Structurally Dynamic Cellular Automata

by

Stephen Michael Majercik

A thesis submitted to the Department of Computer Science and the faculty of the University of Southern Maine in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

July 22, 1994

©Stephen Michael Majercik 1994

The University of Southern Maine
School of Applied Science

8 August 1994

We hereby recommend that the thesis of Stephen Michael Majercik, entitled *Structurally Dynamic Cellular Automata*, be accepted as partial fulfillment of the requirements of the Degree of Master of Science in Computer Science.

Committee:

_____
Chairperson

_____

_____

Accepted:

_____
Dean, School of Applied Science

Acknowledgments

I would like to thank Steve Fenner, my thesis advisor, for generously sharing his time and ideas. His patience while I searched for a thesis topic and his inspiration after I found one were immeasurably helpful.

I would like to thank David Briggs and Bob Boothe for serving on my thesis committee. Their excellent suggestions were invaluable.

I would also like to thank Charles Welty, John Heath, Bruce MacLeod, and Rukmini Vijaykumar. I greatly appreciate the knowledge and enthusiasm each of them brought to lectures in the classroom and discussions outside the classroom.

Finally, I would like to thank my wife, Faith, for her loving support and understanding, even during those times when she felt like a computer widow. She has helped make this thesis a reality in uncountable ways, for which I am grateful.

Abstract

We describe structurally dynamic cellular automata (SDCA), cellular automata with dynamic cell links first proposed by Ilachinski and Halpern as a promising model for the simulation and study of naturally occurring processes. We introduce three new models of SDCA, defining these models precisely as models of computation rather than as tools with which to model physical processes, and we prove some results that begin to delineate their computational capabilities and their relationship to conventional cellular automata. We show a partial simulation hierarchy of SDCA classes and prove that SDCA are faster than conventional cellular automata. Finally, we describe a CA-universal SDCA, an SDCA that is capable of simulating any conventional cellular automaton of the same dimension. The speed of our simulator represents a significant improvement on a previous result which constructed a CA-universal conventional cellular automaton.

Contents

1 Introduction **1**

1.1 Cellular Automata 1

1.2 Structurally Dynamic Cellular Automata 2

1.3 Overview of Thesis 3

2 Background **8**

3 Notation and Definitions **13**

3.1 Preliminaries 13

3.2 Conventional Cellular Automata 14

3.3 Structurally Dynamic Cellular Automata 15

3.3.1 The Ilachinski and Halpern Model 15

3.3.2 Problems with the Ilachinski and Halpern Model 19

3.3.3 Alternative SDCA Models 23

4 Results **36**

4.1 Introduction 36

4.2 The Hierarchy of SDCA Models 37

4.3 Comparison of CA and SDCA 44

5 An FLLL CA-Universal SDCA **53**

5.1 Introduction and Overview 53

5.2 Preliminary Conversions 56

5.3 The Block Level 59

5.4 The Cellular Level 61

5.5 Simulation of Evolution 65

5.6 Construction of Simulation Module Groups and Rule Tables 70

5.6.1 Simulation Module Group Construction 70

5.6.2 Rule Table Construction 73

5.7 Calculation of Simulation Module Group Size 76

6 Conclusions and Further Work **80**

List of Figures

4.1 Simulation of automaton A by automaton B 40

5.1 Configuration of groups simulating nonquiescent cells 60

5.2 Groups simulating nonquiescent cells plus idle simulation module group 61

5.3 Available links in a simulator cell 61

5.4 The neighborhood path 62

5.5 Two groups of three operational simulation modules 63

5.6 An operational binary tree in a rule table 64

5.7 Construction segment for a simulation module group 71

5.8 Construction segment for a rule table binary tree 74

5.9 Intermediate stage of rule table binary tree construction 75

5.10 Completed rule table binary tree 75

Chapter 1

Introduction

1.1 Cellular Automata

A cellular automaton (CA) is a model of parallel computation or, more generally, a model of a complex system, in which a possibly infinite d-dimensional array of finite state automata (FSA) updates itself in parallel according to some homogeneous local transition rule. This transition rule is the same for every FSA, or *cell*, in the array and operates on information local to the cell to which it is being applied. The rule determines the state of a given cell in the next time step by looking at the state of that cell and the states of a finite, local collection of *neighbors* of that cell. The size and configuration of this neighborhood is fixed through time and is the same for each cell in a given CA, but different CA may use different neighborhoods. For example, Conway's Game of Life [3], a well-known 2-dimensional CA, uses the 9-neighbor Moore neighborhood, consisting of the cell itself and the eight adjacent cells in the horizontal, vertical, and diagonal directions. The von Neumann neighborhood, in contrast, is a 5-neighbor neighborhood consisting of the cell itself and the adjacent

cells in the horizontal and vertical directions. Thus, the state of each cell in the CA at time $t+1$ is a function of the state of that cell and its neighboring cells at time t.

Cellular automata, particularly 1-dimensional cellular automata, are similar to Turing machines in some respects. A CA is divided into cells similar to the tape squares of a Turing machine, and both models of computation allow only a finite set of cell or tape symbols. But there are some obvious differences. Cellular automata have no distinguished tape head; computation proceeds in parallel across the entire array. And cellular automata do not in general have a terminal alphabet or a distinguished halting state. In what follows, we will always restrict our cellular automata to a finite number of state changes at any given time step; given this restriction, Turing machines can clearly simulate cellular automata. And, since cellular automata can simulate Turing machines (Smith [19] has shown that any Turing machine with m symbols and n states can be simulated by a 1-dimensional CA with $m+2n$ states), these two models of computation are equivalent. Cellular automata are thus capable of universal computation; we merely embed a universal Turing machine in a CA. But cellular automata can compute directly, without an embedded Turing machine. In fact, for some language recognition problems, cellular automata operating without an embedded Turing machine, can recognize the language in a time strictly faster than any Turing machine [19].

1.2 Structurally Dynamic Cellular Automata

In [21], Ilachinski and Halpern introduced a new type of cellular automaton: the structurally dynamic cellular automaton (SDCA) "in which binary [0,1] *value-con-*

figurations and the (conventionally quiescent) underlying *topological structure* are dynamically coupled." The essential, new feature of this model was that the connections between the cells in the CA are allowed to change. This means that given certain conditions links between cells may be created or destroyed; the neighborhood of each cell is now dynamic rather than fixed throughout the automaton. Thus, the state and link configurations of an SDCA are *both* dynamic and are continually altering each other.

Ilachinski and Halpern intended the SDCA model to be an improvement over conventional CA in simulating various natural processes. They suggested that SDCA would be useful for modeling crystal growth, pattern formation, and neural networks. We feel that SDCA would also be particularly appropriate for modeling human brain function. Relatively distant neurons in the brain can be directly connected, and experimental evidence indicates that the brain encodes new experiences by performing the functional equivalent of "rewiring" parts of itself [2]. The dynamic connection structure of an SDCA, and particularly its ability to dynamically forge links between cells which are arbitrarily far apart, along with the potential to *self-organize* (shared with conventional CA), would seem to make the SDCA model an especially attractive candidate for modeling the human brain.

1.3 Overview of Thesis

Although Ilachinski and Halpern are clear about their intention to create a cellular automaton model in which both the cell values *and* the topology are dynamic, their definition of the model is not as clear. Certain issues, critical from a computation

theoretic standpoint, are left unaddressed or vague. Given that they were primarily motivated by a desire to construct a model that would allow more realistic simulations of naturally occurring processes, it is not surprising that the construction of their model appears to have been governed more by a physical intuition of what would be useful than by concerns for mathematical precision.

The purpose of this thesis is to define SDCA more precisely as a model of computation rather than as a tool with which to model physical processes, and to prove some results that delineate some of the computational capabilities of SDCA and their relationship to conventional CA.

In Chapter 2 we describe previous work in the area of cellular automata.

In Chapter 3 we review the conventional CA model and the Ilachinski and Halpern SDCA model. We point out some weaknesses of the Ilachinski and Halpern SDCA model and formulate a number of alternative SDCA models. Our alternative models differ primarily in the number of links each cell is allowed and in the type of neighborhood information the local transition functions are allowed to have:

- The Unbounded Links Relative Location (ULRL) Model allows an unbounded, though finite, number of links for each cell, and the transition functions have access to the state and exact *relative location* of each cell in the neighborhood.
- The Bounded Links Relative Location (BLRL) Model puts a bound on the number of links for each cell. The neighborhood information is the same as in the ULRL Model.
- The Unbounded Links Labeled Links (ULLL) Model allows an unbounded,

though finite, number of *labeled links* for each cell. The label set is infinite, and the transition functions have access to the state of each neighbor and the label of the link connecting it to the central cell.

- The Bounded Links Labeled Links (BLLL) Model puts a bound on the number of labeled links for each cell. The label set is still infinite and the neighborhood information is the same as in the ULLL Model.

- The Finite Labels Labeled Links (FLLL) Model puts a bound on the number of labeled links for each cell *and* a bound on the size of the label set. The neighborhood information is the same as in the ULLL Model.

- The Unbounded Links Symmetric (ULS) Model allows an unbounded, though finite, number of links for each cell, and the transition functions have access only to the distribution of states in the neighborhood, *i.e.*, how many cells in the neighborhood are in each possible state.

- The Bounded Links Symmetric (BLS) Model puts a bound on the number of links for each cell. The neighborhood information is the same as in the ULS Model.

These models are presented in considerable detail, establishing precisely what we mean by an SDCA of each type.

In Chapter 4, we will for the most part avoid this level of detail, presenting somewhat less formally the proofs of our results concerning the computational capabilities of these models. We will present results that identify a partial simulation hierarchy among the SDCA models we have defined.

We will prove an exponential lower bound on the slowdown involved when a conventional 1-dimensional CA attempts to simulate an arbitrary 1-dimensional SDCA. Specifically, we show that the Firing Squad Synchronization Problem, which requires linear time on a conventional CA, can be solved in log time on an SDCA:

Theorem. *The Firing Squad Synchronization Problem for n cells can be solved by a ULS SDCA in time $\lceil \log n \rceil + 1$ using $n - 1$ links for each cell.*

We will show that this lower bound is not tight and that a conventional CA can simulate a BLRL SDCA or a BLS SDCA with no loss of time:

Theorem. *Given a 1-dimensional BLRL or BLS SDCA A with k states and a maximum neighborhood radius r, there is a conventional 1-dimensional CA B with $2^{2r}k$ states and neighborhood radius $2r$ that simulates A with no loss of time.*

We will show that, given a 1-dimensional, conventional, nearest-neighbor CA, a ULRL SDCA can simulate this CA with a speedup of 2:

Theorem. *Given an arbitrary 1-dimensional conventional CA A with neighborhood radius 1, there exists a 1-dimensional ULRL SDCA B which can simulate A with a speedup of 2.*

In Chapter 5 we will show that even an FLLL SDCA, a much more restricted SDCA model, can operate as a universal CA simulator, although with some slowdown:

Theorem. *There exists a 1-dimensional FLLL SDCA which can simulate an ar-*

bitrary k-state 1-dimensional conventional CA with neighborhood radius r with $\mathcal{O}(k^{2r}\sqrt{2r\log k})$ *slowdown.*

This slowdown represents a significant improvement over a previous result [1].

Finally, in Chapter 6 we will summarize our results and identify areas for further investigation.

Chapter 2

Background

In 1948 von Neumann [32] introduced cellular automata for the purpose of investigating the reproductive capabilities of digital machines. He constructed a complicated 29-state cellular automaton, later simplified by Codd [4], which was capable of both *self-reproduction* and *universal computation*. Self-reproduction in this context refers to the ability of a cellular automaton to reproduce its own finite, initial configuration of cell states at another unoccupied location in the array of cells in which the cellular automaton is embedded. Universal computation refers to the ability of a cellular automaton to carry out any effective computation given an appropriate initial configuration. Current interest in cellular automata began with a paper by Wolfram [33] in 1983 in which he proposed using cellular automata to investigate naturally occurring systems which seem to evolve from disordered states to more ordered states. In the introduction to this paper, Wolfram summarized the characteristics of CA which make this model of computation attractive for such work:

> Cellular automata are sufficiently simple to allow detailed mathematical analysis, yet sufficiently complex to exhibit a wide variety of complicated phenomena. Cellular automata are also of sufficient generality to provide

> simple models for a very wide variety of physical, chemical, biological, and other systems. The ultimate goal is to abstract from a study of cellular automata general features of "self-organizing" behavior and perhaps to devise universal laws analogous to the laws of thermodynamics.

Also, as noted in [17], the dynamics of CA respect certain properties of naturally occurring systems:

- Information travels a finite distance in a finite amount of time.
- Both the transition rules in a CA and the laws of physics in naturally occurring systems are independent of one's position in the system.

One of the primary motivations for studying CA has therefore been to improve our understanding of complex natural systems. Since von Neumann, CA have been used to model many complex physical and biological processes [27, 24, 26]. They have been used to investigate issues surrounding the origins of life [23, 24, 26] and to design a reliable parallel computer [8]. In a more speculative vein, it has been suggested that quantum fields may actually be cellular automata [31] and that future computers may be large crystals of smart molecules grown artificially, each crystal operating like a cellular automaton [7]. If CA really are good models of these natural systems, then to the extent that CA can be divided into relatively few classes (according to some quantitative or qualitative measures) and to the extent that we understand the structure of this CA space and why these classes exist, then our study of CA may allow us to draw some very general conclusions about the nature and behavior of complex natural systems.

Wolfram was the first to address the issue of classification of CA. In [35], he described an extensive experimental study of 1-dimensional cellular automata which suggested that all 1-dimensional cellular automata could be divided into four classes depending on their asymptotic qualitative behavior on arbitrary random initial configurations:

- Class 1 automata exhibit *fixed point* behavior. A Class 1 automaton exhibits no dependence on initial conditions and, given any initial configuration, a Class 1 automaton evolves (usually rather quickly) to a stable configuration (a configuration which never changes).
- Class 2 automata exhibit *periodic* behavior. A Class 2 automaton exhibits limited dependence on initial conditions and, given any initial configuration, ultimately repeats a fixed cycle of configurations.
- Class 3 automata exhibit *chaotic* behavior. A Class 3 automaton exhibits the sensitive dependence on initial conditions typical of deterministic chaos and, given any initial configuration, never reaches a repeating cycle of configurations, displaying pseudo-random behavior.
- Class 4 automata, unlike the first three classes, do not have a clear analogue in dynamical systems theory. Their asymptotic behavior is characterized by complex localized structures (configurations of states) which may propagate. Conway's Game of Life is an example of a Class 4 two-dimensional automaton.

The classes can be arranged hierarchically according to the degree to which any structures (local configurations of states) are preserved and the information contained

in these structures is propagated. Class 1 CA preserve structures extremely well but do not propagate the information contained in these structures at all. Class 2 CA preserve structures well (although not as rigidly as Class 1 CA) and may propagate the information contained in these structures over a limited area (during the course of their periods). Class 3 CA do not preserve structures or propagate their information at all. They propagate any information indiscriminately in all directions, thereby destroying structures and the information contained therein almost as soon as they form. Class 4 CA seem to represent a transition phase between Class 2 CA and Class 3 CA. They preserve structures *and* can propagate the information contained in these structures over arbitrary distances. In other words, they neither preserve structures so thoroughly (in a fixed location) that it limits information propagation, nor do they propagate information so indiscriminately that structure preservation deteriorates. Although not all Class 4 CA are capable of universal computation, all CA that are capable of universal computation belong to Class 4. In fact, it appears that most useful computation takes place in Class 4 automata. In [25], Langton argued that Class 4 CA, and therefore computation in general, represent a *phase transition* between the regularity of Class 2 CA and the chaos of Class 3 CA.

These classes can also be characterized by quantitative measures, such as cellular information entropy (a measure of the predictability/randomness of the CA's evolution), mutual information between cells, growth dimension (a measure of how rapidly information disperses through the CA; similar to Lyapunov exponents), and length of transients before asymptotic behavior sets in [25]. For example, Class 3 CA tend to have high entropy, low mutual information, a high growth dimension (close to the maximum), and short transients, while Class 4 CA tend to have moder-

ate entropy, relatively high mutual information, a moderate growth dimension, and arbitrarily long transients.

Researchers have also studied CA from a theoretical perspective. In [34] Wolfram attempted to describe cellular automata behavior in terms of computation theory, viewing cellular automata as computers "whose time evolution processes the information specified by their initial configurations." CA have been studied as language recognizers [20] and as continuous functions on compact topological spaces [13, 14, 15, 16, 17, 18, 22]. For the latter characterization, let S be the finite set of cell states of a CA. S with the discrete topology is a compact space. Let d be the dimension of the CA and let $\mathbf{Z}$ denote the integers. Let $\mathcal{S} = S^{\mathbf{Z}^d}$ be the space of functions from cells in the CA to states. Then $\mathcal{S}$ with the product topology is also a compact space (by Tychonoff's theorem [29, p. 205]). If we view a configuration of the CA as a function in $\mathcal{S}$, assigning a state to every cell in the CA, then we can characterize the transition function of the CA as a continuous map of $\mathcal{S}$ onto itself which commutes with all primitive shift maps (maps which shift all cell states in the CA by one cell along a particular dimension). Continuity here is equivalent to saying that the transition function uses only a finite number of local cells in the automaton to determine the next state of any given cell. The fact that the transition function commutes with all shift maps means that the transition function is homogeneous; its operation is the same everywhere in the automaton. In [11], Hedlund showed the equivalence of this characterization of CA with the characterization described in the first paragraph of this chapter (to be formalized in Chapter 3).

Chapter 3

Notation and Definitions

3.1 Preliminaries

We let $\mathbf{N} = \{0, 1, 2, \ldots\}$ denote the natural numbers (nonnegative integers) and let $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ denote the integers.

We define the following set of projection functions for n-tuples:

$$\pi_i^n(x_1, \ldots, x_n) = x_i \text{ for } 1 \leq i \leq n.$$

If P is a predicate, we use the following notation to denote the truth value of P:

$$[P] = \begin{cases} 1 & \text{if } P \text{ is True} \\ 0 & \text{if } P \text{ is False} \end{cases}$$

If f and g are functions, fg denotes the *composition* of f with g as follows:

$$fg(x) = f(g(x))$$

For basic definitions of recursive functions and computability theory, see any of [5, 12, 9, 30].

3.2 Conventional Cellular Automata

Our definition of a conventional cellular automaton follows Culik, *et al.* in [17] with minor variations. Formally, a CA is a quadruple (S, d, T, s) where:

- S is a finite set of cell *states*, including a single *quiescent* state q. In what follows, we will at times, without loss of generality, assume that states are natural numbers with some number, usually 0, designated as the quiescent state.
- d is the *dimension* of the array, infinite in all directions (isomorphic to $\mathbf{Z}^d$).
- $T \subset \mathbf{Z}^d$ is a finite sequence of translation vectors describing the *neighborhood space*, or *neighborhood*, of a given cell (including that cell). We say that a cell j is a *neighbor* of cell i if its coordinate vector in the array is arrived at by adding one of the translation vectors in T to the coordinate vector of i. In a 1-dimensional CA, for example, the sequence of translation vectors would list the neighbors in order from the leftmost neighbor to the rightmost neighbor. When the ordering of the neighbors is not important, we will sometimes intentionally confuse the sequence with the underlying set.
- $s : S^{|T|} \rightarrow S$ is a *local state transition function* specifying the state transitions of every cell. The function s may depend on the ordering of the neighbors of a cell and always satisfies $s(q, \ldots, q) = q$.

A configuration $c \in S^{\mathbf{Z}^d}$ of a CA is a function assigning a nonquiescent state to a finite number of cells in the array, and the quiescent state to all other cells. Given a cell i in a CA, we let $\vec{i} = (i_1, \ldots, i_d) \in \mathbf{Z}^d$ represent the coordinates of cell i. Then $c(\vec{i})$ yields the state of cell i in the CA. A global transition function $G_s : S^{\mathbf{Z}^d} \rightarrow S^{\mathbf{Z}^d}$ results from the simultaneous application of the local transition function s to each cell in the array. Given cell i with coordinates $\vec{i}$ and a sequence of translations defining the neighborhood space $T = \langle t_1, \ldots, t_{|T|} \rangle$, let:

$$T_i = \langle n_i^1, \ldots, n_i^{|T|} \rangle = \langle c(t_1(\vec{i})), \ldots, c(t_{|T|}(\vec{i})) \rangle$$

be the sequence of states of the neighbors of cell i. Then we define G_s as follows:

$$G_s(c)(\vec{i}) = s(T_i) = s(n_i^1, \ldots, n_i^{|T|})$$

The action of G_s on cell i is defined by the output of the local state transition function s when applied to the sequence of states of the neighbors of i.

3.3 Structurally Dynamic Cellular Automata

3.3.1 The Ilachinski and Halpern Model

Overview

In [21], Ilachinski and Halpern proposed a new cellular automaton model called a structurally dynamic cellular automaton. The essential new feature of this model was that the connections between the cells, which are initially distributed on a d-dimensional array, are allowed to change according to rules similar in nature to the

state transition rules associated with conventional CA. This means that given certain conditions, specified by *link transition rules*, links between cells may be created or destroyed.

In the Ilachinski and Halpern (IH) Model, an SDCA consists of a finite set of binary-valued cells (called "sites" by Ilachinski and Halpern) numbered 1 through N whose connectivity is specified by an N x N connectivity matrix in which l_{ij} equals 1 if cells i and j are connected; 0 otherwise. These cells are initially arranged on a d-dimensional array with an initial, usually regular, pattern of connections. The *effective* dimensionality, defined as the average ratio of next-nearest neighbors to nearest neighbors, may change during the evolution of the SDCA depending on the evolution of the connections between cells. Cell values and connections change according to local transition rules which depend on some combination of states in the immediate neighborhood of a cell.

We will describe here the *totalistic* local transition rules, which depend only on the sum of the states of the relevant neighbors. This is the simplest type of rule used by Ilachinski and Halpern but it still conveys the flavor of their model. We will briefly describe two other types of rules suggested by Ilachinski and Halpern as well. We note here that much of the notation used by Ilachinski and Halpern reflects their background in physics and sometimes clashes with typical computer science notation. We will frequently translate their intentions into more typical computer science notation.

Following Ilachinski and Halpern, we define the distance between two cells i and j, D_{ij}, as the number of links in the shortest path between i and j. D_{ij}^t indicates

the distance between i and j at time t. We say that i and j are *direct neighbors* if $D_{ij} \leq 1$, and that i and j are *next-nearest neighbors* if $D_{ij} = 2$.

The State Transition Function

Ilachinski and Halpern assume that the each cell in an SDCA can be in one of only two integer-valued states, 0 (the quiescent state) and 1. Let c_i^{t+1} be the state of cell i at time $t+1$, and let V_i^t be the sum of the states of the direct neighbors of i at time t. Let $\alpha \subset \mathbf{N}$ be a subset of natural numbers. In order to determine c_i^{t+1}, we apply the local state transition function, ϕ, to V_i^t as follows:

$$c_i^{t+1} = \phi(V_i^t) = [V_i^t \in \alpha]$$

If $V_i^t \in \alpha$, then cell i will be in state 1 at time $t+1$; otherwise cell i will be in state 0. Since α specifies the action of the local state transition function, the nature of this set is critical, but we postpone discussion of the nature of this set, as well as that of similar sets used below to define the link transition functions, to Section 3.3.2.

We note here that this state transition function can easily be extended to an automaton with n nonquiescent states by specifying n disjoint α sets, $\alpha_1, \ldots, \alpha_n$, and modifying ϕ as follows:

$$c_i^{t+1} = \phi(V_i^t) = \sum_{j=1}^{n} [V_i^t \in \alpha_j)]j$$

V_i^t is either in exactly one alpha set (say the k^{th} alpha set), or in none of the alpha sets. In the former case, all terms in the summation are 0 except the k^{th} term, which

has the value k; cell i will therefore go into state k at time $t+1$. In the latter case, all terms are 0 and cell i will go into state 0 at time $t+1$.

Link Transition Functions

There are two types of link transition functions in an IH SDCA: *couplers* and *decouplers*. Let l_{ij}^{t+1} be the status of the link between cells i and j at time $t+1$ (0 = not linked, 1 = linked), and let V_{ij}^{t} be the sum of the states of the direct neighbors of i and j at time t (not double counting any neighbors to which both i and j are connected).

Let $\varepsilon \subset \mathbf{N}$ be a subset of natural numbers which will specify the action of the local coupler function ω. The local coupler function acts on V_{ij}^{t} to determine l_{ij}^{t+1} as follows:

$$l_{ij}^{t+1} = \omega(V_{ij}^{t}) = [V_{ij}^{t} \in \varepsilon \wedge D_{ij}^{t} = 2] \tag{3.1}$$

If $V_{ij}^{t} \in \varepsilon$, then link ij will be created at time $t+1$; otherwise link ij will not exist at time $t+1$.

Let $\beta \subset \mathbf{N}$ be a subset of natural numbers which will specify the action of the local decoupler function ψ. The local decoupler function acts on V_{ij}^{t} to determine l_{ij}^{t+1} as follows:

$$l_{ij}^{t+1} = \psi(V_{ij}^{t}) = (1 - [V_{ij}^{t} \in \beta])l_{ij}^{t} \tag{3.2}$$

where l_{ij}^{t} is the current link, if any, between i and j. If $V_{ij}^{t} \in \beta$, then link ij will not exist at time $t+1$; otherwise link ij remains as it was at time t.

A configuration of an IH SDCA at time t is specified by a *state vector*:

$$|G\rangle_t = |v_1^t, \ldots, v_N^t; \{l_{ij}^t\}\rangle$$

where $v_1^t, \ldots, v_N^t$ are the states of the N nodes at time t, and a global transition function Γ is defined by:

$$|G\rangle_{t+1} = \Gamma|G\rangle_t = (\Phi\Psi\Omega)|G\rangle_t$$

where Φ, Ψ, and Ω are the global transition functions obtained by applying the local transition functions, ϕ, ψ, and ω, simultaneously to all cells and cell pairs in the SDCA. We note that Ilachinski and Halpern use the notation $(\Phi\Psi\Omega)$ to indicate that Φ, Ψ, and Ω are applied to $|G\rangle_t$ simultaneously, and not to indicate the composition of these functions.

3.3.2 Problems with the Ilachinski and Halpern Model

While Ilachinski and Halpern are clear about their intention to create a cellular automaton model in which both the cell values and the connection structure are dynamic, their definition of the model leaves certain issues, critical from a computation theoretic standpoint, unaddressed or vague.

First, their definition of an SDCA allows only a finite number of cells. Is this an intentional part of the model, or merely a concession to the limitations encountered in programming SDCA simulations on the computer?

Second, what is the nature of the sets α, β, and ε that define the actions of the local state and link transition functions? The examples provided by Ilachinski

and Halpern seem to indicate that these sets must be finite. Clearly, if the number of cells in the SDCA is finite, finite α, β, and ε sets will suffice to describe any possible transition functions. What if, however, we wish to expand the model to allow an infinite array of cells (as we do in Section 3.3.3)? In a subsequent paper [10], examples used by Halpern and Caltagirone seem to indicate that these sets need only be recursive; we will take this viewpoint as the appropriate one for an IH SDCA with an infinite array of cells.

Third, why are the links in their model bidirectional? While bidirectional links make sense in certain settings (*e.g.*, where links are modeling the mutual influence of entities that affect each other through the same channel of communication), unidirectional links would seem more natural for various applications. For example, since synapses in the brain are unidirectional, two neurons can affect each other only if there are at least two pathways between them, one to transmit impulses from neuron A to neuron B and one to transmit impulses from neuron B to neuron A. In Section 3.3.3 we will describe an alternative SDCA model (the Labeled Links Model) which allows us to model unidirectional links as well as bidirectional links.

Fourth, there is a problem with Ilachinski and Halpern's formulation of a coupler function. Just as the decoupler function (Equation 3.2) needs the factor l_{ij} to prevent it from inadvertently *linking* two unlinked cells, the coupler function (Equation 3.1) needs a factor to prevent it from inadvertently *unlinking* two linked cells. As it stands, the coupler function would unlink any two currently linked cells to which it was applied. This problem can be corrected and, at the same time, the coupler and decoupler functions can be combined into a single link transition function by

summing Ilachinski and Halpern's coupler and decoupler functions. Let λ be this combined link transition function. To determine the status of the link between cells i and j at time $t+1$ we apply λ to V_{ij}^t as follows:

$$l_{ij}^{t+1} = \lambda(V_{ij}^t) = [V_{ij}^t \notin \beta \wedge D_{ij}^t = 1] + [V_{ij}^t \in \varepsilon \wedge D_{ij}^t = 2]$$

where l_{ij}^{t+1}, V_{ij}^t, β, and ε are the same as before. If i and j are direct neighbors, the second term becomes zero and the first term determines the status of the existing link during the next time step. If i and j are next-nearest neighbors, the first term becomes zero and the second term determines whether a link should be forged between i and j during the next time step. If $D_{ij}^t > 2$ then both terms become zero; i and j are not currently connected and cannot become connected during the next time step (since they are not next-nearest neighbors).

Fifth, there are features of the Ilachinski and Halpern Model which appear to be operating at cross purposes. For example, their state transition functions are of two types—totalistic, which we have already seen, and *outer-totalistic*, a variation in which the next state of a cell depends on the current state of that cell and the sum of the states of the cell's direct neighbors *excluding* the cell itself. Like totalistic transition functions, however, outer-totalistic transition functions are completely symmetric; *i.e.*, the *positions* of a cell's neighbors are unknown and play no part in the state transitions. In fact, state transition functions are ignorant of even the *number* of cells in each possible state in the neighborhood.

This is not the case, however, for the different link transition functions they present. In addition to the totalistic link transition functions described above, Ilachinski and Halpern also describe *outer-totalistic* link transition functions and

restricted-totalistic link transition functions. All three types of link transition functions, as presented by Ilachinski and Halpern, require that we be able to partition the set of direct neighbors of a given pair i and j into three subsets:

$$\begin{aligned} A_{ij} &= \{i, j\} \\ B_{ij} &= \{k \mid D_{ik} = 1 \wedge D_{jk} = 1\} \\ C_{ij} &= \{k \mid D_{ik} = 1 \oplus D_{jk} = 1\} \end{aligned}$$

where $\oplus$ denotes *exclusive or*. A_{ij} contains only the cell pair i and j. B_{ij} contains all direct neighbors (other than i and j) which i and j have in common. And C_{ij} contains all direct neighbors not in A_{ij} or B_{ij}, *i.e.*, those direct neighbors (other than i and j) which i and j do *not* have in common. Totalistic link transition functions depend only on the sum of the states of the cells in $A_{ij} \cup B_{ij} \cup C_{ij}$. Outer-totalistic link transition functions depend on two sums, the sum of the states of the cells in A_{ij} and the sum of the states of the cells in $B_{ij} \cup C_{ij}$. And restricted-totalistic link transition functions depend on three sums, the sum of the states of the cells in A_{ij}, the sum of the states of the cells in B_{ij}, and the sum of the states of the cells in C_{ij}. The necessity of partitioning the neighborhood in this fashion indicates that the link transition functions, unlike the value transition functions, must know the position of a cell's neighbors. The inconsistent natures of the value transition functions and link transition functions argue for a splitting of the Ilachinski and Halpern Model into several, more consistent models of varying degrees of power.

Finally, the types of transition functions allowed in the Ilachinski and Halpern Model appear to us to be too restrictive. All of the transition functions described

above depend only on one or more sums of subsets of a cell's neighbors. This excludes all transition functions that might depend on some other attribute of the neighborhood (*e.g.*, a majority state transition function, yielding state 1 if there are more 1's than 0's in the neighborhood; 0 otherwise).

3.3.3 Alternative SDCA Models

Overview

For our purposes, we will define an SDCA as a quadruple (S, d, σ, λ) where:

- S is a finite set of cell *states*, including a single *quiescent* state q. In what follows, we will at times, without loss of generality, assume that states are natural numbers with some number, usually 0, designated as the quiescent state.
- d is the *dimension* of the underlying array, which is now infinite in all directions (isomorphic to $\mathbf{Z}^d$).
- σ is a *local state transition function* specifying the state transitions of a cell (based on the states of that cell's direct neighbors, *i.e.*, its *neighborhood.*).
- λ is a *local link transition function* specifying the status transitions of the links between all possible pairs of cells (based on the states of each pair's direct neighbors).

The exact nature of the local transition functions σ and λ will depend on which of three variations on the SDCA model (described below) we are using. We will, however, always require the following:

- The local transition functions σ and λ must be recursive.
- Neighborhoods must be finite.
- The local state transition function σ must always map a completely quiescent neighborhood to the quiescent state.
- Links between cells whose neighborhoods are completely quiescent are never altered.
- The local link transition function λ must forge links only between next-nearest neighbors.

The local transition functions of the model variations will differ primarily in the type of neighbor information that these functions act upon. In all models, the transition functions have access to the states of the neighbors in a neighborhood. The models differ in the degree to which the transition functions can associate these states with *specific* neighbors. In other words, given a state in a neighborhood, how much do the transition functions know about the source of that state? In the Relative Location Model, the transition functions know the state and exact relative location of each neighbor, and can therefore identify exactly the source of any state in the neighborhood. In the Labeled Links Model, links are labeled and the transition functions know the state of each neighbor and the label of the link through which that neighbor is connected; exact location of the neighbor is not specified. In the

Symmetric Model, the transition functions only know the number of neighbors in a particular state; all information about the source of the states in the neighborhood is lost. Furthermore, we will distinguish between *unbounded links* models, in which there is no bound on the number of links a cell can have, and *bounded links* models, in which there is such a bound.

In addition, we will impose the following restrictions on the configurations of an SDCA regardless of the model we are using:

- Any configuration of an SDCA may contain only a finite number of cells in nonquiescent states, and this quality will be preserved by any legitimate computation.
- The neighborhood structure, or relative link pattern, of all cells in any configuration must be homogeneous except for a finite number of cells, and this quality will also be preserved by any legitimate computation.

A configuration $c \in \Theta = S^{\mathbf{Z}^d} \times \{0,1\}^{(\mathbf{Z}^d)^2}$ of an SDCA is a *state pair*. The first element is a function assigning a state to each cell. The second element is a function specifying the status of each possible link (0 = not linked, 1 = linked). Borrowing our earlier notation for conventional CA, we let $\vec{i} = (i_1, \ldots, i_d) \in \mathbf{Z}^d$ represent the coordinates of cell i in an SDCA. Then $(\pi_1^2(c))(\vec{i})$ yields the state of cell i in the SDCA. Similarly, $(\pi_2^2(c))(\vec{i},\vec{j})$ yields the status of the link between cells i and j in the SDCA.

Given cell i at coordinates $\vec{i}$, let $T_i = \{n_i^1, \ldots, n_i^{|T_i|}\}$ be the set of neighbors of i, however neighbors are defined for the model with which we are concerned. It

is useful to think of each neighbor n as a packet of information about a neighbor, the exact contents of which will depend on the particular model we are using. Note that T_i may vary over time depending on the link evolution of the SDCA. Then we define $\Sigma : \Theta \to S^{\mathbf{Z}^d}$, the global state transition function which results from the simultaneous application of σ to each cell in the array as follows:

$$\Sigma(c)(\vec{i}) = \sigma(T_i) = \sigma(n_i^1, \ldots, n_i^{|T_i|})$$

Similarly we define $\Lambda : \Theta \to \{0,1\}^{(\mathbf{Z}^d)^2}$, the global link transition function which results from the simultaneous application of λ to each possible link in the array as follows:

$$\Lambda(c)(\vec{i},\vec{j}) = \lambda(T_i, T_j) = \lambda((n_i^1, \ldots, n_i^{|T_i|}), (n_j^1, \ldots, n_j^{|T_j|}))$$

Practically, we should apply λ only to those cells which are direct neighbors—to check for decoupling—and to those cells which are next-nearest neighbors—to check for coupling. If the neighborhood sets of cells i and j are disjoint ($T_i \cap T_j = \emptyset$), then λ is just the identity function and $\Lambda(c)(\vec{i},\vec{j}) = \pi_2^2(c)(\vec{i},\vec{j})$. Also, in the same way that the state of a quiescent cell cannot change if its neighborhood is completely quiescent, we will require that the status of the link between two quiescent cells cannot change if both of their neighborhoods are completely quiescent. Note that both Σ and Λ are homogeneous due to their dependence on the homogeneous local functions σ and λ.

Finally, we define a global transition function $\Gamma : \Theta \to \Theta$ as follows:

$$\Gamma(c) = (\Sigma(c), \Lambda(c))$$

The Relative Location Model

In the *Relative Location Model* all the local transition functions have access to the exact *relative location* of each neighbor as well as the state of the neighbor. For our purposes in this section, a neighbor $n_i \in S \times \mathbf{Z}^d$ of the cell i is a pair specifying the state and *relative* location of the neighboring cell. The first element of the pair indicates the state of the neighbor and the second element is a d-tuple specifying the location of n_i relative to i. The absolute coordinates of the neighbor are obtained by adding the coordinate vector of i and the relative location vector of n_i. For example, if the coordinate vector of i in a 2-dimensional automaton is $(4, 10)$ and the relative location vector of a neighbor, n_i, is $(-2, 6)$, then the absolute coordinates of n_i would be $(2, 16)$. Note that the cell i itself can be specified in the form of a neighbor by writing $((\pi_1^2(c))(i), (0, \ldots, 0))$ where $(\pi_1^2(c))(i)$ is the state of cell i and the relative location vector is the zero vector.

Let $W = S \times \mathbf{Z}^d$ be the set of all possible neighbors. Let $\mathcal{F}_W$ be the set of finite, nonempty, partial functions mapping $\mathbf{Z}^d$ to W. We call these *neighborhood functions*, since the neighborhood of a cell can be specified by a function $f \in \mathcal{F}_W$. The function f is defined for those cells which are in the neighborhood, and its range, which is a finite, nonempty subset of W, yields all the neighbor information. The local state transition function $\sigma : \mathcal{F} \rightarrow S$ is a mapping from neighborhood functions—specifying finite, but arbitrarily large, neighborhoods—to the state set of the SDCA. The local link transition function $\lambda : \mathcal{F} \times \mathcal{F} \times \{0, 1, 2\} \rightarrow \{0, 1\}$ is a mapping from a pair of neighborhood functions specifying the neighborhoods of two cells i and j and a digit specifying the current status of the link between cells i and

j ($0 = i$ and j are neither direct neighbors or next-nearest neighbors, $1 = i$ and j are direct neighbors, $2 = i$ and j are next-nearest neighbors) to the status 0 (no direct link between i and j at the next time step) or 1 (a direct link between i and j at the next time step). Information about the current link status is necessary so that the link transition function does not try to link two cells which are not next-nearest neighbors, or unlink two cells which are not currently direct neighbors.

If we put no further restrictions on our transition functions, however, we encounter two problems. First, arbitrary transition functions would allow a Relative Location Model SDCA to solve arbitrary, noncomputable problems. Since we are allowing nonrecursive transition functions, we could, for example, encode the Halting Problem in the transition function. In other words, arbitrary transition functions would make a Relative Location Model SDCA *too* powerful. Second, we would have no effective way of *describing* arbitrary transition functions as finite objects. We solve both of these problems by requiring that the transition functions, σ and λ, be recursive functions.

Even with this restriction, the Relative Location Model is still quite powerful. The only significant limitation on a Relative Location Model SDCA is the lower bound on the time needed for two cells to communicate with each other. Two cells i and j at a distance of D_{ij} require at least $\log D_{ij}$ time steps to forge a link (by constantly halving the length of the minimum path between i and j) and thereby exchange information. In fact, given a group of cells in an SDCA in which the maximum distance between any two cells is r, a Relative Location Model SDCA can perform any calculation requiring the states of this group of cells in $\log r$ time.

This level of power is not appropriate for all applications. One way of restricting the power of a Relative Location Model SDCA is to put a bound on the number of links each cell can have. In this way we could distinguish between ULRL (Unbounded Links Relative Location) SDCA and BLRL (Bounded Links Relative Location) SDCA. But this restriction does not solve the real problem presented by the Relative Location Model—the fact that a cell knows the relative location of all cells connected to it. This assumption is realistic only from a certain perspective. For example, each processor of a parallel computer whose processors are arranged on an array could reasonably be expected to know its location and communicate that information to all of its neighbors. But if we intend to use our SDCA to model certain naturally occurring processes (human brain function, for example) where this expectation makes no sense, we will need a different model. We therefore introduce two alternatives to the Relative Location Model, each of which requires that the exact relative location of a cell's neighbors *not* be known during state and link transitions.

Labeled Links Model

The *Labeled Links Model* removes the local transition functions' knowledge of the exact relative locations of a cell's neighbors without completely sacrificing the ability to differentiate among different orientations of the neighborhood. This model is intended as a more realistic alternative to the Relative Location Model; it is more realistic as a model for those processes in which each cell, or unit, in the process can be expected to be able to differentiate among its possible inputs, but not on the basis of the location of their source. Consider, for example, neurons in the brain.

They respond differently to different neurotransmitters with no apparent knowledge concerning the location of the source of these neurotransmitters. A computer network may also behave in this manner; input from different ports in a machine may be treated differently irrespective of what is connected to that machine through these ports.

In the Labeled Links Model we require that the local transition functions not have access to the exact locations of the neighbors in the neighbor vector upon which they are operating, but we do allow the links to be *labeled* so that the transition functions can at least differentiate one *link* from another. Although links are undirected, two linked cells may label their common link differently. In fact, these labels allow us to construct an SDCA with unidirectional links, merely by using link labels to distinguish between the *input* and *output* links of a cell. We note here that the operation of a Labeled Links SDCA is invariant under any change of labels that preserves all previous distinctions among labels. Like the Relative Location Model SDCA, the Labeled Links Model SDCA can have unbounded links (ULLL SDCA) or bounded links (BLLL SDCA). We describe ULLL SDCA here.

Let the links be labeled by natural numbers. Define a neighbor of cell i to be a pair (q, n) where $q \in S$ indicates the state of the neighboring cell and $n \in \mathbf{N}$ is the label of the link connecting that neighbor to i. Thus, W, the set of all possible neighbors of a given cell i, equals $S \times \mathbf{N}$. (Cell i is specified as the direct neighbor connected through the 0^{th} link, *i.e.*, $(c_1^2(\vec{i}), 0)$ is the neighbor representation of cell i.) As before, let $\mathcal{F}_W$ be the set of neighborhood functions mapping $\mathbf{Z}^d$ to W. Then the local state transition function $\sigma : \mathcal{F} \rightarrow S$ is a mapping from

nonempty neighborhood functions to states in S. The local link transition function $\lambda : \mathcal{F} \times \mathcal{F} \times \{0,1,2\} \rightarrow \{0,1\}$ maps a pair of nonempty neighborhood functions and a digit to 0 or 1. The neighborhood functions specify the neighbor information of the direct neighbors of the two cells i and j to which the link transition function is being applied, and the digit specifies the current status of the link between i and j ($0 = i$ and j are neither direct neighbors or next-nearest neighbors, $1 = i$ and j are direct neighbors, $2 = i$ and j are next-nearest neighbors). The result indicates the status of the link in the next time step (0 = not linked, 1 = linked).

At this point we introduce another very important potential restriction on the Labeled Links Model. All of the models we have described so far—ULRL, BLRL, ULLL, BLLL—share the attribute of requiring transition functions with an infinite domain. This is true even for bounded links models, since each of the links in a BLRL SDCA can be connected to any cell in the infinite array, and each of the links in a BLLL SDCA can be labeled by any natural number. Our requirement that the transition functions be recursive allows us at least to be able to describe these functions finitely. But a more natural finitary model—one which would respect the standard conventional CA notion that the local transition function can be represented in a finite rule table—seems highly desirable. The capabilities of such a model would not be inflated by the power we get from an infinite rule table, and we could more readily attribute the benefits offered by such a model to the dynamic nature of its connection structure rather than the size of its rule table.

We could implement this restriction in a BLRL SDCA by designating a finite set of cells in the array (which would include all nonquiescent cells in the initial

configuration) and requiring that all subsequent links in the evolution of the SDCA remain within this finite set. This is unsatisfactory for at least two reasons. Such a limitation would put a bound on the size of the nonquiescent portion of any configuration in any computation of the SDCA. This is much too severe a restriction, excluding many interesting problems in which the nonquiescent portion of the SDCA must be able to grow in an unbounded fashion. At the same time, such a model would still be *too* powerful in one respect; the transition functions would still have access to the exact relative locations of each of a cell's neighbors. For these reasons we reject this finite locations version of the BLRL SDCA.

These problems are avoided when we implement the finite rule table restriction in a BLLL SDCA. In this case, we merely add the restriction that the set of available labels be a finite subset of the natural numbers rather than the entire set of natural numbers. We will call this model the Finite Labels Labeled Links (FLLL) Model. Since each cell in an FLLL SDCA has a bounded number of links and each one of these links can be labeled by one of a finite set of labels, we obtain a finite rule table. We avoid restricting the growth of the nonquiescent portion of the SDCA since none of our limitations prevent any cell from being able to link to any other cell over time. And since this is a Labeled Links SDCA we avoid the unwanted power of the transition functions knowing the exact relative locations of a cell's neighbors.

This is a much stronger restriction than putting a bound on the number of links. As we will show in Section 4.2, any BLRL SDCA can be simulated by a BLLL SDCA with no loss of time, exactly because we still have an infinite set of labels which we can use to encode the infinity of possible neighbor locations in the BLRL

SDCA. It seems highly unlikely that a BLRL SDCA can be simulated by an FLLL SDCA, especially with no loss of time, since our finite label set gives us no direct way of encoding the location information necessary for the simulation of a BLRL SDCA.

The FLLL SDCA is our first naturally finitary SDCA model. It is, in fact, one of only two naturally finitary models in our collection of models. The Bounded Links Symmetric (BLS) Model, a bounded links version of the Symmetric Model described in the next section, is our other naturally finitary model. In our opinion, however, the BLS Model sacrifices too much power, and in the following chapters, particularly Chapter 5, we will focus our attention on the FLLL SDCA Model as the most natural SDCA analogue of the conventional CA. We note here that although the FLLL model does not require either that the number of links be equal to the number of labels or that the links of every cell be labeled distinctly, the FLLL SDCA described in Chapter 5 operates with an equal number of links and labels, and each link of a cell has a distinct label.

The Symmetric Model

In the *Symmetric Model*, we impose a more severe restriction on the Relative Location Model. We require that the local transition functions have no means whatsoever of differentiating among different orientations of a neighborhood. This is quite a severe restriction and we include this model primarily as a bridge between the more powerful Relative Location and Labeled Links Models and the possibly less powerful Ilachinski and Halpern Model. Like the Relative Location and Labeled Links Models, Symmetric Model SDCA can have unbounded links (ULS SDCA) or bounded links

(BLS SDCA). Here we describe ULS SDCA.

We alter the local transition functions σ and λ as follows. Suppose the SDCA has n states. Without loss of generality, let $S = \{1, \ldots, n\}$. Let i be a cell in the SDCA and let $\vec{n_i} \in \mathbf{N}^n$ be an n-dimensional vector such that $(n_i)_k$ is equal to the number of neighbors of i in state k. Then the local state transition function $\sigma : \mathbf{N}^n \rightarrow S$ is a mapping from vectors in $\mathbf{N}^n$ to states in S, where the vector indicates the distribution of states among the direct neighbors of the cell to which the state transition function is being applied. The local link transition function $\lambda : \mathbf{N}^n \times \{0, 1, 2\} \rightarrow \{0, 1\}$ maps a vector in $\mathbf{N}^n$ and a link status digit to 0 or 1. Here the vector indicates the distribution of states among the direct neighbors of the two cells i and j to which the link transition function is being applied, and the digit specifies the current status of the link between i and j ($0 = i$ and j are neither direct neighbors or next-nearest neighbors, $1 = i$ and j are direct neighbors, $2 = i$ and j are next-nearest neighbors). The result indicates the status of the link in the next time step (0 = not linked, 1 = linked).

In both the Relative Location Model and the Labeled Links Model, the local transition functions retain access to the state of the central cell(s) in their calculations (the Relative Location Model through the neighbor whose relative location vector is $\vec{0}$, and the Labeled Links Model through the neighbor connected by the 0^{th} link). We can modify the symmetric model to have this feature as well. Let the local state transition function $\sigma : S \times \mathbf{N}^n \rightarrow S$ be a mapping from pairs to states in S. The first element of the pair is the state of the central cell and the second element is a vector indicating the distribution of states among the direct neighbors

of the central cell not including the central cell. The local link transition function $\lambda : S^2 \times \mathbf{N}^n \times \{0,1,2\} \rightarrow \{0,1\}$ would be a mapping from triples to 0 or 1. Here the first element of the triple is a state pair indicating the states of the two cells to which the link transition function is being applied the second element is a vector indicating the distribution of states among the direct neighbors of the two central cells not including those central cells, and the third element of the triple is the current link status digit.

Chapter 4

Results

4.1 Introduction

In Chapter 3 we described three SDCA models as alternatives to the Ilachinski and Halpern SDCA model: the Relative Location Model, the Labeled Links Model, and the Symmetric Model. We noted that each of these models has an unbounded links (UL) version, in which there is no upper limit on the number of neighbors a cell can have, and a bounded links (BL) version, in which such an upper limit is imposed. In addition, we have a finite labels version of the Labeled Links Model, and the Ilachinski and Halpern Model. This gives us eight SCDA models:

- ULRL (Unbounded Links Relative Location SDCA)
- BLRL (Bounded Links Relative Location SDCA)
- ULLL (Unbounded Links Labeled Links SDCA)
- BLLL (Bounded Links Labeled Links SDCA)

- FLLL (Finite Labels Labeled Links SDCA)
- ULS (Unbounded Links Symmetric SDCA)
- BLS (Bounded Links Symmetric SDCA)
- IH (Ilachinski and Halpern SDCA)

4.2 The Hierarchy of SDCA Models

It is natural to inquire into the relationships among these classes of SDCA. We do so by defining a *simulation relationship* and investigating the structure imposed on the SDCA classes by this relationship. First, we need to make clear what we mean by one automaton simulating another. Our definition of simulation, though similar to that in [17], contains some significant changes which make it better suited to our purposes. These changes occur mostly in the function used to transform configurations of the simulated automaton into configurations of the simulating automaton, and are motivated by our desire to use this definition for an SDCA simulating a conventional CA as well as for a conventional CA simulating a conventional CA.

When an SDCA simulates a conventional CA it is very likely that, in order to take advantage of an SDCA's special linking capabilities, the simulating SDCA will not preserve the geography of the simulated CA. For example, if the states of the cells in the nonquiescent portion of the initial configuration in a simulated 1-dimensional CA are, in order, *abc*, the groups of cells in the SDCA which are simulating these nonquiescent cells may be arranged in a completely different order, such as *acb*. Or, as is the case in the simulator we will be describing in Chapter 5, information

necessary to the operation of the automaton may be encoded in a group of cells at a great distance from the cells which actually need that information. The dynamic linking of an SDCA allows this by making the particular location of any cell largely irrelevant. In addition, it is possible that the SDCA will generate groupings of cells which help it to perform the simulation but which, from the point of view of the simulated CA, are garbage.

For these reasons, we define separate *encoding* and *decoding* functions. The encoding function transforms configurations (specifically the initial configuration) in the simulated automaton to configurations in the simulating automaton, and the decoding function transforms configurations in the simulating automaton to configurations in the simulated automaton. We would like these functions to be simple in a natural way—they should only transform configurations (*i.e.*, not perform any of the required simulation) and the input to the functions should be limited to a finite portion of the configuration they are transforming. We will call such functions *structurally-defined.*

Definition 4.1 *Let C_A and C_B be configuration spaces of automata A and B respectively (conventional or structurally dynamic). A function $e : C_A \rightarrow C_B$ which transforms configurations of A into configurations of B is a structurally-defined encoding function if it is 1-1 recursive and uses only a finite amount of information in a given configuration of A to do the encoding, otherwise transforming quiescent cells in A to quiescent cells in B. The encoding function has access to the rule table of A. This rule table information can be encoded as part of the corresponding configuration in B, but we will require that no rule table information be used to perform any of*

the simulation during the encoding process. A decoding function $d : C_B \rightarrow C_A$ is structurally-defined if it is recursive and uses only a finite amount of information in a given configuration of B to do the decoding, otherwise transforming quiescent cells in B to quiescent cells in A. The decoding function does not have access to the rule tables of A or B. Finally, the encoding and decoding functions must satisfy the following relationship:

$$de = id_{C_A}$$

Since we require that all legitimate automaton configurations have a finite number of nonquiescent cells, it is possible that the encoding and decoding functions will operate by looking at the entire finite range containing the nonquiescent cells and transforming it in one step to the nonquiescent range in the other automaton. This is how the encoding and decoding functions in our universal simulator in Chapter 5 will operate. An important special case arises when the encoding and decoding functions are *homogeneous*, transforming the cells of a configuration individually and uniformly, and the encoding function ignores the rule table of A. We will refer to this as *oblivious* encoding and decoding.

We define simulation as follows:

Definition 4.2 *Given a d-dimensional cellular automaton A (conventional or structurally dynamic) with configuration space C_A and global transition function F_A, and a d-dimensional cellular automaton B (conventional or structurally dynamic) with configuration space C_B and global transition function F_B, we say that B* simulates *A if there exist $m \geq 1$, $n \geq 1$, and structurally-defined functions $e : C_A \rightarrow C_B$ and $d : C_B \rightarrow C_A$ such that given any configuration $c_0 \in C_A$ of cellular automaton A,*

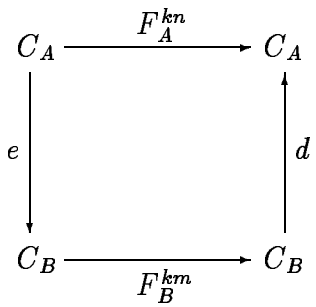

Figure 4.1: Simulation of automaton A by automaton B

and any $k \geq 1$*:*

$$F_A{}^{kn}(c_0) = d(F_B{}^{km}(e(c_0)))$$

or, equivalently, that the diagram in Figure 4.1 commutes. We say that B *can simulate* n *steps of* A *in* m *steps. If* $m > n$*, we say that* B *simulates* A *with a* slowdown *of* m/n*. If* $m = n$*, we say that* B *simulates* A *with* no loss of time*. If* $m < n$*, we say that* B *simulates* A *with a* speedup *of* n/m*.*

In other words, B simulates n steps of A in m steps if we can encode any initial configuration of A using e, run B on this encoded configuration for m steps in B, decode the resulting configuration using d, and get the same result as if we ran A on its initial configuration for n steps in A. The factor of k is intended to capture the notion that B must be able to simulate any *multiple* of n steps of A—B is not a legitimate simulator if it simulates n steps of A successfully but can then carry the simulation no further.

Simulation is a transitive relationship. If automaton B can simulate automaton A and automaton C can simulate automaton B, then automaton C can simulate automaton A. In most cases, where $n = 1$ in both simulations, this is obvious. For example, if B takes 20 steps to simulate 1 step of A (using encoding function e_{AB} and

decoding function d_{AB}), and C takes 5 steps to simulate 1 step of B (using encoding function e_{BC} and decoding function d_{BC}), then C will be able to simulate 1 step of A in 100 steps. Given $c_0 \in C_A$, we would have the following simulation:

$$F_A{}^k(c_0) = d_{AB}(d_{BC}(F_C{}^{100k}(e_{BC}(e_{AB}(c_0)))))$$

But suppose instead that B takes 20 steps (m_{AB}) to simulate 7 (n_{AB}) steps of A, and C takes 50 steps (m_{BC}) to simulate 30 steps (n_{BC}) of B. We need to find m_{AC}, n_{AC}, e_{AC}, and d_{AC} to show that C can simulate A. As in the previous example, we compose the encoding and decoding functions of the two existing simulations and let $e_{AC} = e_{BC}e_{AB}$ and $d_{AC} = d_{AB}d_{BC}$. To find the required m_{AC} and n_{AC}, let $\mathrm{lcm}(i,j)$ be the *least common multiple* of the integers i and j. Then

$$m_{AC} = \frac{\mathrm{lcm}(m_{AB}, n_{BC})}{n_{BC}} m_{BC}$$

and

$$n_{AC} = \frac{\mathrm{lcm}(m_{AB}, n_{BC})}{m_{AB}} n_{AB}.$$

In our example, $m_{AC} = 100$ and $n_{AC} = 21$, so C takes 100 steps to simulate 21 steps of A.

We define a simulation relationship among SDCA classes as follows:

Definition 4.3 *Given two SDCA classes, C_1 and C_2, we say that $C_1 \leq_s C_2$ if, given any SDCA $A_1 \in C_1$ there is an SDCA $A_2 \in C_2$ that can simulate A_1. We say that $C_1 \equiv_s C_2$ if $C_1 \leq_s C_2$ and $C_2 \leq_s C_1$. We say that $C_1 <_s C_2$ if $C_1 \leq_s C_2$ and $C_1 \not\equiv_s C_2$.*

Certain relationships are immediately obvious. Any bounded links SDCA can be simulated by an unbounded links SDCA of the same model, and an FLLL SDCA can be simulated by a BLLL SDCA:

- $BLRL \leq_s ULRL$
- $FLLL \leq_s BLLL \leq_s ULLL$
- $BLS \leq_s ULS$

Since a Relative Location SCDA or a Labeled Links SDCA can ignore the cell location or link label information available and act like a Symmetric SDCA, any ULS SDCA can be simulated by a ULRL SDCA or a ULLL SDCA, and any BLS SDCA can be simulated by a BLRL SDCA or a BLLL SDCA:

- $ULS \leq_s ULRL$
- $ULS \leq_s ULLL$
- $BLS \leq_s BLRL$
- $BLS \leq_s BLLL$

Because of the infinite number of labels available in a Labeled Links SDCA, a ULRL SDCA can be simulated by a ULLL SDCA and a BLRL SDCA can be simulated by a BLLL SDCA. In either case, we can construct a bijection between location coordinates in the Relative Location SDCA and the natural number link labels in the Labeled Links SDCA. We label the links in the Labeled Links SDCA simulator

so that the labels encode the location information that would be available in the Relative Location SDCA we are trying to simulate. Thus, the transition functions in the Labeled Links SDCA have all the information necessary to mimic the transition functions in the Relative Location SDCA and the simulation is straightforward. Thus we have:

- $ULRL \leq_s ULLL$
- $BLRL \leq_s BLLL$

Combining these relationships, we have:

- $ULS \leq_s ULRL \leq_s ULLL$
- $BLS \leq_s BLRL \leq_s BLLL$

An IH SDCA can be simulated by a ULRL SDCA since a ULRL SDCA would have all the information necessary to construct the neighborhood partition used in an IH SDCA. And since $ULRL \leq_s ULLL$, an IH SDCA can also be simulated by a ULLL SDCA:

- $IH \leq_s ULRL$
- $IH \leq_s ULLL$

4.3 Comparison of CA and SDCA

In the following theorems we will begin to explore the relative capabilities of conventional CA and SDCA. How much more power do we get from an SDCA? Theorem 4.1 uses the Firing Squad Synchronization Problem (FSSP) to answer this question by putting a lower bound on the slowdown of a 1-dimensional conventional CA acting as a universal simulator of 1-dimensional SDCA.

Definition 4.4 *A d-dimensional CA or SDCA is* CA-universal *(*SDCA-universal*) if it can simulate any other d-dimensional CA (SDCA). We note that the slowdown (or speedup) of a universal automaton need not be constant across all possible simulated automata.*

The 1-dimensional FSSP presents us with a contiguous block of n cells in a 1-dimensional CA. At time $t = 0$, the leftmost cell in this block (the general) is in state G and all the other cells in the block (the soldiers) are in state R (Ready). We require a transition function that will force the general and soldiers to enter the firing state F simultaneously and for the first time at some time t_F. Since it is well known that a conventional 1-dimensional CA requires at least time $2n - 2$ to solve the FSSP [28], Theorem 4.1 establishes an exponential lower bound on the slowdown of a 1-dimensional, SDCA-universal, conventional CA.

Theorem 4.1 *The Firing Squad Synchronization Problem for n cells can be solved by a ULS SDCA in time* $\lceil \log n \rceil + 1$ *using* $n - 1$ *links for each cell.*

Proof. We construct the required SDCA as follows. Let $S = \{G, R, A, F\}$ (A = Aim). At time $t = 0$ the leftmost cell of the firing squad is in state G and all other cells in the firing squad are in state R. Each cell is linked to the cells immediately to its left and right (except for the end cells which have only a left or right link). The automaton functions according to the following state and link transition rules:

- State Transition Function:
  - If cell i is in state R and at least one of its neighbors is in state G or A, cell i goes into state A in the next time step.
  - If cell i is in state G or A, and it does not have at least one neighbor in state R, cell i goes into state F in the next time step.
  - Otherwise, the state of a cell does not change.
- Link Transition Function: Forge all possible links between squad members; never remove any links.

Informally, the SDCA constructs links between each general/soldier and every other general/soldier. Since at every time step the length of the path covered by the longest link doubles, constructing the necessary $n-2$ additional links requires $\lceil \log n \rceil$ time. Until all the links have been constructed every cell is either in state R or is connected to a cell in state R and is thus prevented from firing. In the time step after all the links have been constructed, all the remaining R cells in the squad enter state A. Since there are no longer any cells in state R, the entire squad enters state F in the next time step. □

If we strengthen our hypothesis somewhat we can solve the 1-dimensional FSSP in $\mathcal{O}(\log n)$ time using *constant* links for each cell.

Theorem 4.2 *The Firing Squad Synchronization Problem (FSSP) for n cells, where n is a power of 2, can be solved by a ULS SDCA in time* $\log n + 1$ *using at most 2 links for each cell.*

Proof. Let $S = \{G, R, A, F\}$. At time $t = 0$ the leftmost cell of the firing squad is in state G and all other cells in the firing squad are in state R. Each cell is linked to the cells immediately to its left and right (except for the end cells which have only a left or right link). The automaton functions according to the following state and link transition rules:

- State Transition Function:
  - If cell i is in state R and at least one of its neighbors is in state G or A, cell i goes into state A in the next time step.
  - If cell i is in state G or A and has no neighbors other than itself, cell i goes into state F in the next time step.
  - Otherwise, the state of a cell does not change.
- Link Transition Function: Forge all possible links between squad members; remove all current links between squad members.

Informally, each general/soldier extends its right and left links whenever possible, doubling them at each time step. Each squad member is connected to at least one

other squad member until step $\log n$, at which point the link that existed in step $\log n - 1$ is removed and link doubling makes it impossible for any squad member to forge a new link with any other squad member. At step $\log n$, all links between squad members disappear and all squad members initially in state R have at some point been connected to a cell in state G or A and are now in state A. At step $\log n + 1$ the entire squad enters state F. □

This exponential slowdown obtained when a conventional CA simulates an SDCA can be improved if we place certain restrictions on the SDCA that the conventional CA is trying to simulate.

Definition 4.5 *If the neighborhood of each cell in a 1-dimensional CA consists of the cell itself and the r cells to either side of the cell, we say that the CA has a* neighborhood radius *of r. If the neighborhood of each cell in a 1-dimensional SDCA contains only cells from the conventional CA neighborhood of radius r, the SDCA is said to have a* maximum neighborhood radius *of r.*

Theorem 4.3 *Given a 1-dimensional Relative Location or Symmetric SDCA $A = (S_A, 1, \sigma, \lambda)$ with k states and a maximum neighborhood radius r, there is a conventional 1-dimensional CA $B = (S_B, d, T, f)$ with $2^{2r}k$ states and neighborhood radius $2r$ that simulates A with no loss of time.*

Proof. Clearly, this restriction means that we have a BLRL or BLS SDCA. Given the bound r on the neighborhood radius of A, each cell has at most $2r$ neighbors (other than itself), and hence 2^{2r} possible connection patterns between itself

and its neighbors. Given this finite number of connection patterns, we can construct states for B that encode the current state of the cell *and* the current connection pattern of that cell. Thus, for each $s \in S_A$ we construct 2^{2r} states in S_B encoding the possible connection patterns:

$$S_B = \{\{0,1\}^r s \{0,1\}^r \mid s \in S_A\}$$

The state of a given cell in SDCA A in the next time step can be determined by examining the states of all cells in a radius of r; similarly, the state of cell i in the simulating CA B need only have access to the cells in a radius of r. In order to calculate whether an existing link from cell i to cell j in A will remain, however, we need to know the states of all the cells in the neighborhood of cell i *and* the states of all the cells in the neighborhood of cell j: that is, B needs access to the states of all cells in a radius of $2r$. The same information is necessary to determine whether two indirectly linked cells will be linked in the next time step. Hence, in order for a cell in the simulating CA to calculate its state (*i.e.*, the state and connection pattern of the corresponding cell in the SDCA), it needs to know its own state and the states of all neighbors in a radius of $2r$.

Note that we cannot use this technique to allow a conventional CA to simulate a bounded neighborhood Labeled Links SDCA. Even when the neighborhood is bounded in a Labeled Links SDCA, we still have an infinite number of labels with which to label the finite number of links and hence an infinite number of connection patterns.

Clearly, we can reduce this neighborhood radius of $2r$ by a factor of $1 \leq k \leq 2r$ by allowing the simulating automaton to operate k times slower, thereby giving the

necessary state information time to be transmitted to where it is needed. But we can do better using the *grouping* technique of Mazoyer and Reimen in [28]. Given a 1-dimensional CA A with state set S_A and an arbitrary neighborhood radius r, we can construct a 1-dimensional CA B with an arbitrary neighborhood radius p that simulates A m times faster (m arbitrary) by constructing states for B that encode the state information in $\rho = mr/p$ cells of A. Thus, each group of ρ cells in A is translated into a single cell in B, and cells in B thus have all the information they need to operate faster and with a smaller neighborhood. For example, if A has a neighborhood radius of $2r$ and we wish to simulate it using a CA with neighborhood radius of r and no speedup, we would need to form groups of $(2r/r) = 2$ in our translation of configurations in A to configurations in B. This would mean that if there were k states in the original CA, we would need k^2 states in the simulating CA. Thus, given the simulating CA in Theorem 4.3, we can simulate *that* CA with a CA with $(2^{2r}k)^2$ states, a neighborhood radius of r and no loss of time. □

Next we turn our attention to SDCA simulations of conventional CA. If an SDCA is more powerful than a conventional CA, then an arbitrary conventional CA should be able to be simulated by an SDCA either faster or, at least, with no loss of time. In fact, as the following theorem shows, we can achieve a speedup of 2 in certain simulations of this type.

Theorem 4.4 *Given an arbitrary 1-dimensional conventional CA A with neighborhood radius 1, there exists a 1-dimensional ULRL SDCA B which can simulate A with a speedup of 2.*

Proof. Assume without loss of generality that there are no quiescent cells embedded in the nonquiescent cells of the initial configuration of A (just replace any embedded quiescent cells with cells in an additional "embedded quiescent state" which is distinguishable from the quiescent state but which is treated like the quiescent state by the transition function). This nonquiescent block and its surrounding quiescent cells are transformed unchanged and in the same order to B and start out connected to their nearest left and right neighbors. In one time step, B simulates one step of A and constructs additional links twice the length of the current links between all cells whose neighborhoods are not completely quiescent. No existing links are deleted.

Now each cell which might change state during the next *two* time steps of A is connected to all the nonquiescent cells that could influence its state during these next two time steps. Thus, the transition function of B has the information it needs to simulate *two* steps of A in the next time step of B. Since a given cell is connected to all the cells that could influence it in two steps of A's evolution, and since B knows the relative location of each of these neighbors, the state transition function of B can compute the state of the cell after two time steps of A. B continues in this fashion, extending the doubled links into the quiescent portion of the simulator, simulating two steps of A at a time. □

Given the information available to a ULRL SDCA, this result is not surprising. But can we achieve comparable results using an apparently less powerful SDCA model? We answer this question by investigating a stronger question. Can we construct a *CA-universal* SDCA, using an apparently less powerful SDCA model,

which operates faster than a CA-universal *CA*?

We argued in Chapter 3 that results obtainable using infinitary SDCA models might not give us a clear picture of the additional capabilities provided by structural dynamism, since these latter capabilities might be confused with the those provided by the power inherent in an infinite rule table. According to this argument, it is not sufficient to construct, for example, a CA-universal ULS SDCA. Although we feel it to be highly likely that $ULRL \not\leq_s ULS$, and therefore that ULS SDCA are strictly weaker than ULRL SDCA, a ULS SDCA still has an infinite rule table. In order to avoid this problem, we need to construct a CA-universal SDCA using one of our naturally finitary models *i.e.*, an FLLL SDCA or BLS SDCA.

In [1], Albert and Culik constructed a 1-dimensional CA-universal conventional CA that simulates any 1-dimensional k-state CA with neighborhood radius r. The simulation, however, results in an $\mathcal{O}(k^{8r})$ slowdown. In the next chapter we will show that it is possible to construct a 1-dimensional CA-universal FLLL SDCA that is significantly faster than the Albert and Culik CA-universal CA. Specifically, we will prove the following theorem:

Theorem 4.5 *There exists a 1-dimensional FLLL SDCA which can simulate an arbitrary k-state 1-dimensional conventional CA with neighborhood radius r with* $\mathcal{O}(k^{2r}\sqrt{2r\log k})$ *slowdown.*

Although the CA-universal SDCA we construct uses substantially more states than the Albert and Culik CA-universal conventional CA (approximately 4.5 times more states), it is much faster; our CA-universal SDCA operates with an

$\mathcal{O}(k^{2r}\sqrt{2r \log k})$ slowdown, a substantial improvement over Albert and Culik's $\mathcal{O}(k^{8r})$ slowdown. The next chapter will be devoted to a description of this FLLL SDCA.

Chapter 5

An FLLL CA-Universal SDCA

5.1 Introduction and Overview

In [1], Albert and Culik construct a 1-dimensional CA-universal conventional CA that simulates any 1-dimensional k-state CA with neighborhood radius r with an $\mathcal{O}(k^{8r})$ slowdown. In this chapter we describe the construction of U_{FLLL}, a 1-dimensional CA-universal FLLL SDCA that simulates any 1-dimensional k-state CA with neighborhood radius r with an $\mathcal{O}(k^{2r}\sqrt{2r\log k})$ slowdown. Although U_{FLLL} uses more states than the Albert and Culik CA-universal conventional CA (approximately 4.5 times as many), it is significantly faster. In our description of this FLLL *simulator*, all CA and SDCA are assumed to be 1-dimensional unless otherwise noted.

Given an arbitrary conventional CA A with k states and a neighborhood radius of r, we first convert A into an equivalent CA A'' which is easier for our simulator to handle. Following this conversion, the manner in which our simulator simulates the evolution of an initial configuration of A'' is straightforward. The state of each

cell in the initial configuration of A'' is encoded in binary in a group of cells in the simulator, one cell for each bit of the state. A group of cells representing a cell of A'', say cell i, is connected with every other group of cells representing a neighbor of cell i in A''. In addition, each group of cells is connected, directly or indirectly, to a block of cells in the simulator whose states encode the local state transition function of A''. During one *cycle* of the simulator, in which time one step of A'' is simulated, the links between these groups of cells are continuously reconfigured so as to allow each group of cells encoding a state to "read" both the states of the cells in its neighborhood and the rule table. In this way, each group of cells is able to determine what state it needs to encode during the next time step.

Before we describe the details of the operation of our CA-universal FLLL SDCA, we provide some intuition of why an FLLL SDCA would allow us to build a faster simulator. First, dynamic links create something like a random access capacity in the automaton. Given $\mathcal{O}(\log d)$ time, an SDCA can create a link between any two cells a distance of d apart. So any cell in an SDCA has the potential to access the state of any other cell in the SDCA. In this respect, an SDCA acts less like an automaton and more like a CPU with random access to memory. For example, the simulator described in this chapter encodes the simulated rule table as a group of binary-tree-like blocks of cells, and the simulator performs a binary search on these structures in order to calculate the next state of a simulated cell. Although a cell in a conventional CA also has the potential to access the state of any other cell, it cannot *permanently link* to an arbitrary cell and so does not have the fast access available in an SDCA once a link has been constructed.

Second, unlike a conventional CA, the links in an SDCA can connect cells arbitrarily far apart. Since these links are dynamic, a small number of links can be used to transfer state information quickly between entire regions of the automaton arbitrarily far from each other. Thus, once in place, these links can communicate state information much more quickly than a conventional CA, which would be forced to propagate this information one cell at a time. Finally, the link labels in an FLLL SDCA allow us to encode information that facilitates the operation of the SDCA. In our CA-universal SDCA we will often use link label configurations to encode information necessary for controlling the simulation.

The most powerful aspect of an SDCA, at least potentially, is its ability to change its connection topology in response to the changing conditions under which it is operating. For example, it might be possible to build an SDCA which would act to forge a new link between two cells if those two cells exhibited a high degree of correlation in their behavior. We will have more to say about this in Chapter 6, but we note here that our simulator does not take advantage of this potential power of structural dynamism. Its connection structure, though dynamic, is always dynamic in the same way, regardless of the CA it is simulating.

We note here that the encoding and decoding functions, e and d, and the parameters defining the rate of simulation, m and n, are not fixed for all cellular automata simulated by our simulator. The specific transformation functions and rate of simulation depend on the number of states in the simulated automaton and will thus vary depending on the automaton we are simulating. The transformation functions e and d and the parameters m and n are, however, fixed for each particular

CA simulated by our simulator and so our simulation definition is satisfied for each CA simulated.

The necessary preliminary conversion of the CA A into the CA A'' is described in Section 5.2. In Section 5.3, we describe the how certain groupings of cells are organized into *blocks* of cells in the simulator. We describe the individual cells in the simulator in more detail in Section 5.4. In Sections 5.5 and 5.6 we describe the details of the operation of the simulator. Finally, in Section 5.7 we calculate the optimal value for a particular parameter of the simulator.

5.2 Preliminary Conversions

We start by converting the CA A into an equivalent CA which is easier to simulate. First, using the grouping technique discussed following the proof of Theorem 4.3, we construct a k^r-state conventional CA A' with neighborhood radius 1 which simulates A with no loss of time. Second, we construct another conventional CA, A'', which is a *1-way* version of A'.

Definition 5.1 *A conventional CA with neighborhood radius of 1 is* 1-way *if the local state transition function depends only on the state of the cell to which the transition function is being applied and the state of its right neighbor. In other words, the state of a cell has no impact on the state transition of the cell to its right.*

We use a construction from [1] to build A''. This construction works by inserting an intermediate transition step in which each cell changes to a "double-state"

which encodes both that cell's current state and its right neighbor's current state. If the original CA had j states we will need j^2 *additional* states for these double-states. After this intermediate step, each cell has in its 1-way neighborhood the state information for three consecutive cells and so can compute the next state for the middle cell of those three consecutive cells. This process shifts the nonquiescent portion of the automaton to the left by one cell and so has the impact of allowing the left boundary of the nonquiescent portion to advance to the left while preventing the right boundary of the nonquiescent portion from advancing to the right. Note that because of the intermediate transition step, required to collect the neighborhood state information that is being passed to the left, this 1-way automaton operates with a slowdown of 2. Thus, A'' is a $k^{2r} + k^r$-state *1-way* conventional CA, still with neighborhood radius 1, which simulates A' two times slower. To simplify our subsequent notation, we let $\kappa = k^{2r} + k^r$. It is this κ-state CA, A'', that our CA-universal FLLL SDCA will simulate.

Albert and Culik also convert A into a form easier to simulate, but they assume that A's neighborhood radius r is already 1. In order to make our results comparable with theirs, we have assumed that they would need to perform the same conversion we did to arrive at the CA A' with neighborhood radius 1, and would thus suffer the same increase in the size of the state set. Given this CA A' with neighborhood radius 1, Albert and Culik perform the conversion to a 1-way CA A'', but they do not stop here. Their simulator also requires that the simulated CA be totalistic (a conventional CA is totalistic if its local state transition function depends only on the *sum* of the states of the cells in the neighborhood). For this reason, they convert the 1-way CA A'' into a totalistic, 1-way CA A'''.

A''', which has $3(k^{2r}+k^r)$ states and neighborhood radius 1, simulates A'' with no loss of time and preserves its 1-way quality, by "coloring" the states of A'' in such a way that adding the states in the neighborhood of a cell yields a sum from which it is possible to extract the state and position of each of the individual neighbors. More specifically, if we have k states in A'', we construct the states of A''' in $k+1$-ary notation such that the state of a cell is represented by a single digit in a particular position in a $k+1$-ary number. If we color the cells in a neighborhood appropriately, when we add the states of the cells the sum will be a $k+1$-ary number in which there is a single 0, indicating the start of the neighborhood sequence, and the digits after the 0 will give us the states of the neighborhood sequence (wrapping around to the leftmost digit when we reach the rightmost digit).

It is this version of the CA A which Albert and Culik's simulator simulates. There is a problem, however, with an assumption they make about the nature of the states in A'''. They assume that the $3(k^{2r}+k^r)$ states in A''' can be represented by $3(k^{2r}+k^r)$ consecutive integers, but this cannot be done without losing the critical property provided by the coloring of the states. To see this, let us suppose that A'' has j states. A''' works by introducing 3 colorings of each of the states in A'' (yielding $3j$ states for A''') such that if we add any two states of different colors we get a sum that is different from the sum of any *other* two states of different colors. Thus, the operation of A''' depends on having $3j^2$ distinct sums available. But if the $3j$ states of A''' are consecutive integers, they are capable of providing only $6j-1$ distinct sums. For Albert and Culik's simulator to be able to simulate A''' in the manner they outline, A''' would need to have $\mathcal{O}(k^{4r})$ states.

This increase in the number of states in A''' has no impact on the number of states Albert and Culik require for their simulator, and it is clear that they were primarily trying to minimize the state set of their simulator. However, the increase in the states of A''' has a significant impact on the slowdown of their simulator. Their simulator works by encoding each cell of A''' in a block of cells in the simulator. Each block of cells encodes the state of a cell in A''' and the *entire* rule table of A'''. Since the number of cells required for the encoding of the rule table is quadratic in the number of states of A''', and the simulator must go through the entire rule table to simulate one time step of A''', their simulator operates with a slowdown of $\mathcal{O}(k^{8r})$ (not $\mathcal{O}(k^2)$ as claimed).

5.3 The Block Level

We assume, without loss of generality, that the κ states of the CA A'' are consecutive integers starting with 0, and that 0 is the quiescent state. A cell of the CA A'' in state i is represented in our simulator by a block of $\lceil \log \kappa \rceil$ cells containing the binary representation of i. Such a block of cells is called a *simulation module*; the details of its operation will be explained in subsequent sections. Each simulation module is connected to a binary-tree-like encoding of the rule table of A''. This allows us to store the rule table in $\mathcal{O}(\kappa^2 \log \kappa)$ cells We arrange links such that a *group* of simulation modules can share a copy of the rule table. This sharing means that we can significantly reduce the cost (in time) associated with building new rule tables to accommodate an expanding simulation. The sharing does slow the simulator down, but the slowdown can be minimized by *pipelining* the simulation modules' usage

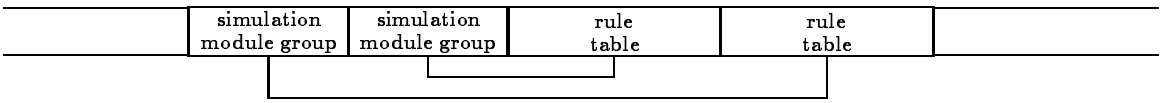

Figure 5.1: Configuration of groups simulating nonquiescent cells

of the rule table. For the purpose of describing the operation of our simulator, we will assume that s simulation modules share a rule table; in the last section of this chapter we will calculate the optimal value for s.

We take advantage of an attribute of the CA we are simulating and an attribute of SDCA to simplify the construction of our simulator. Since the simulated CA is 1-way and thus expands only to the left, and since cells arbitrarily far apart in an SDCA can be connected, we can group all the necessary copies of the rule table to the right of the simulation modules simulating the cells of the original CA. As the nonquiescent area of the simulated CA expands to the left, the active simulation modules simulating them also expand to the left, while copies of the rule table expand to the right as needed. For example, suppose that the initial configuration in the simulated CA was composed of $2s$ cells. There would then be two groups of s simulation modules each to simulate these cells, and we would need two copies of the rule table to support these groups, arranged as show in Figure 5.1. Note that a link between a simulation module group and a rule table in this diagram represents all the links that would be necessary to connect the simulation module group and the rule table. These links will be described later. In addition we construct one group of s idle simulation modules just to the left of the initial two groups to allow for expansion of the active frontier, and we connect this group to a third copy of

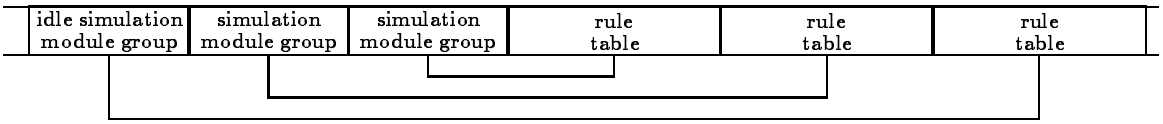

Figure 5.2: Groups simulating nonquiescent cells plus idle simulation module group

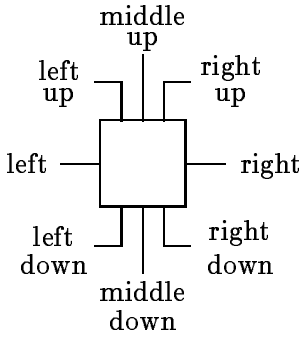

Figure 5.3: Available links in a simulator cell

the rule table (Figure 5.2). In the minimum amount of time it could take for the active frontier of simulation modules to expand completely into the group of idle simulation modules, the simulator will construct another group of idle simulation modules connected to another copy of the rule table to allow for further expansion.

In the next two sections, we will describe the operation of the two major components of the simulator: the actual simulation of the evolution of the simulated CA A'', and the construction of new simulation module groups and rule tables.

5.4 The Cellular Level

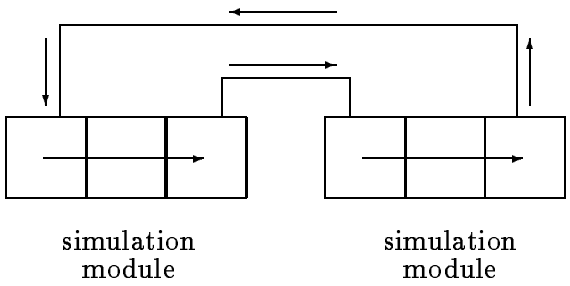

Figure 5.4: The neighborhood path

Cells in the simulator have 8 links: 1 left link, 1 right link, 3 up links (left, middle, and right), and 3 down links (left, middle, and right) as shown in Figure 5.3. Each of these links has a particular function in the operation of the simulator, though the function for a particular link may vary depending on whether it is the link of a cell in a simulation module or the link of a cell in a rule table and whether that cell is in the process of being constructed or is already in operation in the simulator.

In simulation module cells, the left and right links are used to connect the cells in a simulation module; adjacent simulation modules are not connected *via* their left and right links, but rather through their up links and down links. The left and right up links are used in conjunction with the left-right links to create, for each simulation module, a path which passes through each cell in that simulation module and each cell in the neighboring simulation module to the right (Figure 5.4).

Since we are simulating a 1-way automaton, our simulator needs to be able to "collect", for each simulation module, the $\log \kappa$ states encoding the simulated state in that simulation module and the $\log \kappa$ states encoding the simulated state in that simulation module's right neighbor. This path allows the simulator to process this

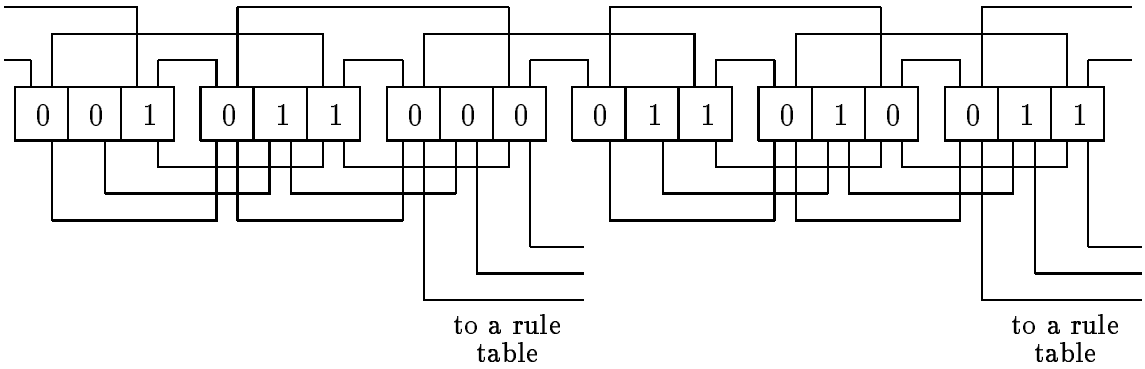

Figure 5.5: Two groups of three operational simulation modules

information for each simulation module by adjusting the down links connecting the simulation module to its rule table so as to follow the path. The middle up links are not used in simulation module operation. The down links serve to connect the simulation modules to the rule table. We will call the rightmost simulation module in a group the *gatekeeper* since this module controls access to the rule table for all the other simulation modules in its group. The down links of the gatekeeper connect it directly to the rule table. The down links of all the other simulation modules in a group connect each cell in a simulation module to the corresponding cell in the simulation module to the right, thus indirectly connecting all of these simulation modules to the rule table. Up links and down links are also used in the construction of simulation module cells. Two groups of three operational simulation modules can be visualized as shown in Figure 5.5.

The rule table is composed of $\lceil \log \kappa \rceil$ blocks (one for each of the $\lceil \log \kappa \rceil$ "bits", or cells, encoding a simulated state in a simulation module). Each block has a topology similar to that of a binary tree and; we will call them *rule table trees*. The idea here is that each rule table tree encodes the information necessary for the

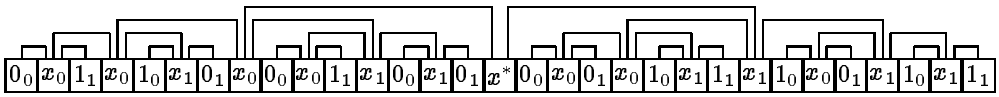

Figure 5.6: An operational binary tree in a rule table

simulation module cell to which it is connected to determine what state it should be in in the next step of simulated time. In a rule table, the left and right links are necessary only in the construction of the rule table but, with one exception, they are left intact after construction. The left-right links between each rule table tree are severed after construction to separate the trees. The up links provide the pathways through the rule table trees. The left and right up links are links to a cell's children in the tree (not used in leaf cells) and the middle up link is a link to a cell's parent in the tree (not used in the root cell). Down links are used to connect the rule table to the simulation modules. During simulation, the simulation modules in each group traverse, in a pipelined fashion, a root-to-leaf path in each rule table tree to obtain the information needed to determine the states of their cells in the next simulated time step. Each group of simulation modules does this in parallel. The down links from the rule table cells (which connect to down links of the simulation module cells) indicate which cell in the tree a simulation module cell is currently connected to. Up links and down links are also used during construction of the rule table. An operational binary tree in a rule table can be visualized as shown in Figure 5.6.

The state of each cell in the rule table, except for the root cell, is subscripted with a 0 or a 1 to indicate whether it is its parent's left or right child respectively. These subscripts are necessary for the simulation modules to navigate correctly through their associated rule table trees. We will explain this further in the next

section.

5.5 Simulation of Evolution

The manner in which the simulator simulates the evolution of A'' is quite simple. The state of a cell in A'' is encoded in binary in a simulation module in the simulator. Each cell, or "bit" of the state, in the simulation module has access to the appropriate tree in a rule table. These links move through a $2\lceil \log \kappa \rceil$ circuit of cells comprised of the cells in their own simulation module and the cells in the simulation module to their right (the right neighbor). The 0/1 values that the links encounter on this circuit guide the other end of the links down particular paths in the rule table trees to which they are connected. When each link completes its circuit, it is connected to a leaf in a tree which contains the new value for one of the cells in the simulation module (the cell immediately to the right of the current connection in the simulation module). The cells in the simulation module copy the correct states for the next simulated time step from the leaves and the links travel back up the rule table trees to the root in preparation for the next simulation cycle.

To describe the operation of a group of simulation modules we will start at a point where each simulation module in the group contains the binary representation of the current state of the cell it is simulating. To simplify the explanation, we will focus on the operation of the gatekeeper, referring to the operation of the other simulation modules as necessary to describe their participation.

Each cell in the gatekeeper is linked *via* its left down link to the root cell of

the appropriate rule table tree in the corresponding rule table. (Each cell in a non-gatekeeper simulation module is connected *via* its right down link to the left down link of the corresponding cell in the neighboring simulation module on the right.) In one time step the gatekeeper down links dissolve and reconnect to the left-child cell of the root (if the simulation module cell is in state 0) or the right-child cell of the root (if the simulation module cell is in state 1). This is where we need the 0/1 subscripts in the rule table tree states. A link to a rule table tree cell can always forge a new link to either the left or right child of that cell. If a link to the left child is indicated, it will link to the cell with a 0 subscript; if a link to the right child is needed, it will link to the cell with a 1 subscript.

The states of the cells in the simulation module change to indicate that a rotation of the links in the simulation module must take place. (This change of state is indicated in our diagrams by an r subscript.) In the next time step, all links from the simulation module cells to the rule table rotate one cell to the right in the simulation module, thereby connecting the next cell in the simulation module to the current point in the root-to-leaf path that is being navigated in the rule table trees. The link attached to the right-most cell in each simulation module will now be attached to the left-most cell of the neighboring simulation module on the right. This 2-step process is repeated $2\lceil \log \kappa \rceil$ times, the values of the cells in each simulation module and its neighboring simulation module to the right guiding the root-to-leaf paths in the binary trees.

We note here that the simulator uses the link labels to keep track of where it is in the simulation modules. Down links from the simulation module cells to the

rule table start out as *left* down links (from the perspective of the simulation module cells). As soon as a link crosses over into the neighboring simulation module to the right, it connects to the simulation module cell *via* that cell's *right* down link. When that link has processed the right-most cell of the neighboring simulation module and comes back to the left-most cell of the home simulation module, it links to that cell *via* that cell's *middle* down link.

When the down links of a simulation module have completed a circuit of the $2\lceil \log \kappa \rceil$ cells in that simulation module and right neighbor simulation module, they will be connecting the leaf of a rule table tree to the cell in the simulation module *just to the left* of the cell which should assume the value stored in the leaf for the next step of simulated time. In other words, the rule table trees are constructed such that when leaves have been reached in each tree (simultaneously) each leaf contains the state (0 or 1, ignoring subscripts) which the *next* simulation module cell in rotation should assume during the next step of simulated time. For example, suppose that the cells in a 2-cell simulation module S_1 and its right neighbor simulation module S_2 at the beginning of this process are in states 01 and 10 respectively, and that the rule table of the simulated automaton specifies that when a cell is in state 1 (01) and its right neighbor is in state 2 (10), that cell should be in state 2 (10) in the next time step. The states in the rule table trees would be set so that when leaves are reached in the trees, one more rotation of simulation module links will result in the left cell of simulation module S_1 being connected to a 1 leaf and the right cell of simulation module S_1 being connected to a 0 leaf after the root-to-leaf trips take place. The simulation module cells copy their new states from the leaves, obliterating their old states. We can afford to lose the old states at this point since the left neighbor

simulation module (the module that needed these old states to calculate *its* new states) has already been through this part of its circuit and no longer needs the old states.

The non-gatekeeper simulation modules in the group gain access to the rule table trees by "following" the rule table links of the gatekeeper. In other words, in the same time step in which the gatekeeper makes its first move down the rule table trees, the next simulation module (to the left of the gatekeeper) uses its links to the gatekeeper to forge new links to the roots of the rule table trees. While the gatekeeper is rotating its links, the next simulation module makes *its* first move down the binary trees. Because there is a 1-step lag in a simulation module cell's navigation of the trees (to allow for rotation of the links in the simulation module), links from two adjacent simulation modules will collide. We use the left and right down links of the rule table tree cells to accommodate this collision. Whenever a simulation module cell first connects to a rule table tree cell, it connects *via* the *right* down link of the rule table tree cell. In the next time step of the simulation, at the same time that this link is rotating in the simulation module, it switches to the *left* down link of the rule table tree cell, thus leaving the right link of that rule table cell free for the next simulation module link which wants to connect to that cell. In this pipelined manner, each simulation module in the group gains access to the rule table in $\mathcal{O}(s)$ time.

When the cells of a simulation module have their new values for the next simulated time step they go into a state which indicates that they must make their way back *up* the binary trees in preparation for the next cycle of the simulation.

(This state is indicated in our diagrams by a u subscript.) This upward movement takes place *via* the middle down links of the rule table cells and is accomplished with no time lag since no rotation of the links is necessary during the upward trip. When the down links of a simulation module have returned to their original state (the gatekeeper connected to the rule table tree root cells and all other simulation modules connected to the simulation module to the right) the simulation module cells return to their original (unsubscripted) states, indicating that they are ready for the next simulation cycle.

A simulation module waiting for the next simulation cycle will wait until the simulation module to its right indicates that *it* is ready for the next simulation cycle as well. So the next simulation cycle for a group of simulation modules will not start until the gatekeeper is ready for the next cycle. But the gatekeeper will not be ready until its right neighbor simulation module (which is the left-most simulation module of the group to the right of the gatekeeper's group) is ready. And this left-most simulation module in the group to the right will not be ready exactly when every left-most simulation module in every group is ready, *i.e.*, when all the simulation modules in each group have their new values. Thus, the next simulation cycle will start exactly when all the simulation modules in the groups have completed the current simulation cycle.

5.6 Construction of Simulation Module Groups and Rule Tables

The construction of new groups of simulation modules and the construction of new rule tables proceed along similar general lines. A segment of cells in the simulator are given states in the initial configuration that encode the instructions for building the new section. The appropriate links are constructed in the new section, after which the cells in the new section enter the appropriate initial states. This link construction proceeds from right to left in simulation module groups and from root to leaves in rule table trees. Concurrently with link construction, the initial instruction states are copied to the quiescent areas immediately to the left (right) of the current simulation module group (rule table) construction, thus setting the stage for another round of construction.

5.6.1 Simulation Module Group Construction

Suppose we wish to construct a group of three simulation modules, each of which contains three cells. The initial new construction segment with its instruction states would appear as show in Figure 5.7.

Each simulation module cell is in state l (the leftmost cell), m (a middle cell), or r (the rightmost cell). The gatekeeper is marked by special states, here indicated by asterisk superscripts. The rightmost gatekeeper cells in the area of new construction are already linked to the leftmost cell simulator in the group of idle simulation modules to the right of the new construction through their *circuit return* (cr) and

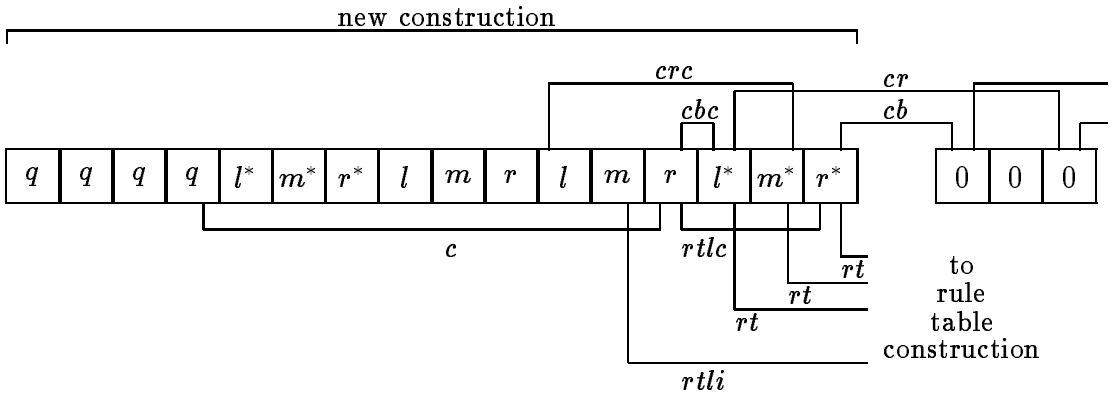

Figure 5.7: Construction segment for a simulation module group

circuit bridge (*cb*) links. The *crc* (*circuit return constructor*) and *cbc* (*circuit bridge constructors*) construct the necessary circuit return and circuit bridge links, and operate as follows. Each end of the link moves one cell to the left in alternating time steps. When the link connects two cells which need to be linked (*i.e.*, if the *crc* is connecting an l (or l^*) cell on the left and an r (or r^*) cell on the right, or a *cbc* is connecting an r (or r^*) cell on the left and an l (or l^*) cell on the right) a link is left behind as the constructor moves forward. For example, the *crc* in our diagram is connecting an l cell on the left and an r^* cell on the right, and so needs to leave a link behind as it moves forward. In one time step, the current link connecting the l cell and the r^* cell dissolves and is replaced by a link between the r cell (to the left of the l cell) and the r^* cell. In the next time step, this link dissolves and new links are created between the l and r^* cells and the r and m^* cells immediately to their left. We now have a circuit return link between the l and r^* cells and the *crc* has advanced to the left. This process repeats continuously, forging addition links where necessary and advancing. The *cbc* operates in a similar fashion.

The rule table (rt) links connect the gatekeeper cells to the leftmost cells of the

rule table which is being constructed at the same time as the new group of simulation modules. These links will remain connected to the gatekeeper cells throughout the period of new construction. The rule table cells to which they are connected will change during construction, but we will describe this when we deal with rule table construction. The rule table link constructor (*rtlc*) operates in a fashion similar to that of the *crc* and *cbc*, except that it leaves a link behind every time it moves forward, thus creating links between each cell in a simulation module and its corresponding cell in both the left neighbor simulation module and the right neighbor simulation module.

The rule table link initiator (*rtli*) starts out linked to the cell two cells to the left of the first gatekeeper in the new construction. We do this to avoid link congestion in the construction process; the *rtli* is always slightly ahead of the circuit return, circuit bridge, and rule table link constructors. The function of the *rtli* is to set up the initial links between the gatekeeper and its corresponding rule table. This link alternately moves one cell to the left in the new simulation module group construction and one cell to the right in the new rule table construction. When this link encounters the beginning of a new gatekeeper (an r^* cell) it remains linked to this cell, advancing the right end of the link until it encounters the first cells of the corresponding rule table. When it finds these, it constructs the necessary links between the gatekeeper and the rule table, and moves on, looking for the next gatekeeper.

The link marked *c* is the copier link. It copies the construction instructions in the construction section into the quiescent cells to the left of the construction

section, laying the groundwork for the next round of construction. Whenever the left end of this link is connected to a quiescent cell, it copies the state of the cell at its right end into the quiescent cell and moves its right end one cell to the left. In the next time step, it advances its left end one cell to the left so that the left end is connected to the next quiescent cell in line, and the entire process is repeated. In this fashion, the construction instructions are copied at a rate of one instruction every two time steps. Since the links are constructed immediately behind this copier wave, the entire construction process is accomplished at the same rate.

As soon as the cr, cb, and rt links of a simulation module have been constructed, that cell simulator detaches itself from the area of new construction. A gatekeeper, at this point, will wait until its rt links (which have been moving to the right in the rule table construction area) have connected to the root cell in the correct binary tree in its corresponding rule table. As each simulation module cell makes this connection, it enters the 0 state. When all the simulation module cells have entered the 0 state, the simulation module is a fully operational, though still idle simulation module. When a non-gatekeeper simulation module detaches from the area of construction, it waits until the simulation module to its right becomes operational, at which point its cells enter the 0 state and it becomes operational as well.

5.6.2 Rule Table Construction

We will describe the construction of a single rule table tree (construction of all the trees in a rule table occurs in parallel). Suppose we wish to construct a binary tree with sixteen leaves whose values are 0110010000111011 from left to right. The

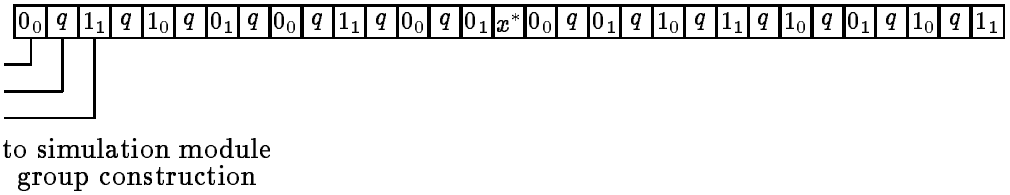

Figure 5.8: Construction segment for a rule table binary tree

construction instructions would be as shown in Figure 5.8. (The down links are the initial construction links attaching the rule table to its corresponding simulation module group. They were described in the Simulation Module Group Construction section and their behavior will be explained after we describe the construction of the tree links.)

A subscript of 0 (1) indicates that that cell is the left (right) child of the cell to its immediate right (left). These subscripts allow us to construct the parent links of the leaves in the first two steps of construction. In the first time step, a cell with a 0 (1) subscript constructs a link from its middle up link to the left (right) up link of the cell two cells to its right (left). This link dissolves in the next time step, but allows the cell to construct a similar link between itself and the cell to its immediate left (right). The parent links of the leaves are now in place.

The rest of the tree links are constructed from the root down. In a two-step process like that described in the previous paragraph, the root cell (in state x^*) forms links with the cells immediately to its left and right. These links move to the left (right) at one-third speed (*i.e.* the left (right) end of the link moves one cell to the left (right) every third time step). At the same time the root cell sends out two

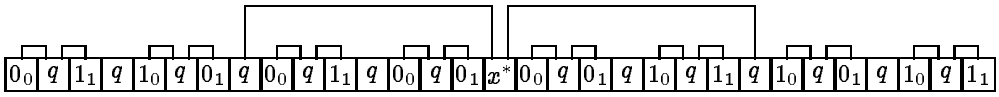

Figure 5.9: Intermediate stage of rule table binary tree construction

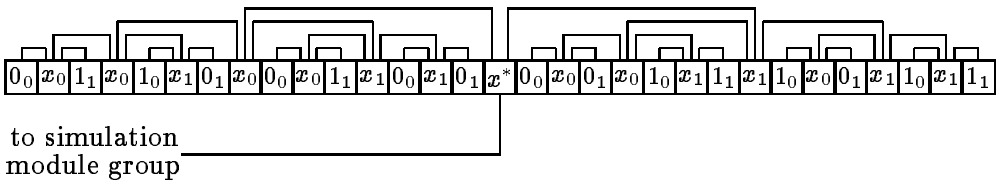

Figure 5.10: Completed rule table binary tree

signals moving at full speed (one cell per time step), one to the right and one to the left. These signals are reflected off the left (right) end of the tree and move back toward the root. The left (right) link and the left (right) signal meet in the cell which is the left (right) child of the root. The nature of the signal indicates that this is a left (right) child and so that cell enters the state x_0 (x_1). At this point, the tree appears as shown in Figure 5.9. The rest of the tree links are constructed recursively following the same process. The finished tree appears as show in Figure 5.10.

The rt links move their right end one cell to the right at every time step. When the first link (which is the link from the l^* cell in the gatekeeper) reaches the root cell it waits until the link behind it (from the m^* gatekeeper cell) catches up. At this point, the root cell has two down links and enters the state x^2, indicating this fact. The l^* link moves one cell to the right, while the m^* link waits for the link following it (in this case the r^* link) to catch up. In this fashion, all the rt links pass through

the root cell, which enters state x^2 as soon as the first two rt links connect to the root cell and remains in that state until only one link (the r^* link is connected to it). The last link to pass through the root cell when it is in state x^2 remains there, which is exactly what we want—the rt links are arranged such that every time they pass over a root cell in the rule table, they leave the last link in their sequence attached to that root. In this way, the rt links are attached correctly to the roots of the trees in the rule table.

The $rtli$ was described in the Simulation Module Group Construction Section. Finally, the c link is a copier link which copies the initial construction instructions for a rule table into the section of quiescent cells immediately to the right of the current construction. Its operation is analogous to that of the c link in the Simulation Module Group construction, but one additional point needs to be made. The tree links are being constructed at the same time that the copier link is copying, so the copier may encounter cells whose state has been changed by the tree link construction. In most cases, the original instruction state can be recovered from the new state. The only cells which will have been changed beyond recognition are the quiescent cells that get changed to x_0 or x_1. When the copier encounters cells in this state it copies nothing, leaving the cell to which it is copying in the quiescent state, and moves on to the next cell.

5.7 Calculation of Simulation Module Group Size

We want a simulation module group to be as small as possible (to minimize the slowdown induced by the process of sharing a rule table) yet large enough to give

us sufficient time to construct a new simulation module group and a new rule table, both of which are constructed at a rate of one new cell every two time steps. We will assume that the rule table is the larger of the two constructions, calculate the optimal value of s, the number of simulation modules in a group, to give us sufficient time to construct a new rule table, and then check to see that this also gives us sufficient time to construct a new simulation module group.

Consider the root-to-leaf trip performed during the simulation of a step of A''. Since there are $2\lceil \log \kappa \rceil$ simulation module cells in the circuit guiding this trip, and each of these cells advances the link to the next level in the tree, we will need a tree with a depth of $2\lceil \log \kappa \rceil$. This means that each tree in a rule table will contain $2^{2\lceil \log \kappa \rceil + 1} - 1$ cells. Since we need a tree for each of the $\lceil \log \kappa \rceil$ cells in a simulation module, the entire rule table will need a total of $\lceil \log \kappa \rceil (2^{2\lceil \log \kappa \rceil + 1} - 1)$ cells. Each cell requires two time steps to construct, so we will need $2\lceil \log \kappa \rceil (2^{2\lceil \log \kappa \rceil + 1} - 1)$ time steps to construct a rule table.

If there are s simulation modules in a group, we have, at the beginning of the simulation, s idle simulation modules to expand into if necessary. This means we have st time steps to construct another rule table (and simulation module group), where t is the time needed to simulate one step of A'' (since each simulated step of A'' may "use up" one of our idle simulation modules). The size of t is determined by the left-most simulation module in the group; it will take this module the most time to link to the rule table (since it must wait its turn behind all the other modules in its group) and the most time for its links to return to their initial position. Since we are pipelining this process, it will need $s - 1$ time steps to get to the roots of the

rule table trees. The trip down the tree takes $4\lceil \log \kappa \rceil$ time steps (2 time steps for each of the $2\lceil \log \kappa \rceil$ levels in the tree) and the trip back up the tree takes $2\lceil \log \kappa \rceil$ time steps (1 step per level). Finally, it will take $s - 1$ time steps for this module to "walk" back down the line of simulation modules in the group and restore its links to their original position. Thus:

$$t = 2s + 6\lceil \log \kappa \rceil - 2$$

and we have

$$st = 2s^2 + (6\lceil \log \kappa \rceil - 2)s$$

time steps for our construction work. Solving the following quadratic equation gives us the optimal value for s:

$$2s^2 + (6\lceil \log \kappa \rceil - 2)s = 2\lceil \log \kappa \rceil (2^{2\lceil \log \kappa \rceil + 1} - 1) \tag{5.1}$$

To get an order estimate for s, we can simplify Equation 5.1 as follows:

$$s^2 + (\log \kappa)s - \kappa^2 \log \kappa = 0$$

which gives us an $\mathcal{O}(\kappa\sqrt{\log \kappa})$ approximation for s. More careful analysis yields a value of $1.5\kappa\sqrt{\log \kappa}$ for s.

Is this value of s sufficient for the construction of a simulation module group? Since there are $\lceil \log \kappa \rceil$ cells in each simulation module, there are $1.5\kappa\lceil \log \kappa \rceil \sqrt{\log \kappa}$ cells in a group, each requiring two time steps to construct. We have

$$st = 4.5\kappa^2 \log \kappa + 9\kappa\lceil \log \kappa \rceil \sqrt{\log \kappa} - 3\kappa\sqrt{\log \kappa}$$

times steps for this construction, which is clearly sufficient.

Now that we have a value for s we can calculate t, the slowdown of our simulator:

$$t = 3\kappa\sqrt{\log \kappa} + 6\lceil \log \kappa \rceil - 2.$$

Thus, our simulator simulates A'' with an $\mathcal{O}(\kappa\sqrt{\log \kappa})$ slowdown. Since A'' simulates A' two times slower, and A' simulates A with no loss of time, our simulator also simulates A with an $\mathcal{O}(\kappa\sqrt{\log \kappa})$ slowdown. Substituting in the actual value of κ (in terms of k, the number of states in the original arbitrary CA A, and r, the neighborhood radius of A), we arrive at an $\mathcal{O}(k^{2r}\sqrt{2r \log k})$ slowdown for our simulator.

Chapter 6

Conclusions and Further Work

Our results in Chapters 4 and 5 show that an SDCA is a more efficient model of computation than a conventional CA. We showed that a ULRL SDCA can simulate a conventional CA with a speedup of two. Even when we restricted ourselves to an FLLL SDCA, one of our two finitary models, we were able to construct a CA-universal FLLL SDCA that was significantly faster than the CA-universal conventional CA in [1].

An SDCA can solve the Firing Squad Synchronization Problem in log time, an exponential improvement over the optimal linear time required by a conventional CA. It follows from this result that any SCDA-universal conventional CA will suffer an exponential slowdown. If we restrict certain capabilities of the SDCA, a simulating CA can do better. We showed that if we put a bound on the maximum neighborhood radius of an SDCA, thereby destroying its ability to connect cells arbitrarily far apart, a conventional CA can simulate that SDCA with no loss of time. We note here that putting a bound on the *number* of neighbors in an SDCA, rather than the

connection distance, did not result in an SDCA which could be easily simulated by a conventional CA, suggesting that an SDCA's ability to connect cells arbitrarily far apart contributes more to the SDCA's capabilities than its ability to allow each cell an arbitrary number of neighbors.

Each of our results depends critically on one or both of the following abilities of an SDCA:

- the ability to connect two arbitrarily distant cells i and j in $\log D_{ij}$ time
- the ability to connect arbitrarily many cells to a given cell (in unbounded links models)

These abilities allow an SDCA to transmit information from one cell to another much more efficiently than a conventional CA. Furthermore, in an unbounded links model, any given cell can transmit/accept information to/from arbitrarily many other cells. It seems to be this ability to expedite information flow that gives an SDCA much of its power.

None of our results, however, exploit what is potentially the most powerful aspect of an SDCA—its ability to modify its connection structure in unexpected ways in response to the demands of the computation. While the links in each of the SDCA in our results are dynamic, they are dynamic in a fixed way that depends on the initial configuration and is not modified by the subsequent computation. For example, the links in our CA-universal FLLL SDCA in Chapter 5 are fixed initially depending on the CA we are simulating, and then change in a predictable fashion regardless of the computation it is simulating. It would be interesting to try to create

an SDCA which modifies its connection structure in a way which *does* depend on the evolution of the computation. An SDCA which models some aspect of the learning process (or any process which must be responsive to the current state of the process) might profitably be able to exploit this aspect of an SDCA. Could we construct an SDCA that would operate like a computer that can change its hardware, constructing links, say, between memory locations that frequently exchange data? In this respect, we note the similarities between SDCA and the Hardware Modification Machines (HMM) of [6]. Each unit of an HMM is a finite state machine that accepts inputs from a bounded number of other units and produces an output based on these inputs and its own current state. The units operate in parallel and, at each time step, a unit can redirect one or more of its input links to any other unit that is two or fewer links away from it. The primary difference between an SDCA and an HMM is that the links in an HMM are unidirectional; each unit is allowed only a bounded number of input links, but an unlimited number of units are allowed to accept the output of that unit.

We investigated the ability of SDCA to act as universal simulators of conventional CA. It would be interesting to explore their ability to act as universal simulators of SDCA. Can we create a reasonable SDCA-universal SDCA for a particular class of SDCA?

It would also be interesting to try to use the theory of communication complexity to derive some lower bounds for ULRL SDCA. This theory studies the amount of information two cooperating parties must exchange in order to solve a particular problem. As we noted when defining the Relative Location Model, the only sig-

nificant limitation of this model is the time required for two cells to communicate with each other. Given this communications bottleneck, can we use results from communication complexity to put lower bounds on this powerful SCDA model?

Finally, how realistic are these SDCA models? Could any of the bounded links models, for example, be used as the basis for a practical parallel computer? What would the principles of organization be for such a system? Would it be possible to make an SDCA-based system fault-tolerant?

Bibliography

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

[2] M. A. Arbib and J. A. Robinson. *Natural and Artificial Parallel Computation.* MIT Press, 1990.

[3] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways For Your Mathematical Plays*, volume 2. Academic Press, 1982.

[4] E. F. Codd. *Cellular Automata.* Academic Press, 1968.

[5] M. D. Davis and E. J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science.* Academic Press, 1983.

[6] P. W. Dymond and S. A. Cook. Hardware complexity and parallel computation. In *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science*, pages 360–372, 1980.

[7] P. Gács. Statement of research interests, 1993.

[8] P. Gács and J. Reif. A simple three-dimensional real-time reliable cellular array. Technical Report 85/002, Boston University, August 1985.

[9] Jr. H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967. Reprinted. MIT Press. 1987.

[10] P. Halpern and G. Caltagirone. Behavior of topological cellular automata. *Complex Systems*, 4:623–651, 1990.

[11] G. A. Hedlund. Endomorphisms and automorphisms of the shift dynamical system. *Mathematical Systems Theory*, 3:320–375, 1969.

[12] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Co., 1979.

[13] M. Hurley. Attractors: persistence and density of their basins. *Transactions of the American Mathematical Society*, 269(1):247–271, 1982.

[14] M. Hurley. Attractors in cellular automata. *Ergodic Theory and Dynamical Systems*, 10:131–140, 1990.

[15] M. Hurley. Varieties of periodic attractor in cellular automata. *Transactions of the American Mathematical Society*, 326(2):701–726, 1991.

[16] M. Hurley. Attractors in restricted cellular automata. *Proceedings of the American Mathematical Society*, 115(2):563–571, 1992.

[17] K. Culik II, L. P. Hurd, and S. Yu. Computation theoretic aspects of cellular automata. *Physica D*, 45:357–378, 1990.

[18] K. Culik II, J. Pachl, and S. Yu. On the limit sets of cellular automata. *SIAM Journal on Computing*, 18(4):831–842, 1989.

[19] A. R. Smith III. Simple computation-universal cellular spaces. *Journal of the ACM*, 18(3):336–353, 1971.

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

[21] A. Ilachinski and P. Halpern. Structurally dynamic cellular automata. *Complex Systems*, 1:503–527, 1987.

[22] J. Kari. The nilpotency problem of one-dimensional cellular automata. *SIAM Journal on Computing*, 21(3):571–586, 1992.

[23] C. G. Langton. Studying artificial life with cellular automata. *Physica D*, 22:120–149, 1986.

[24] C. G. Langton, editor. *Artificial Life*. Addison-Wesley Publishing Co., 1989.

[25] C. G. Langton. *Computation at the edge of chaos: Phase transitions and emergent computation*. PhD thesis, University of Michigan, 1991.

[26] C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors. *Artificial Life II*. Addison-Wesley Publishing Co., 1992.

[27] P. Manneville, N. Boccara, G. Y. Vichniac, and R. Bidaux, editors. *Cellular Automata and Modeling of Complex Physical Systems*. Springer-Verlag, 1989.

[28] J. Mazoyer and N. Reimen. A linear speed-up theorem for cellular automata. *Theoretical Computer Science*, 101:59–98, 1992.

[29] A. J. Sieradski. *An Introduction to Topology and Homotopy*. PWS-KENT Publishing Company, 1992.

[30] R. I. Soare. *Recursively Enumerable Sets and Degrees*. Springer-Verlag, 1987.

[31] K. Svozil. Are quantum fields cellular automata? *Physics Letters A*, 119(4):153–156, 1986.

[32] J. von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966. edited and completed by A. W. Burks.

[33] S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, 1983.

[34] S. Wolfram. Computation theory of cellular automata. *Communications in Mathematical Physics*, 96:15–57, 1984.

[35] S. Wolfram. Universality and complexity in cellular automata. *Physica D*, 10:1–35, 1984.