A sweep begins with the read-write-head (RWH) scanning ¢ and the machine in a distinguished state q_0. In the i^{th} sweep, the machine reads a_i and moves right of ¢ into a non-q_0 state. It continues moving right, rewriting non-λ symbols by non-λ symbols and changing states except into q_0. When the RWH reads a λ, it rewrites it by a non-λ symbol and resets to the leftmost cell in state q_0 to begin the next sweep. When \$ is first read, the machine completes the $(n+1)^{th}$ sweep, writes a \$ on the $(n+1)^{th}$ tape cell, and resets to ¢ in state q_0. Subsequent sweeps are performed between ¢ and \$ without expanding the workspace. \$ is assumed to be always available for reading after the input is exhausted. The input is accepted if the machine eventually enters an accepting state at the end of a sweep.

Several techniques for programming an SA have been described in [CIV88]. For a full description of how the techniques are implemented on an SA, the reader is referred to [CIV88].

Lemma 2.6 *[CIV88, IJ87, IPK85b] A language is accepted by an OCA if and only if it is accepted by an SA. Further, if the OCA runs in linear time, then the SA needs exactly n sweeps, and vice versa. A language is accepted by an lCA if and only if it is accepted by an SA in cn sweeps for some constant c.*

Note that this result also shows that $lCA \subseteq OCA$.

The next result gives the linear speed-up for CA and OCA. It is easy to see how speed-up is achieved for CA; the input moves left and is packed, k letters per cell, in the leftmost n/k cells, then these n/k cells synchronise themselves using a firing squad algorithm, and then they simulate the original CA at the rate of k steps per transition. k is to be chosen suitably depending on what speed-up is required. For OCA, the proof is not so trivial since synchronisation needs two-way communication. It has been proved directly in [BC84, CC84]; a simple proof using the sequential machine characterisation of OCA (which, itself, is a non-trivial proof!) appears in [IPK85b].

Lemma 2.7 *[BC84, CC84, IPK85b, Smi72] If L can be accepted in $n + R(n)$ time by a CA or an OCA, then it can be accepted by a CA or an OCA in $n + R(n)/k$ time, for any*

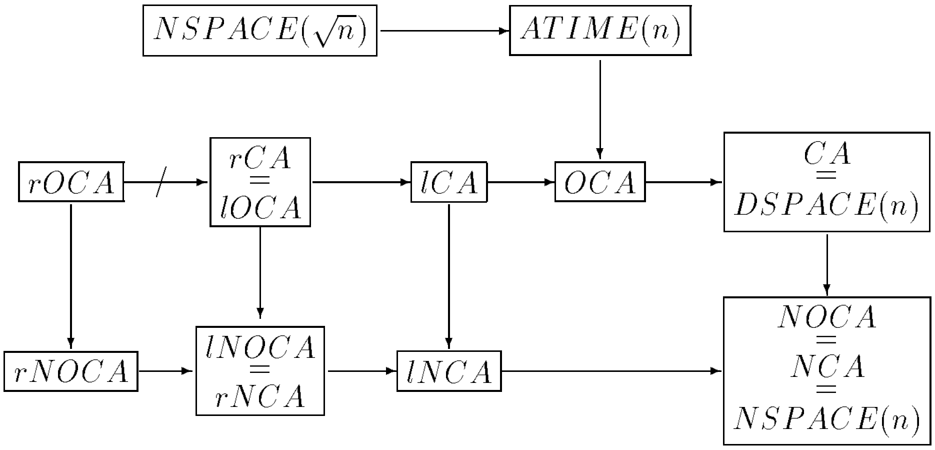


Figure 2.4: Known results about CA language classes

positive integer k. Thus a linear-time CA or OCA can be speeded up to $n(1+\epsilon)$ time, for any positive constant ϵ.

Thus multiplicative constants can be scaled down in CA operations. It has also been shown [BC84, CC84, IKM85, IPK85b] that additive constants can be done away with altogether, giving the following:

Lemma 2.8 *For CA and OCA, and for $T(n) \geq n-1$, $T(n)+c$ time can be speeded up to $T(n)$ time.*

The known containments amongst the CA and OCA classes are summarised in the following theorem and represented in Figure 2.4. In the figure, a (crossed) arrow denotes (proper) containment.

Theorem 2.9
$$\begin{array}{rrcl} (i) & rOCA & \subset & rCA \\ (ii) & rCA & = & lOCA \\ (iii) & lCA & \subseteq & OCA \end{array}$$

(i) is easily shown using a pumping lemma kind of argument on the time-space unrolling of $rOCA$; see [CC84, CGS84a]. The language $\{a^{2^n} \mid n \geq 0\}$ is a language in the difference. To show (ii) consider the time-space diagrams of an rCA and a $(2n)$-time $lOCA$ (use Lemma

a	b	c	d
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P
Q	R	S	T
	U	V	W
		X	Y
			Z

Computation of C,
a $2n$-time OCA

a	b	c	d
F E	G $\#$	H $\#$	$?$ $\#$
O N	P M	$?$ $\#$	
W V	$?$ U		
$?$ Z			

Simulation by C',
an rCA

Figure 2.5: An rCA simulating an $lOCA$

2.7). It is easy to construct a mapping between these two diagrams. An example is shown in Figure 2.5; for a formal proof, see [CC84]. This result, along with Lemma 2.6, gives us a sequential machine characterisation of rCA. (iii) follows from Lemma 2.6.

Despite the constraint of one-way communication, OCA are remarkably powerful. The class OCA has been shown to contain QBF (the language consisting of quantified Boolean formulae which evaluate to *True*), which is $PSPACE$-complete; see [IJ87]. It has also been shown to contain the classes $NSPACE(\sqrt{n})$ and $ATIME(n)$; again, see [IJ87] for proofs. Thus the class OCA lies between $NSPACE(\sqrt{n})$ and $DSPACE(n)$, and a proper containment between OCA and CA would also properly separate these classes, improving Savitch's result [BDG88, HU79]. It is conjectured that the containment is indeed proper, though no proof has yet been found. The problem has also been posed in terms of "resetting" deterministic linear-bounded automata; refer [Iba91].

There is stronger evidence that lCA are properly contained in OCA; since lCA are easily seen to lie in P and since OCA contains QBF, any proof that OCA are only as powerful as lCA (eg., $lCA = CA$, $lOCA = OCA$ etc.) will immediately imply that $P = PSPACE$. On the other hand, a proof to the contrary seems extremely difficult to obtain.

Surprisingly, once nondeterminism is introduced, one-way communication is no longer a restriction in the unbounded-time case, and the following result is easily seen [Dye80]:

Lemma 2.10 *$NOCA = NCA = NSPACE(n)$, the class of context-sensitive languages.*

The equivalence of NCA and $NSPACE(n)$ is more or less as described before Lemma 2.3 for the deterministic case. For an $NOCA$ to simulate an NCA, each cell has to guess its right neighbour's state, and special signals have to travel across the array verifying that the guesses were correct. The technique is described in detail in [Dye80].

It is also known that $rNOCA$ contains an NP-complete problem [IK84]; thus even for time-bounded classes, nondeterminism appears to add significantly to an OCA computation. (Recall that for the deterministic case, even lCA are within P, as seen from a straightforward Turing machine simulation.)

It is not known whether NCA are more powerful than CA. In fact, whether this containment of $DSPACE(n)$ in $NSPACE(n)$ is proper is one of the really longstanding open problems in formal language theory, and is usualy phrased as: "Are there context-sensitive languages which are not deterministic context-sensitive?" [HU79].

Some typical examples of languages known to lie in the real-time and linear-time CA and OCA classes are mentioned below.

$rOCA$ contains:

$\{ww^R \mid w \in \Sigma^*\}$

All linear context-free languages

$\{a^n b^n c^n \mid n \geq 1\}$

Dyck's languages

$\{x\$y\$z \mid |x| = |y|, \langle z\rangle = \langle x\rangle + \langle y\rangle\}$

rCA contains:

$\{ww \mid w \in \Sigma^*\}$

$\{a^n b^m \mid n \textit{ divides } m\}$

$\{a^n b^m \mid m \textit{ divides } n\}$

$\{a^{2^n} \mid n \geq 0\}$

$\{a^n \mid n \textit{ is a prime number }\}$

lCA contains:

$$\Big\{c^{\lceil log_2 |x_1| \rceil} x_1 c x_2 \# \ldots \# x_k \mid x_i \in \{0,1\}^+, |x_{i+1}| = \lceil log_2 |x_i| \rceil$$
$$\textit{the number of 1s in } \langle |x_k| \rangle \textit{ equals the number of } 1s \textit{ in } x_k\}$$

$$\Big\{c^{\lceil log_2 |x_1| \rceil} x_1 c x_2 \# \ldots \# x_k \mid x_i \in \{0,1\}^+, |x_{i+1}| = \lceil log_2 |x_i| \rceil, |x_k| = 1,$$
$$\textit{the number of 1s in } \langle k \rangle \textit{ equals the number of } 1s \textit{ in } x_1\}$$

OCA contains:

QBF, the set of fully quantified Boolean formulae which evaluate to true

All languages in $NSPACE(\sqrt{n})$

$ATIME(n)$, the class of languages accepted by alternating Turing machines in linear time

All languages that can be accepted by multihead 2-way nondeterministic pushdown automata operating in $c^{n/logn}$ time for some constant c

To summarise, the following problems posed variously in [Smi72, BC84, IJ88] are still open.

(a) Are linear-time CA more powerful than real-time CA?

(b) Are non-linear-time CA more powerful than linear-time CA?

(c) Are non-linear-time CA more powerful than real-time CA?

(d) Are CA more powerful than OCA?

(e) Are real-time CA closed under reversal? In [IJ88] it has been shown that this is the case if and only if the answer to (a) is No.

(f) Are linear-time CA more powerful than real-time CA when restricted to single letter input alphabets (unary alphabets and tally languages)?

Chapter 3

Time-varying Cellular Automata

In this chapter, we introduce time-varying transitions in CA and study the effect on the time complexity of language recognition. The idea of time-variant structures is not new. Time-variation in the context of language generators (grammars) has been studied before, the motivation being obtaining different classes of languages by regulating the usage, rather than the form, of the rewriting rules. Time variations have been shown to vastly increase the generative power of regular grammars [Sal73, KD84, KD85]. Analogously, time-varying finite-state automata (FSA) and pushdown automata (PDA) have been shown to be far more powerful language acceptors than their non-time-varying counterparts [KD86, KS88]. In another context, flexible cellular structures, where the global transition applied at each time step need not be the same, have also been studied in [Nas79], but there the object of study is to express non-realisable global transitions as a composition of a finite number of realisable global transitions. This work does not deal with such considerations. Here we are concerned with the enhancement of language recognition capabilities because of time-varying transitions.

3.1 Time-Varying Cellular Automata ($TVCA$)

A $TVCA$ is similar to a CA except in one respect: the transition function which specifies the next state of a cell in terms of its current state and the current state of its neighbours is not fixed. Instead, there is a finite set of local transition functions, and at each time step, exactly one of them is used, depending on how many time steps have elapsed since the $TVCA$ operation began. The effective transition function of the $TVCA$ thus has, as arguments, the current states of the cell and its immediate neighbours, and the current time $t \in \mathbf{N}$ ($\mathbf{N}$ is the set of natural numbers). Note that once Q is fixed, the number of distinct possible transition functions is finite.

Definition 3.1 *A k-time-varying CA is defined by a $(2k+2)$-tuple*

$$C = (Q, \#, p_1, p_2, \ldots, p_{k-1}, \delta_1, \delta_2, \ldots, \delta_k, A)$$

where

- *Q is a finite set of states*
- *p_i, $i = 1$ to $k-1$, are unary predicates over natural numbers,* $p_i : \mathbf{N} \rightarrow \{T, F\}$
- *δ_j, $j = 1$ to k, are distinct transition functions, $\delta_j : Q \times Q \times Q \rightarrow Q$*
- *$\#$ is the boundary state*
- *$A \subseteq Q$ is the set of accepting states.*

The transition function δ of C can be described as follows:

$$\delta(x, y, z, i) = \begin{cases} \textit{if} & p_1(i) & \textit{then} & \delta_1(x, y, z) \\ \textit{elseif} & p_2(i) & \textit{then} & \delta_2(x, y, z) \\ & \vdots & & \vdots \\ \textit{elseif} & p_{k-1}(i) & \textit{then} & \delta_{k-1}(x, y, z) \\ \textit{else} & & & \delta_k(x, y, z) \end{cases}$$

Thus δ is a transition function $\delta : Q \times Q \times Q \times \mathbf{N} \rightarrow Q$ specified in terms of p_i, $i = 1$ *to* $k-1$, and δ_j, $j = 1$ *to* k. Acceptance is defined as for CA.

The operation of the $TVCA$ is such that at every time instant t, there is a unique transition function to be applied to all cells. Thus the set $\mathbf{N}$ is partitioned into k blocks B_i, $i = 1$ *to* k, such that at all time instants t in block B_j, the CA applies transition function δ_j. The blocks B_j can be represented through tally languages (languages over a unary alphabet) instead of unary predicates. Each predicate p_i represents membership of strings, over a unary alphabet, in a particular language L_i. Specifically, if the unary alphabet is $\{0\}$, then $L_i = \{0^j \mid p_i(j) = T\}$. Clearly, $B_1 = \{j \mid p_1(j) = T\}$, and thus B_1 corresponds directly to L_1; ie. , $B_1 = \{j \mid 0^j \in L_1\}$. However,

$$B_2 = \{j \mid p_1(j) = F \wedge p_2(j) = T\}$$

Thus B_2 corresponds to $L_2 - L_1$. In general, block B_j corresponds to the language $L_j - (L_1 \cup L_2 \cup \ldots \cup L_{j-1})$, for $j = 2 \ to \ k-1$. Block B_k corresponds to $\overline{(L_1 \cup L_2 \cup \ldots \cup L_{k-1})}$.

Our interest here is in constraining the languages L_i controlling the $TVCA$ to belong to specific classes, and then studying the capabilities of the $TVCA$. All the classes we will consider in this thesis are closed under union and complementation; so the $TVCA$ can be specified in such a way that

$$B_i = \{j \mid 0^j \in L_i\}$$

for every i. This gives an alternative definition of $TVCA$ as follows:

Definition 3.2 *A k-time-varying CA is defined by a $(2k+3)$-tuple*
$C = (Q, \#, L_1, L_2, \ldots, L_k, \delta_1, \delta_2, \ldots, \delta_k, A)$ *where*

- Q *is a finite set of states*
- $L_i \subseteq \{0\}^*, i = 1 \ \ to \ \ k$*, are tally languages partitioning* $\{0\}^*$
- δ_j*,* $j = 1 \ \ to \ \ k$*, are distinct transition functions* $\delta_j : Q \times Q \times Q \rightarrow Q$
- $\#$ *is the boundary state*
- $A \subseteq Q$ *is the set of accepting states.*

The transition function δ of C can be described as follows:

$$\delta(x, y, z, i) = \delta_j(x, y, z) \qquad \textit{if and only if } 0^i \in L_j$$

Let $\mathcal{L}$ be any class of languages over the unary alphabet $\{0\}$. In the following sections, k-$\mathcal{L}TVCA$ (k-$r\mathcal{L}TVCA$ and k-$l\mathcal{L}TVCA$) denote k-$TVCA$ which are controlled by languages belonging to class $\mathcal{L}$ (and which accept languages in real time and linear time respectively). $\mathcal{L}TVCA$ (respectively $r\mathcal{L}TVCA$, $l\mathcal{L}TVCA$) denotes the union, over all finite k, of the classes k-$\mathcal{L}TVCA$ (k-$r\mathcal{L}TVCA$, k-$l\mathcal{L}TVCA$). Other classes, including those which consider one-way communication only, are similarly defined.

3.2 Control Imposed by Different Classes of Languages

In this section we consider different instances of the class $\mathcal{L}$ of controlling languages, and see how the classes k-$\mathcal{L}TVCA$, k-$r\mathcal{L}TVCA$ and k-$l\mathcal{L}TVCA$ get modified.

Definition 3.3 *A language $L \subseteq \{0\}^*$ is said to be ultimately periodic if there exist natural numbers $n_0 \geq 0$ and $p \geq 1$ such that*

$$\forall n \geq n_0,\ 0^{n+p} \in L \textit{ if and only if } 0^n \in L$$

Theorem 3.4 *If all languages in $\mathcal{L}$ are ultimately periodic, then a k-$\mathcal{L}TVCA$ can be simulated by a CA (which is not time-varying) with no loss of time. If the $TVCA$ uses one-way communication, so will the simulating CA.*

Proof: Essentially, $n_0 + p$ copies of the state set are created, and the transition function depends on which copy of the state is an argument. Copies of the boundary state need not be created. Accepting states can be suitably defined. Formally, let the k-$\mathcal{L}TVCA$ be given by

$$C = (Q, \#, L_1, L_2, \ldots, L_k, \delta_1, \delta_2, \ldots, \delta_k, A)$$

We will construct an equivalent non-time-varying CA $C' = (Q', \#, \delta', A')$ as follows: Let n_0 and p be as in the definition of ultimate periodicity for C. Define sets Q_i, $i = 2$ *to* $n_0 + p - 1$, to be disjoint copies of Q; thus $Q_i = \{q_i \mid q \in Q - \{\#\}\}$. Then $Q' = \left(\bigcup_{i=2}^{n_0+p-1} Q_i\right) \cup Q$. Let A_i be the restriction of Q_i to copies of states in A; then $A' = \left(\bigcup_{i=2}^{n_0+p-1} A_i\right) \cup A$. δ' is defined as follows: $\delta'(a, b, c) = q_2$ where $q = \delta(a, b, c, 1)$ and $b \neq \#$.
$\delta'(a_i, b_i, c_i) = q_{i+1}$ where $q = \delta(a, b, c, i)$ and $i < n_0 + p - 1$.
$\delta'(a_i, b_i, c_i) = q_j$ where $i = n_0 + p - 1$ and $q = \delta(a, b, c, i)$ and $j = n_0$.
$\delta'(a, \#, b) = \#$ for any $a, b \in Q'$.

It is easy to see that C' so defined simulates C and that the simulation is step-for-step, ie. , with no loss of time. ∎

Proposition 3.5 *[Gin66] Tally regular languages are ultimately periodic.*

Corollary 3.6 *If a $TVCA$ is controlled by regular languages, then it can be simulated by a CA with no loss of time.*

This result is easier to see directly rather than as a consequence of Theorem 3.4. Since the regular languages involved are all over unary alphabets, each cell in the simulating CA can internally simulate the finite-state automata (FSA) accepting these languages and independently determine which of the k transition functions to apply.

Proposition 3.7 *The class of tally languages accepted by $rOCA$ is exactly equal to the class of tally regular sets.*

Proof: In [CGS84a, CC84] it has been shown that all regular languages can be accepted by $rOCA$. To see the converse when restricted to tally sets, let $C = (Q, \#, \delta, A)$ be an $rOCA$ accepting a tally language L. We can construct a finite-state machine M accepting L as follows: $M = (q_0 \cup (Q \times Q), \{0\}, \delta', q_0, F)$ where

$\delta'(q_0, 0) = [\delta(\#, 0), \delta(0, 0)]$,

$\delta'([a, b], 0) = [\delta(a, b), \delta(b, b)]$, and

$F = \{[a, b] \mid a \in A\}$.

For instance, if the cells of C change states as shown in Figure 3.1, then M goes through states $q_0, AB, CD, EF, GH, \ldots$. ∎

Corollary 3.8 *If a $TVCA$ is controlled by $rOCA$ languages, then it can be simulated by a CA with no loss of time.*

These results are, in some sense, negative; they show conditions under which $TVCA$ are no better than CA. We shall now look at some positive results. For any class $\mathcal{L}$, let $\mathcal{L} \mid_t$ denote the class of all tally languages in $\mathcal{L}$.

#	O	O	O	O	O
	A	B	B	B	B
		C	D	D	D
			E	F	F
				G	H
					I

Figure 3.1: The unrolling of an $rOCA$

Lemma 3.9 *$\forall \mathcal{L}$, $\mathcal{L}\,|_t \subseteq 2\text{-}r(\mathcal{L}\,|_t)TVOCA$.*

Proof: Let $L \in \mathcal{L}\,|_t$. We can design a $TVOCA$ with controlling language L, to accept L. Let the input string be 0^n. The $TVOCA$ sends a signal \$ from its leftmost cell to its rightmost cell. The transition functions δ_1 and δ_2 are identical on arguments which do not contain \$. If a cell's neighbourhood contains \$, then δ_1 puts it into an accepting state whereas δ_2 does not. Since \$ reaches the rightmost cell (which is the accepting cell) at time n, this cell enters an accepting state at time n if and only if δ_1 is used, ie., if and only if the input belongs to L. Thus L can be accepted by a $2\text{-}rTVOCA$ with controlling language $L \in \mathcal{L}\,|_t$, ie., by a $2\text{-}r(\mathcal{L}\,|_t)TVOCA$. ∎

Corollary 3.10

$$\begin{aligned} CA\,|_t &\subseteq 2\text{-}r(CA\,|_t)TVOCA \\ OCA\,|_t &\subseteq 2\text{-}r(OCA\,|_t)TVOCA \end{aligned}$$

On the other hand, we can show that allowing the controlling language to be only as powerful as CA or OCA languages cannot increase the power of even an $lTVCA$ beyond the classes CA or OCA respectively. These results are intuitively obvious, and, in the case of CA control, easy to show. However the proof in the case of OCA control is surprisingly non-trivial. The proofs given below are for $k = 2$; the extensions to arbitrary k are straightforward.

Theorem 3.11

$$\begin{aligned} \forall k \quad k\text{-}l(CA\,|_t)TVCA &\subseteq CA \\ \forall k \quad k\text{-}l(OCA\,|_t)TVCA &\subseteq OCA \end{aligned}$$

Proof: To show that $2\text{-}l(CA\ |_t)TVCA \subseteq CA$, let $\psi(A)$ be an $lTVCA$ controlled by language A, where A can be accepted by CA ϕ. Let the constant for ψ be c. Then we can construct a CA ϕ' which, for the i^{th} step of $\psi(A)$, first simulates ϕ on the i length input 0^i and then uses the outcome to simulate ψ. Since ψ is an lCA, ϕ has to be simulated on inputs upto length cn. This requires a cn length array, which can be compacted onto the n length array of ϕ'. ϕ' thus simulates $\psi(A)$.
Clearly, this argument can be generalised to $k\text{-}TVCA$.

To show that $k\text{-}l(OCA\ |_t)TVCA \subseteq OCA$, we will use the sequential machine characterisation for OCA. We will show that any language in the class $2\text{-}l(OCA\ |_t)TVCA$ can be accepted by an SA. This can be generalised to $k\text{-}TVCA$, and will thus prove the theorem.

Let L be a language accepted by an $lTVCA$ ψ controlled by the OCA language A. A is accepted by an OCA φ. Choose constant c sufficiently large so that L is accepted by ψ in $T(n) < cn$ time steps. Let the input be $a_1a_2\ldots a_n$. Construct SA M accepting L as follows:

The SA operates in sweeps, reading a_i at the beginning of the i^{th} sweep. In the first sweep, M creates $2c$ subcells in the first worktape cell, and puts a boundary marker **b** on the c^{th} subcell. It also puts markers [and] on the $(c+1)^{th}$ subcell, and writes a_1 on the $(c+1)^{th}$ subcell (along with the [and]). In subsequent sweeps while reading the input, it creates $2c$ new subcells per sweep (in the first λ cell read). It also moves **b** and [c subcells right, and] $c+1$ subcells right. The characters a_1 to a_{i-1} are shifted c subcells right, and a_i is written, with], beyond them. Thus the worktape is partitioned by **b** into two parts such that after the i^{th} sweep, each part has ci subcells. In the second part the first i subcells are marked off between [and], and hold the input read so far, one character per subcell. Each subcell in the left part holds the unary character 0 (apart from possibly **b**).

When M starts getting \$ as input, it begins the actual simulation. M places a $\star$ on subcell 1 to indicate that membership of input 0^1 in A is to be determined. φ is simulated on input 0^{cn} in the left part. Since an OCA has only two arguments in its transition function, the left cell's state and the current cell's state, the state information can be updated in a left-to-right sweep. Let $c(i,t)$ denote the state of the i^{th} cell of φ at time t. Because of one-way communication, $c(i,t)$ is the same for all input lengths $n \geq i$. So simulating φ on input 0^{cn} also gives simulations of φ on input 0^i, $i \leq cn$. If in any sweep the i^{th} subcell in the left part enters an accepting state of φ, the subcell is marked with a **Y** indicating that input 0^i belongs to A; whenever it enters a rejecting state, the subcell is marked **N**. (Since OCA are closed under complementation, accepting and rejecting states can be defined.)

In any sweep, if M encounters a **Y** (**N**) in a subcell marked $\star$, then the current query to A has been answered, so M moves the $\star$ one subcell right to query A on the next input. It then simulates one transition of ψ in the region between and including the subcells marked [and] in the right part, using transition δ_1 (δ_2). However, since ψ is a two-way CA, a simulation in a left-to-right sweep will shift its configuration one unit right. The [and] markers are also correspondingly shifted. Since ψ operates within time cn, the right part is provided with cn subcells to allow for the shifting configuration. In any sweep if an accept state of ψ is written on the subcell marked [, then this means that the lCA ψ has accepted the input. So M completes this sweep by moving right in a final state.

In sweeps where neither **Y** nor **N** are found on the $\star$ subcell, the $\star$ is kept where it is and the right part is left unchanged.

Thus the membership of strings in the controlling language is answered in the cn left subcells. As and when an answer to the next query is available, the corresponding transition step of the $TVCA$ is simulated in n subcells in the right part.

When M attempts to move $\star$ beyond **b**, all cn queries to A have been answered and this sweep will complete the operation of ψ. So by this time if M has not found an accept state in the [subcell, it moves right in a rejecting state.

The worktape profile for one such SA is shown in Figure 3.2 (c). (The † symbol is printed by the SA on the first subcell in the $(n+1)^{th}$ sweep to allow the SA to tell this sweep apart from subsequent sweeps. This is crucial because only in this sweep should the SA print a $\star$ on the first subcell.) ■

Corollary 3.12

$\forall k$

$$(k\text{-}r(CA\mid_t)TVOCA)\mid_t = (k\text{-}r(CA\mid_t)TVCA)\mid_t = (k\text{-}l(CA\mid_t)TVCA)\mid_t = CA\mid_t$$

$\forall k$

$$(k\text{-}r(OCA\mid_t)TVOCA)\mid_t = (k\text{-}r(OCA\mid_t)TVCA)\mid_t = (k\text{-}l(OCA\mid_t)TVCA)\mid_t = OCA\mid_t$$

	a_1	a_2	a_3
δ_1	a	b	c
δ_2	d	e	f
δ_2	g	h	i
δ_1	j	k	
δ_2	l		

(a) lCA ψ on input $a_1a_2a_3$

0	0	0	0	0
z_1^1	z_2^1	z_3^1	z_4^1	z_5^1
$Y\,z_1^2$	z_2^2	z_3^2	z_4^2	z_5^2
z_1^3	z_2^3	$N\,z_3^3$	z_4^3	z_5^3
z_1^4	$N\,z_2^4$	z_3^4	z_4^4	z_5^4
z_1^5	z_2^5	z_3^5	z_4^5	z_5^5
z_1^6	z_2^6	z_3^6	$Y\,z_4^6$	z_5^6
z_1^7	z_2^7	z_3^7	z_4^7	z_5^7
z_1^8	z_2^8	z_3^8	z_4^8	z_5^8
z_1^9	z_2^9	z_3^9	z_4^9	$N\,z_5^9$

(b) Computation of OCA φ on 0^5

(Accepting states are marked **Y**, rejecting states **N**)

Figure 3.2: An SA simulating an $lCA(OCA)$ computation (Part (c) on next page)

input	cell 1				cell 2				cell 3			
0	0	b	$[a_1]$	.	.	.	.	.	.	.	.	.
0	0	0	0	b	$[a_1$	$a_2]$	.	.	.	.	.	.
0	0	0	0	0	0	b	$[a_1$	a_2	$a_3]$	.	.	.
$\$$	$\dagger_{*}z_1^1$	z_2^1	z_3^1	z_4^1	z_5^1	b	$[a_1$	a_2	$a_3]$	.	.	.
$\$$	$\dagger_{Y}z_1^2$	${}_{*}z_2^2$	z_3^2	z_4^2	z_5^2	b	.	$[a$	b	$c]$	.	.
$\$$	$\dagger_{Y}z_1^3$	${}_{*}z_2^3$	${}_{N}z_3^3$	z_4^3	z_5^3	b	.	$[a$	b	$c]$	.	.
$\$$	$\dagger_{Y}z_1^4$	${}_{N}z_2^4$	${}_{*N}z_3^4$	z_4^4	z_5^4	b	.	.	$[d$	e	$f]$	.
$\$$	$\dagger_{Y}z_1^5$	${}_{N}z_2^5$	${}_{N}z_3^5$	${}_{*}z_4^5$	z_5^5	b	.	.	.	$[g$	h	$i]$
$\$$	$\dagger_{Y}z_1^6$	${}_{N}z_2^6$	${}_{N}z_3^6$	${}_{Y}z_4^6$	${}_{*}z_5^6$	b	.	.	.	.	$[j$	k
$\$$	$\dagger_{Y}z_1^7$	${}_{N}z_2^7$	${}_{N}z_3^7$	${}_{Y}z_4^7$	${}_{*}z_5^7$	b	.	.	.	.	$[j$	k
$\$$	$\dagger_{Y}z_1^8$	${}_{N}z_2^8$	${}_{N}z_3^8$	${}_{Y}z_4^8$	${}_{*}z_5^8$	b	.	.	.	.	$[j$	k
$\$$	$\dagger_{Y}z_1^9$	${}_{N}z_2^9$	${}_{N}z_3^9$	${}_{Y}z_4^9$	${}_{N}z_5^9$	${}_{*}b$	.	.	.	.	.	$[l$

(c) SA simulating $\psi(\varphi)$

Figure 3.2: An SA simulating an $lCA(OCA)$ computation

The proof does not extend to include k-$(CA\ |_t)TVCA$ (unbounded-time CA), because an unbounded-time $TVCA$ can make an unbounded number of queries about membership of strings, of arbitrary length, in the controlling language. A CA is constrained by space and so cannot find query responses for arbitrarily long strings. As a matter of fact, there is a $TVCA$ controlled by a CA oracle which accepts languages provably not acceptable by CA; this result follows from the following theorem and the space hierarchy theorem [BDG88] which strictly separates $DSPACE(n)$ from $DSPACE(2^n)$. Before stating the theorem we prove a simple proposition.

Proposition 3.13 *There is a CA which, given input $\langle n\rangle$, puts its rightmost cell into a special state* **S** *after exactly n steps.*

Proof: The CA with input $\langle n\rangle$ counts n time steps by subtracting 1, at each time step, from the number in its array. The leftmost "active" cell has two bits of the number. Initially the leftmost cell of the CA is the leftmost "active" cell and behaves as if it has $1b_k$, where b_k is the leftmost bit of $\langle n\rangle$. Subtraction occurs at the rightmost end, with "borrow" signals travelling left as and when generated, and the leftmost "active" cell gradually shifting right.

An example for $\langle n\rangle = 011$, is shown in Figure 3.3. Since the input is 011, the number in question is the binary number 1011 (the leading 1 is implicit), ie. 11 in decimal notation. The leftmost cell thus sets its state to 10 initially and is the leftmost "active" cell. The rightmost cell subtracts 1 from its contents at each step. When it has only 0, it sets the 0 to 1 and also sets a **b** flag in its state, indicating that it has "borrowed" a 1 from its left neighbour. A cell which sees this **b** flag in its right neighbour's state subtracts one from its own contents – if necessary, borrowing a 1 from its left neighbour in a similar fashion. The exception is when a cell needs to borrow a 1 and finds that its left neighbour is the leftmost "active" cell and contains 10. Clearly, this cell will become the leftmost "active" cell after the borrowing, so it directly sets itself to 11. A cell with a 10, on seeing its right neighbour set itself to 11, knows that it is no longer required to be "active", and sets itself to some special state X. The computation ends when the rightmost cell becomes the leftmost "active" cell and decrements its contents to 00.

■

$t = 0$	#	0	1	1	#
1	#	10	1	0	#
2	#	10	1	1_b	#
3	#	10	0	0	#
4	#	10	0	1_b	#
5	#	10	11	0	#
6	#	X	11	1_b	#
7	#	X	10	0	#
8	#	X	10	11	#
9	#	X	X	10	#
10	#	X	X	01	#
11	#	X	X	**S**	#

Figure 3.3: Computing n from $\langle n \rangle$ on a CA

Theorem 3.14 $DSPACE(2^n) \subseteq 2\text{-}(CA\mid_t)TVCA$

Proof: Let L be any language in DSPACE(2^n). Consider the tally version $TALLY(L)$ defined as $\{0^n \mid \langle n \rangle \in L\}$. Since the length of 0^n is exponential in the length of $\langle n \rangle$, it is easy to see that for any language L in $DSPACE(2^n)$, $TALLY(L)$ is in $DSPACE(n)\mid_t = CA\mid_t$. Now it is easy to construct a $TVCA$ which has δ_1 and δ_2 alike everywhere except in the presence of **S**. The input to the $TVCA$ is $\langle n \rangle$, and both δ_1 and δ_2 compute n in time, as in the above proposition. In the presence of **S**, δ_1 puts the $TVCA$ in an accepting state and δ_2 puts it in a rejecting state. Such a $TVCA$, with a controlling language $TALLY(L)$ from $CA\mid_t$, can thus accept any language L in $DSPACE(2^n)$. ■

Corollary 3.15 $CA \subset 2\text{-}(CA\mid_t)TVCA$

Corollaries 3.8 and 3.12 consider two extremes of control, $rOCA$ and OCA. In between these extremes we have the classes rCA and lCA. Using these classes as classes of controlling languages, we obtain some interesting results.

Theorem 3.16 k-$r(rCA\,|_t)TVCA \subseteq lCA$

Theorem 3.17 k-$l(rCA\,|_t)TVCA = lCA$

We postpone the proofs of these results to chapter 5 (Theorems 5.11 and 5.12), since their presentation is more appropriate there. There the proofs will be presented for the case $k = 2$; the extension to larger k is straightforward. Actually these results can also be obtained by a modification of the proof for Theorem 3.11, and by using the result that SAs operating in n or cn sweeps accept rCA or lCA languages respectively (refer Lemma 2.6). However in chapter 5, we will give a direct construction of corresponding lCA. This direct construction is more useful because it will be generalised to show further results; the SA-based proof cannot be generalised in this manner.

Theorem 3.17 says that linear-time CA do not become more powerful through the addition of real-time CA control. It seems difficult to find whether their power is increased through lCA control. If this is the case, then Theorem 3.11 will immediately imply that lCA are properly contained in OCA and CA, a question that has been open for a long time. Similarly, a proof that the power of real-time CA is augmented by rCA control will imply, from Theorem 3.16, that lCA are strictly more powerful than rCA, another longstanding open question.

3.3 The k-function Hierarchy

The power of a $TVCA$ depends not only on the nature of the controlling languages but also on the number of languages, ie. on the size of the partition on $\mathbf{N}$ that is defined by the $TVCA$. The previous section explored the effect of varying the nature of the languages, ie. varying the nature of the partition. In this section, we consider the effect of varying the size; the importance of the parameter k is studied.

For a tally language L, let $2L$ and $2L-1$ denote the languages $\{0^{2j} \mid 0^j \in L\}$ and $\{0^{2j-1} \mid 0^j \in L\}$ respectively. We shall say that a class of tally languages $\mathcal{L}$ is closed under doubling if $\forall L \in \mathcal{L}$, $2L$, $2L-1$ also belong to $\mathcal{L}$. Similarly, we shall say that a class of tally languages $\mathcal{L}$ is closed under compressed composition if for every L_i, L_j in $\mathcal{L}$, the language $L_{ij} = \{0^k \mid 0^{2k-1} \in L_i \wedge 0^{2k} \in L_j\}$ also belongs to $\mathcal{L}$.

Lemma 3.18 *$rCA\mid_t$, $lCA\mid_t$, $OCA\mid_t$ and $CA\mid_t$ are closed under union, doubling and compressed composition.*

Proof:

(a) Closure under union is straightforward.

(b) Closure under doubling for $rCA\mid_t$ languages will be shown in chapter 5 (Lemma 5.10).

Closure under doubling for $lCA\mid_t$ and $CA\mid_t$ can be seen as follows. Let L be an $lCA\mid_t$ or $CA\mid_t$ language. $2L$ and $2L-1$ are to be accepted. On input 0^{2j} or 0^{2j-1}, the extreme cells of the CA send a signal inwards. These signals meet and mark out the left portion of the input array, of length j. On this portion, a linear-time firing squad algorithm (refer Lemma 2.4) is run to synchronise the operation of all the cells. When they are synchronised, the CA accepting L is run and membership of the input is determined accordingly. Clearly, this takes linear time if L can be accepted in linear time.

To see closure under doubling for $OCA\mid_t$, let L be an $OCA\mid_t$ language. Consider an SA which functions as follows: On input 0^{2j} or 0^{2j-1}, the SA writes the input on its worktape and also places a marker **B** on the midpoint of the input. (This is achieved by moving the marker one cell right in every *other* sweep.) When the entire input is read, it simulates the OCA on the portion of the input upto and including the marker. Thus the OCA is simulated on input 0^j. Since SA accept the same languages as OCA (Lemma 2.6), it follows that $2L-1$ and $2L$ are OCA languages.

(c) Closure under compressed composition for $rCA\mid_t$, $lCA\mid_t$ and $CA\mid_t$ is seen as follows. Let L_i and L_j be the languages to be compressed, and let C_1 and C_2 respectively be the CAs accepting them. The input array is of length n, so membership of 0^{2n-1} in L_i and of 0^{2n} in L_j is to be determined. In the first time step, the CA changes state so that each cell has two channels, and each channel has two input characters, ie. 00. The exception is the rightmost cell, which has 0# in its first channel and 00 in its second channel. The CA now simulates C_1 in its first channel and C_2 in its second channel, at double speed. This is possible because the initial input to these CA is already packed two characters per cell. Thus the language L_{ij} can be accepted, within real time or linear time if C_1 and C_2 are rCA or lCA.

Closure of $OCA\mid_t$ is seen in a similar fashion. The difference is that the rightmost cell does not know that it is a boundary cell, but the leftmost cell does. So when the two channels are created, each cell except the leftmost cell puts 00 in both channels. The leftmost cell puts #0 in the first channel and 00 in the second channel. Now C_1 and C_2 are simulated in the two channels, as in the previous case.

■

Theorem 3.19 *Let $\mathcal{L}$ be any class of tally languages closed under union and doubling. Then, for $k > 1$, a $(k+1)$-$\mathcal{L}TVCA$ can be simulated by a k-$\mathcal{L}TVCA$, in twice as much time, and with thrice as many states.*

Proof: Consider $k = 2$. The basic idea is that the 3-$TVCA$ has to make a three-way decision at every time-step. This decision can be split into a sequence of two two-way decisions. This would require twice as much time. Different copies of the original state set are used to indicate which decision is to be taken. The idea can be extended to larger k. The choice between the k^{th} and $(k+1)^{th}$ functions is deferred by one time step. This can be diagrammatically represented as in Figure 3.4.

The construction is formally described as follows:

Let $C = (Q, \#, L_1, \ldots, L_{k+1}, \delta_1, \ldots, \delta_{k+1}, A)$ be a $(k+1)$-$\mathcal{L}TVCA$. We construct an equivalent k-$\mathcal{L}TVCA$ C' as follows.

$$C' = (Q', \#, L'_1, \ldots, L'_k, h_1, \ldots, h_k, A)$$

where

$Q' = \{q, \overline{q}, \overline{\overline{q}} \mid q \in Q\}$

$L'_i = \{0^{2j-1}, 0^{2j} \mid 0^j \in L_i\}$ for $i = 1$ *to* $k-2$

$L'_{k-1} = \{0^{2j-1}, 0^{2j} \mid 0^j \in L_{k-1}\} \cup \{0^{2j} \mid 0^j \in L_k\}$

$L'_k = \{0^{2j-1}, 0^{2j} \mid 0^j \in L_{k+1}\} \cup \{0^{2j-1} \mid 0^j \in L_k\}$

These languages also belong to $\mathcal{L}$, since $\mathcal{L}$ is closed under doubling and union.

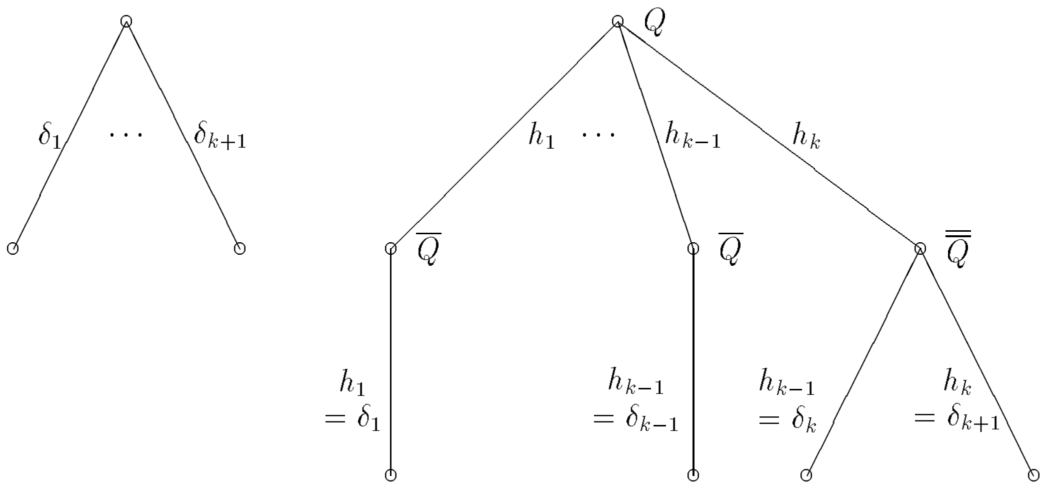


Figure 3.4: Reducing a $(k+1)$-$\mathcal{L}TVCA$ to a k-$TVCA$ in twice as much time

$h_i(a,b,c) = \overline{b}$ for $i = 1$ *to* $k-1$

$h_k(a,b,c) = \overline{\overline{b}}$

$h_i(\overline{a},\overline{b},\overline{c}) = \delta_i(a,b,c)$ for $i = 1$ *to* $k-1$

$h_{k-1}(\overline{\overline{a}},\overline{\overline{b}},\overline{\overline{c}}) = \delta_k(a,b,c)$

$h_k(\overline{\overline{a}},\overline{\overline{b}},\overline{\overline{c}}) = \delta_{k+1}(a,b,c)$

It is straightforward to see that C' simulates C. ■

Figure 3.4 shows how to reduce a $(k+1)$-function computation to a k-function computation by splitting a decision into two stages. A simple improvement upon this, ensuring that a k-way decision is made at each stage, shows that a k^2-$\mathcal{L}TVCA$ can be simulated by a k-$\mathcal{L}TVCA$ in twice as much time, though the state size will be slightly larger. See Figure 3.5.

Corollary 3.20 *Let $\mathcal{L}$ be any class of tally languages closed under union and doubling. The hierarchy*

$$2\text{-}l\mathcal{L}TVCA \subseteq 3\text{-}l\mathcal{L}TVCA \subseteq \ldots \subseteq k\text{-}l\mathcal{L}TVCA \subseteq (k+1)\text{-}l\mathcal{L}TVCA \subseteq \ldots$$

collapses to the class 2-$l\mathcal{L}TVCA$*, and the hierarchy*

$$2\text{-}r\mathcal{L}TVCA \subseteq 3\text{-}r\mathcal{L}TVCA \subseteq \ldots \subseteq k\text{-}r\mathcal{L}TVCA \subseteq (k+1)\text{-}r\mathcal{L}TVCA \subseteq \ldots$$

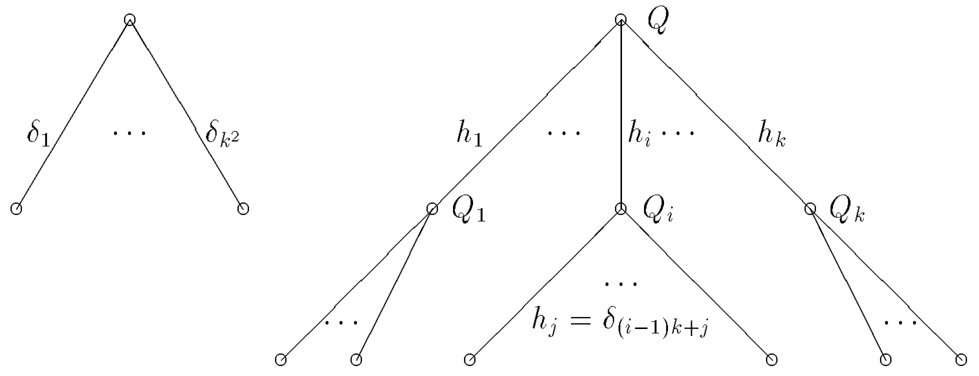


Figure 3.5: Reducing a k^2-$\mathcal{L}TVCA$ to a k-$TVCA$ in twice as much time

is contained in this class 2-$l\mathcal{L}TVCA$.

The analogous results for OCA classes also hold, by a similar argument.

Some non-trivial classes which do satisfy these conditions of closure under union and doubling are $rCA\mid_t$, $lCA\mid_t$ and $CA\mid_t$. However, we have seen in the previous section that rCA control for any k cannot enhance the power of $lTVCA$ (Theorem 3.17). Thus for $\mathcal{L} = rCA\mid_t$, we have the following stronger statement:

Corollary 3.21 *Let* $\mathcal{L} = rCA\mid_t$. *The hierarchy*

$$lCA \subseteq 2\text{-}l\mathcal{L}TVCA \subseteq 3\text{-}l\mathcal{L}TVCA \subseteq \ldots \subseteq k\text{-}l\mathcal{L}TVCA \subseteq (k+1)\text{-}l\mathcal{L}TVCA \subseteq \ldots$$

collapses to the class lCA, *and the hierarchy*

$$2\text{-}r\mathcal{L}TVCA \subseteq 3\text{-}r\mathcal{L}TVCA \subseteq \ldots \subseteq k\text{-}r\mathcal{L}TVCA \subseteq (k+1)\text{-}r\mathcal{L}TVCA \subseteq \ldots$$

is contained in this class lCA.

Thus the first statement of Corollary 3.20 is meaningfully applicable for $\mathcal{L} = lCA\mid_t$ and $\mathcal{L} = CA\mid_t$. By this corollary, when we consider $lTVCA$ with lCA or CA control, it is sufficient to consider the case of 2-$TVCA$ only.

Reducing k in the case of real-time $TVCA$ does not seem to be so easy. However, some results showing the trade-off between time and two-way communication have been obtained.

Theorem 3.22 *Let $\mathcal{L}$ be a class of tally languages closed under union and doubling. For $k > 1$, a $(k+1)$-$r\mathcal{L}TVCA$ with state set Q can be simulated by a k-$l\mathcal{L}TVOCA$, in twice as much time, and with $O(\|Q\|^3)$ states.*

Proof: This proof combines the proof of Theorem 3.19 and the proof for $rCA = 2n$-time OCA (Theorem 2.9(ii), Figure 2.5). The first stage in the new 2-stage decision is used by the OCA to obtain the desired neighbourhood information of the CA. However, as the OCA simulates the CA, the configuration slides to the right. Thus if $c(i,t)$ and $\overline{c}(i,t)$ represent the states of the CA's and OCA's i^{th} cell at time t, then $\overline{c}(i,2t) = c(i-t,t)$. As a result, $c(1,n)$, which is the state denoting acceptance, is represented at $\overline{c}(n+1,2n)$, which is beyond the OCA's accepting cell. To counter this, we make $\overline{c}(i,2t)$ also contain, as a component, the state which the $(i-t+1)^{th}$ CA cell would have entered at time t if cell i had been the rightmost cell. Then $\overline{c}(n,2n)$ will contain $c(1,n)$, denoting acceptance. A typical computation of C' is shown in Figure 3.6. Note that this simulation will work only if acceptance occurs exactly at time n, and not within time n. This is not restrictive since any rCA can be modified to accept its input at exactly time n, by sending a signal from one extreme of the input to the other.

Thus at even time steps $2k$, the first components of the states of the $lOCA$ reflect the states of the rCA at time k. The diagonal states of the rCA are stored in the second component of the rightmost cell of the $lOCA$. The construction is described formally below:

Let $C = (Q, \#, L_1, \ldots, L_{k+1}, \delta_1, \ldots, \delta_{k+1}, A)$ be a $(k+1)$-$r\mathcal{L}TVCA$. We construct an equivalent k-$l\mathcal{L}TVOCA$ C' as follows.

$C' = (Q', \#, L'_1, \ldots, L'_k, h_1, \ldots, h_k, A')$ where

$Q' = Q \cup \{[ab], \overline{[ab]}, [a,b], [ab,c], \overline{[ab,c]} \mid a,b,c \in Q\}$

$L'_i = \{0^{2j-1}, 0^{2j} \mid 0^j \in L_i\}$ for $i = 1$ *to* $k-2$

$L'_{k-1} = \{0^{2j-1}, 0^{2j} \mid 0^j \in L_{k-1}\} \cup \{0^{2j} \mid 0^j \in L_k\}$

$L'_k = \{0^{2j-1}, 0^{2j} \mid 0^j \in L_{k+1}\} \cup \{0^{2j-1} \mid 0^j \in L_k\}$

These languages also belong to $\mathcal{L}$, since $\mathcal{L}$ is closed under doubling and union.

$A' = \{[a,b] \mid b \in A\}$.

$h_i(a,b) = [ab]$ for $i = 1$ *to* $k-1$.

$h_k(a,b) = \overline{[ab]}$

input	A	B	C	D	E
δ_1	a	b	c	d	e
δ_3	f	g	h	i	
δ_2	j	k	l		
δ_1	m	n			
δ_2	p				

time-space unrolling of a 3-$rTVCA$

input	A	B	C	D	E
h_1	$\#A$	AB	BC	CD	DE
$h_1 = \delta_1$	$\#, \star$	$a, \star$	$b, \star$	$c, \star$	d, e
h_2	$\overline{\#\#, \star}$	$\overline{\#a, \star}$	$\overline{ab, \star}$	$\overline{bc, \star}$	$\overline{cd, e}$
$h_2 = \delta_3$	$\#, \star$	$\#, \star$	$f, \star$	$g, \star$	h, i
h_2	$\overline{\#\#, \star}$	$\overline{\#\#, \star}$	$\overline{\#f, \star}$	$\overline{fg, \star}$	$\overline{gh, i}$
$h_1 = \delta_2$	$\#, \star$	$\#, \star$	$\#, \star$	$j, \star$	k, l
h_1		$\#\#, \star$	$\#\#, \star$	$\#j, \star$	jk, l
$h_1 = \delta_1$			$\#, \star$	$\#, \star$	m, n
h_2				$\overline{\#\#, \star}$	$\overline{\#m, n}$
$h_1 = \delta_2$					$\#, p$

time-space unrolling of corresponding 2-$lTVOCA$

($\star$ = don't-care state)

Figure 3.6: Reducing a 3-$r\mathcal{L}TVCA$ to a 2-$l\mathcal{L}TVOCA$

These two rules are used at time $t = 1$.

$h_i([ab], [bc]) = [\delta_i(a, b, c), \delta_i(b, c, \#)]$ for $i = 1$ *to* $k - 1$.

$h_{k-1}(\overline{[ab]}, \overline{[bc]}) = [\delta_k(a, b, c), \delta_k(b, c, \#)]$

$h_k(\overline{[ab]}, \overline{[bc]}) = [\delta_{k+1}(a, b, c), \delta_{k+1}(b, c, \#)]$

These rules are used at time $t = 2$. At the leftmost cell, the first argument is not of the form $[ab]$ or $\overline{[ab]}$ but is #. For this cell, the above rules are used with # instead of a.

$h_i([a, b], [c, d]) = [ac, d]$ for $i = 1$ *to* $k - 1$.

$h_k([a, b], [c, d]) = \overline{[ac, d]}$

These rules are used at odd time steps $t = 2k + 1$ for $k \geq 1$.

$h_i([ab, c], [bd, e]) = [\delta_i(a, b, d), \delta_i(b, d, e)]$ for $i = 1$ *to* $k - 1$

$h_{k-1}(\overline{[ab, c]}, \overline{[bd, e]}) = [\delta_k(a, b, d), \delta_k(b, d, e)]$

$h_k(\overline{[ab, c]}, \overline{[bd, e]}) = [\delta_{k+1}(a, b, d), \delta_{k+1}(b, d, e)]$

These rules are used at even time steps $t = 2k$ for $k > 1$. ■

Similarly, we can also prove that the improved simulation depicted in Figure 3.5 is possible in this case; ie. a k^2-$r\mathcal{L}TVCA$ with state set Q can be simulated by a k-$l\mathcal{L}TVOCA$, in twice as much time, and with $O(k\|Q\|^3)$ states.

Thus $rTVCA$s can be simulated by $lTVOCA$s, with fewer functions. For the reverse direction, ie. simulating an $lTVOCA$ by an $rTVCA$, the first problem which arises is the potential distinction between $2n$ time and linear time. In the non-time-varying case, any linear-time CA can be sped up to $2n$ time. The same does not appear to hold for $TVCA$. Suppose we restrict ourselves to $2n$-time $TVOCA$. Then a containment from $lTVOCA$ to $rTVCA$ can be shown, but the condition to be satisfied by $\mathcal{L}$, the class of controlling languages, is now different. $\mathcal{L}$ is required to be closed under compressed composition. This result is stated in the next theorem.

Theorem 3.23 *Let $\mathcal{L}$ be a class of tally languages closed under compressed composition. For $k > 1$, a $2n$-time k-$\mathcal{L}TVOCA$ can be simulated by a real-time k^2-$r\mathcal{L}TVCA$.*

Proof: Let C be a k-$\mathcal{L}TVOCA$ accepting a language in time $2n$. Consider a typical computation of C as shown in Figure 3.7. We draw a diagonal from the top left corner (cell 1 at $t = 1$) to the bottom right corner (cell n at $t = 2n - 1$) and fold the array of cell states

	a	b	c	d
δ_1	A	B	C	D
δ_1	E	F	G	H
δ_2	I	J	K	L
δ_2	M	N	O	P
δ_2	Q	R	S	T
δ_1		U	V	W
δ_1			X	Y
δ_2				Z

$2n$-time 2-$\mathcal{L}TVOCA$

	a	b	c	d
$h_{1,1}$	F E	G $\#$	H $\#$	$\#$ $\#$
$h_{2,2}$	O N	P M	$\#$ $\#$	
$h_{2,1}$	W V	$\#$ U		
$h_{1,2}$	$\#$ Z			

Simulating 4-$r\mathcal{L}TVCA$

Figure 3.7: $2n$-time k-$\mathcal{L}TVOCA$ simulated by k^2-$r\mathcal{L}TVCA$

along this diagonal (the dotted line in the figure). Alternate rows of the folded array, with the rows shifting leftward, can be computed by a CA C', as shown in the same figure.

For a formal construction, let $C = (Q, \#, L_1, \ldots, L_k, \delta_1, \ldots, \delta_k, A)$ be the given k-$l\mathcal{L}TVOCA$. We construct the k^2-$r\mathcal{L}TVCA$ C' as follows.

$C' = (Q', \#, L'_1, \ldots, L'_m, h_1, \ldots, h_m, A')$ where $m = k^2$,

$Q' = Q \cup (Q \times Q)$

$L'_{k(i-1)+j} = \{0^t \mid 0^{2t-1} \in L_i, 0^{2t} \in L_j\}$ for $i = 1$ *to* $k, j = 1$ *to* k. All these languages will belong to $\mathcal{L}$, since $\mathcal{L}$ is closed under compressed composition.

For $a, b, c, d, e, f \in Q$,

$h_{k(i-1)+j}(a, b, c) = [\delta_j(\delta_i(a, b), \delta_i(b, c)), \#]$

$h_{k(i-1)+j}(\#, b, c) = [\delta_j(\delta_i(\#, b), \delta_i(b, c)), \delta_j(\#, \delta_i(b, c))]$

$h_{k(i-1)+j}([a, b], [c, d], [e, f]) = [\delta_j(\delta_i(a, c), \delta_i(c, e)), \delta_j(\delta_i(f, d), \delta_i(d, b))]$

$$h_{k(i-1)+j}(\#, [c,d], [e,f]) = [\delta_j(\delta_i(d,c), \delta_i(c,e)), \delta_j(\delta_i(f,d), \delta_i(d,c))]$$
$$A' = Q \times A$$ ■

Corollary 3.24 *For $\mathcal{L}$ closed under union, doubling, and compressed composition, the classes of languages accepted by $2n$-time k-$\mathcal{L}TVOCA$ and by k^2-$r\mathcal{L}TVCA$ coincide.*

3.4 Conclusions

In this chapter we have introduced $TVCA$, which are CA with an external global control imposed uniformly on all the cells. The power of such CA has been examined when the nature and the extent of the control is varied. The results are well summarised in Figure 3.8. In the next few chapters of this thesis we will interpret this external control and time-variation in different ways, giving rise to CA which are relativised, nondeterministic, probabilistic or alternating. $TVCA$ provide a unifying framework for studying all these models. Within the context of $TVCA$ themselves some interesting problems are still open, notably, whether $k+1$ functions are better than k for rCA. As can be seen from the Corollary 3.20, there is no difference between k and $k+1$ for linear-time $TVCA$ and for $k > 1$. This does not appear to be the case for real-time $TVCA$, even if the controlling languages are also constrained to be in rCA. A proof that $k+1$ is indeed better than k will, from Theorem 3.16, give a proper separation of the classes rCA and lCA. A proof to the contrary seems difficult to obtain. Thus a solution to this problem in either way would be of considerable interest and significance.

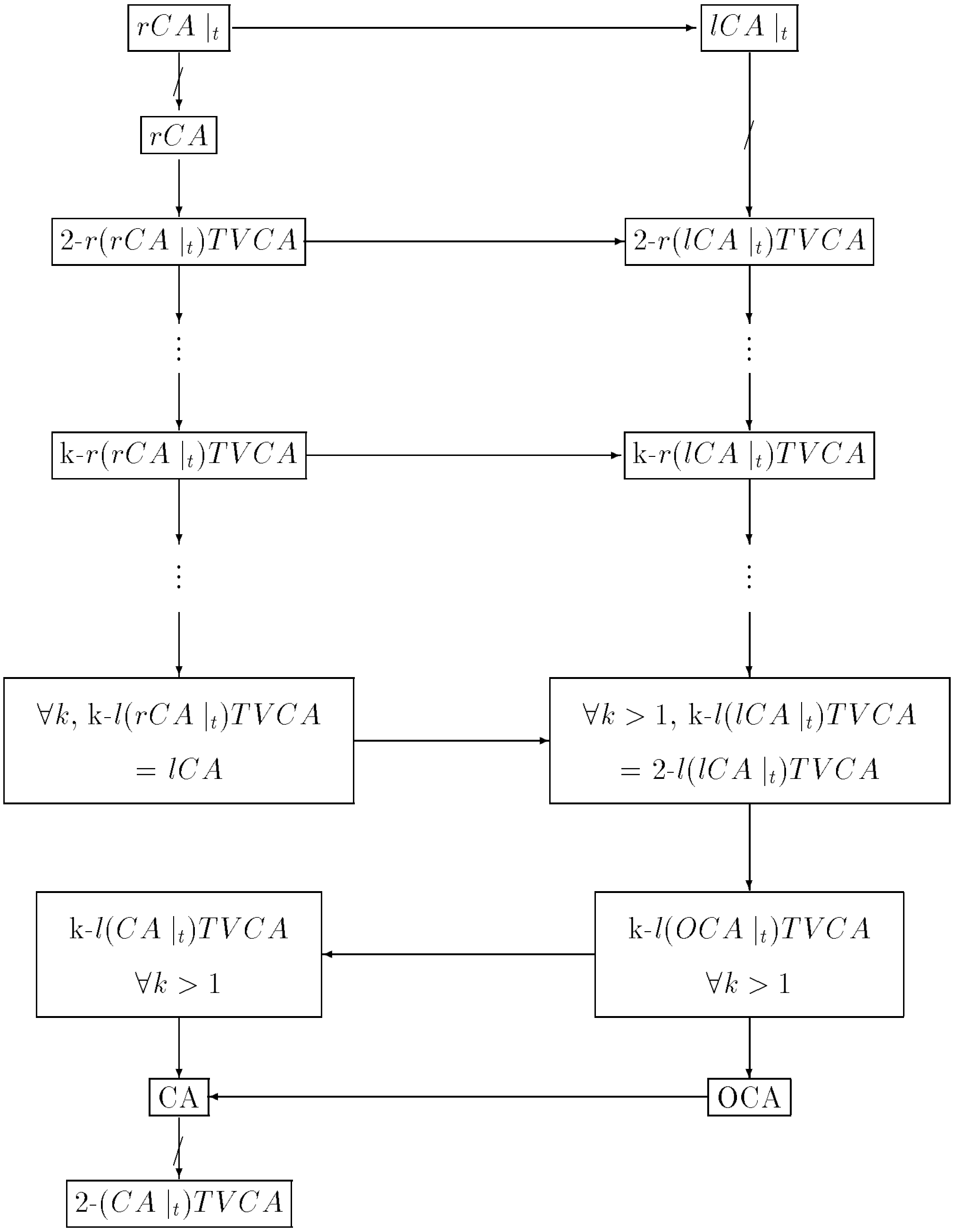


Figure 3.8: Inclusions amongst some $TVCA$ classes

Chapter 4

A Study of the Oracle Access Mechanism provided by TVCA as Relativised CA

In this chapter, we interpret the notion of time-varying CA, introduced in the previous chapter, as an oracle access mechanism for relativising CA. The effect of the time-variation described here is similar to the effect of allowing a machine to access an oracle and query it on some specific membership problem. Such oracle accesses have been used to define relativised Turing machines, and the study of these machines has thrown up an enormously complex body of work, providing a lot of useful insights into the structure of important complexity classes. Motivated by the hope of obtaining similar useful insights in the context of CA classes, we formalise the notion of relativised cellular automata from time-varying cellular automata ($TVCA$), and explore the power of such an oracle access mechanism. The study of such relativised CA has also led us to re-examine the CA complexity classes under different restricted relativisations; this work will be taken up in the next chapter.

4.1 TVCA as Relativised CA

A frequently used technique in structural complexity theory is relativisation. The idea is to study the power of a resource-bounded class, assuming that answers to instances of some specific problem can be obtained at no extra cost. (The cost of framing the question itself is, of course, counted.) Machines describing such classes are said to be relativised, and the problem for which they can obtain answers at no extra cost is called the oracle. Thus a relativised Turing machine is a Turing machine with a separate tape for writing oracle queries, and three special states $q_?$, q_y, q_n. The state $q_?$ is used to ask whether the string on the query tape belongs to the oracle set. If this is so, then at the next time step the machine enters state q_y; else it enters state q_n. The computation then proceeds normally, until a fresh query is made by entering the state $q_?$ again. Relativisation of Turing machines and the study of the classes they define has provided a lot of insight into the structural properties of important complexity classes [BDG88, BDG90]. We hope that similar useful insights into the structure of the CA complexity classes rCA, lCA and CA can be obtained through relativisation.

However, the notion of relativisation in the context of CA is difficult to formalise, because the control of the computation is not centralised but is distributed over all the cells. Besides, there is no tape on which to write out queries to the oracle. Even if queries were to be written in a separate component of the state of each cell, synchronising the query procedure would become contrived and cumbersome. We adopt the approach of using implicit oracle querying as provided by $TVCA$. In $TVCA$, the transition rule is not fixed, but depends on how many time steps have elapsed since the $TVCA$ operation began. Thus an oracle is implicitly queried on inputs $t = 1, 2, \ldots$, and its replies tell the CA which transition rule is to be used. We consider the simplest case where there are only two possible rules to be used, δ_1 and δ_2; ie. the $TVCA$ is a 2-$TVCA$. The usage of δ_1 and δ_2 is controlled by a tally set L. The $TVCA$ thus acts as if it has an oracle answering queries about the membership of strings in L. The queries are quite restricted; the oracle must be queried at each time step, and it must be queried on strings $0^1, 0^2, \ldots$ in that order. However, even this restrictive notion of querying appears to be quite non-trivial, and is the subject of this and the next chapter.

With this interpretation of 2-$TVCA$ as oracle CA, we can now specify a 2-$TVCA$ as $C(L)$, where $C = (Q, \#, \delta_1, \delta_2, A)$ and L is the oracle (the controlling language of the $TVCA$). When the oracle is an empty set, this denotes the CA $(Q, \#, \delta_2, A)$. Classes of CA $\mathcal{S}$ with a particular oracle L are denoted $\mathcal{S}(L)$; eg. $rCA(L)$, $CA(L)$ etc. Classes of CA $\mathcal{S}$ relative to a set of tally oracles $\mathcal{L}$ are denoted $\mathcal{S}(\mathcal{L})$; eg. $CA(rCA)$ etc. By $\mathcal{S}(\mathcal{L})$ we will mean both the class of machines in $\mathcal{S}$ using an oracle from $\mathcal{L}$ and the class of languages accepted by such machines.

Note that while the controlling languages of the $TVCA$ are naturally described by tally languages, there is no reason why they may not be also described by non-tally languages. To generalise the relativisation concept to non-tally languages—without loss of generality we consider languages over a binary alphabet $\{0, 1\}$—we formulate the oracle queries in a different way. The operation of a 2-$TVCA$ is now expressed as follows:

$$\delta(a, b, c, i) = \begin{cases} \delta_1(a, b, c) & \text{if } \langle i \rangle \in L \\ \delta_2(a, b, c) & \text{otherwise} \end{cases}$$

where $\langle i \rangle$ is the standard binary representation of the natural number i, as described in chapter 2.

Thus the $TVCA$ is now interpreted as a relativised CA, where the oracle is a binary language. By itself, this does not seem to be significantly different from the relativised CA decribed earlier, where tally oracles are used to express the same partition of $\mathbf{N}$. However there is a crucial difference. If a tally oracle is used, then $\forall L$, $L \in rOCA(L)$. This follows from Lemma 3.9. If a binary oracle is used, then the oracle answers the query about the membership of x after $2^{|x|}$ steps (since there are those many strings of length $|x|$ or less), so a naive approach will require an exponential amount of time. In fact, with a binary oracle L, if L can be accepted in a polynomial amount of time by such a CA, then this means that L is self-reducible, and membership of x in L can be answered in polynomial time by querying the oracle only on strings which are bounded in length by $c \log |x|$, for some c. This is a much stronger notion of reducibility than that obtained through relativised CA with tally oracles, and is a motivating factor for studying relativised CA with binary oracles.

4.2 Querying Tally Oracles

The oracle access mechanism as defined above for relativising CA is quite non-standard. As such, it makes sense to first examine how basic equalities and inclusions behave when the corresponding classes are relativised. In this section we primarily focus on the relationship between CA and $DSPACE(n)$, where the Turing machines defining the latter class can be relativised in various ways. In the unrelativised case, it is well known (Lemma 2.3) that the two classes are equal.

We assume, without loss of generality, that the $DSPACE(n)$ machine has a single worktape, apart from possibly a query tape. We first consider the bounded query relativisation model for Turing machines as described in [Bus88]; here the space bound for the machine also applies to the query tape. Let us compare relativised CA with such relativised $DSPACE(n)$ machines. Relativised $DSPACE(n)$ Turing machines (we denote this class by$DSPACE^L(n)$, for oracle L) have the advantage of being able to query the oracle on strings in any order. On the other hand, since the space bound also applies to the query tape, a relativised $DSPACE(n)$ Turing machine can query its oracle only on strings $0^1, 0^2, \ldots, 0^{cn}$ for some constant c. A CA can query its oracle on strings of any length, since it is not time-bounded. This gives us the following result. Here we consider only tally oracles.

Theorem 4.1 *Under the bounded query model, for any tally language A,*

$$DSPACE^A(n) \subseteq CA(A).$$

Proof: Let M be a $DSPACE(n)$ machine with oracle A. M can query A on strings $0^1, 0^2, \ldots, 0^{cn}$ for some constant c. However the querying may take place in any order. So the simulating CA C, in the first cn steps, merely collects the oracle responses in its n length array, putting c answers into each cell. This is done by letting only the leftmost cell be sensitive to the oracle during this phase. The other cells merely provide storage space for the responses. After cn time steps are over (detected by sending a signal from end to end at speed $1/c$), the CA synchronises itself and starts simulating M. The worktape and the query tape of M are stored in different components of the state of each cell, with each CA cell holding the contents of c squares of each tape. The tape head position is marked by a special symbol. As long as M does not enter state $q_?$, it can be simulated by the CA in a straightforward fashion. When M enters state $q_?$, the CA suspends simulation to look up the oracle response which it had stored in the earlier part of the computation. Based on this, state q_y or q_n is entered and the simulation resumed. Such a simulation requires that after the first cn steps, the operation of the CA is no longer time-varying. But since relativised CA have implicit oracle querying, the CA must be made insensitive to its oracle by setting a flag, in the leftmost cell, after cn steps. δ_1 and δ_2 are designed to differ only in the leftmost cell, and even there, only if the flag is not set. Thus the $DSPACE^A(n)$ machine is correctly simulated by the CA. ∎

In fact, since the $DSPACE(n)$ machine must run within $O(2^{cn})$ time or enter a rejecting infinite loop, the simulation outlined above can be modified so that the CA runs for at most $O(2^{cn})$ time. So a relativised $DSPACE(n)$ Turing machine can be simulated by an $O(2^{cn})$ time-bounded relativised CA.

The above result appears to hold because of the difference in the potential query space of the $DSPACE(n)$ machine and the CA. But, note that in the simulation of the $DSPACE(n)$ machine, the CA does not make any additional queries; despite the additional implicit querying provided by the $TVCA$ mechanism, the CA is sensitive only to oracle responses to queries from the $DSPACE(n)$ machine's query space. And yet, the reverse containment is not true, simply because the CA has access to a larger query space. In fact, it is quite easy to construct a recursive oracle A such that $CA(A)$ is not contained in $DSPACE^A(n)$, by exploiting the additional querying that the CA can do. (The construction is not outlined here; however, it is very similar to the oracle construction for separating lCA and CA, as in Theorem 4.9.) Such a construction, thus, does not tell us much about the relative computing powers of these two models. To compare the computing power, we must allow both models to have the same potential query space; otherwise the comparison may be unfair. We may relax the space bound on the query tape for relativised $DSPACE(n)$ Turing machines. Consider the unrestricted query model studied in [Bus88, LL76]. Here no space bound applies to the query tape. This agrees with there being no time bound and hence no query bound on a relativised CA. Comparing such machines, we have the following result:

Theorem 4.2 *Under the unrestricted query model, for tally oracle A,*

$$CA(A) \subseteq DSPACE^A(n).$$

Proof: The $DSPACE(n)$ machine begins with the initial configuration on its tape. Consider the i^{th} step in the CA operation. At this step, either δ_1 or δ_2, depending on whether or not

0^i belongs to A, is applied to all cells synchronously. If the $DSPACE(n)$ machine knows which of these rules is to be applied, then it can update the contents of its tape according to this rule in $O(n)$ time. To find out which rule to apply, it must query its oracle on 0^i. Since at the previous stage the oracle was queried on 0^{i-1}, all it has to do is append a 0 to the string on the query tape. Thus to simulate one step of the CA, the $DSPACE(n)$ machine does the following: Append 0 to the string on the query tape. Query the oracle. If state q_y is entered, update the worktape array using transition function δ_1; else update using transition function δ_2. ■

Note that this will not work in a model where the query tape is erased after each query. Such is the case for the deterministic query mechanism studied in [RST84]; this mechanism was defined primarily to study relativised nondeterministic machines and works as follows: The machine proceeds normally as long as the query tape is blank. The moment something is written on the query tape, the machine starts acting in a deterministic fashion until the query is actually made. When the query is answered, the query tape is automatically erased and the machine reverts to being a nondeterministic machine. Here we are not considering nondeterminism at all. But the automatic erasing of the query tape after each query does make a difference to the space-bounded computation we are interested in, as can be seen below. Apart from query tape erasure, we also impose a space bound $S(n)$ — which is not necessarily the same as the worktape space bound — on the query tape. This allows the machine to make upto $S(n)$ distinct queries to a tally oracle. An $S(n)$-time CA also has the same set of potential queries. Comparing these classes, we have the following interesting result:

Theorem 4.3 *In the deterministic query model, for every tally oracle A, a $DSPACE(n)$ machine with oracle A and query tape space bound $S(n)$ can simulate an $S(n)$-time CA with oracle A, if $S(n)$ belongs to $O(2^{cn})$.*

Proof: The $DSPACE(n)$ machine begins with the initial configuration on its tape. Consider the i^{th} step in the CA operation. At this step, either δ_1 or δ_2, depending on whether or not 0^i belongs to A, is applied to all cells synchronously. If the $DSPACE(n)$ machine knows which of these rules is to be applied, then it can update the contents of its tape according to this rule in $O(n)$ time. To find out which rule to apply, it must query its oracle on 0^i. But for this it must know i. Since the query tape gets erased after each query, the current value of i must be stored in the worktape. The worktape is linear-space-bounded, and can store numbers upto 2^{cn} only, in binary notation. This is sufficient if $S(n)$ belongs to $O(2^{cn})$. Thus the $DSPACE(n)$ machine can simulate the CA. ■

As a corollary, we obtain the result that the containment $lCA \subseteq DSPACE(n)$ always relativises, with the lCA relativised via $TVCA$ and the $DSPACE(n)$ machine relativised in any of the three ways described above.

Corollary 4.4 $\forall A, lCA(A) \subseteq DSPACE^A(n)$

We do not know whether Theorem 4.3 can be strengthened to show the reverse simulation as well. We suspect that it cannot, because the CA oracle access mechanism, as mentioned earlier, is quite restrictive. Also, bounding the potential query space of the CA means bounding its running time. Since the running time of the $DSPACE(n)$ machine has not been bounded, (it is only implicitly bounded by 2^{cn}, the number of distinct configurations. The $S(n)$ cells on the query tape are used only with a unary alphabet, and hence generate only $S(n) \in O(2^{cn})$ configurations.) the comparison is again biased, this time against the CA. So the converse is not likely to be true. But if $S(n)$ is in $\theta(2^{cn})$, then the running times of the $DSPACE(n)$ machine and the CA are the same and the query space is also the same. Even in this case, we suspect that the converse does not hold, the constraint now being the order in which querying is permitted. However, we have not been able to prove this.

Another thing which is also not known is which types of oracles A leave the relativised class $CA(A)$ closed under complementation. $DSPACE^A(n)$ is closed under complementation for any A (except if unrestricted querying is allowed), since it is a space-bounded class with a space-constructible bound. Thus the closure/non-closure of $CA(A)$ appears to have a major impact on the difference between relativised CA and relativised $DSPACE(n)$ Turing machines.

Before closing this section we would like to mention that as a direct consequence of Theorem 3.11, the $lCA \subseteq OCA$ containment also relativises for all tally oracles within OCA.

4.3 Querying Non-Tally Oracles

In this section we study relativisation of CA as in the previous section, but with respect to non-tally oracles. Consider the analogue of Theorem 4.1 in this light. A $DSPACE^A(n)$ machine can query the oracle A on any string of length upto cn, for some constant c, and in any order. The $CA(A)$ can also query A on any string, but in a specific order. So to be able to simulate the $DSPACE^A(n)$ machine, it should first collect and store all oracle replies that could possibly be required. But there are 2^{cn} such strings; storing all the responses is no longer possible within the real-space-bounded array of the CA! Already the first result fails to carry over.

On the other hand consider Theorem 4.2, the unrestricted query model. As long as the query tape can also be used as a worktape (to increment its contents), the same simulation holds. However, if the query tape is write-only (ie. the query tape head cannot move left) then the construction fails.

The deterministic query model eliminates this problem by erasing the tape contents after each query. Now the $DSPACE(n)$ machine needs to remember the last query made on its worktape. In an analogue of Theorem 4.3, we now have the following conditions:

Theorem 4.5 *A CA with binary oracle A, running in time $S(n)$, can be simulated by a $DSPACE(n)$ machine with the same oracle and with a $\log(S(n))$ bound on the query tape space, provided $S(n)$ belongs to $O(2^{cn})$.*

4.4 Separation Results

In this section we show how to construct oracles separating various CA language classes. We also show how to effect strong separations via immune sets. A set X is said to be immune to a class $\mathcal{L}$ ($\mathcal{L}$-immune) if X is infinite and contains no infinite subset belonging to $\mathcal{L}$. An oracle L strongly separates classes $\mathcal{L}_1$ and $\mathcal{L}_2$, where $\mathcal{L}_1 \subseteq \mathcal{L}_2$, if $\mathcal{L}_2(L)$ contains a set that is $\mathcal{L}_1(L)$-immune.

We will construct oracle sets A and B such that $lCA(A) \neq CA(A)$ and $rCA(B) \neq lCA(B)$. We will then generalise the construction to obtain sets C and D such that C

(respectively, D) strongly separates lCA from CA (respectively, rCA from lCA). All these separations hinge around the fact that in our model of relativisation, a time bound imposes a stringent bound on the potential query space. Before doing so we will show some intermediate results.

We first need to fix an enumeration of rCA and lCA. We assume that the state set of a CA is $\{0, 1, \ldots, k\}$ for some finite k, and that # is the state 0. The number of distinct transition rules is $m = k^{(k+1)^2 k}$ (since # always maps to # and no other symbol maps to #), and the number of possible sets of accepting states is 2^k (since # cannot be an accepting state). Thus a $TVCA$ can be specified, without the oracle language, as a 4-tuple (k, i_1, i_2, j) where i_1, i_2 are integers between 1 and m specifying the transition rules, and j is an integer between 1 and 2^k specifying A. Let $\phi_i, i = 1, 2, \ldots$ be an ordering of such 4-tuples. This serves as an enumeration of CA as well as rCA. To enumerate lCA, we order pairs (ϕ_i, c_j) where ϕ_i is the machine and c_j specifies the constant for linear-time acceptance. Let this ordering be ψ_i, an enumeration of lCA. Since we allow the set of accepting states to be empty, CAs accepting the empty set will occur infinitely often in both these enumerations.

Proposition 4.6 *Let $f : \Sigma^+ \to \mathbf{N}$ be a CA-time-constructible function, and let A be some set acting as an oracle. If A is tally, then the set $L_{ft}(A)$, given by*

$$L_{ft}(A) = \{x \in \Sigma^+ \mid 0^m \in A, \text{ where } m = f(x)\}$$

can be accepted by a CA, with tally oracle A, in time $f(x)$. Thus if $f_1 : \mathbf{N} \to \mathbf{N}$ is the function defined as $f_1(n) = \max_{|x|=n} f(x)$, then $L_{ft}(A)$ can be accepted by a CA, with oracle A, in time $f_1(n)$. Similarly, if A is non-tally, then the set $L_f(A)$, given by

$$L_f(A) = \{x \in \Sigma^+ \mid \langle m \rangle \in A, \text{ where } m = f(x)\}$$

can be accepted by a CA, with oracle A, in time $f_1(n)$.

Proof: Since f is CA-time-constructible, we can design a relativised CA where both δ_1 and δ_2 compute $m = f(x)$. At time instant m, only δ_1 puts the CA into an accepting state. Thus $L_{ft}(A)$ or $L_f(A)$ is accepted. ■

Lemma 4.7 *1. $f : \{0\}^+ \to \mathbf{N}$, where $f(0^n) = 2n$, is CA-time-constructible.*

2. *$g : \{0\}^+ \rightarrow \mathbf{N}$, where $g(0^n) = n^2$, is CA-time-constructible.*

3. *$h : \{0,1\}^+ \rightarrow \mathbf{N}$, where $h(\langle n \rangle) = n$, is CA-time-constructible in a weak sense— the* rightmost *cell enters a special state after exactly n steps.*

Proof: (a) This is straightforward — the CA just has to send a signal from right to left at half speed.

(b) This is achieved as follows. At $t = 1$, the rightmost cell enters a special bounding state **b** and also sends a signal \$ left. \$ travels upto a cell marked **b** and then returns to the rightmost cell before setting out leftwards again. Every time \$ reaches a **b** cell, the **b** marker moves one unit left. Thus the \$ goes through excursions of length $2 \times 1, 2 \times 2, \ldots, 2 \times i, \ldots$. When the \$ reaches the leftmost cell, the number of steps elapsed is $\left[\sum_{i=1}^{n-1}(2 \times i)\right] + n = n^2$. An example is shown in Figure 4.1.

(c) This has been shown in Proposition 3.13. ■

For tally sets A and B, let
$$\begin{aligned} L_1(A) &= \{0^s \mid 0^{s^2} \in A\} = \sqrt{A}, \text{ and} \\ L_2(B) &= \{0^s \mid 0^{2s} \in B\} = \tfrac{1}{2}B. \end{aligned}$$

Then, in the above notation, $\sqrt{A} = L_{gt}(A)$ and $\frac{1}{2}B = L_{ft}(B)$, where f and g are as in Lemma 4.7. Clearly, $f_1(n) = 2n$ and $g_1(n) = n^2$. The next lemma now follows from the preceding two results.

Lemma 4.8 $\forall A, \ \sqrt{A} \in CA(A).$
$\forall B, \ \frac{1}{2}B \in lCA(B).$

By direct diagonalisation we can now construct sets A and B such that $\sqrt{A} \notin lCA(A)$ and $\frac{1}{2}B \notin rCA(B)$, giving the following result.

Theorem 4.9 *(a) There exists an oracle A such that $lCA(A) \neq CA(A)$.*

(b) There exists an oracle B such that $rCA(B) \neq lCA(B)$.

Proof: (a) Let $\psi_1, \psi_2, \ldots$ be an enumeration of relativised lCA, with constants $c_1, c_2, \ldots$. For any tally set X, $\sqrt{X}$ can be accepted by a CA with oracle X. We will incrementally construct a tally set A such that for any relativised lCA ψ_i, the language accepted by ψ_i with oracle A differs from $\sqrt{A}$. This will prove the theorem's first assertion.

t	0	0	0	0	#
1	.	.	.	**b** $\overset{\$}{\leftarrow}$	#
2	.	.	**b**	$\overset{\$}{\rightarrow}$	#
3	.	.	**b**	$\overset{\$}{\leftarrow}$	#
4	.	.	**b** $\overset{\$}{\leftarrow}$	.	#
5	.	**b**	$\overset{\$}{\rightarrow}$	.	#
6	.	**b**	.	$\overset{\$}{\rightarrow}$	#
7	.	**b**	.	$\overset{\$}{\leftarrow}$	#
8	.	**b**	$\overset{\$}{\leftarrow}$	.	#
9	.	**b** $\overset{\$}{\leftarrow}$	.	.	#
10	**b**	$\overset{\$}{\rightarrow}$	.	.	#
11	**b**	.	$\overset{\$}{\rightarrow}$	.	#
12	**b**	.	.	$\overset{\$}{\rightarrow}$	#
13	**b**	.	.	$\overset{\$}{\leftarrow}$	#
14	**b**	.	$\overset{\$}{\leftarrow}$	.	#
15	**b**	$\overset{\$}{\leftarrow}$	.	.	#
16	**b** $\overset{\$}{\leftarrow}$	.	.	.	#

Figure 4.1: Computing n^2 on a CA.

Stage 0: $A_0 = \emptyset, m_0 = 0$.

Stage i: Choose the smallest integer m_i satisfying

1. $c_i m_i < m_i^2$
2. $\forall j < i$, $c_j m_j < m_i^2$
3. $\forall j < i$, $m_i \neq m_j$

The first condition ensures that with input 0^{m_i}, ψ_i does not get a chance to query A on $0^{m_i^2}$. The second condition ensures that the inclusion/exclusion of $0^{m_i^2}$ in/from A does not change the behaviour of the $TVCA$ already considered. The third condition ensures that a string once excluded from A cannot subsequently be included in A.

Simulate ψ_i on 0^{m_i} for $c_i m_i$ steps, using A_{i-1} as oracle. If 0^{m_i} is accepted, then $A_i = A_{i-1}$, else $A_i = A_{i-1} \cup \{0^{m_i^2}\}$.

$$A = \lim_{n \to \infty} A_n = \{w \mid w \text{ belongs to all but finitely many } A_n\}$$

Since our construction never deletes strings from any A_n, $A = \bigcup_{n>0} A_n$.

Claim: The set A so constructed satisfies "$\sqrt{A}$ cannot be accepted by any relativised lCA with oracle A".

Proof of claim: By contradiction. Assume that for some i, $\psi_i(A)$ accepts $\sqrt{A}$. On input 0^{m_i}, ψ_i queries A on strings of length upto $c_i m_i$. By our construction,

$$A \cap \{0^j \mid j \leq c_i m_i\} = A_{i-1} \cap \{0^j \mid j \leq c_i m_i\}$$

Thus on input 0^{m_i}, $\psi_i(A)$ behaves as $\psi_i(A_{i-1})$. Suppose $\psi_i(A_{i-1})$ accepts 0^{m_i}. Then our construction excludes $0^{m_i^2}$ from A_i and A. Thus 0^{m_i} is in $L(\psi_i(A_{i-1})) - \sqrt{A}$. On the other hand, if $\psi_i(A_{i-1})$ does not accept 0^{m_i}, then $0^{m_i^2}$ is included in A_i. Since words are never deleted from A, it remains in A, and hence 0^{m_i} is in $\sqrt{A}$. Thus 0^{m_i} is in $\sqrt{A} - L(\psi_i(A_{i-1}))$. In either case, $\psi_i(A)$ cannot be correctly accepting $\sqrt{A}$.

Essentially, our construction ensures that $\forall i, 0^{m_i} \in L(\psi_i(A)) \triangle \sqrt{A}$. This proves the claim.

(b) Let $\phi_i, i = 1, 2, \ldots$ be an enumeration of relativised rCA. We will incrementally construct a tally set B such that $\frac{1}{2}B$ is not accepted by any $\phi_i(B)$. Since $\frac{1}{2}B$ can be accepted by an lCA with oracle B, the assertion will be proved.

Stage 0: $B_0 = \emptyset$.

Stage i: Simulate ϕ_i on the string 0^i for i steps, using oracle B_{i-1}. If ϕ_i accepts the input, then $B_i = B_{i-1}$, else $B_i = B_{i-1} \cup \{0^{2i}\}$.

$$B = \lim_{n \to \infty} B_n = \{w \mid w \text{ belongs to all but finitely many } B_n\}$$

Claim: The set B so constructed satisfies "$\frac{1}{2}B$ cannot be accepted by any relativised rCA with oracle B".

This proof is similar to that in (a). Here we ensure that $\forall \text{i}, 0^i \in L(\phi_i(B)) \triangle \frac{1}{2}B$. ∎

Corollary 4.10 *There exists an oracle X such that $rCA(X) \neq lCA(X) \neq CA(X)$.*

Proof: In the above theorem, we have seen how to construct oracles A and B separating CA from lCA and lCA from rCA respectively. If the construction of A is modified so that only odd length strings are chosen (choose odd m_i), the separation is still valid. Now the even length strings can be used to separate lCA from rCA, as in the construction of B. ϕ_i will now be simulated on the string 0^{2i} instead of 0^i, and according to the outcome 0^{4i} may or may not be added to the oracle. It is easy to see that the oracle so constructed simultaneously separates CA from lCA and lCA from rCA. ∎

We now show that these results can be strengthened to strong separations. We will use delayed diagonalisation to show the following.

Theorem 4.11 *(a) There is a tally oracle C such that $CA(C)$ contains an $lCA(C)$-immune set.*

(b) There is a tally oracle D such that $lCA(D)$ contains an $rCA(D)$-immune set.

Proof: (a) Let $\psi_1, \psi_2, \ldots$ be an enumeration of oracle lCA, with constants $c_1, c_2, \ldots$. For any tally set X, $\sqrt{X}$ can be accepted by a CA with oracle X. We will incrementally construct a set C such that

1. $\sqrt{C}$ is infinite, and
2. For any relativised lCA ψ_i, the language accepted by ψ_i with oracle C, $L(\psi_i(C))$, is not a subset of $\sqrt{C}$, except, possibly, if it is finite. This condition, for the i^{th} lCA, is called requirement i.

This will prove the theorem's first assertion.

Stage 0 $C_0 = \emptyset$, $k_0 = 0$, $S_0 = \emptyset$.

(C_i holds the oracle constructed at stage i. S_i holds all indices less than or equal to i, for which the requirement is not yet satisfied at stage i.)

Stage n: $S_n = S_{n-1} \cup \{n\}$

Choose the smallest integer k_n satisfying

(i) $\forall i \leq n,\ c_i k_n < k_n^2$

(ii) $\forall i, j < n,\ c_i k_j < k_n^2$

(iii) $\forall i < n,\ k_i \neq k_n$

If, for some $i \in S_n$, 0^{k_n} is accepted by lCA ψ_i using oracle C_{n-1},

then let i_n be the smallest such index.

$$S_n = S_n - \{i_n\}$$

$$C_n = C_{n-1}$$

else

$$C_n = C_{n-1} \cup \{0^{k_n^2}\}$$

$$C = \lim_{n\to\infty} C_n = \{w \mid w \text{ belongs to all but finitely many } C_n\}$$

Since our construction never deletes strings from any C_n, $C = \bigcup_{n>0} C_n$. Similarly, let $S = \lim_{n\to\infty} S_n$.

Condition (i) ensures that none of the first n lCA can query the oracle on $0^{k_n^2}$ when the input is 0^{k_n}. The second condition ensures that the behaviour of the lCA already considered will not be affected by the inclusion/exclusion of $0^{k_n^2}$ in C. The third condition ensures that a string once included in (excluded from) C will not be subsequently excluded (included, respectively). So from conditions (ii) and (iii) we can conclude that 0^{k_n} is accepted by ψ_i with oracle C_{n-1}, for $i \leq n$, if and only if 0^{k_n} is accepted by ψ_i with oracle C.

Claim 1: $\sqrt{C}$ is infinite. To prove this, it suffices to prove that C is infinite. Suppose that C is finite. Then there is an n_0 such that after stage n_0, C does not grow any more, ie. the construction never follows the **else** part of the algorithm. This means that in all subsequent stages, one index enters S and one index leaves it; S does not grow in size any more. But there are infinitely many lCA accepting the empty set; none of the corresponding indices can ever leave S, since to leave S, the lCA must accept some string 0^{k_n}. So the size of S must grow unboundedly, a contradiction. Hence C and $\sqrt{C}$ are infinite.

Claim 2: $\forall i > 0$, if $L(\psi_i(C))$ is infinite, then $L(\psi_i(C))$ is not a subset of $\sqrt{C}$ (ie. $L(\psi_i(C))$ contains a string not in $\sqrt{C}$).

Suppose that there is a j contrary to the claim. Then the index j is in S. $L(\psi_j(C))$ is contained in $\sqrt{C}$, so it contains strings of the form 0^{k_n} only. Let n_0 be the first stage of construction satisfying

- $j \leq n_0$ (ie. j has already entered S).
- n_0 is large enough so that all of the indices less than j either have already left S or will remain in S forever.
- $0^{k_{n_0}} \in L(\psi_j(C))$.

Such an n_0 will exist, since by assumption $L(\psi_j(C))$ is infinite. From the preceding discussion, we know that $0^{k_{n_0}}$ is accepted by ψ_j with oracle C_{n_0-1}. Thus at stage n_0, the construction will follow the **then** part and choose $i_{n_0} = j$. So no string is added to C at this stage. This means that $0^{k_{n_0}^2} \notin C$, and therefore $0^{k_{n_0}} \notin \sqrt{C}$. But $0^{k_{n_0}} \in L(\psi_j(C))$, so $L(\psi_j(C))$ is not contained in $\sqrt{C}$.

Thus the only indices which never leave S, ie. for which the requirement is not explicitly satisfied, must correspond to lCA accepting finite sets. This shows that $\sqrt{C}$ is $lCA(C)$-immune.

(b) Let $\phi_i, i = 1, 2, \ldots$ be an enumeration of oracle rCA. We incrementally construct a set D such that $\frac{1}{2}D$ is infinite, and, for any relativised rCA ϕ_i, the language accepted by ϕ_i with oracle D is not a subset of $\frac{1}{2}D$, except, possibly, if it is finite. Since $\forall D$, $\frac{1}{2}D$ can be accepted by an lCA with oracle D, the assertion will be proved.

Stage 0: $D_0 = \emptyset$, $k_0 = 0$, $S_0 = \emptyset$.

Stage n: $S_n = S_{n-1} \cup \{n\}$

Let $k_n = n$.

If for some $i \in S_n$, 0^{k_n} is accepted by rCA ϕ_i using oracle D_{n-1},

then let i_n be the smallest such index.

$$S_n = S_n - \{i_n\}$$

$$D_n = D_{n-1}$$

else $D_n = D_{n-1} \cup \{0^{2k_n}\}$.

$$D = \lim_{n\to\infty} D_n = \{w \mid w \text{ belongs to all but finitely many } D_n\}$$

The proof that $\frac{1}{2}D$ is $rCA(D)$-immune is similar to the proof in part (a) above. ∎

The proofs of both Theorem 4.9 and Theorem 4.11 are identical in nature to the corresponding oracle constructions for separating P and NP; refer [BDG90].

Before closing this section we would like to point out that the separation results of Theorems 4.9 and 4.11 hold even if non-tally oracles are considered.

Theorem 4.12 $\exists A \subseteq \{0,1\}^*, \quad lCA(A) \neq CA(A)$

$\exists B \subseteq \{0,1\}^*, \quad rCA(B) \neq lCA(B)$

Proof: In Theorem 4.9 we have shown the existence of a tally oracle A such that $\sqrt{A} \in CA(A) - lCA(A)$. The basic idea was that given an input x of length n, an lCA does not have enough time to count upto n^2. The same idea can be used here to diagonalise out of the class $lCA(A)$ where A is a binary oracle. We now use the sets $L_g(A)$ and $L_f(B)$ from Lemma 4.7, instead of the sets $L_{gt}(A)$ and $L_{ft}(B)$. In fact, the language in the difference, by this construction, will still be tally, even though the separating oracle is not. The same argument can be used for the other separation. ■

Theorem 4.13 *(a) There is a non-tally oracle C such that $\sqrt{C} \in CA(C) - lCA(C)$ and $\sqrt{C}$ is $lCA(C)$-immune.*

(b) There is a non-tally oracle D such that $\frac{1}{2}(D) \in lCA(D) - rCA(D)$ and $\frac{1}{2}(D)$ is $rCA(D)$-immune.

Proof: As above. ■

4.5 Conclusions

In this chapter we have considered the interpretation of time-varying CA as relativised CA. The oracle access mechanism defined by $TVCA$ is quite non-standard, and in sections 4.2 and 4.3 we have compared it with the standard relativisation of Turing machines. It would be interesting to strengthen these comparative statements by proving the conjecture that the converses of Theorems 4.2 and 4.3 are false. We find that even this restrictive oracle access mechanism suffices to exhibit separations, and even strong separations, of the relativised classes rCA, lCA and CA. Thus it is to be hoped that the study of such relativised CA may provide more insight into the structure of the CA complexity classes.

It is of independent interest to propose different mechanisms for relativising CA; these would correspond to different reducibilities amongst languages using CA as the model of computation. One such reducibility has been considered briefly in [BC84] under the name Generalised CA (GCA). In a GCA, no single cell is designated as an accepting cell. Instead, the entire configuration after $T(n)$ steps is checked. Let the input be x, and let the GCA "transform" it to y in $T(n)$ steps. Now the GCA is said to accept x if and only if y belongs to some prespecified language L. Let the language so accepted by the GCA be L'. Then, in a sense, L' has been reduced to L; the GCA performs some computation, then makes a single query to an oracle L, and reports acceptance or rejection accordingly. This corresponds to a form of many-one CA reducibility. Under this reducibility, it has been shown [BC84] that languages CA-reducible to rCA in real time are in lCA. This is analogous to our result of Theorem 3.16, which uses a $TVCA$ form of CA-reducibility (this kind of reducibility is not many-one, since several queries are required). However, under the reducibility of [BC84], the $rCA(rCA)$ class coincides with the class lCA, whereas in the $TVCA$ form of reducibility, we only have a containment in Theorem 3.16. Such generalised acceptance criteria have been further studied in [IPK85a, KM90, SW83] as well.

Chapter 5

Language Classes Defined by Time-bounded Relativised CA with CA Oracles

In the previous chapter, $TVCA$ were interpreted as relativised CA. The focus of study in that chapter was the power of the oracle access mechanism. Separation results were obtained exploiting this mechanism; however, the separating oracles constructed were not CA languages.

In this chapter we will examine the behaviour of the relativised CA language classes, specifically the classes rCA and lCA, when the oracle classes themselves are CA and relativised CA classes. Some such results have already been mentioned in the preceding chapters in the context of time-varying CA; they will be restated here in the context of relativised CA. Besides, we will construct a hierarchy of languages built up from rCA and lCA at the base level and with each level obtained by using the class at the preceding level as the oracle class. We will show some non-trivial interesting properties of this CA hierarchy.

5.1 Relativised CA Language Classes

We first go over some elementary results. The following theorem is a direct consequence of Corollary 3.8, and states that as an oracle class, the class $rOCA\mid_t$ has no effect on the classes rCA, lCA and CA. Thus the problem of whether rCA are properly contained in lCA remains unchanged in this relativised world.

Theorem 5.1
$$\begin{aligned} rCA(rOCA\,|_t) &= rCA \\ lCA(rOCA\,|_t) &= lCA \\ CA(rOCA\,|_t) &= CA \end{aligned}$$

At the other extreme, when the classes $CA\,|_t$ and $OCA\,|_t$ are used as oracles, $rCA\,|_t$ become as powerful as $lCA\,|_t$, as seen in the following theorems which are special cases of Corollary 3.12.

Theorem 5.2 $rCA(CA\,|_t)\,|_t = lCA(CA\,|_t)\,|_t = CA\,|_t$.

Theorem 5.3 $rCA(OCA\,|_t)\,|_t = lCA(OCA\,|_t)\,|_t = OCA\,|_t$.

In these theorems, showing the containments from left to right requires only the oracle to be tally, not the accepted languages. Thus, with minor modifications, we can also show that

Theorem 5.4
$$\begin{aligned} rCA(CA\,|_t) &\subseteq lCA(CA\,|_t) \subseteq CA \subset CA(CA\,|_t) \\ rCA(OCA\,|_t) &\subseteq lCA(OCA\,|_t) \subseteq OCA \end{aligned}$$

Thus for the oracle class below rCA, ie. $rOCA$, relativisation does not alter the $rCA \stackrel{?}{=} lCA$ question. For the oracle classes above lCA, ie. OCA and CA, relativisation merges $rCA\,|_t$ and $lCA\,|_t$. The question naturally occurring at this point is: What happens under relativisation with respect to classes between $rOCA$ and OCA? This motivated us to construct the cellular automata hierarchy. This hierarchy is obtained by repeatedly relativising the classes $rCA\,|_t$ and $lCA\,|_t$, using the previously obtained classes as oracles. In our study, we consider only tally sets. This may seem very restrictive at first, because tally sets are often inadequate in capturing the complexity of various classes. However they sometimes suffice to express strong inter dependencies [Boo74]. For instance, tally sets are present in $NP - P$ if and only if $DEXT \neq NEXT$. Even when only tally sets are considered, the problem $rCA \stackrel{?}{=} lCA$ is open [IJ88]. Though many conjecture that the classes (of tally sets) are distinct, no answer is forthcoming. Book has shown (Theorem 2, [Boo74]) that if every tally language in $DSPACE(n)$ is in P, then $EXPSPACE = EXPTIME$ and every tally language in $PSPACE$ belongs to P. Since lCA languages are in P and since $DSPACE(n) = CA$, we can conclude that if, for tally languages, $lCA = CA$, then $EXPSPACE = EXPTIME$ and every tally language in $PSPACE$ belongs to P. Our work here is based on the conjecture that if the classes rCA, lCA and CA are distinct, then there are tally sets in the difference.

The CA hierarchy is formally defined as follows:

Definition 5.5 *The cellular automata hierarchy ($CAH\mid_t$) of tally languages is the structure formed by the classes $rrCA_k$, $lrCA_k$, $llCA_k$ and $rlCA_k$, for each $k \geq 0$, where*

1. $rrCA_0 = rlCA_0 = rCA\mid_t$
2. $llCA_0 = lrCA_0 = lCA\mid_t$
3. $rrCA_{k+1} = rCA(rrCA_k)\mid_t$
4. $lrCA_{k+1} = lCA(rrCA_k)\mid_t$
5. $llCA_{k+1} = lCA(llCA_k)\mid_t$
6. $rlCA_{k+1} = rCA(llCA_k)\mid_t$

Also, $CAH\mid_t = \bigcup_{k\geq 0}(rrCA_k \cup rlCA_k \cup lrCA_k \cup llCA_k)$.

Some elementary properties of the cellular automata hierarchy are given below.

Proposition 5.6 *(a)* $llCA_0 \subseteq lrCA_1$.

(b) $\forall k \geq 0, llCA_k \subseteq rlCA_{k+1}$.

(c) $\forall k \geq 0,$
$$\begin{aligned} rrCA_k &\subseteq rrCA_{k+1} \\ lrCA_k &\subseteq lrCA_{k+1} \\ rlCA_k &\subseteq rlCA_{k+1} \\ llCA_k &\subseteq llCA_{k+1} \end{aligned}$$

(d) $\forall k \geq 0,$
$$\begin{aligned} rrCA_k &\subseteq lrCA_k \subseteq llCA_k \\ rrCA_k &\subseteq rlCA_k \subseteq llCA_k \end{aligned}$$

Proof: (a), (b), (c): Obvious, because the empty set belongs to all these classes, and because with oracle A, A can be accepted in real time.
(d): This is proved by induction. The assertion is obviously true for $k = 0$. Assume it is true upto $k-1$. Now $rrCA_k$ and $rlCA_k$ are both real time CA, but the oracle set of $rlCA_k$, by the induction hypothesis, contains the oracle set of $rrCA_k$. So $rrCA_k \subseteq rlCA_k$. $rlCA_k$ and $llCA_k$ both have the oracle set $llCA_{k-1}$, but the CA from the class $rlCA_k$ can use only real time, while CA from the class $llCA_k$ can use linear time. So $rlCA_k \subseteq llCA_k$. The other inclusions are similarly shown. Thus the assertions are true for all k. ■

Theorem 5.7 $CAH \mid_t \subseteq (OCA \cap P) \mid_t$.

Proof: By statement (d) of the previous proposition, it suffices to show that $\forall k$, $llCA_k \subseteq (OCA \cap P) \mid_t$. This is shown by induction. $llCA_0 = lCA \mid_t$ is clearly in the class $(OCA \cap P) \mid_t$. Let $llCA_{k-1}$ be in $(OCA \cap P) \mid_t$. Then $llCA_k$ is contained in the class $lCA(OCA \mid_t) \mid_t$, which by Theorem 5.4 is contained in $OCA \mid_t$. Also, $llCA_k$ is contained in the class $lCA(P \mid_t) \mid_t$, which can be easily seen to be contained in $P(P) \mid_t = P \mid_t$. ■

This result, along with Book's results [Boo74], immediately yields the following corollary:

Corollary 5.8 *If $CAH \mid_t = CA \mid_t$, then $EXPSPACE = EXPTIME$ and every tally language in $PSPACE$ belongs to P.*

This suggests that while the power of the class lCA may be increased somewhat due to repeated relativisations with respect to previously obtained classes, it is unlikely to increase sufficiently to equal the class $CA \mid_t$, or even $OCA \mid_t$.

The following theorem is mentioned in this section essentially for completeness; the actual proof is provided only in the next section.

Theorem 5.9 *If $rrCA_0 = llCA_0$, then $\forall k$, $rrCA_k = rlCA_k = lrCA_k = llCA_k = rCA \mid_t$. Consequently, $CAH \mid_t = rCA \mid_t$.*

This theorem says that if the classes $rCA \mid_t$ and $lCA \mid_t$ are equal, then for tally sets, linear time can be brought down to real time even in the presence of any oracle from $CAH \mid_t$. Consequently, the entire hierarchy collapses.

5.2 The Structure of the Cellular Automata Hierarchy

In this section we show some interesting inclusions in the cellular automata hierarchy. We first need some preliminary results.

Lemma 5.10 *Let L be a (tally) language accepted by rCA C. We can effectively construct CAs C' and C'' which, on input O^n, do the following:*

(a) At time step i, the accepting cell of C' specifies whether or not $O^i \in L$.

(b) At time step $2i-1$, the accepting cell of C'' specifies whether or not $O^i \in L$.

Proof: (a) Consider the time-space unrolling of C. In this diagram, the unrollings of C on inputs 0^i and 0^{i+1} differ only in the i^{th} diagonal from right to left. So we can construct C' so that each cell stores the corresponding values in the unrollings of two input lengths. This allows C to be simulated on all input lengths. An example is shown in Figure 5.1.

More specifically, let $c^n(i,t)$ ($\overline{c}^n(i,t)$) denote the state of the i^{th} cell of C (C'), on input 0^n, at time t. Then $\overline{c}^n(i,t)$ contains both $c^n(i,t)$ and $c^{i+t-1}(i,t)$. $\overline{c}^n(1,t)$ will now contain $c^t(1,t)$ as the second component of its state for $t < n$, denoting membership of 0^t in L, and at $t = n$ it will contain $[c^n(1,n), \$]$. To achieve this, let δ be the transition function of C. Then h, the transition function of C', is given by the following rules. The first four rules give the transitions at $t = 1$ and the other rules are used at subsequent steps.

$$\begin{aligned}
h(\#,0,\#) &= [\delta(\#,0,\#), \$] \\
h(\#,0,0) &= [\delta(\#,0,0), \delta(\#,0,\#)] \\
h(0,0,0) &= [\delta(0,0,0), \delta(0,0,\#)] \\
h(0,0,\#) &= [\delta(0,0,\#), \$] \\
h(\#,[c,d],[e,f]) &= [\delta(\#,c,e), \delta(\#,c,f)] \\
h(\#,[c,d],[e,\$]) &= [\delta(\#,c,e), \$] \\
h([a,b],[c,d],[e,f]) &= [\delta(a,c,e), \delta(a,c,f)] \\
h([a,b],[c,d],[e,\$]) &= [\delta(a,c,e), \$]
\end{aligned}$$

For arguments where h is not specified above, h maps to some don't-care state D.

#	0	0	0	0	0	#
#	a	b	b	b	c	
#	d	e	f	g		
#	h	i	j			
#	k	l				
#	$\mathbf{m}$					

#	0	0	0	0	#
#	a	b	b	c	
#	d	e	g		
#	h	n			
#	$\mathbf{p}$				

#	0	0	0	#
#	a	b	c	
#	d	q		
#	$\mathbf{r}$			

#	0	0	#
#	a	c	
#	$\mathbf{s}$		

#	0	#
#	$\mathbf{t}$	

CA C on inputs 0^5, 0^4, $\ldots$, 0^1

#	0	0	0	0	0	#
#	$a\mathbf{t}$	bc	bc	bc	$c\$$	
#	$d\mathbf{s}$	eq	fg	$g\$$		
#	$h\mathbf{r}$	in	$j\$$			
#	$k\mathbf{p}$	$l\$$				
#	$\mathbf{m}\$$					

CA C' simulating C

Figure 5.1: Simulating a CA on all prefixes of the input in real time

(b) C'' is merely a half-speed version of C'. ■

Note that in part (b), if the machine C'' is treated as a real-time machine (ie. for inputs of length n, the state of the accepting cell at time n is checked), then C'' accepts the language $2L-1$, defined in chapter 3. Slowing it down by one step would allow acceptance of the language $2L$. Thus through this lemma we have also proved that $rCA\mid_t$ are closed under doubling, a result mentioned without proof in Lemma 3.18.

Theorem 5.11 $rrCA_1 \subseteq llCA_0$.

Proof: Let L be an $rrCA_1$ language accepted by an rCA C_1 with oracle L', where L' is accepted by an rCA C_2. The machines of Lemma 5.10 can be used to find responses to all the oracle queries made by C_1. These responses must then be propagated down the array. This involves a delay; so C_2'' rather than C_2' is used. Thus at time $2i-1$, the response to the i^{th} oracle query is available at the leftmost cell. So the i^{th} transition of the leftmost cell of C_1 is also implemented at this cell now. Simultaneously, the oracle response is sent right at unit speed, so that the j^{th} cell implements the i^{th} transition step of C_1 at time $2i-1+j-1$. It is easily verified that at this time, if each cell stores the current and the previous value of the corresponding cell in the simulation of C_1, the arguments to the transition function are indeed available in the cell and its neighbours. An example is shown in Figure 5.2. The oracle queries are answered by C_2'', a half-speed version of the CA C' shown in Figure 5.1. For the behaviour of C_1 as in Figure 5.2 (a), the simulating CA functions as in Figure 5.2 (b), recognising the input in time $2n-1$.

Formally, the transition function of such a CA can be specified in terms of those of C_1 and C_2'' as follows.

Let $C_1 = (Q_1, \#, \delta_1, \delta_2, L', F_1)$ and $C_2'' = (Q'', \#, \delta, F'')$. Define a CA $C = (Q, \#, h, F)$ where

$$Q = \{[u, v, w, x] \mid u, v \in Q_1, w \in Q'' \text{ and } x \in Q'' \cup \{?\}\}$$

u and v hold the old and current states in the simulation of C_1. w holds the state in the simulation of C_2'' and is updated at each time step. The value of w in the leftmost cell is the response to the oracle query, and is propagated right at unit speed in x. When x holds a ?, u and v are not updated. When it holds a state from Q'', then v is stored in u and v is

#	0	0	0	0	0	#
#	A	B	B	B	C	
#	D	E	F	G		
#	H	I	J			
#	K	L				
#	M					

(a) $rrCA_1$ C on input length 5, with the oracle from Figure 5.1.

#	0	0	0	0	0
#	$0 \mid A$ / $at \mid at$	$0 \mid 0$ / $bc \mid ?$	$0 \mid 0$ / $bc \mid ?$	$0 \mid 0$ / $bc \mid ?$	$0 \mid 0$ / $c\$ \mid ?$
#	$0 \mid A$ / $at \mid ?$	$0 \mid B$ / $bc \mid at$	$0 \mid 0$ / $bc \mid ?$	$0 \mid 0$ / $bc \mid ?$	$0 \mid 0$ / $c\$ \mid ?$
#	$A \mid D$ / $ds \mid ds$	$0 \mid B$ / $eq \mid ?$	$0 \mid B$ / $fg \mid at$	$0 \mid 0$ / $g\$ \mid ?$	$0 \mid 0$ / $\ \mid ?$
#	$A \mid D$ / $ds \mid ?$	$B \mid E$ / $eq \mid ds$	$0 \mid B$ / $fg \mid ?$	$0 \mid B$ / $g\$ \mid at$	$0 \mid 0$ / $\ \mid ?$
#	$D \mid H$ / $hr \mid hr$	$B \mid E$ / $in \mid ?$	$B \mid F$ / $j\$ \mid ds$	$0 \mid B$ / $\ \mid ?$	$0 \mid C$ / $\ \mid at$
#	$D \mid H$ / $hr \mid ?$	$E \mid I$ / $in \mid hr$	$B \mid F$ / $j\$ \mid ?$	$B \mid G$ / $\ \mid ds$	$0 \mid C$ / $\ \mid ?$
#	$H \mid K$ / $kp \mid kp$	$E \mid I$ / $l\$ \mid ?$	$F \mid J$ / $\ \mid hr$	$B \mid G$ / $\ \mid ?$	$C \mid \ $ / $\ \mid ds$
#	$H \mid K$ / $kp \mid ?$	$I \mid L$ / $l\$ \mid kp$	$F \mid J$ / $\ \mid ?$	$G \mid \ $ / $\ \mid hr$	$C \mid \ $ / $\ \mid ?$
#	$K \mid M$ / $m\$ \mid m\$$	$I \mid L$ / $\ \mid ?$	$J \mid \ $ / $\ \mid kp$	$G \mid \ $ / $\ \mid ?$	$\ \mid \ $ / $\ \mid hr$

(b) The simulating CA

Figure 5.2: Simulating an $rrCA_1$ by an $llCA_0$

updated as in C_1, using the previous state u from the left neighbour and the current states v from the cell and its right neighbour.

$$F = \{[u, v, w, x] \mid u \in Q_1, v \in F_1, w \in Q'' \text{ and } x \in Q'' \cup \{?\}\}$$

$$\begin{array}{lll}
h(\#, 0, z) = & [0, A, B, C] \text{ for } z = 0 \text{ or } \#, \text{ where} & \\
& B = \delta(\#, 0, z),\ C = B, \text{ and if } C \in F'' & \\
& & \text{then } A = \delta_1(\#, 0, z) \\
& & \text{else } A = \delta_2(\#, 0, z). \\
h(0, 0, z) = & [0, 0, \delta(0, 0, z), ?] \text{ for } z = 0 \text{ or } \#. & \\
h(\#, b, c) = & [A, B, C, D] \text{ for 4-tuples } b \text{ and } c, \text{ where} & \\
& C = \delta(\#, b_3, c_3), \text{ and} & \\
& \text{if } b_4 \neq ? & \text{then } A = b_1,\ B = b_2,\ D = ? \\
& & \text{else } A = b_2,\ D = C, \text{ if } C \in F'' \\
& & \qquad \text{then } B = \delta_1(\#, b_2, c_2) \\
& & \qquad \text{else } B = \delta_2(\#, b_2, c_2). \\
h(a, b, c) = & [A, B, C, D] \text{ for 4-tuples } a, b, c, \text{ where} & \\
& C = \delta(a_3, b_3, c_3), \text{ and} & \\
& \text{if } a_4 = ? & \text{then } A = b_1,\ B = b_2,\ D = ? \\
& & \text{else } A = b_2,\ D = a_4, \text{ if } D \in F'' \\
& & \qquad \text{then } B = \delta_1(a_1, b_2, c_2) \\
& & \qquad \text{else } B = \delta_2(a_1, b_2, c_2).
\end{array}$$

(If $c = \#$, then we take c_i to be $\#$ for $i = 1$ to 4.) ■

Theorem 5.12 $lrCA_1 = llCA_0$.

Proof: $llCA_0 \subseteq lrCA_1$ follows from Proposition 5.6 (a). $lrCA_1 \subseteq llCA_0$ can be shown as above, packing c cells of C_2'' together to simulate C_2'' on input 0^{cn} within an n length array. ■

Note that in the above proofs, the crucial point is that the oracle classes contain only tally sets. The accepted language itself need not be tally; thus we can also conclude that

$rCA(rCA\mid_t)$ is contained in lCA which is equal to $lCA(rCA\mid_t)$. In other words, $rCA\mid_t$ is useless as an oracle class if the CA is allowed even as much as linear time. These proofs easily generalise to k controlling languages and thus prove Theorems 3.16 and 3.17.

Theorem 5.9 of the previous section now follows from the above two theorems, by a simple inductive argument.

Proof of Theorem 5.9: We know that $rrCA_0 \subseteq rrCA_1 \subseteq llCA_0$. So if $rrCA_0 = llCA_0$, then $rrCA_0 = rrCA_1$. Let $rrCA_k = rrCA_0$. $rrCA_{k+1}$ is the class of languages accepted by rCA using oracles from $rrCA_k$, ie. oracles from $rrCA_0$, and so equals the class $rrCA_1$. But this is equal to $rrCA_0$, under our assumption. So $rrCA_{k+1} = rrCA_0$. Thus by induction, $\forall k$, $rrCA_k = rrCA_0$. From the definition, it then follows that $\forall k$, $lrCA_k = lrCA_1$ which equals $rrCA_0$ by assumption. The other classes are similarly shown to be equal to $rrCA_0$. Thus if $rrCA_0 = llCA_0$, then the CAH collapses to the smallest class $rrCA_0 = rCA\mid_t$. ■

We now show that the results of Lemma 5.10 and Theorems 5.11 and 5.12 'translate upwards'; they also hold at higher levels of the CA hierarchy. The following lemma essentially states that Lemma 5.10 (b) relativises if the oracle classes are $rrCA_k$ classes. Thus all $rrCA_k$ classes are closed under doubling.

Lemma 5.13 *If $L \in rrCA_k$, then $2L-1 = \{0^{2i-1} \mid 0^i \in L\} \in rrCA_k$.*

Proof: Consider $L \in rrCA_0$. Let L be accepted by rCA C. On input 0^{2m-1}, simulate machine C'' described in Lemma 5.10, and also send a signal $\mathbf{S}$ at unit speed from the rightmost cell to the left. $\mathbf{S}$ reaches the accept cell when it is specifying membership of 0^m in L. So C'' will accept (reject) 0^{2m-1} if $0^m \in L$ ($0^m \notin L$), in time $2m-1$, ie. in real time. So C'' is an rCA accepting $2L-1$.

Assume that the statement of the lemma is true for k. Let $L \in rrCA_{k+1}$. L is accepted by an rCA C using oracle $L' \in rrCA_k$. By our assumption, $2L'-1 \in rrCA_k$. Construct C'' as above, using oracle $2L'-1$. The resulting CA accepts $2L-1$ in real time; hence $2L-1 \in rrCA_{k+1}$. So the statement of the lemma is also true for $k+1$.

Thus by induction, the statement is true for all k. ■

In Theorem 5.11, we are essentially proving that $rrCA_1 \subseteq lrCA_0$. Using the above lemma, this generalises as follows.

Theorem 5.14 *For $k \geq 0$, $rrCA_{k+1} \subseteq lrCA_k$.*

Proof: For $k = 0$, this is proved in Theorem 5.11. Consider $k > 0$. Let C be an $rrCA_{k+1}$ CA using oracle L. $L \in rrCA_k$, so L is accepted by an rCA using oracle $L' \in rrCA_{k-1}$. Now $2L' - 1$ is also in $rrCA_{k-1}$. A CA using oracle $2L' - 1$ can accept the same language as C, in linear time, as described in Theorem 5.11. But this lCA uses an oracle from $rrCA_{k-1}$, and hence will belong to $lrCA_k$. Hence the theorem. ■

Similarly, reading Theorem 5.12 as $lrCA_1 = lrCA_0$ and translating it upwards in an identical fashion, we get

Theorem 5.15 *For $k \geq 0$, $lrCA_{k+1} \subseteq lrCA_k$.*

This, along with Proposition 5.6, immediately yields

Corollary 5.16 $\forall k \geq 0$, $lrCA_k = lCA\mid_t$.

This corollary clearly generalises Theorem 5.12; not only are $rCA\mid_t$ (ie. $rrCA_0$) languages useless as oracles for the class lCA, but so are all languages in the classes $rrCA_k$, for any k. Thus the $rrCA_k$ languages seem to be quite limited in their power.

Now we can combine all these known results to obtain an overall picture of the CA hierarchy. The structure of the cellular automata hierarchy is as shown in Figure 5.3.

The following series of propositions shows how this structure changes under various assumptions of equality of certain classes.

Proposition 5.17 *If $rrCA_0 = rrCA_1$, then all the $rrCA$ classes are equal.*

Proof: Obvious, as seen in proof of Theorem 5.9. ■

Proposition 5.18 *$rrCA_1 = lrCA_1$ if and only if $rrCA_1 = rlCA_1$. In this case, the structure in Figure 5.4 results.*

Proof: From Figure 5.3, it is obvious that $rrCA_1 = rlCA_1$ implies $rrCA_1 = lrCA_1$. Assume that $rrCA_1 = lrCA_1$. From Figure 5.3, we see that this implies $rrCA_1 = rrCA_2 = llCA_0$. So $rlCA_1 = rCA(llCA_0)\mid_t = rCA(rrCA_1) \mid t = rrCA_2 = rrCA_1$. Further, since under this assumption we have $rrCA_2 = rrCA_1$, it is clear that $\forall k > 0$, $rrCA_k = rrCA_1$. ■

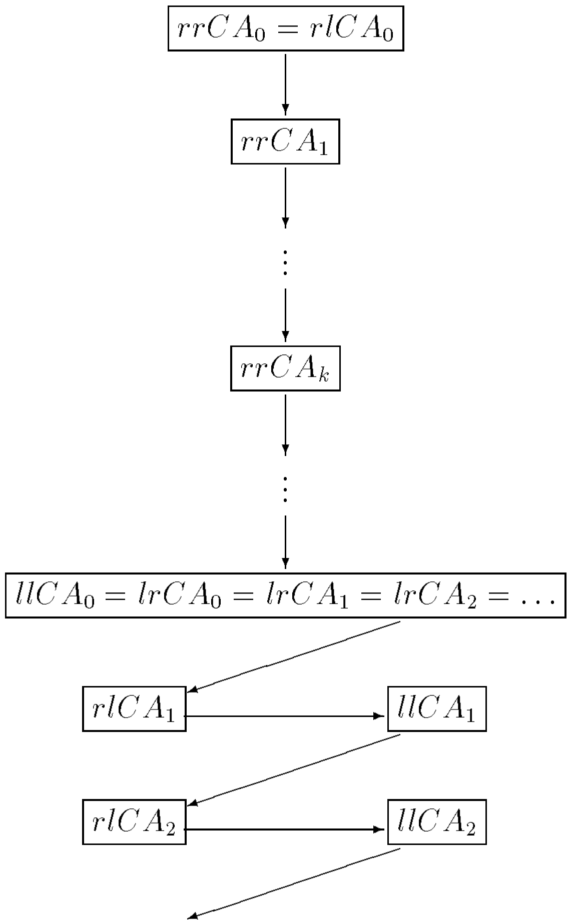


Figure 5.3: The structure of the CAH

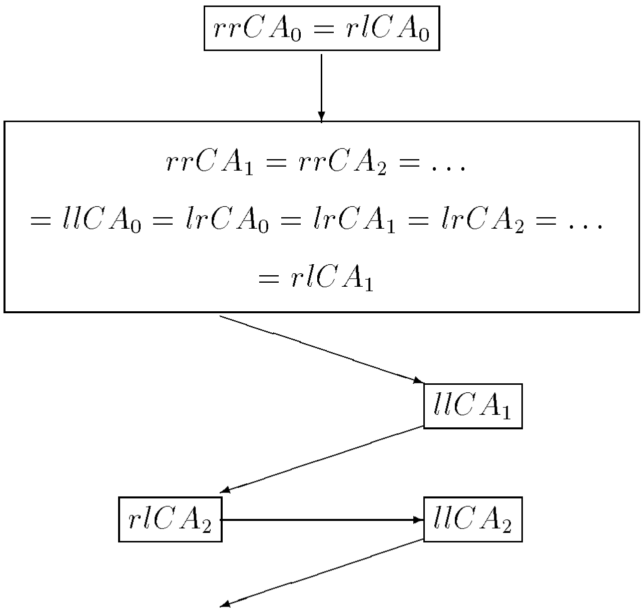


Figure 5.4: The CAH, assuming $rrCA_1 = lrCA_1$

$rrCA_1 = rlCA_1$ means that $rCA\ |_t$ and $lCA\ |_t$ as oracles add equally to the class $rCA\ |_t$. $rrCA_1 = lrCA_1$ means that $rCA\ |_t$ and $lCA\ |_t$ coincide relative to the class of oracles $rCA\ |_t$. Equivalently, since $lCA(rCA\ |_t) = lCA$, this also means that with an rCA oracle, the class $rCA\ |_t$ rises up to equal $lCA\ |_t$. These equalities imply each other and also imply that the $rrCA_k$ classes are not distinct for $k > 0$.

Proposition 5.19 *If $rrCA_1 = llCA_1$, then the cellular automata hierarchy has only two distinct classes: $rCA\ |_t = rrCA_0 = rlCA_0$, and $lCA\ |_t$, which is equal to all the remaining classes.*

Proof: $rrCA_1 = llCA_1$ clearly implies $rrCA_1 = lrCA_1$. So from the above proposition we immediately conclude that $\forall k \geq 1$, $rrCA_k = rrCA_1$. Further, since $llCA_1 = llCA_0$, $\forall k \geq 0$ $llCA_k = llCA_0$. This also implies that $\forall k \geq 1$, $rlCA_k = llCA_0$. Thus $rrCA_0$ and $rlCA_0$ are identical, and all other classes are identical; the CAH has at most two distinct classes. ■

Proposition 5.20 *$lrCA_1 = llCA_1$ if and only if $llCA_0 = llCA_1$. In this case, the CAH has the structure shown in Figure 5.5.*

$$
\begin{array}{c}
\boxed{rrCA_0 = rlCA_0} \\
\downarrow \\
\boxed{rrCA_1} \\
\downarrow \\
\vdots \\
\downarrow \\
\boxed{rrCA_k} \\
\downarrow \\
\vdots \\
\downarrow \\
\boxed{\begin{array}{c} lrCA_0 = lrCA_1 = lrCA_2 = \ldots \\ = rlCA_1 = rlCA_2 = \ldots \\ = llCA_0 = llCA_1 = llCA_2 = \ldots \end{array}}
\end{array}
$$

Figure 5.5: The CAH, assuming $lrCA_1 = llCA_1$

Proof: Obvious. ■

A point worth examining is whether proper containments translate upwards. Equalities do; we have, by a straightforward argument,

$$rrCA_k = rrCA_{k+1} \Leftrightarrow \forall m > k,\ rrCA_m = rrCA_k$$

$$llCA_k = llCA_{k+1} \Leftrightarrow \forall m > k,\ llCA_m = llCA_k$$

5.3 Conclusions

This work attempts to study the structure of the tally languages, if any, separating CA, lCA and rCA. We have also restricted the language classes CA, lCA and rCA to tally sets. A similar hierarchy of CA language classes can be constructed if non-tally languages are considered for acceptance and as oracles. It is easily verified that Proposition 5.6 (a), (c), (d) continue to hold. (b) does not appear to, because, as mentioned in section 4.1, once we consider non-tally oracles, the containment $A \in lCA(A)$ does not necessarily hold. ($A \in CA(A)$ does hold, from Proposition 4.6, since on input $x = \langle m \rangle$, m is CA-time-constructible; refer Proposition 3.13.) A straightforward algorithm for recognising an $llCA_1$ language (non-tally oracle) by a CA requires $O(n \log n)$ time, while for tally oracles such an algorithm runs in $O(n^2)$ time. However, even for $rrCA_1$ languages with non-tally oracles we have been unable to improve the $O(n \log n)$ upper bound, whereas $rrCA_1$ languages with tally oracles can be accepted by lCA; refer Theorem 5.11. Of course, such differences are to be expected, since tally sets are very low in information content.

An unanswered question is whether or not rCA and lCA coincide over unary alphabets. If this is the case, then Theorem 5.9 states that the CA hierarchy collapses. In [IJ88] it is conjectured that these classes do not coincide. Our work is motivated by a weaker conjecture — namely, that if the classes rCA and lCA are distinct, then there are tally sets in the difference.

Another aspect which deserves more study is finding languages complete for CA and lCA, where completeness will have to be suitably defined. Such complete languages may admit a relativisation under which the classes rCA, lCA and CA coincide. This, along with

Theorem 4.9, will provide a contradictory relativisation of these problems, but may also provide more information about the nature of the CA complexity classes.

Chapter 6

Nondeterministic Computation on Cellular Array Models

In this chapter, we consider a new model of nondeterminism based on the structure of time-varying automata, and, imposing this model upon CA and OCA, investigate the power of the resulting classes. The notion of a time-varying CA ($TVCA$) was introduced in chapter 3, and in chapters 4 and 5 $TVCA$ were interpreted as relativised CA, ie. CA which compute with some help from an oracle. In this paper we further generalise this notion, and interpret the computation of a $TVCA$ as a nondeterministic computation. The description and definition of this model is presented in section 6.1. In section 6.2, we compare this form of nondeterminism with the traditional notion, and try to place our nondeterministic CA classes amid the traditionally defined classes. In section 6.3, some restricted forms of nondeterministic $TVCA$ computation are studied, with the intention of trying to identify how much nondeterminism is required, if at all, to enhance the power of a particular class. In section 6.4, some closure properties of nondeterministic CA classes are studied.

6.1 Preliminaries and Definitions

In a $TVCA$, the transition function to be applied to each cell depends not only on the states of cells in the neighbourhood but also on the number of time steps elapsed since the CA operation began. The dependence on time is expressed in the following way: a set of transition functions $\delta_1, \delta_2, \ldots, \delta_k$ is associated with the CA, and δ, the effective transition function of the CA, agrees with one of $\delta_1, \delta_2, \ldots, \delta_k$ depending on the time. In other words,

$$\delta(a, b, c, i) = \delta_{f_i}(a, b, c)$$

where $a, b, c \in Q$, $i \in \mathbf{N}$, and δ_{f_i} is the transition function used at time $t = i$. The manner in which f_i is chosen thus crucially affects the overall computation.

One important fact to note about $TVCA$ is that speed-up does not necessarily hold. Neither Lemma 2.7 nor Lemma 2.8 can be shown to trivially apply to $TVCA$. Since we are essentially interested in the dependence of running time on input length, we will still continue to ignore additive constants, and treat $(T(n)+c)$-time as equivalent to $T(n)$-time. However for multiplicative constants, there is a trade-off, as described in section 3.3. To be more precise, consider speeding up the operation of a k-$TVCA$ by a factor of 2. Even assuming that an initial phase achieves the required packing of the input, to be able to simulate two steps of the $TVCA$ in one step calls for the ability to simulate k^2 different combinations of the form $\delta_i\delta_j$. So the simulating $TVCA$ will need k^2 different transition functions. Thus speed-up is achieved at the cost of the number of functions required. Conversely, the number of functions can be reduced at the expense of slowing down the computation — a k-$TVCA$ operating in $T(n)$ time can be simulated by a 2-$TVCA$ operating in $(\log_2 k)T(n)$ time (refer Theorem 3.19). Since the slowing down is only by a constant factor, for (linear-time) $TVCA$ it is sufficient to consider 2-$TVCA$. But for real-time computation, it appears that k is a crucial parameter; whether $k+1$ functions are better than k for real-time $TVCA$ is an open problem raised in that chapter.

In chapters 4 and 5, 2-$TVCA$ have been interpreted as relativised CA. A tally language $L \subseteq 0^*$ is the oracle, and δ is now expressed as follows:

$$\delta(a, b, c, i) = \begin{cases} \delta_1(a, b, c) \text{ if } 0^i \in L \\ \delta_2(a, b, c) \text{ otherwise} \end{cases}$$

Note that for a 2-$TVCA$ operating in time $T(n)$, there are $2^{T(n)}$ possible computation paths, and the structure of L determines which of these paths is chosen. In the preceding chapters, we have examined how varying the complexity of the oracle L affects the computational power of the $TVCA$.

In this chapter we relax the notion of a single computation path being checked for acceptance. First we define the characteristic bit strings of a language and of a $TVCA$ computation path as follows:

Definition 6.1 *The characteristic bit string of a language $L \subseteq \Sigma^*$ is a bit string $a_0a_1a_2\dots$ where each $a_i \in \{0,1\}$, and for the standard enumeration of Σ^*, $a_i = 0$ if and only if the i^{th} word of Σ^*, w_i, is in L.*

Thus for a tally language L, $a_i = 0$ if and only if $0^i \in L$.

For a 2-$TVCA$, on input w of length n, a $T(n)$-time computation path is a sequence $w = w_0, w_1, \dots, w_{T(n)}$ where for each i, $|w_i| = |w_0|$, and for $i > 0$, w_i can be obtained from w_{i-1} by applying either δ_1 or δ_2. It is an accepting computation if $w_{T(n)}$ is an accepting configuration, ie. the leftmost state is an accepting state.

Definition 6.2 *The characteristic bit string of a $T(n)$-time computation path is a $T(n)$-length bit string $b_1b_2\dots b_{T(n)}$ where $b_i = 0$ if w_i can be obtained from w_{i-1} by applying δ_1, and $b_i = 1$ otherwise.*

Note that this definition assigns a unique bit string as the characteristic bit string for a given computation. However, more than one bit string may still determine the same computation. This could happen if, from a particular configuration, both δ_1 and δ_2 lead to the same next configuration. The unique characteristic bit string is that bit string which uses δ_1 wherever possible and thus, when interpreted as an integer, has least numerical value.

Consider Figure 6.1, a binary tree. The root node holds w_0. The left (right) child of a node holding c holds the configuration obtained by applying δ_1 (δ_2) to c. This binary tree, of height $T(n)$, gives all possible computations of a $T(n)$-time 2-$TVCA$ on input w_0. A bit string of length $T(n)$ picks out a particular path in this tree. In a relativised CA operation, the unique computation path whose characteristic bit string is a prefix of the characteristic bit string of the oracle is picked, and the input is accepted if and only if this computation path ends in an accepting configuration. Instead, we can check whether at all there exists an accepting computation, thus giving a nondeterministic interpretation to the $TVCA$. This notion is formalised below.

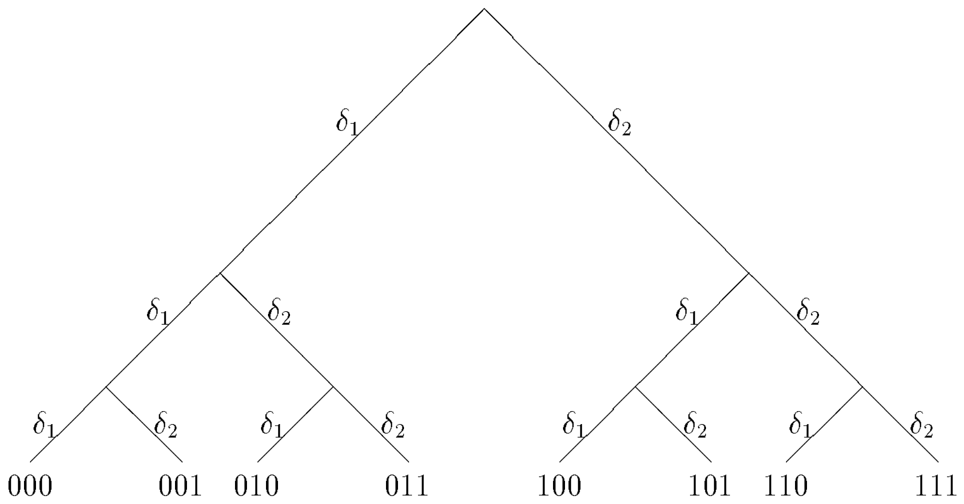


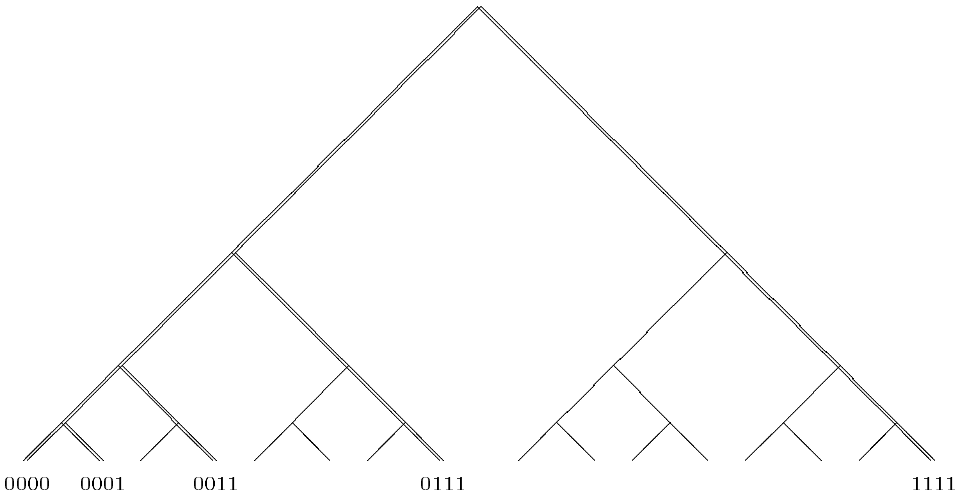
Figure 6.1: Binary tree of possible computations and characteristic bit strings

Definition 6.3 *A nondeterministic TVCA (NTVCA) is a construct* $C = (Q, \#, \delta_1, \delta_2, A)$ *defined as a 2-TVCA. A string* w *is accepted by* C *in time* $T(n)$ *if* $\exists \alpha \in \{0,1\}^{T(|w|)}$ *such that the computation path of* C *beginning with* w *and with characteristic bit string* α *is an accepting computation.*

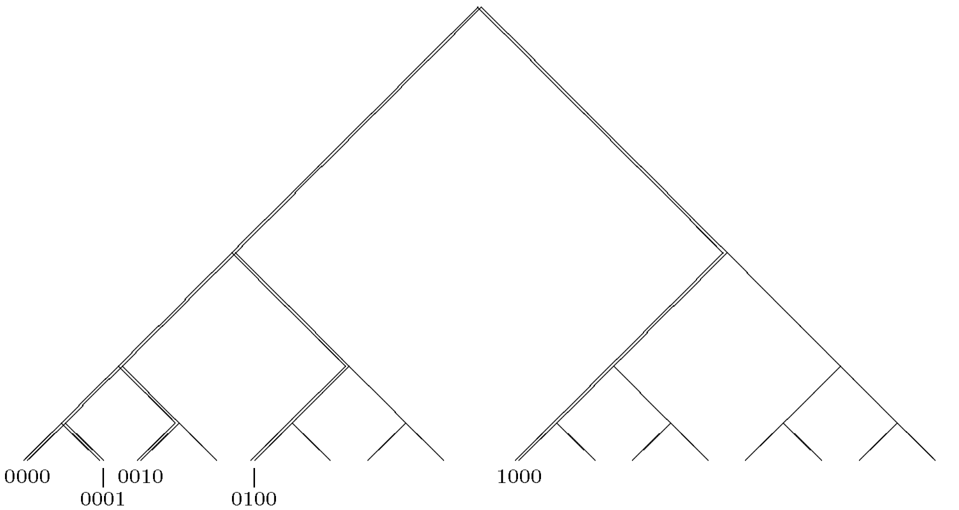
For a $T(n)$-time computation, an $NTVCA$ as defined above looks at all the $2^{T(n)}$ computation paths. We can define restricted versions, where only certain computation paths, whose characteristic bit strings possess some special properties, are of interest. This takes us closer to relativised CA, where exactly one computation path is of interest. Two such restrictions are defined below.

Definition 6.4 *A 1-turn (1-kink) NTVCA is a TVCA* C *which accepts input* w *if and only if there is an accepting computation of* C *on* w*, with a characteristic bit string of the form* 0^*1^* *(* $0^*(\epsilon + 10^*)$ *).*

A 1-turn $NTVCA$ uses only δ_1 for some time and then switches over to using only δ_2. The nondeterminism is in deciding when to switch from δ_1 to δ_2. So for a $T(n)$-time 1-turn $NTVCA$, there are $T(n) + 1$ computation paths of interest. A 1-kink $NTVCA$ can use δ_2 at most once; again, for a $T(n)$-time 1-kink $NTVCA$, there are $T(n) + 1$ computation paths of interest. These paths are shown in Figure 6.2.



1-turn paths



1-kink paths

Figure 6.2: Restricted nondeterminism computation paths

Definitions 6.2 and 6.3 can be generalised to k-$TVCA$ as well. The different computation paths of the $TVCA$ can now be represented in a k-ary tree, and the characteristic string for a specific computation path will be a string over an alphabet of size k.

6.2 Nondeterministic Computation on TVCA

In this section we examine some of the results concerning NCA and see which of these hold for the new mode of nondeterminism defined in section 6.1. Even seemingly trivial results need to be re-examined, largely because speed-up does not hold for 2-$TVCA$. In what follows, we assume that the $TVCA$ are 2-$TVCA$, unless otherwise stated. First we consider the unbounded-time classes of $NTVCA$.

Lemma 6.5 *$NTVCA \subseteq NSPACE(n)$; a $T(n)$-time $NTVCA$ can be simulated by an $NSPACE(n)$ machine in $O(nT(n))$ time.*

Proof: A $T(n)$-time $NTVCA$ can be simulated in $O(nT(n))$ time by an $NSPACE(n)$ machine which simulates one step of the $NTVCA$ as follows. It first decides, nondeterministically, whether to use δ_1 or δ_2, and then moves down the entire array, deterministically updating the state of each cell accordingly. Clearly, real space suffices for such a simulation, and the $NSPACE(n)$ machine needs $O(nT(n))$ time (n steps for each step of the $NTVCA$).

■

Lemma 6.6 *$NOCA \subseteq NTVOCA$; an $NTVOCA$ can simulate a $T(n)$-time $NOCA$ in $O(nT(n))$ time.*

Proof: Let $C = (Q, \#, \delta, A)$ be an $NOCA$, where δ maps $Q \times Q$ to subsets of Q. Let k be the size of the largest subset of Q in the range of δ. We will construct a $(k+1)$-$NTVOCA$ C' accepting the same language as C. Then, as described in section 6.1, an equivalent 2-function $NTVOCA$ can be constructed.

Each cell of C can independently choose one of upto k options when making a transition according to δ. But in C', at a single time step, all cells must use the the same option. So to simulate the n independent choices made by C in one step on an n length input, C' needs n

steps, where at each step exactly one cell of C' makes a transition and all other cells merely maintain their state. Now the first k distinct transition functions of C' can implement the k options provided by δ. The leftmost cell of C' sends a pulse right at unit speed. As this pulse passes through a cell, that cell makes a state transition. When the pulse reaches the right end, all cells have updated their states and one step of C has been simulated. One row in the time-space unrolling of C appears as a diagonal in the time-space unrolling of C'.

The problem which now arises is that the leftmost cell does not know when to send out the next pulse. Pulses should be at least n time steps apart, but, in the absence of two-way communication, counting upto n is not possible. So the correct spacing of pulses has to be guaranteed separately. The leftmost cell sends a pulse whenever δ_{k+1} is used. If at this time the previous pulse has not reached the right end, this can be detected by a cell which still has the travelling pulse. This cell will now put the OCA into a rejecting configuration. Otherwise, simulation of the next row in the time-space diagram of C begins with this pulse. If the pulses are more than n steps apart, then in between there will be some idle steps, when C' does nothing. However for a $T(n)$-time computation path of C, there will be a computation path of C' where the pulses are exactly n steps apart; this computation path will be of length $nT(n)$. Figure 6.3 shows different spacing of pulses and the resulting CA operation.

With this construction, $T(n)$ steps of the $NOCA$ are simulated by the $NTVOCA$ in $nT(n)$ steps. This $NTVOCA$ can be converted to one having only two transition functions, with a slowing down only by a constant factor. This is the required $NTVOCA$. ■

Note: Strictly speaking, ensuring that the pulses are at least n time steps apart is not necessary. If the pulses overlap, then some cells have to make related choices. But these choices could have been made even if all cells were acting independently. What one needs to ensure is that there are computation paths where the pulses are so spaced, guaranteeing the checking of all possible choices. Then the other paths, with overlapping pulses, are already simulated on some of these paths, and therefore need not be explicitly made rejecting paths. However, we have presented this construction to bring out, more clearly, the step-by-step simulation of the $NOCA$.

From these two lemmas and Lemma 2.10, we can now conclude:

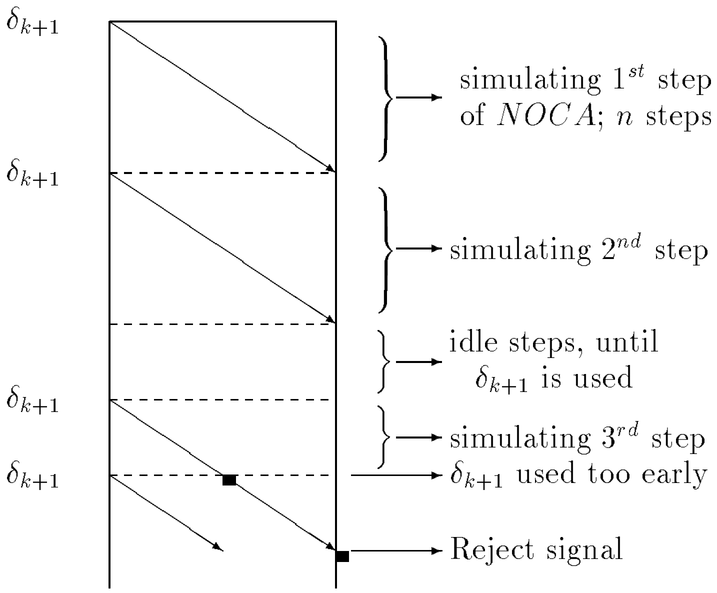


Figure 6.3: An $NTVOCA$ simulating an $NOCA$

Theorem 6.7 $NTVOCA = NTVCA = NSPACE(n)$.

The following lemma further strengthens the statement $NTVCA \subseteq NCA$; it claims that a real-time simulation is possible.

Lemma 6.8 *An $NTVCA$ can be simulated by an NCA with no loss of time. If the $NTVCA$ uses only one-way communication, so does the simulating NCA.*

Proof: In an $NTVCA$, all cells must use the same transition function, at any given time instant. This condition can be enforced in an NCA as follows: Each cell of the NCA nondeterministically uses δ_1 or δ_2 at any time instant. Additionally, each cell also records, in its state, which transition function was used. From time step $t = 2$ onwards, each cell also checks that the cells in its neighbourhood used the same transition function as itself at the previous step. If this is not the case, a reject signal is generated and sent to the accepting cell. Thus if the NCA accepts its input, it must be along a computation path where all cells had used the same transition function at each time instant; ie. it must be along a computation path corresponding to the $NTVCA$. ∎

We now look at the time-bounded $NTVCA$ classes. The next result highlights the difficulty of determining membership for real-time $NTVCA$ languages. A similar result for real-time $NOCA$ also exists [IK84]. However we have not been able to improve our result to real-time $NTVOCA$. We can only conclude, form the NP-completeness of the $rNOCA$ membership problem and from Lemma 6.6, that the membership problem is NP-complete for $NTVOCA$ running in $O(n^2)$ time.

Theorem 6.9 *The class of real-time $NTVCA$ languages contains an NP-complete language.*

Proof: Consider the language of satisfiable Boolean formulas in 3-clause conjunctive normal form 3-$CNFSAT$. Let the formulas be coded as follows:

$$v_1 ¢ v_2 ¢ \ldots ¢ v_m \$ F_1 \wedge F_2 \wedge \ldots \wedge F_k$$

where

- $v_i \in \{0,1\}^*$,
- $|v_i| = |v_j|$ for each i, j,
- $v_i \neq v_j$ for $i \neq j$, and
- each F_i is of the form $w \vee x \vee y$, where w, x and y are of the form $0v_t$ or $1v_t$ for some t.

Thus the formula has a list of variables, coded as equal length bit strings separated by ¢s, followed by a \$, followed by a set of clauses separated by $\wedge$s, where each clause has three terms separated by $\vee$s, and each term is either $0v$, representing the variable v, or $1v$, representing the negation of the variable v, for some variable v.

This language is well known to be NP-complete [BDG88, HU79]. Consider the following $NTVCA$ accepting it. The $NTVCA$ operates in four phases.

Phase 1: A signal **Assign** travels from the leftmost cell upto the \$. As it travels, it assigns a value to each variable encountered. Value 0 is assigned if δ_1 is used and value 1 is assigned otherwise. The assigned value can be stored in the cell holding the leftmost bit of the encoded variable.

Phase 2: From here onwards the operation of the $NTVCA$ is deterministic. In other words, δ_1 and δ_2 differ only in the presence of signal **Assign**; elsewhere they are identical. In the second phase, which begins when **Assign** reaches \$, the string to the left of \$ begins moving right, in a separate channel. Each cell transfers its contents to its right neighbour only after the right neighbour has transferred its own contents further right. Whenever a ¢ (or \$, initially) reaches an $\vee$ or an $\wedge$, it temporarily halts.

Phase 3: When a ¢ or \$ is positioned on an $\vee$ or $\wedge$, the $|v_i|$ cells to its left have the codings of two variables in their two channels. The cell containing ¢ or \$ sends a signal left to check if the codings match. If they do, then the value assigned to the variable in the second channel is copied to the first channel (its negation is copied if the code in the first channel is preceded by 1), and the signal returns to the cell from where it originated. If the codings don't match, the signal returns directly, without doing anything. When the signal returns to its originating cell, the string in the second channel resumes its rightward movement.

Phase 4: When a left boundary marker reaches the rightmost cell, all variables in each F_i have been assigned values. Now an **Evaluate** signal moves leftwards, checking whether this assignment satisfies the formula.

Checking the syntax of the input can be done alongwith these phases.

It can be easily verified that if $|v_i| = p$, then the total time required is linear in p, m and k. (Phase 1 needs pm time. Phase 2 and 3 together need $3p + 4$ time for each variable occurrence after the \$ — there are $3k$ such occurrences — plus an additional pm steps in the staggered rightwards movement. Phase 4 needs as many steps as the length of the input.) The length of the input, n, is $m(p + 1) + k(3p + 2) + k - 1$. Clearly, the total time required is linear in n.

Let the $NTVCA$ accepting this language operate in cn time. Then consider the language $L' = \{x\$^{(c-1)|x|} \mid x \in L\}$. Using the above algorithm and ignoring the trailing \$s, L' can be accepted by a real-time $NTVCA$. But L' is also NP-complete, since it is a simple padded version of L. Hence the theorem. ■

We next look at the relationship $lOCA = rCA$ (refer Theorem 2.9), in the context of nondeterminism. For the traditionally defined nondeterministic classes, the equality $rCA = lOCA$ continues to hold, since the speed-up of $lOCA$ to $2n$-time and the simulation of an rCA by a $2n$-time $lOCA$ and vice versa are not affected by nondeterminism in the transition function. This is not the case for $NTVCA$. In fact, an equivalent result does not seem to hold, but we have a restricted version.

Theorem 6.10 *The class of $NTVOCA$ C running in time $2n$ and satisfying the condition*

$$\forall x \in \Sigma^* \, (x \in L(C) \Leftrightarrow \exists \textit{ a computation path accepting } x, \textit{ whose characteristic bit string}$$

$$a_1 a_2 \ldots a_{2|x|} \textit{ has } a_{2i-1} = a_{2i} \textit{ for } i = 1 \textit{ to } |x|\Big)$$

is exactly equivalent to the class of real-time $NTVCA$.

Proof: The condition imposed on the accepting computation paths simply means that it suffices to check computation paths where each transition function is always used for two consecutive instants. Though $2n$ steps are allowed for the computation, effectively only n steps are allowed for the nondeterministic choice. Under such conditions, the equivalence of $2n$-time $NTVOCA$ and real-time $NTVCA$ can be shown in a manner identical to the proof that $rCA = lOCA$ ([BC84], also refer Figure 2.5). ■

As for the containment $lCA \subseteq OCA$ from Theorem 2.9, we show that even linear-time $NTVCA$ are no more powerful than deterministic OCA. We do not know whether a similar result holds for $lNCA$. We do know, however, that the containment holds if both classes are made nondeterministic using the traditional notion, ie. that $lNCA$ are contained in $NOCA$, because $NOCA$ and NCA have the same power.

Theorem 6.11 *Linear-time $NTVCA \subseteq OCA$.*

Proof: We resort to the sequential machine characterisation of OCA to prove this result; given a linear-time $NTVCA$, we will construct a sweeping automaton SA accepting exactly the same language.

Let C be an NTVCA running in cn time. A valid computation path thus has a cn length characteristic bit string. We design the SA to generate all cn length strings in lexicographic

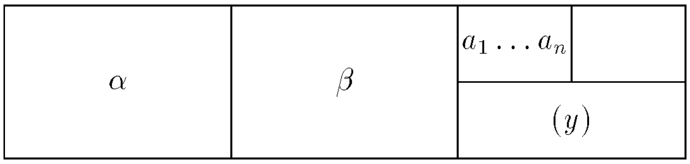


Figure 6.4: Worktape of an SA accepting an $lNTVCA$ language

order, and, for each string, to trace out the corresponding computation path. The SA will accept its input if it ever finds an accepting computation in this process.

As the SA reads its input, it shifts and packs symbols on the tape. When the entire input has been read, the worktape will be partitioned into three areas as shown in Figure 6.4.

The first area is a counter of length cn, and holds the string α. The second area is also of length cn, for holding the bit string β currently being tested. Initially both α and β are set to 0^{cn}. The third area is of length cn and has two tracks. The first track has a permanent copy of the input x in its leftmost n cells, and is blank elsewhere. The second track is initially a copy of the first track, and is to be used for tracing out the computation corresponding to the string in the second area. This requires cn space and not n space because the SA can only move from left to right, while the $NTVCA$ has two-way communication. So in simulating each step of the $NTVCA$, the SA shifts the configuration one cell right.

While reading \$, ie. after all the input has been read, α is incremented in each sweep. Simultaneously, a marker moves right, one cell per sweep, over the string β. If the marker is on a 0 (1, respectively), then the $NTVCA$ configuration y, which is stored on the second track of the third area, is updated as per δ_1 (δ_2, respectively). When the marker reaches the end of the second area, a full computation path has been traced. The marker is now erased, and the third area is reset to its initial status. It remains unchanged in subsequent sweeps until α overflows. When this happens (every 2^{cn} sweeps), the next bit string is generated in the second area (β is incremented). The marker is placed again on its leftmost bit, and the tracing out of the corresponding computation path begins in the third area. Thus all computation paths are traced, and an accepting computation, if any, can be found by the SA. ■

6.3 Modelling Restricted Nondeterminism

We now consider $TVCA$ where only specific computation paths are of interest. Specifically, we consider the 1-turn and 1-kink $NTVCA$ defined in section 6.1. These classes are important in that they help us identify the amount of nondeterminism needed to enhance the power of other classes. To make this clearer, note that a $T(n)$-time $NTVCA$ has $2^{T(n)}$ computation paths. Picking any one of these involves choosing $T(n)$ bits, corresponding to the characteristic bit string of the chosen computation path. A $T(n)$-time 1-turn $NTVCA$, on the other hand, has only $T(n)+1$ computation paths of interest. Picking any one of these involves picking one of the $T(n)$ positions in the characteristic bit string where the $TVCA$ switches over from using δ_1 to using δ_2. Since making a choice from $T(n)$ positions would involve setting $\log(T(n))$ bits, the "amount" of choice, due to nondeterminism, available to an $NTVCA$ and to a 1-turn $NTVCA$ differ by an exponential factor. As is to be expected, we will show that the 1-turn classes are quite weak compared to the other $NTVCA$ classes. First we show that 1-turn and 1-kink are equivalent notions; an $NTVCA$ of one type can be simulated by an $NTVCA$ of the other. Before this, we first show an intermediate result.

Lemma 6.12 *Let r be a regular expression denoting a subset R of $\{0+1\}^*$. Given any $NTVCA$ C, we can produce a modified $NTVCA$ C' which performs the same computation as C, but additionally, along each computation path, also indicates whether the bit string determining the computation path belongs to R.*

Proof: Let $C = (Q, \#, \delta_1, \delta_2, A)$ be an $NTVCA$, and let $M = (Q_1, \{0,1\}, \delta, q_0, F)$ be a deterministic finite-state machine (FSM) accepting R. We construct the required $NTVCA$ C' to function as follows: The states of C' are 2-tuples. The first component of each cell, put together, gives the configuration of C. In the second component, which is initially q_0, the state of M while processing the bit string corresponding to the current computation path is recorded. This component is updated as follows: If it contains $p \in Q_1$, then on using δ_1 (δ_2, respectively) it is changed to $\delta(p,0)$ ($\delta(p,1)$, respectively). Thus along any computation path, at any given time step, the second component of the state of each cell holds the same value. The bit string determining the computation path is in R if and only if this value is in F. ∎

This is in fact a weak result in that each cell is able to recognise R by acting as an FSM in isolation. By collectively using all cells in the array, some non-regular subsets of bit strings can also be recognised; however, for our purposes now, regular sets suffice.

Theorem 6.13 *$T(n)$-time 1-turn $NTVCA$ = $T(n)$-time 1-kink $NTVCA$.*

Proof: Consider simulating a 1-turn $NTVCA$ by a 1-kink $NTVCA$. Let the 1-turn $NTVCA$ be $C = (Q, \#, \delta_1, \delta_2, A)$. We define a 1-kink $NTVCA$, with transition functions h_1 and h_2, and with one unmarked state and one marked state corresponding to each state in Q. h_1 on unmarked states acts as δ_1. h_2 on unmarked states acts as δ_2 and also marks the resulting states. Subsequently, all operation is on the marked version of the states. h_1 on marked states acts as δ_2. (If h_2 encounters marked states, then the result is immaterial, since this does not correspond to a 1-kink path.) Thus the 1-turn path 0^i1^j using δ_1 and δ_2 is simulated by the 1-kink path 0^i10^{j-1} using h_1 and h_2.

The other inclusion can be similarly shown. ■

Since 1-turn and 1-kink nondeterminism allow less choice, it is to be expected that the classes they define are contained in the unrestricted nondeterminism classes. This is shown in the proof of the following theorem.

Theorem 6.14 *$T(n)$-time 1-turn $NTVCA \subseteq T(n)$-time $NTVCA$.*

Proof: A 1-turn $NTVCA$ must have an accepting computation with a characteristic bit string $0^i1^{T(n)-i}$ to accept its input. We can design an $NTVCA$ which uses the transition functions of the given 1-turn $NTVCA$, and also checks the regular expression 0^*1^* along its computation paths, as described in Lemma 6.12. A state is an accepting state if and only if its first component is an accepting state for the 1-turn $NTVCA$ and its second component is an accepting state for an FSM accepting 0^*1^*. Thus if the $NTVCA$ has an accepting computation, then it must be along a 1-turn path. Hence the $NTVCA$ accepts exactly the same language as the 1-turn $NTVCA$, and within the same time. ■

Lastly, we show how restricted the 1-turn nondeterministic class is — even when allowed linear time, it is contained in P. This is in direct contrast to Theorem 6.9, which shows the existence of an NP-complete language in real-time $NTVCA$.

Lemma 6.15 *Linear-time 1-turn $NTVCA \subseteq P$.*

Proof: A linear-time 1-turn $NTVCA$ has $cn+1$ computation paths of interest, where c is some constant. Each path is of length cn; thus it can be traced out by a sequential Turing machine in $O(cn^2)$ time. So all such paths can be checked in $O(n^3)$ time; hence the corresponding language is in P. ■

In fact, for 1-turn nondeterminism, all $NTVCA$ requiring polynomial time ($T(n)$ is $O(n^k)$ for some k) are contained in P.

This last result shows the limitations of 1-turn nondeterminism. However, we believe that even this much nondeterminism can increase the power of a class. As a specific example, consider any language L and define $\exists MID(L)$ as follows:

$$\exists MID(L) = \{xyz \in \Sigma^* \mid |x| = |z|, y \in L\}$$

For any L in rCA, we can show that $\exists MID(L)$ can be accepted by a real-time 1-turn $NTVCA$ (see Figure 8.5; the construction will be described in chapter 8). We do not know whether, for L in rCA, $\exists MID(L)$ can always be accepted by an rCA. Similarly, if we define $\exists PRE(L)$ as follows:

$$\exists PRE(L) = \{xy \in \Sigma^* \mid x \in L\}$$

then we can show that for any L in lCA, $\exists PRE(L)$ is in linear-time 1-turn $NTVCA$ (see Figure 8.2; the construction will be described in chapter 8). We do not know of any lCA construction to accept $\exists PRE(L)$. However, if L is an rCA language, then $\exists PRE(L)$ can also be shown to be an rCA language; this follows from an extension of Lemma 5.10 to non-tally sets. These and other such closure properties will be examined in chapter 8.

The idea behind examining 1-turn $NTVCA$ is essentially to see how many distinct computation paths need to be checked for acceptance. The concept can be generalised to k-turn for some constant k, and finite-turn. A k-turn $NTVCA$ is an $NTVCA$ where an accepting path, if one exists, alternates between using δ_1 and δ_2 at most k times. Similarly, a k-kink $NTVCA$ uses δ_2 at most k times. Clearly, k-turn is contained in $(k+1)$-turn. The non-trivial question is whether the containment is strict. It is easy to see that k-turn and k-kink paths also have characteristic bit strings representable by regular expressions; thus Theorems 6.13

and 6.14 hold for k-turn (k-kink) $NTVCA$ too. As for Lemma 6.15, we basically need to count the number of distinct computation paths of interest in a k-turn $NTVCA$. A combinatorial counting procedure shows that the number of such paths, in a $T(n)$-time $NTVCA$, is $\binom{m}{0} + \binom{m}{1} + \binom{m}{2} + \ldots + \binom{m}{k}$ where $m = T(n)$. This number is polynomially bounded in n for linear-time $T(n) = cn$; thus Lemma 6.15 also holds for linear-time k-turn $NTVCA$. That is, as long as the number of turns allowed on a computation path is bounded by a constant, no matter how large, the power of a linear-time $NTVCA$ is weaker than P. On the other hand, for unbounded turns, even real-time $NTVCA$ has a membership problem which is NP-complete.

6.4 Closure Properties

In this section we examine some closure properties of the language classes defined in the preceding sections.

Theorem 6.16 *If L_1 and L_2 can be accepted by $NTVCA$ in $T(n)$ time, then $L_1 \cup L_2$ can be accepted by an $NTVCA$ in $T(n)$ time.*

Proof: Let C_1 and C_2 be the $NTVCA$ accepting L_1 and L_2 respectively. We can construct an $NTVCA$ C which, on input x, creates two channels in the array of cells. Along each path chosen nondeterministically, it simulates C_1 in one channel and C_2 in the other, using the same characteristic bit string in both channels. If an accepting state is entered in either of the channels, then C accepts x. Clearly, C accepts $L_1 \cup L_2$. ■

Theorem 6.17 *Let L_1 and L_2 be accepted by $NTVCA$ in time $T_1(n)$ and $T_2(n)$ respectively.*

(a) $L_1 \cap L_2$ can be accepted by an $NTVCA$ in $T_1(n) + 2n + T_2(n)$ time.

(b) If $T_1(n)$ is strongly time-constructibile, then $L_1 \cap L_2$ can be accepted by an $NTVCA$ in $T_1(n) + T_2(n)$ time.

Proof: A construction similar to that outlined in the above proof will not work in this case, because even if x belongs to both L_1 and L_2, the accepting computations of C_1 and C_2 need not have the same characteristic bit string. So an $NTVCA$ accepting $L_1 \cap L_2$ must run through all combinations of a computation of C_1 followed by a computation of C_2.

To achieve the time bound in (a), let C_1 and C_2 be the $NTVCA$s accepting L_1 and L_2 respectively. The $NTVCA$ C accepting $L_1 \cap L_2$ begins simulating C_1 along all paths. If the input x belongs to L_1, then acceptance will be detected within $T_1(n)$ steps. When this happens, C initiates a firing squad synchronisation algorithm. This requires $2n$ steps. When the cells synchronise, they start simulating C_2. If x belongs to L_2 as well, this will be detected within another $T_2(n)$ steps. Thus if x is in $L_1 \cap L_2$, C will accept x within $T_1(n) + 2n + T_2(n)$ steps.

If $T_1(n)$ is strongly time-constructible, then the synchronisation stage can be avoided. C simply begins simulating C_1, while simultaneously computing $T_1(n)$. After $T_1(n)$ steps, the whole array of cells switches over to simulating C_2. The leftmost cell accepts its input if and only if both parts of the simulation end in accepting states. ∎

Corollary 6.18 *Linear-time $NTVCA$ are closed under union and intersection.*

Since $NTVCA = NSPACE(n)$, it follows that

Theorem 6.19 *$NTVCA$ are closed under complementation.*

However, it does not seem likely, especially in the light of Theorem 6.9, that real-time or linear-time $NTVCA$ are closed under complementation.

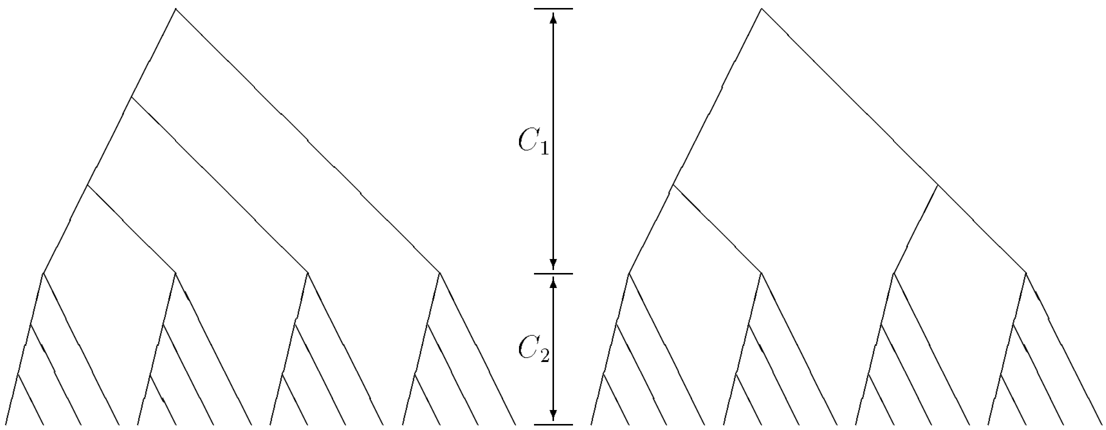
Note that Theorem 6.16 goes through even if we consider k-turn $NTVCA$. Theorem 6.17 does not; applying the method described there will result in the intersection of k-turn $NTVCA$ languages being accepted by a $2k$ or $(2k+1)$-turn (if k is odd) $NTVCA$.

Theorem 6.20 *Let L_1 and L_2 be accepted by k-turn and m-turn $NTVCA$ in $T_1(n)$ and $T_2(n)$ time respectively. Then*

(a) $L_1 \cup L_2$ can be accepted by a $\max(k, m)$-turn $NTVCA$ in $\max(T_1(n), T_2(n))$ time.

(b) $L_1 \cap L_2$ can be accepted by a j-turn $NTVCA$ in $T_1(n) + 2n + T_2(n)$ time, where

$$\begin{aligned} j &= k + m + 1 && \text{if } k \text{ is odd} \\ &= k + m && \text{otherwise} \end{aligned}$$



(a) 1-turn followed by 1-turn gives 3-turn paths

(b) 2-turn followed by 1-turn gives 3-turn paths

Figure 6.5: Intersection of finite-turn $NTVCA$ languages

(c) If $T_1(n)$ is strongly time-constructible, then $L_1 \cap L_2$ can be accepted by a j-turn $NTVCA$ in $T_1(n) + T_2(n)$ time, where j is as in (b).

Proof: (a) is straightforward. (b) and (c) are seen in a fashion similar to that in Theorem 6.17. The number of turns is explained as follows. An accepting path has at most k turns from the simulation of C_1, plus at most m turns from the simulation of C_2, plus possibly one more turn in changing over from the simulation of C_1 to that of C_2, and is thus a $(k+m+1)$-turn path. The additional turn is not needed if k is even, because then the path of C_1 with maximum number of turns ends with δ_1 in use. See Figure 6.5. ■

6.5 Conclusions

In this chapter, we have presented a new mechanism for introducing nondeterministic computations in the cellular automaton model. We have compared our notion of nondeterminism with the traditional notion. We have also defined restricted versions of nondeterministic computations and explored the power of the resulting automata. All this investigation essentially aims at refining the open problems in the containments

$$rOCA \subset rCA = lOCA \subseteq lCA \subseteq OCA \subseteq CA = DSPACE(n) \subseteq NSPACE(n)$$

and thus providing an alternative approach to solving the problems. A lot more investigation still remains to be done. A relatively unexplored area is the use of traditionally defined nondeterminism in time-bounded CA classes, ie. studying classes like $rNCA$ and $lNCA$. We feel that some questions concerning the power of these classes can be answered independent of the longstanding open questions regarding rCA, lCA and CA.

Chapter 7

Probabilistic and Alternating Computations on CA

So far we have seen two interpretations of $TVCA$ in detail — $TVCA$ as relativised CA in chapters 4 and 5, and $TVCA$ as nondeterministic CA in chapter 6. In a relativised CA operation, the unique computation path whose characteristic bit string is a prefix of the characteristic bit string of the oracle is picked, and the input is accepted if and only if this computation path ends in an accepting configuration. When the $TVCA$ is given a nondeterministic interpretation, we check whether at all there exists an accepting computation. We can also check, instead, whether more than half of the computation paths are accepting computations, thus interpreting the $TVCA$ operation as a probabilistic computation. Or, we can mark out specific states as universal or existential, and check if all or some computations respectively from these states end in accepting configurations; this would yield an alternating CA operation. Such computations are briefly examined in this chapter. Section 7.1 considers probabilistic computations, both unrestricted and restricted to specific computation paths. Section 7.2 looks at alternating CA computations, especially in comparison with $ASPACE(n)$ computations of alternating Turing machines.

7.1 Probabilistic Computation on a TVCA

In this section we look at some of the classes obtained by viewing the operation of a $TVCA$ as a probabilistic computation, ie. a computation which is deemed to be accepting if more than half of the sub-computations are accepting. For such probabilistic $TVCA$ ($PTVCA$), we impose the condition that $T(n)$ be CA-time-constructible. This is because when we count the number of accepting computations, we want to have a binary tree, of computations, which is pruned at a particular height $T(n)$. $PTVCA$ are formally defined as follows:

Definition 7.1 *Let $T(n)$ be a CA-time-constructible function. A $T(n)$-time probabilistic $TVCA$ ($PTVCA$) is a construct $C = (Q, \#, \delta_1, \delta_2, A)$ defined as a 2-$TVCA$. Acceptance is defined as follows: A string w is accepted by C if more than half of the $T(|w|)$-time computations of C on w are accepting computations.*

Henceforth, for $PTVCA$, when we talk of a $T(n)$-time computation we implicitly assume that $T(n)$ is CA-time-constructible.

As in the case of $NTVCA$, here too we can define restricted versions, where only certain computation paths are of interest.

Definition 7.2 *A 1-turn (1-kink) $PTVCA$ is a $TVCA$ C which accepts input w if and only if more than half of the computation paths determined by bit strings of the form 0^*1^* ($0^*(\epsilon + 10^*)$) are accepting computations.*

Also, as was done in chapter 6 for $NTVCA$, these definitions can be generalised to k-$TVCA$ as well.

The following result is easily shown.

Theorem 7.3 *$T(n)$-time $NTVCA \subseteq T(n)$-time $PTVCA$*

Proof: Let C be a $T(n)$-time $NTVCA$. On any input w of length n, it has $2^{T(n)}$ computation paths. If even one of these is an accepting path, then C accepts w. To incorporate this condition into a probabilistic computation, we construct a $PTVCA$ C' which, at $t = 1$, executes a dummy step. In this step, δ_1 puts the CA into an accepting configuration from which all configurations resulting from the application of either δ_1 or δ_2 in any order are accepting configurations. δ_2 puts it in the same input configuration with a marker to indicate that the dummy step is over. This configuration is now updated as in C. Thus a computation of C with characteristic bit string $b_1b_2\ldots b_{T(n)}$ is simulated by the computation of C' having characteristic bit string $1b_1b_2\ldots b_{T(n)}$. All computation paths of C' having characteristic bit strings beginning with 0 are accepting computations; exactly half of the total number of computations are of this type. Thus C' will accept w if even one computation path with characteristic bit string of the form 1α is an accepting computation, which happens if and only if the computation of C with characteristic bit string α is an accepting computation, in which case C also accepts w. Thus C' accepts its input exactly when C does.

The $PTVCA$ C' so constructed requires $T(n)+1$ time. To speed it up by one step, we can modify it to get C'' as follows: At the first time step, δ_2, instead of performing a dummy computation, creates two channels in the array. The first channel is filled with the contents obtained by applying δ_1 to w, and the second channel is filled with the contents obtained by applying δ_2 to w. At subsequent steps, each channel is updated according to the transition function in use at that step. Thus each computation of C'' with characteristic bit string 1α holds the results of two computations of C — namely, the computations with characteristic bit strings 1α and 0α — in its two channels. C'' is programmed to accept its input if either of the two channels holds an accepting configuration. Clearly, C'' accepts the same language as C' probabilistically, and does so in $T(n)$ time. ■

Corollary 7.4 *Real-time $NTVCA$ $\subseteq$ real-time $PTVCA$.*
Linear-time $NTVCA$ $\subseteq$ linear-time $PTVCA$.

By a construction similar to that in the proof of Theorem 6.11, we can also show that linear-time $PTVCA$ are no more powerful than deterministic OCA.

Theorem 7.5 *Linear-time $PTVCA \subseteq OCA$*

Proof: Given any linear-time $PTVCA$, we will construct an SA (sweeping automaton) accepting the same language as the $PTVCA$. This will prove the theorem. Most of the details of the construction of the SA are as in the proof of Theorem 6.11; here we will only describe the additional details.

While simulating the $NTVCA$, the SA traced out the different computation paths of the $NTVCA$ in lexicographic order, and accepted the input if any accepting computation path was found. For a $PTVCA$ simulation, the SA must check all computation paths, and count how many of them are accepting computations. For this, a fourth area is created on the worktape, beyond the three areas described earlier. This area has cn cells, and is used as a counter γ initialised to zero. γ is incremented whenever an accepting computation is found. When all computation paths have been checked, the SA checks whether γ contains a number greater than 2^{cn-1}. If this is the case, the SA moves right in an accepting state; otherwise it moves right in a rejecting state. Thus the probabilistic acceptance condition is checked. ∎

We now consider the restricted-paths $PTVCA$ defined in Definition 7.2. Some of the results can be shown in a direct analogue of the corresponding results for $NTVCA$; others are not so simple to see.

Theorem 7.6 *$T(n)$-time 1-turn $PTVCA$ = $T(n)$-time 1-kink $PTVCA$.*

This is shown as in the proof of Theorem 6.13.

Since 1-turn and 1-kink nondeterminism allow less choice, it is to be expected that the classes they define are contained in the unrestricted nondeterminism classes. For $NTVCA$ computation, Theorem 6.14 shows that this is indeed the case. It is also true for probabilistic computation; however, this is not so easy to see. That the containment still holds is shown in the proof of the following theorem.

Theorem 7.7 *$T(n)$-time 1-turn $PTVCA \subseteq T(n)$-time $PTVCA$.*

Proof: A 1-turn $PTVCA$ has $T(n)+1$ computation paths of interest. A $PTVCA$, on the other hand, has to consider $2^{T(n)}$ computation paths. In a simulation of a 1-turn $PTVCA$, $2^{T(n)} - T(n) - 1$ of these carry no information; they correspond to invalid paths. To prevent these computation paths from affecting the overall outcome, we must ensure that exactly half of these are accepting computations. Consider the following method of division of these paths into accepting and rejecting paths:

Invalid paths (ie. of the form $\Sigma^*10\Sigma^*$): $2^{T(n)} - T(n) - 1$.

1. Paths beginning with 0 (ie. of the form $0^+1^+0^+\Sigma^*$): $2^{T(n)-1} - T(n)$ paths.

2. Paths beginning with 1 (ie. of the form $1^+0^+\Sigma^*$): $2^{T(n)-1} - 1$ paths.

   (a) Paths of the form 1^+0^+: $T(n) - 1$ paths.

      (i) Paths with odd number of 1s (ie. of the form 1^k0^j, where $0 < k, j < T(n)$ and k is odd): $\lfloor T(n)/2 \rfloor$ paths.

      (ii) Paths with even number of 1s (ie. of the form 1^k0^j, where $0 < k, j < T(n)$ and k is even): $\lceil T(n)/2 \rceil - 1$ paths.

   (b) Other paths (ie. of the form $1^+0^+1\Sigma^+$): $2^{T(n)-1} - T(n)$ paths.

Make all paths in 1 and 2(a)(i) accepting, and all paths in 2(a)(ii) and 2(b) rejecting. The accept cell can determine the type of the path currently being followed using the procedure described in Lemma 6.12.

Let A and R denote the number of invalid accepting and rejecting paths respectively. Then $A = 2^{T(n)-1} - T(n) + \lfloor T(n)/2 \rfloor$, and $B = 2^{T(n)-1} - T(n) + \lceil T(n)/2 \rceil - 1$. Clearly, if $T(n)$ is odd, then $A = R$, as desired. If $T(n)$ is even, the $A = R + 1$. But in this case, the number of valid paths is itself odd ($T(n) + 1$), and so the 1-turn $PTVCA$ cannot have a tie between the number of accepting and rejecting paths. So this distribution of invalid paths does not introduce any error. Thus in either case, the $PTVCA$ accept its input if and only if the number of valid accept paths exceeds the number of valid reject paths; ie. if and only if the 1-turn $PTVCA$ accepts its input. ∎

For these restricted choice classes also, we show below that an $NTVCA$ class is contained in the corresponding $PTVCA$ class.

Theorem 7.8 *$T(n)$-time 1-turn $NTVCA \subseteq T(n)$-time 1-turn $PTVCA$.*

Proof: A $T(n)$-time 1-turn $NTVCA$ has $T(n) + 1$ computation paths of interest, ie. valid computation paths. If any of these is an accepting computation, then the input is to be accepted. To achieve the same effect in a probabilistic computation, we can construct a $PTVCA$ which has $2T(n) + 1$ valid computation paths of interest. $T(n)$ of these can be designed to always accept the input, and the remaining $T(n)+1$ can simulate the corresponding computation paths of the 1-turn $TVCA$. However, since we want the resulting $PTVCA$ to be 1-turn itself, it must be a $2T(n)$-time $PTVCA$, since only then will $2T(n)+1$ computation paths be considered for acceptance. So this approach, though the most straightforward, does not give us a real-time simulation.

To simulate the 1-turn $NTVCA$ probabilistically in real time using only 1-turn paths, we construct a $T(n)$-time $PTVCA$ where half of the valid computation paths are accepting paths and each of the remaining valid computation paths simulates two distinct computation paths of the $NTVCA$. (This construction is very similar to that outlined in the proof of Theorem 7.3, the difference being that we now restrict our attention to 1-turn paths.) Thus the nondeterministic acceptance criterion of the $NTVCA$ is translated into a probabilistic acceptance criterion.

The division of valid computation paths is done as follows. Note that 1-turn paths are described by bit strings of the form 0^i1^j, where $i + j = T(n)$. Let all paths described by such strings with odd i accept. Let all paths with even i simulate the $NTVCA$ paths with characteristic bit strings 0^i1^j and $0^{i+1}1^{j-1}$. This is done by maintaining two tracks in each cell. The second track is initially empty. As long as δ_1 is being used, only the first track is used. When δ_2 is first used, let the first track contain α. Now the first (second, respectively) track is filled with the result obtained by applying δ_1 (δ_2, respectively) to α. Subsequently, δ_2 is used on both tracks. The leftmost cell enters an accepting state if an accepting state is reached in either track. The $PTVCA$ so constructed accepts the same language as the $NTVCA$. ■

Corollary 7.9 *Real-time 1-turn $NTVCA$ $\subseteq$ real-time 1-turn $PTVCA$.*
Linear-time 1-turn $NTVCA$ $\subseteq$ linear-time 1-turn $PTVCA$.

Finally, we show that even linear-time 1-turn $PTVCA$ are contained in P. This is not an unexpected result, and it merely points out that the weakness of 1-turn or even k-turn computations is not overcome by going from nondeterministic to probabilistic computations.

Theorem 7.10 *Linear-time 1-turn $PTVCA \subseteq P$.*

Proof: Let the 1-turn $PTVCA$ operate in cn time, where $c > 0$ is some constant. There are $cn + 1$ valid computation paths. The characteristic bit strings of these paths can easily be generated in lexicographic order by a deterministic Turing machine. For each such string generated, the corresponding computation path of the $PTVCA$ can be traced out by the Turing machine in $O(n^2)$ time ($O(n)$ time per step, and there are cn steps). Thus all paths can be successively traced out in $O(n^3)$ time. Additionally, the Turing machine can also keep track of how many of these paths ended in accepting configurations. So it can determine, in $O(n^3)$ time, whether the $PTVCA$ accepted its input. ■

Clearly, this argument holds even if the $PTVCA$ requires $p(n)$ time for some polynomial p. The simulating Turing machine will then require $O(np^2(n))$ time, which is again polynomially bounded.

The study in restricted nondeterminism can easily be extended to 2-turn paths and in general k-turn paths for any constant k. As an example, we show below that 1-turn $T(n)$-time $PTVCA$ are contained in 2-turn $T(n)$-time $PTVCA$. As in the proof of Theorem 7.7, we need to show that the paths which are 2-turn but not 1-turn can be divided equally between accepting and rejecting computations, so that they do not affect the overall outcome. A 1-turn path has characteristic bit string 0^*1^*, while a 2-turn path has characteristic bit string $0^*1^*0^*$. The difference is thus characterised by bit strings of the form $0^*1^*100^*$; for strings of length $T(n)$, there are $[T(n)-1]T(n)/2$ such paths. (The position of the last 1 can be chosen in $T(n)-1$ ways, excluding the last position. For each such position i, the position of the first 1 can be chosen in i ways, from 1 to i.) These invalid paths can be partitioned into accepting and rejecting paths as per the division described below. (Such a partition is possible, since paths of each set can be described by regular expressions, and then Lemma 6.12 can be used.) The cardinalities of different sets in the partition are also indicated.

Paths which are 2-turn but not 1-turn (ie. paths of the form $0^*1^+0^+$, ie. $0^i1^j0^k$ where $i \geq 0$, $j,k > 0$): $[T(n)-1]T(n)/2$ paths.

Case 1. $T(n)$ is even, $i+j+k = 2m$ for some m: $2m^2 - m$ paths.

1. $(i+j)$ even. The last 1 can be in positions $2,4,6,\ldots,2m-2$. For each such position j, the first 1 can be in positions $1,\ldots,j$. Total number of such paths, hence, is given by $\Sigma_{j=1}^{m-1}(2j) = m^2 - m$.
2. $(i+j)$ odd: m^2 paths.

(a) $j = 1$, $i \equiv 0 \pmod 4$ (ie. paths of the form $0^{4i}10^k$): $\lceil m/2 \rceil$ paths.

(b) Other paths: $m^2 - \lceil m/2 \rceil$.

Accept on paths from 1 or 2(a), and reject on paths from 2(b).

Case 2. $T(n)$ is odd, $i + j + k = 2m + 1$ for some m: $2m^2 + m$ paths.

1. $(i + j)$ odd. The last 1 can be in positions $1, 3, 5, \dots, 2m - 1$. For each such position j, the first 1 can be in positions $1, \dots, j$. Total number of such paths, hence, is given by $\Sigma_{j=1}^{m}(2j - 1) = m^2$.
2. $(i + j)$ even: $m^2 + m$ paths.

   (a) $j = 1$, $i \equiv 1 \pmod 4$ (ie. paths of the form $0^{4i+1}10^k$): $\lceil m/2 \rceil$ paths.

   (b) Other paths: $m^2 + \lfloor m/2 \rfloor$.

Accept on paths from 2(b), and reject on paths from 1 or 2(a).

Let A_1 and R_1 denote the number of accepting and rejecting paths respectively of the 1-turn $PTVCA$, and A_e and R_e denote the number of invalid accepting and rejecting paths respectively of the 2-turn $PTVCA$. Then A (R respectively), the number of valid accepting (rejecting) paths of the 2-turn $PTVCA$, is given by $A_1 + A_e$ ($R_1 + R_e$, respectively).

In Case 1, $A_e = R_e$ or $A_e = R_e + 1$. Since, in this case, $T(n)$ is even, the number of 1-turn paths is odd, and thus A_1 and R_1 cannot be equal. So A exceeds R if and only if A_1 exceeds R_1.

Similarly, in Case 2, $A_e = R_e$ or $A_e = R_e - 1$. In this case, $T(n)$ is odd, and the number of 1-turn paths is even. Thus if A_1 exceeds R_1, it does so by at least 2. A_1 and R_1 can also be equal, but then the input should be rejected. It is easy to see that in this case too, A exceeds R if and only if A_1 exceeds R_1.

Thus the 2-turn $PTVCA$ accepts the same language as the 1-turn $PTVCA$.

We thus have the following result:

Theorem 7.11 *1-turn $T(n)$-time $PTVCA \subseteq$ 2-turn $T(n)$-time $PTVCA$.*

The other results in this section can be similarly generalised.

Lastly, we consider the closure of $PTVCA$ language classes under some simple operations. Closure under union or intersection does not seem to hold; the proofs of Theorems 6.16 and 6.17 do not carry over, since we now need to *count* the number of accepting computations of C_1 and C_2. On the other hand, for $PTVCA$, closure under complementation is relatively easy to show. Merely exchanging the accepting and the non-accepting states fails in case there are an equal number of accepting and rejecting computations. However, using one extra time step, this difficulty can be overcome, as is seen in the following theorem.

Theorem 7.12 *If L can be accepted by a $PTVCA$ in $T(n)$ time, then $\overline{L}$ can be accepted by a $PTVCA$ in $T(n)+1$ time.*

Proof: Given a $T(n)$-time $PTVCA$ C accepting L, we construct a $(T(n)+1)$-time $PTVCA$ C' accepting $\overline{L}$ as follows. A computation path of C' with characteristic bit string 1α follows the computation of C with characteristic bit string α, and accepts if and only if the computation of C rejects. All computation paths of C' with characteristic bit string beginning with 0 are dummy computations, introduced to take care of the case when the number of accepting and rejecting computations of C, A and R respectively, are tied. Dummy paths ending with 0 accept. Dummy paths ending with 1 reject, the only exception being the path with characteristic bit string $0^{T(n)}1$, which accepts. We can now see that the number of accepting and rejecting computations of C', A' and R' respectively, are

$$\begin{aligned} A' &= R + 2^{T(n)-1} + 1 \\ R' &= A + 2^{T(n)-1} - 1 \end{aligned}$$

where $A + R = 2^{T(n)}$. It is easily verified that $A' > R'$, making C' accept its input, if and only if $A \leq R$, ie. if and only if C rejected its input. Thus C' accepts the complement of the language accepted by C. ■

7.2 Alternating Computations on TVCA

Further generalising the concept of nondeterministic and probabilistic $TVCA$, we now introduce alternation in the CA model of computation. An alternating CA (ACA) is a CA which, at each time step, may globally, ie. at all cells, use either of two transition functions δ_1

and δ_2. The states of Q are partitioned into four classes — accepting states, rejecting states, universal states and existential states. Whether a particular configuration is a universal or an existential configuration is determined by the state of the leftmost cell. The computation tree of an ACA on input w is a binary tree where the root node holds w. The left (right) child of a node holding c holds the configuration obtained by applying δ_1 (δ_2) to c. The input w is said to be accepted if this binary tree has a subtree satisfying the following properties:

The root node of the subtree is the root node of the overall computation tree.

At each node, if the leftmost state in the configuration represented at that node is universal, then both children of the node are present in the subtree.

At each node, if the leftmost state in the configuration represented at that node is existential, then exactly one child of the node is present in the subtree.

At each node, if the leftmost state in the configuration represented at that node is accepting, then the node is a leaf of the subtree.

No leaf of the subtree has a rejecting state as the leftmost state in its configuration.

Such a subtree represents an accepting computation of the ACA.

Investigating the power of such ACA necessarily begins with examining the relationship $DSPACE(n) = CA$. We shall first show that the corresponding equality for alternating computations also holds. Without loss of generality we assume that the Turing machines considered have a single tape.

Lemma 7.13 *$ASPACE(n) \subseteq ACA$.*

Proof: This proof is a slight modification of the proof of Lemma 2.3, where we show that $DSPACE(n) \subseteq CA$. The construction outlined there cannot be used directly because the state of the alternating Turing machine (ATM) at each time step indicates whether the ATM is in a universal or an existential state. So, in the simulating ACA, this state must always be represented at the leftmost cell. The ACA holds tape configurations of the ATM in its array in a folded fashion in two tracks, so that the tape square over which the ATM

head is positioned is always represented at the leftmost cell. When the tape head moves, the ACA correspondingly shifts the contents of the two tracks. For details of such a construction, see Smith's proof [Smi72] that $DSPACE(n) \subseteq CA$. This requires one data movement step after each real simulation step; thus the ACA takes twice as much time as the ATM and finally performs the same computation. ■

Lemma 7.14 $ACA \subseteq ASPACE(n)$.

Proof: This lemma is quite easy to see; given an ACA, the $ASPACE(n)$ machine construction is akin to constructing an $NSPACE(n)$ machine simulating an $NTVCA$ (Lemma 6.5). The Turing machine operates in sweeps, where each sweep requires $O(n)$ time and simulates one step of the CA. For simulating an ACA, the state of the ATM at the beginning of each sweep reflects the state — universal or existential — of the ACA, while the operation within a sweep is deterministic. ■

From the above two lemmas it now follows that

Theorem 7.15 $ACA = ASPACE(n)$.

The time-bounded ACA classes correspond to time-bounded $ASPACE(n)$ computations. We use $ASPTI(s(n), t(n))$ to denote computations of alternating Turing machines which use $s(n)$ space and run in $t(n)$ time, and $ACA(t(n))$ to denote ACA running in $t(n)$ time. The next two lemmas are quite easy to see; they follow from the constructions outlined in Lemmas 7.13 and 7.14.

Lemma 7.16 $ASPTI(n, t(n)) \subseteq ACA(2t(n))$.

Consequently, the $DTIME(n) \subseteq lCA$ containment carries over to alternating computations too.

Corollary 7.17 $ATIME(n) \subseteq lACA$.

Lemma 7.18 $ACA(t(n)) \subseteq ASPTI(n, O(nt(n)))$.

Thus, if *poly* denotes the class of polynomial-valued functions, then

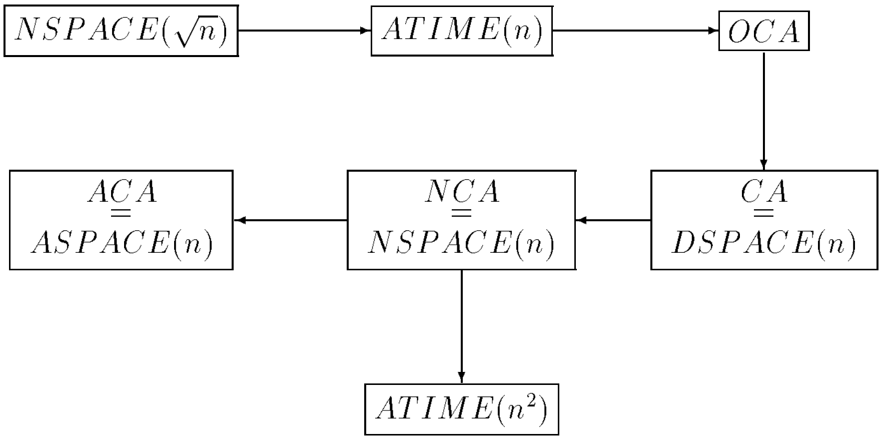


Figure 7.1: Inclusions among CA and ATM classes

Corollary 7.19 $ACA(poly) = ASPTI(n, poly) \subseteq PSPACE$.

It is also quite easy to see, by a process similar to that outlined in Theorem 6.9, that the language of fully quantified Boolean formulas evaluating to *True*, QBF, is in $lACA$. Since QBF is $PSPACE$-complete [BDG88], the membership problem for $lACA$ is also $PSPACE$-complete.

Since it is known that $NSPACE(s(n))$ is contained in $ATIME(s^2(n))$ [HU79, BDG90], we thus have the overall setup shown in Figure 7.1.

7.3 Conclusions

In this chapter we have considered two more interpretations of $TVCA$ — as probabilistic CA and as alternating CA. These are both generalisations of the $NTVCA$ defined in the previous chapter, and have been examined only briefly here. The relations between such language classes are depicted in Figures 7.1 and 7.2. In Figure 7.2, known (ie. existing) classes are shown in ovals and the newly defined classes are shown in boxes. The containments depicted between ovals are known results; the other containments have been shown in this thesis.

The alternating CA classes are useful in considering closures of rCA and lCA languages under various operations. If lCA are not known to be closed under some operation, we would like to identify the smallest CA class containing this closure; this gives us some idea of the complexity of the operation considered. Alternating CA classes help in this respect. Some such closure results are shown in the next chapter, which deals with the closure of CA classes under a wide variety of language operations.

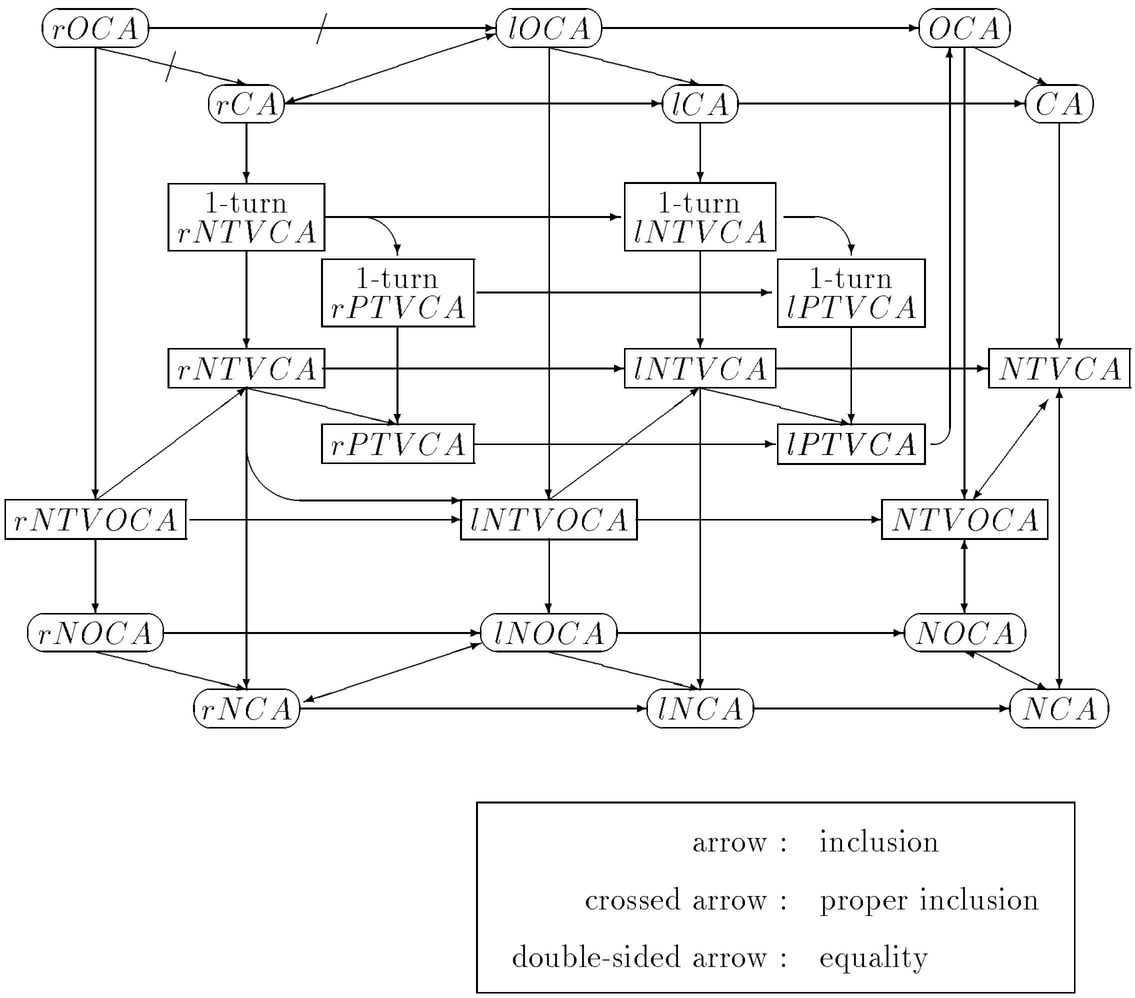


Figure 7.2: Deterministic, nondeterministic and probabilistic CA classes

Chapter 8

Language Operations on Cellular Automata Classes

8.1 Introduction

When studying any language class, an interesting question from the language-theoretic viewpoint is identifying language operations under which the class is closed. In other words, if $\mathcal{L}$ is a language class and $\circ$ is a k-ary operation on languages, then the question is whether $L_1, L_2, \ldots, L_k \in \mathcal{L} \Rightarrow \circ(L_1, L_2, \ldots, L_k) \in \mathcal{L}$ is true. Several language operations have been studied in the literature, especially in relation to the classes defined by the Chomsky hierarchy [HU79]. Operations typically studied are:

- Operations with regular sets: concatenation, intersection, quotient
- Boolean operations: union, intersection, complementation
- Kleene operations: $\star$, concatenation
- Marked operations: marked union ($c_1L_1 \cup c_2L_2$) and marked product (L_1cL_2)
- Others: MIN, MAX, reversal, homomorphisms, inverse gsms etc.

The study of closure properties is important for several reasons:

- Closure properties can be used to show that a language belongs to a particular class.

- Closure properties can be used to show that a language does not belong to a particular class.

- The closure or non-closure of a class under some operation can reflect the proper containment or equality of this class in or with some other class; thus, investigating the closure could provide another approach to proving/disproving a proper containment.

In this chapter, we examine some closure properties of the language classes defined by (real-space-bounded) cellular automata. Many closure properties of these classes, especially under the Boolean and Kleene operations, have already been investigated in detail [BC84, CGS84a, CGS84b, CGS86, IJ87, IJ88, IK84, Smi72]. Some of the known results are listed below:

$rOCA$ This class is closed under union, intersection, complementation, reversal, marked concatenation and marked star, inverse morphisms, injective length-multiplying morphisms, inverse deterministic gsm mappings. It is not closed under letter-to-letter morphisms [CGS84a, CGS84b, CGS86, IK84].

rCA This class is closed under union, intersection, complementation. Its closure under reversal and concatenation is not known. However, it is known that the class is closed under concatenation if it is closed under reversal and that it is closed under reversal if and only if it is equivalent in power to lCA [Smi72, IJ88].

lCA This class is closed under union, intersection, complementation, reversal [Smi72].

OCA This class is closed under union, concatenation, Kleene +, ϵ-free homomorphisms, inverse homomorphisms, intersection, complementation, reversal [IJ87].

Here we consider closures under some other language operations. We consider the following operations:

$$
\begin{aligned}
INIT(L) &= \{x \mid \exists y, xy \in L\} \\
END(L) &= \{x \mid \exists y, yx \in L\} \\
MAX(L) &= \{x \mid x \in L \\
&\qquad \text{and } x \text{ is not a proper prefix of any word in } L\} \\
\exists PRE(L) &= \{xy \mid x \in L\} \\
&= \{a_1 \dots a_n \mid \exists i \in \{1, 2, \dots, n\}, a_1 \dots a_i \in L\} \\
\forall PRE(L) &= \{a_1 a_2 \dots a_n \mid \text{for } 1 \leq i \leq n, a_1 a_2 \dots a_i \in L\} \\
\oplus PRE(L) &= \{a_1 a_2 \dots a_n \mid \\
&\qquad \|\{a_1 a_2 \dots a_i \mid a_1 a_2 \dots a_i \in L, 1 \leq i \leq n\}\| \text{ is odd } \} \\
MIN(L) &= \{x \mid x \in L \text{ and no } w \in L \text{ is a proper prefix of } x\} \\
&= \{a_1 a_2 \dots a_n \in L \mid \text{for } 1 \leq i < n, a_1 a_2 \dots a_i \notin L\} \\
PAD_{m,n}(L) &= \{xy \mid m|x| = n|y|, x \in L\} \\
(1/2)L &= \{x \mid \exists y, |x| = |y|, xy \in L\} \\
(1/3)L &= \{x \mid \exists y, 2|x| = |y|, xy \in L\} \\
MID(1/3)(L) &= \{x \mid \exists y, \exists z, |x| = |y| = |z|, yxz \in L\} \\
\exists MID(L) &= \{xyz \mid |x| = |z|, y \in L\} \\
(1/2)CYCLE(L) &= \{xy \mid |x| = |y|, yx \in L\} \\
CYCLE(L) &= \{xy \mid yx \in L\} \\
SHUFFLE(L) &= \{a_2 a_1 a_4 a_3 \dots a_{2n} a_{2n-1} \mid a_1 a_2 \dots a_{2n} \in L\}
\end{aligned}
$$

Most of these operations preserve regular sets; see [HU79].

The results we prove are summarised in table 8.1.

8.2 Closure Properties

In this section we consider the language classes $rOCA$, rCA, lCA, OCA, and CA. We examine the complexity of the resulting languages when an operation amongst those listed in section 8.1 is applied to languages of these classes. Our first observation is that the operations $INIT$, END and MAX are very powerful; even when applied to $rOCA$ languages, these operations yield undecidable languages.

	$rOCA$	rCA	lCA	OCA	CA
$INIT(L)$	**U**	**U**	**U**	**U**	**U**
$MAX(L)$	**U**	**U**	**U**	**U**	**U**
$END(L)$	**U**	**U**	**U**	**U**	**U**
$\exists PRE(L)$	$rOCA$	rCA	$O(n^2)$ time CA or 1-turn $lNTVCA$	OCA	CA
$\forall PRE(L)$	$rOCA$	rCA	$O(n^2)$ time CA or $lACA$	OCA	CA
$\oplus PRE(L)$	$rOCA$	rCA	$O(n^2)$ time CA or $lACA$	OCA	CA
$MIN(L)$	$rOCA$	rCA	$O(n^2)$ time CA or $lACA$	OCA	CA
$PAD_{m,n}(L)$		rCA	lCA	OCA	CA
$(1/2)(L)$			$lNTVCA$	OCA	CA
$(1/3)(L)$			$lNTVCA$	OCA	CA
$MID(1/3)(L)$			$lNTVCA$	OCA	CA
$\exists MID(L)$		1-turn $rNTVCA$	1-turn $lNTVCA$	OCA	CA
$(1/2)CYCLE(L)$			lCA	OCA	CA
$CYCLE(L)$			$O(n^2)$ time CA or 1-turn $lNTVCA$	OCA	CA
$SHUFFLE(L)$		rCA	lCA	OCA	CA

Table 8.1: Closure properties of CA language classes

Theorem 8.1 *For an arbitrary given $rOCA$ language L and a string x, it is undecidable whether $x \in INIT(L)$ or $x \in END(L)$ or $x \in MAX(L)$.*

Proof: It has been shown in [CGS86] that the emptiness problem for homogeneous trellis automata (given a homogeneous trellis automata, decide whether the language accepted by it is empty) is undecidable, and in [BC84, CC84] that homogeneous trellis automata and $rOCA$ accept exactly the same class of languages. Thus the emptiness problem for $rOCA$ languages is undecidable. The emptiness problem for L can be reduced to the membership problem for $INIT(L)$ as follows: Let L be the $rOCA$ language, $L \subseteq \Sigma^+$. Choose a special symbol $\$ \notin \Sigma$, and define L' as $L' = \$L = \{\$x \mid x \in L\}$. Clearly, L' is also an $rOCA$ language. Now, $\$ \in INIT(L')$ if and only if, for some y, $\$y \in L'$, if and only if, for some y, $y \in L$, if and only if $L \neq \emptyset$. Thus, if membership of \$ in $INIT(L')$ were decidable, then emptiness of L would also be decidable. But this is known to be undecidable. Hence, for an $rOCA$ language L, the membership problem in $INIT(L)$ is undecidable.

The undecidability of the membership problem for $END(L)$ is similarly shown, using the language $L\$$ in the reduction. To show undecidability of $MAX(L)$, $\$L \cup \{\$\}$ is used in the reduction. ■

None of the remaining operations generate undecidable languages from CA languages; in fact it is easy to show that the class of CA languages is closed under all these operations.

Theorem 8.2 *If $L \in CA$, then all the languages described above, except $INIT(L)$, $END(L)$, and $MAX(L)$, are also CA languages.*

Proof: CA languages exactly coincide with the class $DSPACE(n)$. It is easy to see that for any $DSPACE(n)$ language L, the languages obtained through the operations described above can also be accepted by $DSPACE(n)$ machines. ■

Thus when we consider these operations on classes within CA, we know that CA algorithms exist; we will try to find the smallest sub-classes within CA containing the closures of these classes under these operations. Our next two results consider the operations which require prefix membership computations. For $rOCA$ and rCA, such computations can be performed within the same bound; however, for lCA, efficient computations do not seem to be possible.

Theorem 8.3 *Let $\Box \in \{\exists, \forall, \oplus\}$. If L is an $rOCA$ or an $lOCA$ ($= rCA$) language, then $\Box PRE(L)$ and $MIN(L)$ are $rOCA$ or $lOCA$ languages respectively.*

Proof: Let $L \in rOCA$ be accepted by an $rOCA$ C. Let **S** be a signal sent from the leftmost cell to the rightmost cell at unit speed. As **S** passes through the i^{th} cell, this cell is indicating whether or not the prefix of x of length i is in L. **S** can record all these responses and perform the $\Box$ operation on them as it travels right. Thus when it reaches the right end, it can indicate whether some, or all, or an odd number of prefixes, (depending on $\Box$) of x are in L. It can also check whether any proper prefix is in L. So $\Box PRE(L)$ or $MIN(L)$ can be accepted.

If L is an rCA language accepted by rCA C, then Lemma 5.10 outlines the construction of a CA C' which on input x behaves as follows: At time i, the leftmost cell of C' indicates whether or not the prefix of x of length i is in L. The construction given there is for tally languages, but it is easily verified that it carries over for non-tally languages as well. See Figure 8.1. Now C' can be modified so that the leftmost cell also performs the $\Box$ operation on its successive states. Thus the $\Box$ operation is performed on the membership values of all prefixes; thus $\Box PRE(L)$ is accepted in real time. Similarly, the leftmost cell could check that x is in L but no proper prefix of x is in L; thus $MIN(L)$ can be accepted in real time.

■

Theorem 8.4 *Let L be an lCA language, and let $\Box \in \{\exists, \forall, \oplus\}$. Then*

(a) $\Box PRE(L)$ and $MIN(L)$ can be accepted by a CA in $O(n^2)$ time.

(b) $\exists PRE(L)$ can be accepted by a linear-time 1-turn $NTVCA$.

(c) $\forall PRE(L)$, $MIN(L)$ and $\oplus PRE(L)$ can be accepted by a linear-time alternating CA.

Proof: (a) Let L be accepted by an lCA C. Construct a CA C', which, on input $x = a_1 a_2 \dots a_n$, does the following: A signal **S** is sent from the rightmost cell to the left. When **S** first reaches the i^{th} cell, it initiates a firing squad synchronisation algorithm on the cells to its left. When the cells fire ($2i$ steps), they simulate C; thus C is simulated on input $a_1 a_2 \dots a_i$. When this is complete ($2i$ steps), **S** moves one cell left. As and when the membership value

a	b	c	d	a	b	c	a	b	a
1	2	3	4	1	2	11	1	14	16
5	6	7		5	12		15		
8	9			13					
10									

Computation of rCA C on prefixes of $abcd$

a	b	c	d
$1, 16$	$2, 14$	$3, 11$	$4, \$$
$5, 15$	$6, 12$	$7, \$$	
$8, 13$	$9, \$$		
$10, \$$			

Corresponding computation of rCA C'

Figure 8.1: Real-time simulation on all prefixes, for an rCA language

of $a_1a_2\ldots a_i$ in L, for some i, reaches the leftmost cell, it updates the $\Box$ value. When **S** reaches the leftmost cell, $\Box PRE(L)$ is correctly computed here. Since **S** spends $4i$ steps at each cell, the total time required is $O(n^2)$.

The leftmost cell could, instead of computing $\Box$, also check the prefix condition for $MIN(L)$; thus $MIN(L)$ can be accepted in $O(n^2)$ time.
(The firing squad synchronisation algorithm on the first i cells, and the simulation of C on the first $i+1$ cells, can be done in parallel. The time required still remains $O(n^2)$.)
(b) This is seen as follows. The $NTVCA$ begins using δ_1 as the transition function. While δ_1 is being used, a signal travels from right to left at unit speed. If δ_1 is used for n or more steps, then the computation is useless and does nothing. If, however, the one turn to using δ_2 is made at time $n' \leq n$, then at this point a suffix of size n' has been identified; the suffix is the substring in the cells which have already seen the signal pass through them. In the subsequent computation, which uses transition function δ_2, these cells behave as if they have the boundary state #, while the remaining cells behave like cells from the lCA accepting L. Thus only the prefix of length $n - n'$ acts as input to the lCA being simulated. If this simulation ends in an accepting state, the $NTVCA$ accepts its input. In other words, the $NTVCA$ accepts its input if there is some prefix of the input which is accepted by the lCA; ie. the $NTVCA$ accepts $\exists PRE(L)$. A time-space unrolling for such an $NTVCA$ is shown in Figure 8.2. Notice that the unique turn in the characteristic bit string of the computation path also serves to synchronise the cells in the prefix; an explicit firing squad synchronisation algorithm is not required after a prefix has been identified.
(c) This is shown in a manner similar to (b) above. To compute $\forall PRE(L)$, during the first n steps, the CA remains in a universal state, while a signal **S** travels from right to left. δ_1 propagates **S** while δ_2 puts the array into a configuration which uses a different copy of the state set. In this configuration, cells through which **S** has already passed go into a dummy state and do not affect the remaining computation. On the remaining cells, both δ_1 and δ_2 behave like the lCA accepting L. Thus if δ_2 is first used at time t, for $1 \leq t \leq n$, then the ensuing computations check whether $a_1\ldots a_t$ belongs to L. Since for the first n steps the CA is in a universal state, the input is accepted if and only if all such checks return a positive answer. (After n steps, the CA could be in a universal or an existential state; it does not

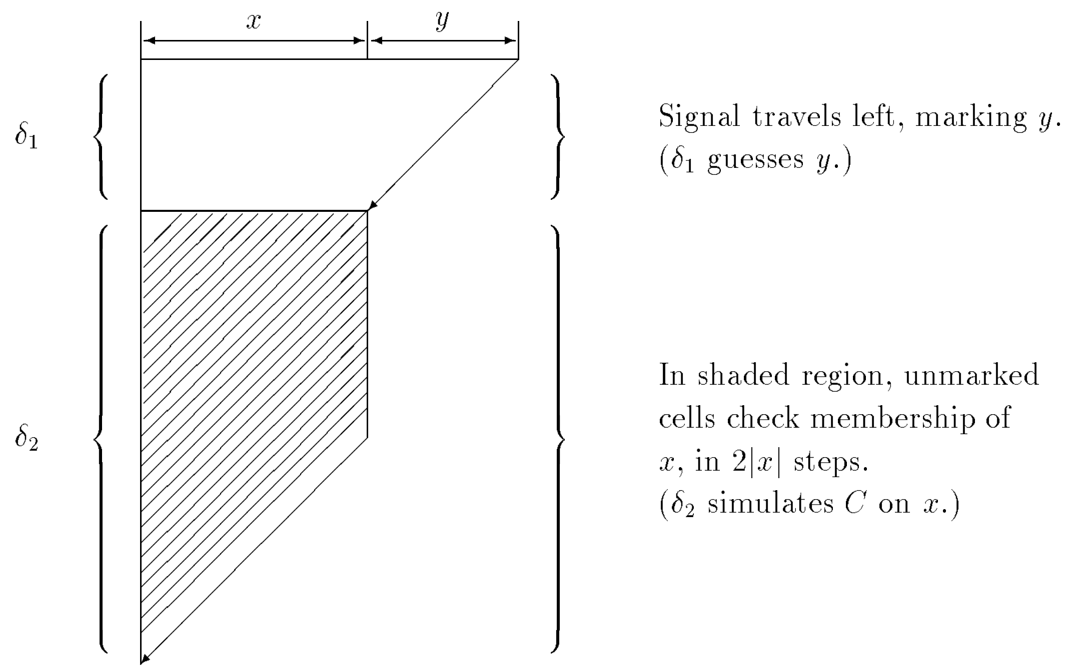


Figure 8.2: For $L \in lCA$, a 1-turn $lNTVCA$ accepting $\exists PRE(L)$

matter, since δ_1 and δ_2 are behaving identically.) Thus $\forall PRE(L)$ is accepted. The tree of computations that the ACA follows for accepting $\forall PRE(L)$, where L is an lCA language, is shown in Figure 8.3.

To accept $MIN(L)$, the same type of computation tree is generated. However, on all branches except the rightmost branch, ie. on all branches which check membership of a proper prefix of x, the lCA accepting $\overline{L}$ (lCA are closed under complementation) is simulated. Thus the ACA checks that no proper prefix of x belongs to L.

To accept $\oplus PRE(L)$, the ACA needs to check that an odd number of prefixes of the input x belong to L. This checking is done in two phases. In the first phase, which is existential, some cells of the input are marked. Let the marked cells be in positions $i_1, i_2, \ldots, i_k$, where $0 \leq k \leq |x|$. (The marking can be done as in the previous constructions, by sending a signal **S** and making only δ_1 mark a cell in the presence of this signal.) The second phase is a universal phase; all tests here must be satisfied. The tests conducted in this phase check that:

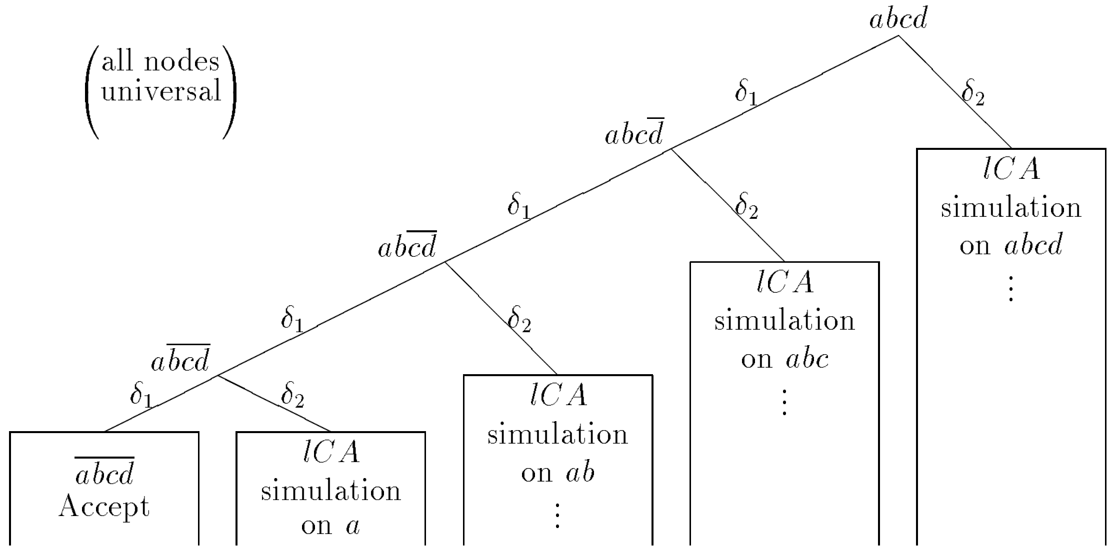


Figure 8.3: For $L \in lCA$, an $lACA$ accepting $\forall PRE(L)$

- k is odd.
- For each marked position i, the prefix $a_1a_2 \ldots a_i$ belongs to L.
- For each unmarked position j, the prefix $a_1a_2 \ldots a_j$ does not belong to L (it belongs to $\overline{L}$).

It is easy to see how these tests can be conducted on an ACA within linear time. Checking that k is odd is trivial; it can be done by a signal travelling across the array at unit speed. The other two checks are similar to the algorithm for recognising $MIN(L)$, the only difference being that some prefixes are being tested for membership in L and some others for membership in $\overline{L}$. In this fashion, $\oplus PRE(L)$ is accepted. ■

The next theorem states that prefixes bearing a fixed length ratio to the overall string can be tested in real or linear time, if L is real- or linear-time CA-testable respectively. We have been unable to find a similar construction for $rOCA$ languages.

Theorem 8.5 *If L is an rCA or an lCA language, then $PAD_{m,n}(L)$ is also an rCA or lCA language respectively.*

Proof: Let L be an rCA language. Consider the language $PAD_{1,1}(L)$. In Lemma 5.10 (b) we have seen that given an rCA, there is an equivalent rCA which, at time $2i-1$, indicates whether the prefix of length i is in L. The construction outlined there is for tally languages, but clearly it holds for non-tally languages as well (as seen in Theorem 8.3). If this rCA is slowed by one step initially, and also sends a signal from right to left at unit speed, then when the signal reaches the left end this cell is denoting membership of x in L, where the input is xy, $|x| = |y|$. So if the rCA accepts accordingly, it accepts exactly the language $PAD_{1,1}(L)$. Now this construction can be modified for any m, n as follows. Let C be an rCA accepting L, and let C', C'' be the rCAs constructed in Lemma 5.10. Then C'', which accepts $PAD_{1,1}(L)$, is obtained by slowing down the operation of C' so that every other step is an idle step. If C' is instead slowed down as follows:

Perform n steps of C'.

Idle for m steps.

then the prefix whose membership is determined in real time is of length $n/(m+n)$ of the total input; thus the resulting rCA accepts $PAD_{m,n}(L)$.

Now consider an lCA language L, accepted by C in linear time. Construct C' which sends signals from left and right inwards at speeds $1/m$ and $1/n$ respectively. The signals meet at a cell, marking a prefix x and suffix y such that $m|x| = n|y|$, as shown in Figure 8.4. From this meeting point, a firing squad algorithm can be initiated on the left portion. When all the cells in the prefix region are synchronised, they simulate C and check if x belongs to L. Clearly, the total time required is linear in L. ∎

$PAD_{1,1}(L)$ checks whether the first half of a given string is in L. On the other hand, $(1/2)L$ checks whether the given string is the first half of some string in L. This appears to be a far more difficult problem, and we have only been able to show its containment in linear-time $NTVCA$ when L is in lCA.

Theorem 8.6 *Let L be an lCA language. Then $(1/2)L$, $(1/3)L$ and $MID(1/3)(L)$ can be accepted in linear time by $NTVCA$.*

Proof: Let $L \subseteq \Sigma^+$, where $\Sigma = \{b_1, b_2, \ldots, b_k\}$. L is accepted by lCA C. We will describe a k-function linear-time $NTVCA$ C' accepting $(1/2)L$. This can be converted to a 2-function

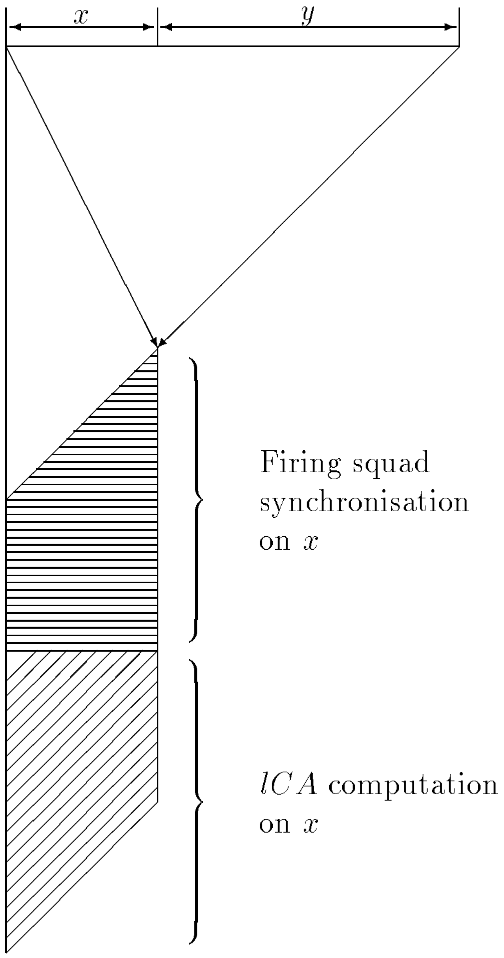


Figure 8.4: lCA accepting $PAD_{2,1}(L)$, where $L \in lCA$

$NTVCA$ with only a constant factor blowup in time, so the 2-function $NTVCA$ will also run in linear time.

Given input x, a string y of the same length as x is to be guessed, and then C simulated on xy. A firing squad synchronisation algorithm is initiated on the array. Simultaneously, a signal **S** travels across the array at unit speed. A cell holding **S** guesses the letter b_i if the transition function δ_i is used. A cell not containing **S** does nothing. Thus after n steps, n letters comprising the string y have been guessed. Simultaneously, the array fires, and C is simulated on xy. The string y is considered to be stored in the array in reverse in a second channel; thus it is as if a single array holding xy is folded in the middle. So the simulation of C can be performed with local neighbourhood information. If for some y, xy belongs to L, then there will be a computation which guessed this y, and it will end in an accepting state. Thus C' correctly accepts $(1/2)L$, and clearly, runs in linear time.

$NTVCA$s to accept $(1/3)L$ and $MID(1/3)(L)$ can be similarly constructed. Since now $2n$ letters are to be guessed, the signal **S** travels at half speed across the array. ■

Notice that in the above construction, the linear-time requirement is not violated even if we consider languages of the form

$$(m/n)L = \{x \mid \exists y,\, n|x| = m(|x| + |y|),\, xy \in L\}$$

for any positive integers m and n, $n > m$. All such languages derived from lCA languages can thus be accepted by $lNTVCA$.

The following result considers $\exists MID(L)$. This is a generalisation of $PAD_{1,1}(L)$; here a string x is padded on both sides by equal length strings, and x has to be checked for membership in L. For one-sided padding, Theorem 8.5 tells us that rCA and lCA languages are preserved. For two-sided padding, however, this does not appear to be the case.

Theorem 8.7 *If L is an rCA or lCA language, then $\exists MID(L)$ can be accepted in real time and linear time respectively by a 1-turn $NTVCA$.*

Proof: This result is shown in a manner similar to that in the proof of Theorem 8.4 (b). Namely, while δ_1 is being used, no actual computation is performed, but signals mark out suitable substrings of the input. Here signals travel at unit speed from both the left and

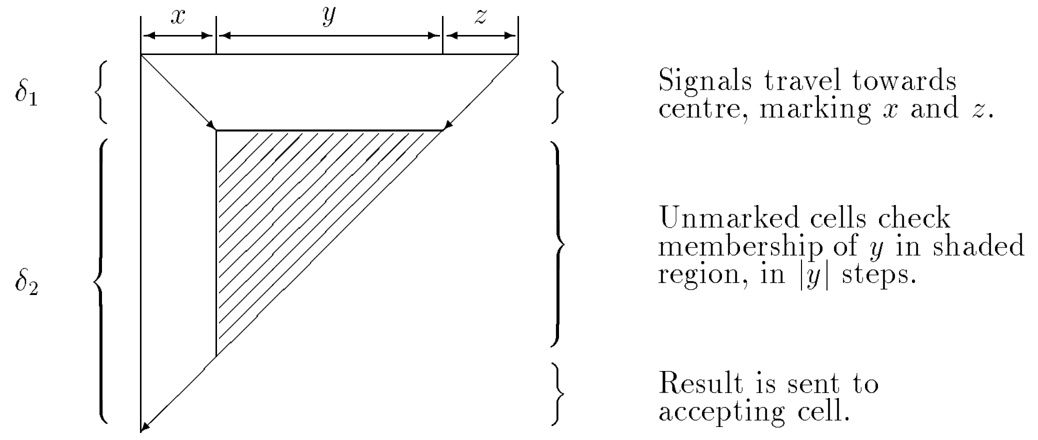


Figure 8.5: For $L \in rCA$, a 1-turn $rNTVCA$ accepting $\exists MID(L)$

right ends. So if the turn occurs at time $n' < n/2$, then a prefix and a suffix of length n' each are marked and discarded, and the rCA or lCA accepting L is subsequently simulated on the centred substring of length $n-2$n'. This simulation is performed using the transition function δ_2. The time-space unrolling for such $NTVCA$s is shown in Figures 8.5 and 8.6. ■

Theorem 8.8 *If L is an lCA language, then $(1/2)CYCLE(L)$ is also an lCA language. $CYCLE(L)$ can be shown to be a linear-time 1-turn $NTVCA$ language; it can also be accepted by a CA in $O(n^2)$ time.*

Proof: Let L be accepted by lCA C. Construct a new CA C' which, in the first $n/2$ steps, marks the midpoint of the array. Let the input be xy, where x and y are of equal length. In the next n steps, the middle cell synchronises the array (using firing squad synchronisation algorithms on both halves). In another n steps, xy is rewritten as yx, using the block-shifting algorithm described in [Smi72]. Simultaneously, a fast synchronisation algorithm (with "generals" at both ends) is run. Thus, when the shifting is complete, the array also fires, and C is simulated on the string yx. Thus membership of yx in L is checked in linear time. Hence $(1/2)CYCLE(L)$ is an lCA language.

To check $CYCLE(L)$, all possible points at which the input can be split must be checked. A 1-turn $NTVCA$ can use the first part of its computation, corresponding to δ_1, to mark

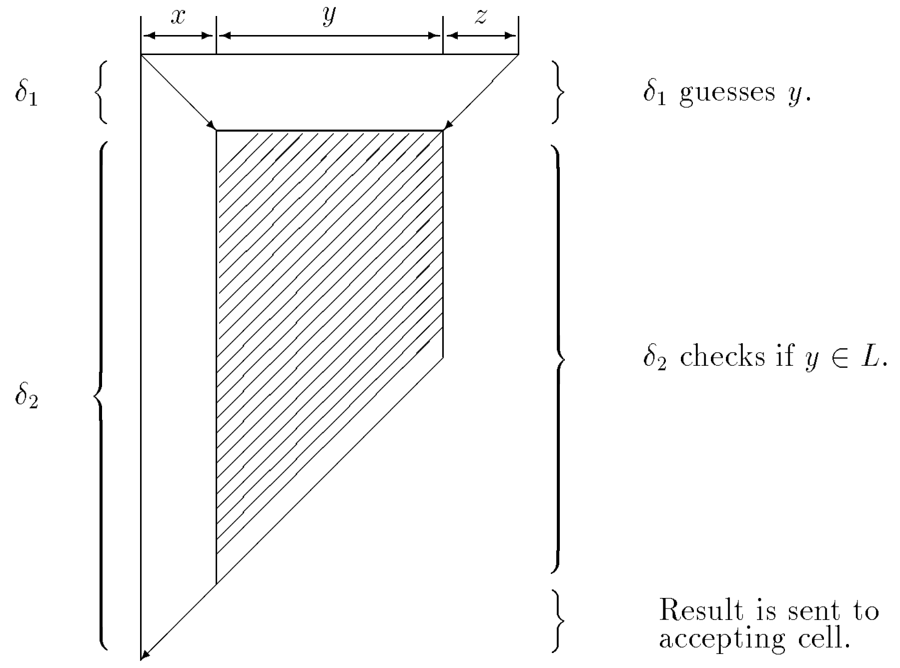


Figure 8.6: For $L \in lCA$, a 1-turn $lNTVCA$ accepting $\exists MID(L)$

the input as xy. Then, when the second part begins, it can deterministically swap x and y as above and test yx for membership in L.

The $O(n^2)$ CA algorithm is the straightforward brute-force method of systematically checking, for each i, whether $a_i \dots a_n a_1 \dots a_{i-1}$ belongs to L. ■

In the next result we improve Theorem 8.2 and show that it holds even when only one-way communication is considered.

Theorem 8.9 *If $L \in OCA$, then all the languages described above, except $INIT(L)$, $END(L)$, and $MAX(L)$, are also OCA languages.*

Proof: First consider the languages $\Box PRE(L)$, where $\Box \in \{\exists, \forall, \oplus\}$ and L is an OCA language accepted by C. For an OCA, the states of a cell do not depend on the computation to its right. Thus on input $a_1 a_2 \dots a_n$, if $c(i,1), c(i,2), \dots, c(i,t)$ are the states the i^{th} cell goes through, then these are the same states the cell will go through on input $a_1 a_2 \dots a_i$. So the i^{th} column in the time-space unrolling of the OCA gives the membership of the i^{th} prefix of the input in L.

To accept $\Box PRE(L)$, at every time instant, the leftmost cell sends a signal travelling right at unit speed. Simultaneously C is being simulated. Additionally, each cell records a $Y/N/?$ in a separate track, depending on whether it has seen an accept or reject state so far. Each signal, as it travels right, computes the $\Box$ operation on the contents of this track. The computation by any signal is aborted if a ? is encountered. At some time, no cell will have ? in this track (since OCA are closed under complementation). The signal sent out at this time will correctly compute $\Box PRE(L)$ and put the rightmost cell into an accepting/rejecting state. The signal can also check whether any proper prefix of x belongs to L; thus $MIN(L)$ can be accepted.
(If $\Box$ is $\exists$, then an easier construction is possible, whereby any cell on entering an accept state sends an accept signal rightwards.)

To show the closure of OCA under the remaining operations, we use the sweeping automaton (SA) model, which is a sequential machine characterisation for OCA (refer Lemma 2.6).

First let us consider $SHUFFLE(L)$, where L is accepted by SA M. Construct an SA M' which functions as follows: In sweep i, $i = 1, 3, \ldots, n-1$, M' merely records the input letter b_i on the first cell of the worktape. In sweep i, $i = 2, 4, \ldots, n$, M' simulates two sweeps of M on input letters b_i and b_{i-1}, in that order. After n sweeps, when \$ is being read, M' directly simulates M. Clearly, M' accepts $SHUFFLE(L)$. What is more, if M accepts L in $S(n)$ sweeps, then M' accepts $SHUFFLE(L)$in $S(n)$ sweeps, giving Corollary 8.10. Examples of the worktape profiles of M and M' on inputs of length 2 and 4 are shown in Figure 8.7.

Next consider $(1/2)CYCLE(L)$, where L is accepted by OCA C. Construct an SA M which functions as follows: While reading the input, M writes the input on its worktape, two letters per cell. It also records the midpoint of the input read so far, by moving a marker **B** one subcell right every other sweep. (A flag **F** in the leftmost cell can toggle in every sweep, indicating when **B** is to be shifted.) Additionally, it also places a $\star$ under the first letter.

When reading \$, in the first $n/2$ sweeps M moves $a_1 \ldots a_{n/2}$ to the region beyond a_n. This is done by moving $\star$ one subcell right in every sweep. When the $\star$ reaches the marker **B**, the shifting is over, and the $\star$ is erased. Now $a_{n/2+1} \ldots a_n a_1 \ldots a_{n/2}$ appears on the worktape

Input	Worktape	Input	Worktape
a_1	z_1^1	a_2	(a_2)
a_2	$z_1^2 \quad z_2^1$	a_1	$z_1^2 \quad z_2^1$

Input	Worktape	Input	Worktape
a_1	z_1^1	a_2	(a_2)
a_2	$z_1^2 \quad z_2^1$	a_1	$z_1^2 \quad z_2^1$
a_3	$z_1^3 \quad z_2^2 \quad z_3^1$	a_4	$z_1^2 \quad z_2^1$
			(a_4)
a_4	$z_1^4 \quad z_2^3 \quad z_3^2 \quad z_4^1$	a_3	$z_1^4 \quad z_2^3 \quad z_3^2 \quad z_4^1$
$\$$	$z_1^5 \quad z_2^4 \quad z_3^3 \quad z_4^2$	$\$$	$z_1^5 \quad z_2^4 \quad z_3^3 \quad z_4^2$
$\$$	$z_1^6 \quad z_2^5 \quad z_3^4 \quad z_4^3$	$\$$	$z_1^6 \quad z_2^5 \quad z_3^4 \quad z_4^3$
$SA\ M$ accepting L		$SA\ M'$ accepting $SHUFFLE(L)$	

Figure 8.7: SAs accepting L and $SHUFFLE(L)$

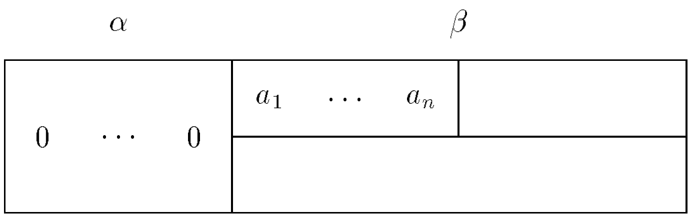
Input	Worktape					
a_1	$F\ \underset{*B}{a_1}\ .$					
a_2	$\overline{F}\ \underset{*B}{a_1}\ a_2$	..				
a_3	$F\ \underset{*}{a_1}\ \underset{B}{a_2}$	$a_3.$	..			
a_4	$\overline{F}\ \underset{*}{a_1}\ \underset{B}{a_2}$	a_3a_4	..	..		
a_5	$F\ \underset{*}{a_1}\ a_2$	$\underset{B}{a_3}\ a_4$	$a_5.$	..	..	
a_6	$\overline{F}\ \underset{*}{a_1}\ a_2$	$\underset{B}{a_3}\ a_4$	a_5a_6	..	..	..
\$	$X\ \underset{*}{a_2}$	$\underset{B}{a_3}\ a_4$	a_5a_6	$a_1.$	..	..
\$	$X\ X$	$\underset{*B}{a_3}\ a_4$	a_5a_6	a_1a_2	..	..
\$	$X\ X$	$\underset{B}{X}\ a_4$	a_5a_6	a_1a_2	$a_3.$	..
\$	$X\ X$	$\underset{B}{X}$	OCA Simulation		.	..
$\vdots$			$\vdots$			

Figure 8.8: SA accepting $(1/2)CYCLE(L)$

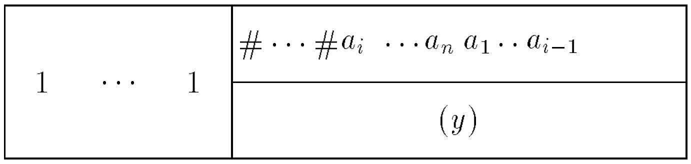
beyond the marker **B**. In subsequent sweeps, M directly simulates C on this string, and moves right into an accepting state if and only if C accepts the string. Thus, M accepts $(1/2)CYCLE(L)$. An example of the worktape profile of M is shown in Figure 8.8.

Note: Clearly, if C is an lCA, then M requires $7n/2$ sweeps for inputs of length n (n sweeps to read the input, $n/2$ sweeps to shift half of the input, and $2n$ sweeps to simulate C). So $(1/2)CYCLE(L)$ is also an lCA language. Thus this gives another proof of one part of Theorem 8.8. The rewriting of the input xy as yx, where $|x| = |y|$, can be done within the first n sweeps while the input is being read. This will save $n/2$ sweeps from the above construction. But this still does not allow us to make any stronger statement about how fast

(a) Worktape after reading the input:



(b) Worktape just before stage i:



(c) Worktape just after stage i begins:

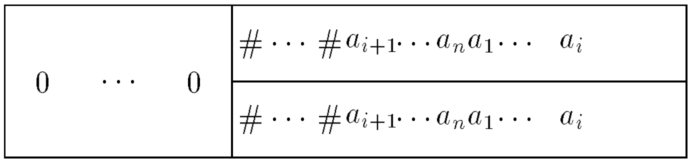


Figure 8.9: Worktape of an SA accepting $CYCLE(L)$

$(1/2)CYCLE(L)$ can be accepted when L is an rCA language.

Consider accepting $CYCLE(L)$. This is an extension of the above algorithm. While reading the input, the SA divides the worktape into two regions α and β, where α is an n length counter initialised to 0, and β is a 2-track region with $2n$ subcells, with the input string initially written on the first track in the first n of these subcells. See Figure 8.9 (a). Once the input is read, the SA operates in stages. The counter is incremented in every stage. Every time the counter overflows, a new stage begins.

Stage i begins with the first track of β holding $\#^{i-1}a_i \dots a_n a_1 \dots a_{i-1}$. See Figure 8.9 (b). In the first sweep of this stage (ie. when the region α has all 0s), # is written in place of a_i, and a_i is written beyond a_{i-1}, as shown in Figure 8.9 (c). The contents of the first track are also copied onto the second track. In subsequent sweeps, C is simulated on the string $a_{i+1} \dots a_n a_1 \dots a_i$ on the second track (string (y)) of β. The counter α can be large enough so that this simulation is completed before it overflows again. If the simulation ends in an accepting state, then the SA M also moves right in an accepting state. Otherwise, when the counter next overflows, it moves on to stage $i+1$. Thus the SA systematically checks all cyclic shifts of the input for membership in L; thus it accepts $CYCLE(L)$.

The algorithms for $(1/2)L$, $(1/3)L$ and $MID(1/3)(L)$ are fairly similar. Here we will only describe the SA accepting $(1/2)L$. Given an input x, the SA systematically generates all strings y of the same length as x, in lexicographic order. For each such y generated, xy is tested for membership in L. Without loss of generality, assume that $L \subseteq \{0,1\}^+$. Clearly, there are $2^{|x|}$ candidates for y. For each such y, the membership of xy in L can be determined by the OCA accepting L in $k^{|xy|} = k^{2|x|} = c^{|x|}$ steps, for some constants k and c. So the SA should generate successive values of y at least $c^{|x|}$ sweeps apart. This is ensured by creating a $|x|$ length c-ary counter α in the initial region of the worktape, as shown in Figure 8.10. Every time the counter overflows, the lexicographically next string y is generated. In subsequent sweeps before the counter overflows again, the OCA accepting L is simulated directly on xy, in a separate track γ. If, for any choice of y, the OCA simulation results in an accepting state, then the SA moves right in an accepting state. Clearly, this SA accepts $(1/2)L$.

Now consider accepting $PAD_{m,n}(L)$. Construct a SA which, while reading its input, also marks out the prefix which is of length $n/(m+n)$ of the total input. (This can be done by setting a fixed-length counter in the initial part of the worktape to count upto $m+n$.) When the entire input is read, the SA can directly simulate the OCA in the marked prefix. This SA will thus accept $PAD_{m,n}(L)$.

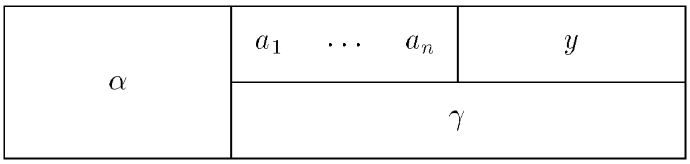


Figure 8.10: Worktape of an SA accepting (1/2)(L)

It remains now to show that for $L \in OCA$, $\exists MID(L)$ is also an OCA language. Note that if L is an lCA language, then $\exists MID(L)$ can be accepted by a 1-turn linear-time $NTVCA$, whereas $MID(1/3)(L)$ requires a linear-time $NTVCA$. It would thus appear that $\exists MID(L)$ is an easier language operation than $MID(1/3)(L)$. However, when $L \in OCA$, the construction for $\exists MID(L)$ appears to be more complex. This is because if the input string is written as xyz where y belongs to L, the OCA or SA must check that x and z are of the same length. It must also do this check for all such y. An SA which correctly does this is described below. Let C be the OCA accepting L.

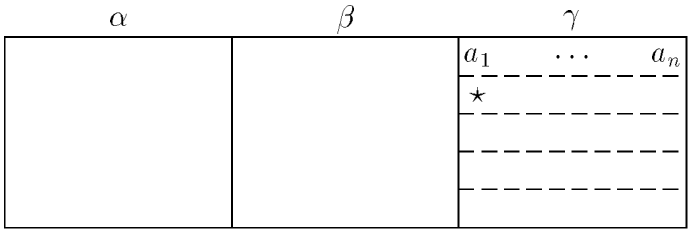
While reading the input, the SA packs and shifts the input in such a way that when the entire input is read, the worktape is partitioned into 3 regions, α, β, γ. α and β are counters of length n, initialised to zero. γ is a 5-track region with n subcells; the input string is written on the first track, with a $\star$ on the second track under the first letter. The worktape at this point appears as shown in Figure 8.11 (a).

α is incremented in every sweep. β is incremented whenever α overflows. Every time β overflows, the $\star$ is moved one subcell right. Then, until α overflows for the first time, C is simulated on the string to the right of (including) the $\star$ position (let the $\star$ be under a_i), in the third track. If any subcell in this simulation enters an accept/reject state, this is recorded in the fourth track. Thus when α overflows again, the fourth track records all $j \geq i$ such that the substring $a_i \dots a_j$ belongs to L. See Figure 8.11 (b).

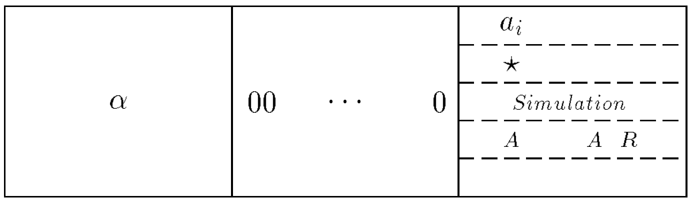
Now, for each such j, the SA must check whether $i-1 = n-j$. This is done by placing two pointers $\dagger$ in track 5, beneath a_1 and a_{j+1}, and moving these one subcell right in each sweep. In these sweeps, the SA checks whether the characters in the first track of the subcells marked by the $\dagger$s are identical. See Figure 8.11 (c). If the pointers simulatneously reach a_i and the end of the string respectively, then a witness to the input being in $\exists MID(L)$ has been found, and the SA accepts its input.

Otherwise, the SA should go on to checking the next value of j. Since information cannot be carried backwards in the SA, the SA does not know exactly when checking a particular j is over. So it just waits until α overflows again before checking for the next j. This allows more than enough time for checking; thus quite a few sweeps of the SA are dummy sweeps. There are at most n values of j to be checked. So by the time β overflows, all checks have

(a) Worktape after reading the input:



(b) Worktape during simulation:



(c) Worktape while checking whether a *middle* string has been found:

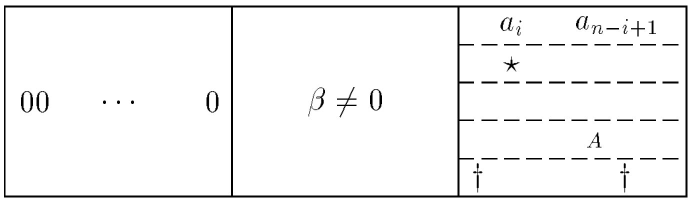


Figure 8.11: Worktape of an SA accepting $\exists MID(L)$

been completed and the SA can begin simulating C on the string beginning from position $i+1$. ■

Corollary 8.10 *If L is an rCA or lCA language, then $SHUFFLE(L)$ is also an rCA or lCA language respectively.*

8.3 Conclusions

In this chapter, we have considered the closure of some language classes defined by cellular automata. The results are summarised in table 8.1. As can be seen, there are quite a few operations for which tight results are not known for $rOCA$ and rCA. (Of course, the class containing the closure of a higher class, say lCA, under that operation, also gives a bound on the closure of the smaller class.) It would be of interest to resolve these questions. We feel that some of these closures are related to the open problem of whether lCA are more powerful than rCA. For instance, we believe that rCA are closed under $(1/2)CYCLE(L)$ only if rCA and lCA have the same power. Finding a proof of this conjecture is a problem worth investigating.

In chapters 6 and 7 of this thesis, some new models of computation, namely nondeterministic, probabilistic and alternating time-varying computation, on cellular arrays, have been defined. These generate classes lying between rCA and OCA. For some of the operations listed above, namely $CYCLE(L)$, $(1/2)L$, $(1/3)L$, $MID(1/3)L$, we have shown that the closure of lCA is contained in some of these classes. This gives some indication of the power of the new computation models. It seems an interesting problem to further tighten such results, in an effort to characterise, more precisely, the power of such nondeterministic classes. Another problem worth studying is finding the closures, under various language operations, of the classes defined by these new models of computation.

Chapter 9

Conclusions

This thesis introduces $TVCA$, which are CA augmented by an external control device, and examines the language recognition capabilities of such CA. The external control and time-variation of the $TVCA$ has been interpreted in various ways — as an oracle answering queries made implicitly by relativised CA, as a model of nondeterministic computation, and as models of probabilistic and alternating computation. Under these different interpretations, a whole lot of new language classes have arisen, lying in between the known CA and $NSPACE(n)$ classes. The inter-relations amongst these classes are shown in Figures 3.8, 7.1 and 7.2.

A major contribution of this work has been to provide new different approaches to solve the $rCA \stackrel{?}{=} lCA \stackrel{?}{=} OCA \stackrel{?}{=} CA$ questions. These questions themselves remain unsolved, but now, proving proper containments can be reattempted via studying the power of the oracle access mechanism or the mode of nondeterminism defined by $TVCA$. Since direct simulations or counterexamples have not hitherto provided a solution, such indirect solutions seem to be called for; we are hopeful that through such techniques these problems will soon be solved.

From a practical viewpoint, $TVCA$ could help in making $VLSI$ implementations of CAs even more easy. For instance, consider the class $rrCA_1$. Languages in this class are all contained in lCA. However, the two rCAs involved in the $rrCA_1$ automaton may both be considerably simpler, or have fewer states, than the one equivalent lCA. In such a case, if a control signal from one rCA (the controlling or "oracle" rCA) can be globally broadcast to another rCA (the time-varying or relativised CA), then it may be easier to implement two rCAs rather than one lCA. Studying the feasibiltiy of such schemes will require a careful examination of the state-space trade-off between $rrCA_k$ and lCA languages, and between $llCA_k$ and OCA languages. Such an investigation is beyond the scope of this thesis; it merits a detailed independent investigation.

From a purely language-theoretic viewpoint, exactly characterising the complexity of the newly defined language classes should be of considerable interest. The problem of whether $k+1$ controlling languages are better than k for real-time $TVCA$ computation, raised in chapter 3, has been left open. Solving this problem will give more insight into the exact nature and the complexity of real-time $TVCA$ computation. We have also left open the problem of defining suitable reducibilities and finding languages complete for $TVCA$ classes. Finding such languages will considerably aid a more systematic investigation into the power of these classes.

The study of closure properties, dealt with in chapter 8, is again of great interest from the language-theoretic point of view. Obtaining tighter bounds on the results obtained there, and also investigating closure under more language operations, could lead to interesting insights into the containment problems.

Bibliography

[BC84] W. Bucher and K. Culik II. On real-time and linear-time cellular automata. *R.A.I.R.O. Informatique theoretique*, 18(4):307–325, 1984.

[BDG88] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*, volume 11 of *EATCS Monograph Series*. Springer-Verlag, Berlin, 1988.

[BDG90] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*, volume 22 of *EATCS Monograph Series*. Springer-Verlag, Berlin, 1990.

[Boo74] R. V. Book. Tally languages and complexity classes. *Information and Control*, 26:186–193, 1974.

[Bus88] J. F. Buss. Relativized alternation and space-bounded computation. *Journal of Computer and System Sciences*, 36:351–378, 1988.

[CC84] C. Choffrut and K. Culik II. On real-time cellular automata and trellis automata. *Acta Informatica*, 21:393–409, 1984.

[CGS84a] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata Part I. *International Journal of Computer Mathematics*, 15:195–212, 1984.

[CGS84b] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata Part II. *International Journal of Computer Mathematics*, 16:3–22, 1984.

[CGS86] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata: stability, decidability and complexity. *Information and Control*, 71:218–230, 1986.

[CIV88] J. H. Chang, O. H. Ibarra, and A. Vergis. On the power of one-way communication. *Journal of the ACM*, 35(3):697–726, July 1988.

[Cod68] E. F. Codd. *Cellular Automata.* ACM Monograph Series. Academic Press, New York, 1968.

[Col69] S. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, C-18(4):349–365, 1969.

[DGT85] J. Demongeot, E. Goles, and M. Tchuente. *Dynamical Systems and Cellular Automata.* Academic Press, New York, 1985.

[DP88] A. K. Das and P. Pal Chaudhari. An efficient on-chip deterministic test pattern generation scheme. In *Proceedings of the 2nd International Workshop on VLSI Design*, pages 250–266, December 1988.

[Dye80] C. Dyer. One-way bounded cellular automata. *Information and Control*, 44:261–281, 1980.

[Gin66] S. Ginsburg. *The Mathematical Theory of Context-Free Languages.* McGraw-Hill Inc., 1966.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[Iba91] O. H. Ibarra. On resetting *DLBA*s. *SIGACT NEWS*, 22(1), 1991. also in EATCS Bulletin 44:190-191, June 1991.

[IJ87] O. H. Ibarra and T. Jiang. On one-way cellular arrays. *SIAM Journal of Computing*, 16(6):1135–1154, December 1987.

[IJ88] O. H. Ibarra and T. Jiang. Relating the power of cellular arrays to their closure properties. *Theoretical Computer Science*, 57:225–238, 1988.

[IK84] O. H. Ibarra and S. M. Kim. Characterizations and computational complexity of systolic trellis automata. *Theoretical Computer Science*, 29:123–153, 1984.

[IKM85] O. H. Ibarra, S. M. Kim, and S. Moran. Sequential machine characterizations of trellis and cellular automata and applications. *SIAM Journal of Computing*, 14:426–447, 1985.

[IKP86] O. H. Ibarra, S. M. Kim, and M. Palis. Designing systolic algorithms using sequential machines. *IEEE Transactions on Computers*, C-35(6):531–542, June 1986. also in *Proceedings of the IEEE Conference on Foundations of Computer Science* (1984).

[IPK85a] O. H. Ibarra, M. Palis, and S. M. Kim. Fast parallel language recognition by cellular automata. *Theoretical Computer Science*, 41:231–246, 1985.

[IPK85b] O. H. Ibarra, M. Palis, and S. M. Kim. Some results concerning linear iterative (systolic) arrays. *Journal of Parallel and Distributed Computing*, 2:182–218, 1985.

[KA87] M. Khare and A. Albicki. Cellular automata used for test pattern generation. In *Proceedings of the International Test Conference*, pages 56–59, September 1987.

[KD84] K. Krithivasan and A. Das. Treating terminals as function values of time. In *Proceedings of the 4th FST&TCS conference*, pages 188–201. Springer-Verlag, 1984. LNCS 181.

[KD85] K. Krithivasan and A. Das. Terminal weighted grammars and picture description. *Computer Vision, Graphics and Image Processing*, 30:13–31, 1985.

[KD86] K. Krithivasan and A. Das. Time-varying finite automata. *International Journal of Computer Mathematics*, 19:103–123, 1986.

[KM90] S. M. Kim and R. McCloskey. A characterization of constant-time cellular automata computation. *Physica D*, 45:404–419, 1990.

[KS88] K. Krithivasan and V. Srinivasan. Time varying pushdown automata. *International Journal of Computer Mathematics*, 24:223–236, 1988.

[Kun79] H. T. Kung. Let's design algorithms for VLSI systems. In L.Seifz, editor, *Proceedings of the Caltech Conference on VLSI*, pages 65–90, Pasadena, California, 1979.

[Kun80] H. T. Kung. *Advances in Computers*, volume 19, chapter The structure of parallel algorithms, pages 65–112. Academic Press, 1980.

[Kun82] H. T. Kung. Why systolic architectures. *Computer magazine*, January 1982. special issue on Highly Parallel Computing.

[LL76] R. Ladner and N. Lynch. Relativization of questions about log-space reducibility. *Mathematical Systems Theory*, 10:19–32, 1976.

[LM68] G. C. Langdon and F. R. Moore. A generalised firing squad problem. *Information and Control*, 12:212–220, 1968.

[Nas79] M. Nasu. Indecomposable local maps of tessellation automata. *Mathematical System Theory*, 13:81–93, 1979.

[Neu66] John von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966. edited and compiled by Arthur W. Burks.

[PTC86] W. Pries, A. Thanailakis, and H. C. Card. Group properties of cellular automata and VLSI applications. *IEEE Transactions on Computers*, C-35(12):1013–1024, 1986.

[RST84] W.L. Ruzzo, J. Simon, and M. Tompa. Space-bounded hierarchies and probabilistic computations. *Journal of Computer and Systems Sciences*, 28:216–230, 1984.

[Sal73] A. Salomaa. *Formal Languages.* Academic Press, 1973.

[Smi71] A. R. Smith III. Cellular automata complexity trade-offs. *Information and Control*, 18:466–482, 1971.

[Smi72] A. R. Smith III. Real-time language recognition by one-dimensional cellular automata. *Journal of Computer and System Sciences*, 6:233–253, 1972.

[Smi76] A. R. Smith III. Introduction to and survey of polyautomata theory. In A. Lindenmayer and G.Rozenberg, editors, *Automata, Languages and Development*, pages 405–422. North Holland, 1976.

[SW83] R. Sommerhalder and S. C. van Westrhenen. Parallel language recognition in constant time by cellular automata. *Acta Informatica*, 19:397–407, 1983.

[UMS82] H. Umeo, K. Morita, and K. Sugata. Deterministic one-way simulation of two-way real-time cellular automata and its related problems. *Information Processing Letters*, 14:158–161, 1982.

[Wak66] A. Waksman. An optimum solution to the firing squad synchronization problem. *Information and Control*, 9:66–78, 1966.

[Wol86] S. Wolfram. *Theory and Applications of Cellular Automata.* World Scientific, Singapore, 1986.

Publications by the author related to the thesis

1. M. Mahajan and K. Krithivasan. Some results on time-varying and relativised cellular automata. *International Journal of Computer Mathematics*, 43(1&2):21-38, 1992.

2. M. Mahajan and K. Krithivasan. Relativised cellular automata and complexity classes. presented at the *National Seminar on Theoretical Computer Science* July 1991 (Madras). also, in *Proceedings of the* 11^{th} *International FST&TCS Conference, LNCS 560*, pages 172-185, December 1991 (New Delhi).

3. M. Mahajan and K. Krithivasan. Languages classes defined by time-bounded relativised cellular automata. *R.A.I.R.O. Theoretical Informatics and Applications*, Vol. **27** (5) (1993), pp. 403–432.

4. K. Krithivasan and M. Mahajan. Nondeterministic, probabilistic and alternating computations on cellular array models. to appear in *Theoretical Computer Science*. preliminary version presented at the *Developments in Language Theory Conference*, Turku, Finland, 12–15 July 1993.

5. M. Mahajan and K. Krithivasan. Language operations on cellular automata classes. *Journal of Mathematical and Physical Sciences*, Vol. **27** (1993).