Chapter 1

Programming Cellular Machines by Cellular Programming

Moshe Sipper

1.1 Introduction

The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen impressive growth in the past few years. Usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, one finds such diverse domains as genetic algorithms, evolution strategies, evolutionary programming, and genetic programming. Central to all these different methodologies is the idea of solving problems by evolving an initially random pool of possible solutions, through the application of "genetic" operators, such that in time "fitter" (i.e., better) solutions emerge (Chapter **??**).

Research in these areas has traditionally centered on proving theoretical aspects, such as convergence properties, effects of different algorithmic parameters, and so on, or on making headway in new application domains, such as constraint optimization problems, image processing, neural network evolution, and more. The implementation of an evolutionary algorithm,

[1]The author is with the Logic Systems Laboratory, Swiss Federal Institute of Technology, IN-Ecublens, CH-1015 Lausanne, Switzerland. E-mail: {moshe.sipper}@di.epfl.ch.

an issue which usually remains in the background, is quite costly in many cases, since populations of solutions are involved, possibly coupled with computation-intensive fitness evaluations. One possible solution is to parallelize the process, an idea which has been explored to some extent in recent years (Chapter **??**). While posing no major problems in principle, this may require judicious modifications of existing algorithms or the introduction of new ones in order to meet the constraints of a given parallel machine.

In this chapter a different approach is taken; rather than ask ourselves how to better implement a specific algorithm on a given hardware platform, we pose the more general question of whether machines can be made to evolve. While this idea finds its origins in the cybernetics movement of the 1940s and 1950s, it has recently resurged in the form of the nascent field of bio-inspired systems and evolvable hardware [19]. The field draws on ideas from evolutionary computation as well as on recent hardware developments.

Our evolving machines are based on the cellular automata model (Chapter **??**). Cellular automata (CA) are dynamical systems in which space and time are discrete. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps according to a local, identical interaction rule. The state of a cell at the next time step is determined by the previous states of a surrounding neighborhood of cells. This transition is usually specified in the form of a *rule table*, delineating the cell's next state for each possible neighborhood configuration. The cellular array (grid) is n-dimensional, where $n = 1, 2, 3$ is used in practice (in this chapter we shall concentrate on $n = 1$ and $n = 2$). A one-dimensional CA is illustrated in Figure 1.1 (based on [15]).

CAs exhibit three notable features, namely, massive parallelism, locality of cellular interactions, and simplicity of basic components (cells). As such they are naturally suited for hardware implementation, with the potential of exhibiting extremely fast and reliable computation that is robust to noisy input data and component failure. A major impediment preventing ubiquitous computing with CAs stems from the difficulty of utilizing their complex behavior to perform useful computations. Designing CAs to exhibit a specific behavior or to perform a particular task is highly complicated, thus severely limiting their applications. This results from the local dynamics of the system, which renders the design of local rules to perform global computational tasks extremely arduous. Automating the design (programming) process would greatly enhance the viability of CAs [16, 25].

The model investigated in this chapter is an extension of the CA model,

Rule table:

neighborhood:	111	110	101	100	011	010	001	000
output bit:	1	1	1	0	1	0	0	0

Grid:

$t = 0$	0	1	1	0	1	0	1	1	0	1	1	0	0	1	1
$t = 1$	1	1	1	1	0	1	1	1	1	1	1	0	0	1	1

Figure 1.1: Illustration of a one-dimensional, 2-state CA. The connectivity radius is $r = 1$, meaning that each cell has two neighbors, one to its immediate left and one to its immediate right. Grid size is $N = 15$. The rule table for updating the grid is shown on top. The grid configuration over one time step is shown at the bottom. Spatially periodic boundary conditions are applied, meaning that the grid is viewed as a circle, with the leftmost and rightmost cells each acting as the other's neighbor.

termed *non-uniform cellular automata* [20] (see Chapter **??**). Such automata function in the same way as uniform ones, the only difference being in the cellular rules that need not be identical for all cells. Our main focus is on the *evolution* of non-uniform CAs to perform computational tasks, using the cellular programming approach. As noted in Chapter **??**, the input to the computation is encoded as an initial configuration and the output is the configuration after a certain number of time steps. We shall first introduce the algorithm, followed by several problems to which it has been applied. We then study a number of related issues, including the evolution of connectivity architectures, asynchronous CAs, evolving ware (evolware), and faulty CAs.

1.2 The cellular programming algorithm

We study 2-state, non-uniform CAs, in which each cell may contain a different rule. A cell's rule table is encoded as a bit string (the "genome"), containing the next-state (output) bits for all possible neighborhood configurations, listed in lexicographic order; e.g., for CAs with $r = 1$, the genome

consists of 8 bits, where the bit at position 0 is the state to which neighborhood configuration 000 is mapped to and so on until bit 7, corresponding to neighborhood configuration 111. Rather than employ a *population* of evolving, uniform CAs, as with genetic algorithm approaches, our algorithm involves a *single*, non-uniform CA of size N, with cell rules initialized at random. Initial configurations are then generated at random, in accordance with the task at hand, and for each one the CA is run for M time steps. Each cell's *fitness* is accumulated over $C = 300$ initial configurations, where a single run's score is 1 if the cell is in the correct state after M iterations, and 0 otherwise. After every C configurations evolution of rules occurs by applying crossover and mutation. This evolutionary process is performed in a completely *local* manner, where genetic operators are applied only between directly connected cells. It is driven by $nf_i(c)$, the number of fitter neighbors of cell i after c configurations. The pseudo-code of the algorithm is delineated in Figure 1.2.

Crossover between two rules is performed by selecting at random (with uniform probability) a single crossover point and creating a new rule by combining the first rule's bit string before the crossover point with the second rule's bit string from this point onward. Mutation is applied to the bit string of a rule with probability 0.001 per bit.

There are two main differences between the cellular programming algorithm and the standard genetic algorithm (Chapter **??**): (a) The latter involves a population of evolving, uniform CAs; all CAs are *ranked* according to fitness, with crossover occurring between *any* two individuals in the population. Thus, while the CA runs in accordance with a local rule, evolution proceeds in a *global* manner. In contrast, the cellular programming algorithm proceeds *locally* in the sense that each cell has access only to its locale, not only during the run but also during the evolutionary phase, and no global fitness ranking is performed. (b) The standard genetic algorithm involves a population of *independent* problem solutions; the CAs in the population are assigned fitness values independent of one another, and interact only through the genetic operators in order to produce the next generation. In contrast, our CA *coevolves* since each cell's fitness depends upon its evolving neighbors. This may also be considered a form of symbiotic cooperation, which falls, as does coevolution, under the general heading of "ecological" interactions (see [15], pages 182-183).

This latter point comprises a prime difference between our algorithm and parallel genetic algorithms, which have attracted attention over the past few years. These aim to exploit the inherent parallelism of evolution-

```
for each cell i in CA do in parallel
    initialize rule table of cell i
    f_i = 0 { fitness value }
end parallel for
c = 0 { initial configurations counter }
while not done do
    generate a random initial configuration
    run CA on initial configuration for M time steps
    for each cell i do in parallel
        if cell i is in the correct final state then
            f_i = f_i + 1
        end if
    end parallel for
    c = c + 1
    if c mod C = 0 then { evolve every C configurations}
        for each cell i do in parallel
            compute nf_i(c) { number of fitter neighbors }
            if nf_i(c) = 0 then rule i is left unchanged
            else if nf_i(c) = 1 then replace rule i with the fitter neighboring rule,
                                    followed by mutation
            else if nf_i(c) = 2 then replace rule i with the crossover of the two fitter
                                    neighboring rules, followed by mutation
            else if nf_i(c) > 2 then replace rule i with the crossover of two randomly
                                    chosen fitter neighboring rules, followed by mutation
                                    (this case can occur if the cellular neighborhood includes
                                    more than two cells)
            end if
            f_i = 0
        end parallel for
    end if
end while
```

Figure 1.2: Pseudo-code of the cellular programming algorithm.

ary algorithms, thereby decreasing computation time and enhancing performance (Chapter **??**). A number of models have been suggested, among them coarse-grained, island models [4, 37, 38], and fine-grained, grid models [14, 39]. The latter resemble our system in that they are massively parallel and local; however, the coevolutionary aspect is missing. As we wish to attain a system displaying global computation, the individual cells do not evolve independently as with genetic algorithms (be they parallel or serial), i.e., in a "loosely coupled" manner, but rather coevolve, thereby comprising a "tightly coupled" system.

1.3 Applications of cellular programming

In this section we study six computational tasks: density, synchronization, ordering, rectangle-boundary, thinning, and random number generation; these are summarized in Table 1.1. Minimal cellular spaces are used: 2-state, $r = 1$ for the one-dimensional case and 2-state, 5-neighbor for the two-dimensional one. Spatially periodic boundary conditions are applied, resulting in a circular grid for the one-dimensional case, and a toroidal one for the two-dimensional case. The total number of initial configurations per evolutionary run was in the range $[10^5, 10^6]$. Performance values reported hereafter represent the average fitness of all grid cells after C configurations, normalized to the range $[0, 1]$; these are obtained during execution of the cellular programming algorithm.

1.3.1 The density task

The one-dimensional density task is to decide whether or not the initial configuration contains more than 50% 1s, relaxing to a fixed-point pattern of all 1s if the initial density of 1s exceeds 0.5, and all 0s otherwise. As noted by [16], the density task comprises a non-trivial computation for a small-radius CA ($r \ll N$, where N is the grid size). Density is a global property of a configuration whereas a small-radius CA relies solely on local interactions. Since the 1s can be distributed throughout the grid, propagation of information must occur over large distances (i.e., $O(N)$). The minimum amount of memory required for the task is $O(\log N)$ using a serial-scan algorithm, thus the computation involved corresponds to recognition of a non-regular language. Note that the density task cannot be perfectly solved by a uniform, two-state CA, as proven by [13]. (This result applies to the above statement of the problem, where the CA's final pattern (i.e., output) is specified as a

Task	Description	Grid
Density	Decide whether the initial configuration contains a majority of 0s or of 1s	1D, r=1 2D, 5-neighbor
Synchronization	Given any initial configuration, relax to an oscillation between all 0s and all 1s	1D, r=1 2D, 5-neighbor
Ordering	Order initial configuration so that 0s are placed on the left and 1s are placed on the right	1D, r=1
Rectangle-Boundary	Find the boundaries of a randomly placed, random-sized non-filled rectangle	2D, 5-neighbor
Thinning	Find thin representations of rectangular patterns	2D, 5-neighbor
Random Number Generation	Generate "good" sequences of pseudo-random numbers	1D, r=1

Table 1.1: List of computational tasks for which cellular machines were evolved via cellular programming.

fixed-point configuration. Interestingly, it has recently been proven that by changing the output specification, namely, the final pattern toward which the system should converge, a two-state, $r = 1$ uniform CA exists that can perfectly solve the density problem [3].)

We studied this task in [23, 25, 29, 30] using non-uniform, one-dimensional, minimal radius $r = 1$ CAs of size $N = 149$. The search space involved is extremely large; since each cell contains one of 2^8 possible rules this space is of size $(2^8)^{149} = 2^{1192}$. In contrast, the size of *uniform*, $r = 1$ CA rule space is small, consisting of only $2^8 = 256$ rules. This enabled us to test each and every one of these rules on the density task, a feat not possible for larger values of r. One of our major results is that evolved non-uniform, $r = 1$ CAs outperform any possible uniform, $r = 1$ CA [23] (for details on the performance comparison see [23, 25]).

For the cellular programming algorithm we used randomly generated initial configurations, uniformly distributed over densities in the range $[0, 1]$, with the CA being run for $M = 150$ time steps (thus, computation time is linear with grid size). We found that non-uniform CAs had coevolved that exhibit performance values as high as 0.93 (in comparison, the maximal performance of *uniform* $r = 1$ CAs is 0.83 [23, 25]). Furthermore, these consist of a grid in which one rule dominates, a situation referred to as

quasi-uniformity [21, 25]. Basically, in a *quasi*-uniform CA the number of distinct rules is extremely small with respect to rule-space size; furthermore, the rules are distributed such that a subset of dominant rules occupies most of the grid.

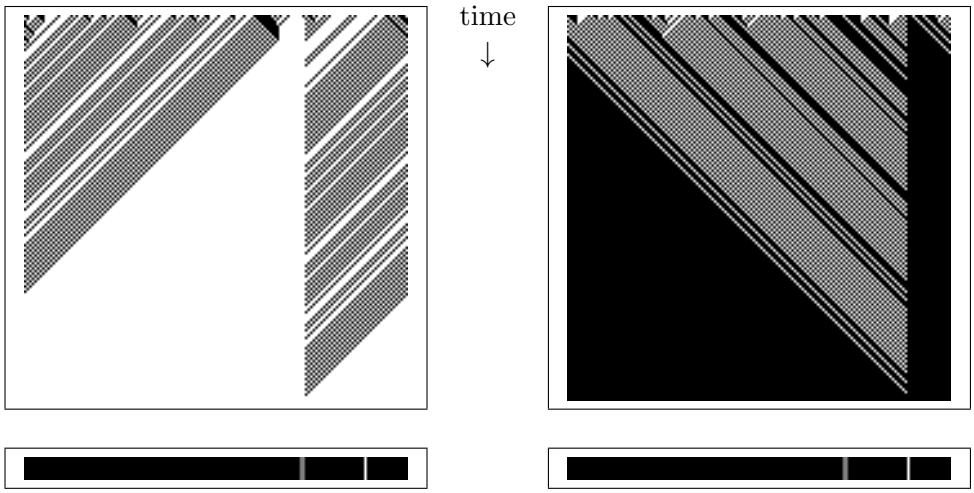


Figure 1.3: One-dimensional density task: Operation of a coevolved, non-uniform, $r = 1$ CA. Grid size is $N = 149$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). Initial configurations were generated at random. Top figures depict space-time diagrams, bottom figures depict rule maps. (a) Initial density of 1s is 0.40. (b) Initial density of 1s is 0.60. The CA relaxes in both cases to a fixed pattern of all 0s or all 1s, correctly classifying the initial configuration.

Figure 1.3 demonstrates the operation of one such coevolved CA along with a rules map, depicting the distribution of rules by assigning a unique gray level to each distinct rule. In this example the grid consists of 146 cells containing rule 226, 2 cells containing rule 224, and 1 cell containing rule 234.[2] The non-dominant rules act as "buffers," preventing information

[2]Rule numbers are given in accordance with Wolfram's convention [40, 42], representing the decimal equivalent of the binary number encoding the rule table. For example, the

from flowing too freely, and making local corrections to passing signals. A detailed investigation of the application of cellular programming to the one-dimensional density task can be found in [23, 25, 30].

The density task can be extended in a straightforward manner to two-dimensional grids, an investigation of which we have carried out, attaining notably higher performance than the one-dimensional case, with values of 0.99; furthermore, computation time, i.e., the number of time steps taken by the CA until convergence to the correct final pattern, is shorter (we shall elaborate upon these improved results in Section 1.4). Figure 1.4 demonstrates the operation of one such coevolved CA. Qualitatively, we observe the CA's "strategy" of successively classifying local densities, with the locality range increasing over time; "competing" regions of density 0 and density 1 are manifest, ultimately relaxing to the correct fixed point.

1.3.2 The synchronization task

The one-dimensional synchronization task was introduced by [5] and studied by us in [7, 24, 25, 26, 27] using non-uniform CAs. In this task the CA, given any initial configuration, must reach a final configuration, within M time steps, that oscillates between all 0s and all 1s on successive time steps. As with the density task, synchronization also comprises a non-trivial computation for a small-radius CA.

We studied non-uniform, one-dimensional, minimal radius $r = 1$ CAs of size $N = 149$. As for the density task, all possible *uniform*, $r = 1$ CAs were first tested on this task. For the cellular programming algorithm we used randomly generated initial configurations, uniformly distributed over densities in the range $[0, 1]$, with the CA being run for $M = 150$ time steps. We found that quasi-uniform CAs had coevolved that exhibit near-perfect performance, which surpasses any possible uniform, $r = 1$ CA. Figure 1.5 depicts the operation of two CAs: a high-performance uniform CA and a coevolved, non-uniform CA. We have also experimented with two-dimensional grids obtaining highly successful results as with the one-dimensional case.

1.3.3 The ordering task

In this task, the one-dimensional CA, given any initial configuration, must reach a final configuration in which all 0s are placed on the left side of the grid and all 1s on the right side (thus the final density equals the initial one,

rule depicted in Figure 1.1 is rule 232.

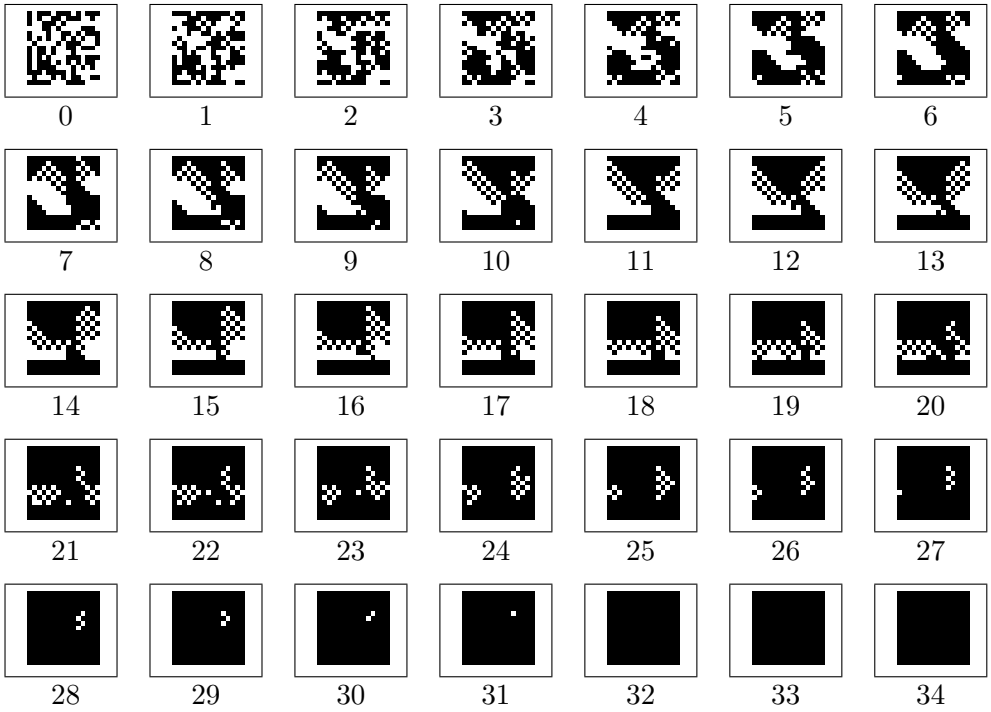
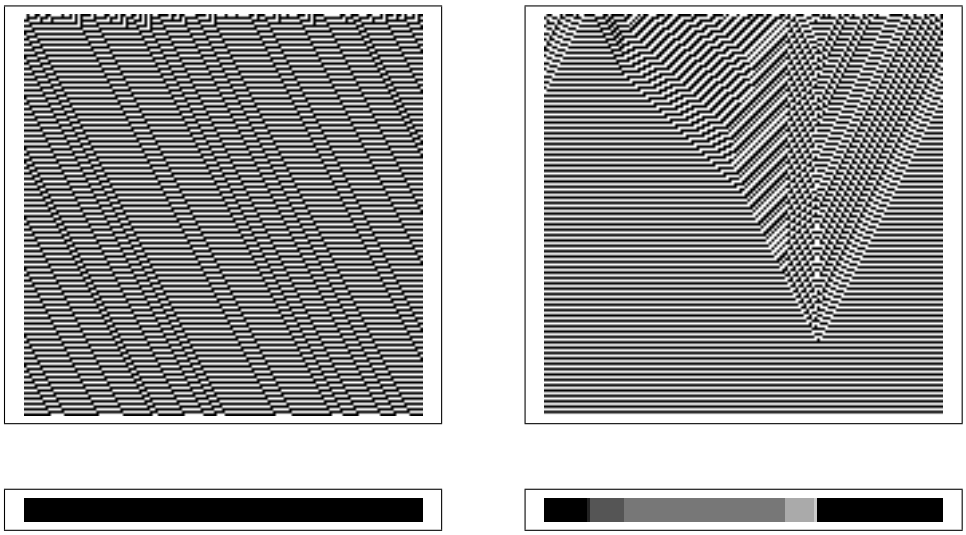


Figure 1.4: Two-dimensional density task: Operation of a coevolved, non-uniform, 2-state, 5-neighbor CA. Grid size is $N = 225$ (15×15). Initial density of 1s is 0.51, final density is 1. Numbers at bottom of images denote time steps.

however the configuration consists of a block of 0s on the left followed by a block of 1s on the right). It is interesting in that the output is not a uniform configuration of all 0s or all 1s as with the density and synchronization tasks. Cellular programming yielded quasi-uniform CAs with fitness values as high as 0.93, one of which is depicted in Figure 1.6. As with the previous tasks we were able to ascertain that this performance level is better than any possible uniform, $r = 1$ CA.



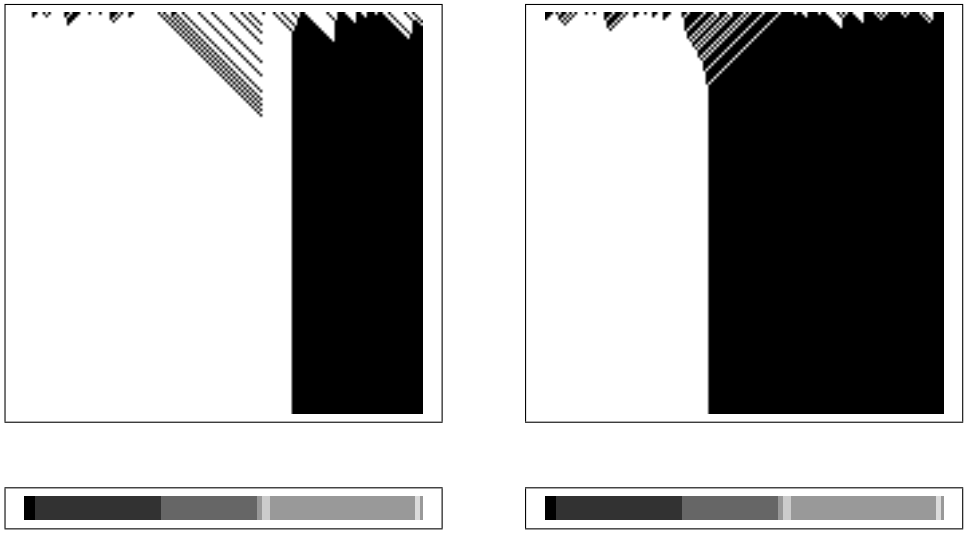
(a) (b)

Figure 1.5: One-dimensional synchronization task: Operation of two $r = 1$ CAs. Grid size is $N = 149$. Initial configurations were generated at random. Top figures depict space-time diagrams, bottom figures depict rule maps. (a) Uniform rule 31 (one of the best-performance uniform CAs for this task). (b) A coevolved, non-uniform, $r = 1$ CA.

1.3.4 The rectangle-boundary task

The possibility of applying CAs to perform image processing tasks arises as a natural consequence of their architecture. In a two-dimensional CA, a cell (or a group of cells) can correspond to an image pixel, with the CA's dynamics designed so as to perform a desired image processing task. Earlier work in this area, carried out mostly in the 1960s and the 1970s, was treated in [18], with more recent applications presented in [1, 9].

The next two tasks involve image processing operations. In this section we discuss a two-dimensional boundary computation: given an initial configuration consisting of a non-filled rectangle, the CA must reach a final configuration in which the rectangular region is filled, i.e., all cells within the confines of the rectangle are in state 1, and all other cells are in state 0. Initial configurations consist of random-sized rectangles placed randomly on the grid (in our simulations, cells within the rectangle in the initial config-



(a) (b)

Figure 1.6: One-dimensional ordering task: Operation of a coevolved, non-uniform, $r = 1$ CA. Top figures depict space-time diagrams, bottom figures depict rule maps. (a) Initial density of 1s is 0.315, final density is 0.328. (b) Initial density of 1s is 0.60, final density is 0.59.

uration were set to state 1 with probability 0.3; cells outside the rectangle were set to 0). Note that boundary cells can also be absent in the initial configuration. This operation can be considered a form of image enhancement, used, e.g., for treating corrupted images. Using cellular programming, non-uniform CAs were evolved with performance values of 0.99, one of which is depicted in Figure 1.7.

Upon studying the (two-dimensional) rules map of the coevolved, non-uniform CA, we found that the grid is quasi-uniform, with one dominant rule present in most cells. This rule maps the cell's state to zero if the number of neighboring cells in state 1 (including the cell itself) is less than two, otherwise mapping the cell's state to one;[3] thus, growing regions of 1s

[3]This is referred to as a totalistic rule, in which the state of a cell depends only on the sum of the states of its neighbors at the previous time step, and not on their individual

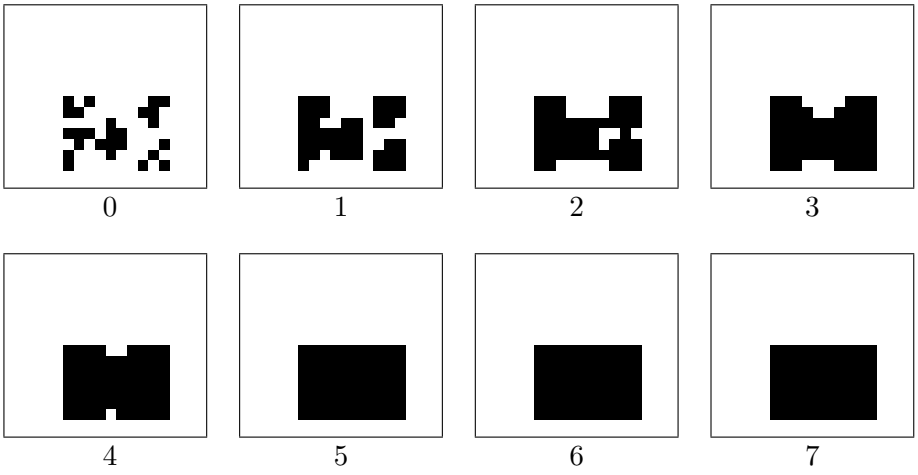


Figure 1.7: Two-dimensional rectangle-boundary task: Operation of a coevolved, non-uniform, 2-state, 5-neighbor CA. Grid size is $N = 225$ (15×15). Numbers at bottom of images denote time steps.

are more likely to occur within the rectangle confines than without.

1.3.5 The thinning task

Thinning (also known as skeletonization) is a fundamental preprocessing step in many image processing and pattern recognition algorithms. When the image consists of strokes or curves of varying thickness it is usually desirable to reduce them to thin representations located along the approximate middle of the original stroke or curve. Such "thinned" representations are typically easier to process in later stages, entailing savings in both time and storage space [8].

While the first thinning algorithms were designed for serial implementation, current interest lies in parallel systems, early examples of which were presented in [18]. The difficulty of designing a good thinning algorithm using a small, local cellular neighborhood, coupled with the task's importance had motivated us to explore the possibility of applying the cellular programming algorithm.

states [40, 42].

Guo and Hall [8] considered four sets of binary images, two of which consist of rectangular patterns oriented at different angles. The algorithms presented therein employ a two-dimensional grid with a 9-cell neighborhood, each parallel step consisting of two sub-iterations in which distinct operations take place. The set of images considered by us consists of rectangular patterns oriented either horizontally or vertically. While more restrictive than that of [8], it is noted that we employ a smaller neighborhood (5-cell) and do not apply any sub-iterations.

Figure 1.8 demonstrates the operation of a coevolved CA performing the thinning task. Although the evolved grid does not compute perfect solutions, we observe, nonetheless, good thinning "behavior" upon presentation of rectangular patterns as defined above (Figure 1.8a). Furthermore, partial success is demonstrated when presented with more difficult images involving intersecting lines (Figure 1.8b).

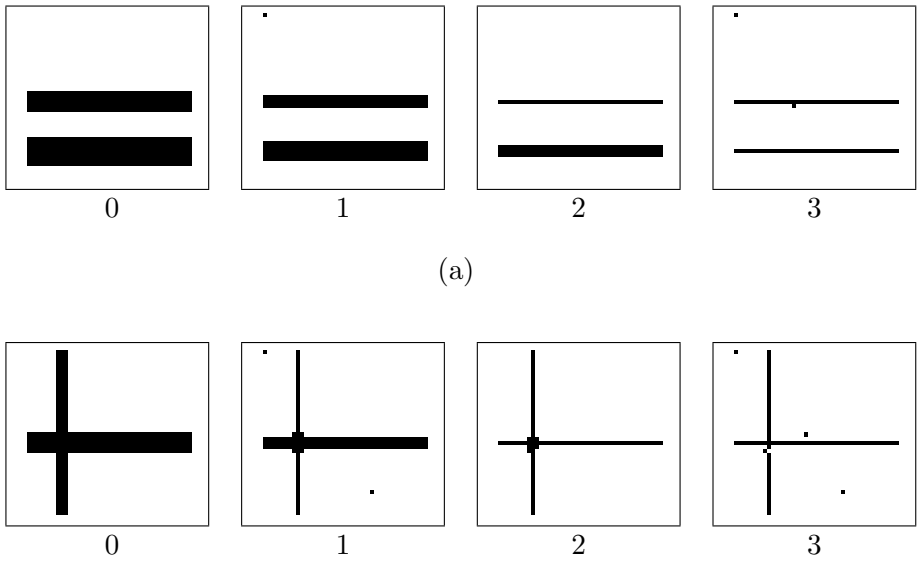


Figure 1.8: Two-dimensional thinning task: Operation of a coevolved, non-uniform, 2-state, 5-neighbor CA. Grid size is $N = 1600$ (40×40). Numbers at bottom of images denote time steps. (a) Two separate lines. (b) Two intersecting lines.

1.3.6 Random number generation

Random numbers are needed in a variety of applications, yet finding good random number generators is a difficult task [17]. To generate a random sequence on a digital computer, one starts with a fixed length seed, then iteratively applies some transformation to it, progressively extracting as long as possible a random sequence. Such numbers are usually referred to as *pseudo*-random, as distinguished from true random numbers resulting from some natural physical process. In the last decade cellular automata have been used to generate random numbers [10, 12, 41].

In [25, 32, 33] we applied the cellular programming algorithm to evolve random number generators. Essentially, the cell's fitness score for a single configuration (refer to Figure 1.2) is the entropy of the temporal bit sequence of that cell, with higher entropy implying better fitness. This fitness measure was used to drive the evolutionary process, after which standard tests were applied to evaluate the quality of the evolved CAs. The results obtained suggest that good generators can indeed be evolved; these exhibit behavior at least as good as that of previously described CAs, with notable advantages arising from the existence of a "tunable" algorithm for obtaining random number generators (Figure 1.9).

1.4 Coevolving cellular architectures

In the previous section we presented the cellular programming approach, by which non-uniform CAs can be coevolved to perform computational tasks. Such CAs comprise a generalization of the original CA model, by removing the uniformity-of-rules constraint. In this section we generalize on a second aspect of CAs, namely, their standard, homogeneous connectivity.

In Section 1.3.1 we noted that the density task can be extended in a straightforward manner to two-dimensional grids, resulting in markedly higher evolved performance coupled with shorter computation times, in comparison to the one-dimensional case. This finding is intuitively understood by observing that a two-dimensional, locally connected grid can be embedded in a one-dimensional grid with local and distant connections. This can be achieved, for example, by aligning the rows of the two-dimensional grid so as to form a one-dimensional array; the resulting embedded one-dimensional grid has distant connections of order $\sqrt{N}$, where N is the grid size. Since the density task is global it is likely that the observed superior performance of two-dimensional grids arises from the existence of distant connections that

enhance information propagation across the grid.

Motivated by this observation concerning the effect of connection lengths on performance, we set out in [29, 30] to quantitatively study the relationship between performance and connectivity on the global density task, in one-dimensional CAs; the results are summarized below (for a detailed account the reader is referred to the aforementioned papers).

We use the term *architecture* to denote the connectivity pattern of CA cells. In the standard one-dimensional model a cell is connected to r local neighbors on either side (in addition to a self-connection), where r is the radius (Chapter **??**). The model we consider is that of non-uniform CAs with non-standard architectures, in which cells need not necessarily contain the same rule nor be locally connected; however, as with the standard CA model, each cell has a small, identical number of impinging connections. In what follows the term *neighbor* refers to a directly connected cell. We employed the cellular programming algorithm to evolve cellular rules for non-uniform CAs whose architectures are fixed (yet non-standard) during the evolutionary run, or evolve concomitantly with the rules; these are referred to as fixed or evolving architectures, respectively. Note that this bears some resemblance to Kauffman's NK model [11] in that connections as well as rules are heterogeneous; however, in our case, while these are *initially* assigned at random, they then *evolve* to perform a veritable *computation*, whereas Kauffman used *random* boolean networks to study issues related to fitness landscapes engendered by arbitrarily complex epistatic couplings. Furthermore, the K parameter, denoting the number of connections per cell, may vary from $K = 1$ to $K = N$, the latter representing a fully connected grid; in our case, the number of impinging connections per cell is kept small (i.e., we concentrate on very small K values).

We considered one-dimensional, symmetrical architectures where each cell has four neighbors, with connection lengths of a and b, as well as a self-connection. Spatially periodic boundary conditions are used, resulting in a circular grid. This type of architecture belongs to the general class of circulant graphs [2], and is denoted by $C_N(a, b)$, where N is the grid size, a, b the *connection lengths* (Figure 1.10). The *distance* between two cells on the circulant is the number of connections one must traverse on the shortest path connecting them.

We surmised that attaining high performance on global tasks requires rapid information propagation throughout the CA, and that the rate of information propagation across the grid inversely depends on the average cellular distance (*acd*). It is straightforward to show that every $C_N(a, b)$

architecture is isomorphic to a $C_N(1, d')$ architecture, for some d', referred to as the *equivalent* d' [25, 29, 30]. We may therefore study the performance of $C_N(1, d)$ architectures, our conclusions being applicable to the general $C_N(a, b)$ case.

To study the effects of different architectures on performance, the cellular programming algorithm was applied to the evolution of cellular rules using fixed, non-standard architectures. We performed numerous evolutionary runs using $C_N(1, d)$ architectures with different values of d, recording the maximal performance attained during the run. Our results showed that markedly higher performance is attained for values of d corresponding to low acd values and vice versa. While performance behaves in a rugged, non-monotonic manner as a function of d, we have found that it is *linearly* correlated with acd (with a correlation coefficient of 0.99, and a negligible p value).

These results demonstrate that performance is strongly dependent upon the architecture, with higher performance attainable by using different architectures than that of the standard CA model. As each $C_N(a, b)$ architecture is isomorphic to a $C_N(1, d)$ one, and as we have found that performance is correlated with acd in the $C_N(1, d)$ case, it follows that the performance of general $C_N(a, b)$ architectures is also correlated with acd. As an example of such an architecture, the operation of a coevolved, $C_{149}(3, 5)$ CA on the density task is demonstrated in Figure 1.11.

Having shown that cellular programming can be applied to the evolution of non-uniform CAs with fixed, non-standard architectures, we then asked whether a-priori specification of the connectivity parameters (a, b or d) is indeed necessary, or can an efficient architecture coevolve along with the cellular rules. Moreover, can heterogeneous architectures, where each cell may have different d_i or (a_i,b_i) connection lengths, achieve high performance? Below we denote by $C_N(1, d_i)$ and $C_N(a_i, b_i)$ heterogeneous architectures with one or two evolving connection lengths per cell, respectively. Note that these are the cell's input connections, on which information is received; as connectivity is heterogeneous, input and output connections may be different, the latter specified implicitly by the input connections of the neighboring cells.

In order to evolve the architecture as well as the rules, the cellular programming algorithm of Section 1.2 is modified, such that each cellular "genome" consists of two "chromosomes." The first, encoding the rule table, is identical to that delineated in Section 1.2, while the second chromosome encodes the cell's connections. The two-level dynamics engendered

by the concomitant evolution of rules and connections markedly increases the size of the space searched by evolution. Our results demonstrated that high performance can be attained, nonetheless, surpassing, in fact, that of the fixed-architecture CAs. Figure 1.12 demonstrates the operation of a coevolved, $C_{129}(1, d_i)$ CA on the density task.

In summary, our main findings concerning the coevolution of cellular architectures are:

1. The performance of fixed-architecture CAs solving global tasks depends strongly and linearly on their average cellular distance. Compared with the standard $C_N(1, 2)$ architecture, considerably higher performance can be attained at very low connectivity values, by selecting a $C_N(1, d)$ or $C_N(a, b)$ architecture with a low *acd* value, such that $d, a, b \ll N$.

2. High performance architectures can be coevolved using the cellular programming algorithm, thus obviating the need to specify in advance the precise connectivity scheme. Furthermore, as was shown in [25, 30], it is possible to evolve such architectures that exhibit low *connectivity cost* per cell as well as high performance (this cost is defined as d_i for the $C_N(1, d_i)$ case and $a_i + b_i$ for $C_N(a_i, b_i)$).

1.5 Asynchronous CAs

One of the prominent features of the CA model is its synchronous mode of operation, meaning that all cells are updated simultaneously. In [25, 35, 36] we investigated the issue of evolving asynchronous CAs to perform the density and synchronization tasks. The grid is partitioned into *blocks* in which synchronous updating takes place (i.e., all cells within a block are updated simultaneously), while the blocks themselves are updated asynchronously (rather than have all blocks updated at once); thus, intra-block updating is synchronous while inter-block updating is asynchronous (a preliminary investigation of a CA-derived model based on the "blocks" idea was carried out in [22]). The number of blocks per grid, $\#_b$, is a tunable parameter, entailing a *scale* of asynchrony, ranging from complete synchrony ($\#_b = 1$) to complete asynchrony ($\#_b = N$). There are two main differences between our investigation and previous ones: (1) rather than consider only complete asynchrony ($\#_b = N$), we introduced the above scale; (2) asynchronous CAs

were previously studied from a more abstract point of view, whereas we were interested in *evolving* them to perform a veritable *computation*.

We introduced three models of asynchrony, previously unstudied in this context, finding that asynchronous CAs can be evolved to perform the computational tasks in question. We concluded that asynchrony presents a more difficult case for evolution, though it is premature to draw any definitive conclusions at this point, since we have only considered two problems, using relatively small-size grids. We feel that successful asynchronous CAs can be evolved, though this will probably entail larger grids (coupled with larger blocks).

1.6 Evolware

Though the results described above were obtained through software simulation, one of the goals is to attain truly evolving ware, *evolware*, with current implementations centering on hardware, while raising the possibility of using other forms in the future, such as *bioware* [24, 25, 26]. In [7, 25, 28] we described a hardware implementation of the cellular programming algorithm, dubbed firefly, thus demonstrating that evolware can indeed be attained (Figure 1.13).

The implementation is based on so-called field-programmable gate array (FPGA) circuits. An FPGA is an array of logic cells, laid out as an interconnected grid, with each cell capable of realizing a logic function (Chapter **??**). The cells, as well as the interconnections, are programmable "on the fly," thus offering an attractive technological platform for realizing, among others, evolware. The features distinguishing this implementation from others in the field of evolvable hardware [19, 25, 31] (see Chapter **??**) are: (1) an ensemble of individuals (cells) is at work rather than a single one; (2) genetic operators are all carried out on-board, rather than on a remote, offline computer; (3) the evolutionary phase does not necessitate halting the machine's operation, but is rather intertwined with normal execution mode. These features entail an *online autonomous* evolutionary process.

1.7 Fault tolerance

Most classical software and hardware systems, especially parallel ones, exhibit a very low level of fault tolerance, i.e., they are not resilient in the face of errors. Indeed, where software is concerned, even a single error can often

bring an entire program to a grinding halt. Future computing systems may contain thousands or even millions of computing elements (e.g., [6]). For such large numbers of components, the issue of resilience can no longer be ignored, since faults will be likely to occur with high probability.

Networks of automata exhibit a certain degree of fault tolerance. As an example, one can cite artificial neural networks, many of which show graceful degradation in performance when presented with noisy input (Chapter **??**). Moreover, the malfunction of a neuron or damage to a synaptic weight causes but a small change in the system's overall behavior, rather than bringing it to a complete standstill. Cellular computing systems, such as CAs, may be regarded as a simple and convenient framework within which to study the effects of such errors. Another motivation for studying this issue derives directly from the work presented in the previous section concerning the firefly machine. We wish to learn how robust such a machine is when operating under faulty conditions.

In [25, 34, 35] we performed a study of fault-tolerance in our evolved CAs, asking how they perform in the face of errors. The CAs in question were those that had evolved to solve either the density or synchronization tasks, with our fault-tolerance investigation picking up upon termination of the evolutionary process. We focused on one type of error where a cell updates its state in a non-deterministic manner: at each time step, the cell's next state is that specified in the rule table, with probability $1 - p_f$, or the complementary one with probability p_f; p_f is denoted the *fault probability*, representing the probability that a cell will incorrectly update its state. Figure 1.14 depicts the operation of two faulty CAs.

Our results showed that the evolved systems exhibit graceful degradation in performance, able to tolerate a certain level of faults. Furthermore, we identified a fault-tolerant range of p_f values, where "good" computational behavior is exhibited, and introduced a number of measures to fine-tune our understanding of the faulty CAs' operation. We studied the error level as a function of time and space, as well as the recuperation time needed to recover from faults.

1.8 Concluding remarks

In this chapter we described the cellular programming approach used to evolve parallel cellular machines, and demonstrated its viability by applying it to the solution of six computational problems. We then studied a num-

ber of related issues, including the evolution of connectivity architectures, asynchronous CAs, evolving ware (evolware), and faulty CAs.

Evolving cellular machines hold potential both scientifically, as vehicles for studying phenomena of interest in areas such as complex systems and artificial life, as well as practically, showing a range of potential future applications ensuing the construction of adaptive systems. This chapter has shed light on the possibility of computing with such machines, and demonstrated the feasibility of their programming by means of coevolution.

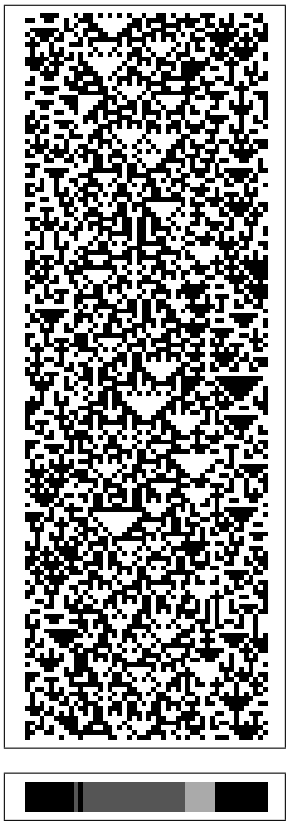


Figure 1.9: One-dimensional random number generator: Operation of a coevolved, non-uniform, $r = 1$ CA. Grid size is $N = 50$. Top figure depicts space-time diagram, bottom figure depicts rules map. Essentially, each cell's sequence of states through time is a pseudo-random bit stream.

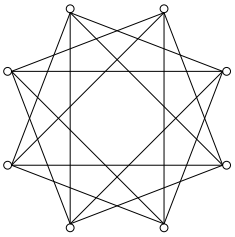
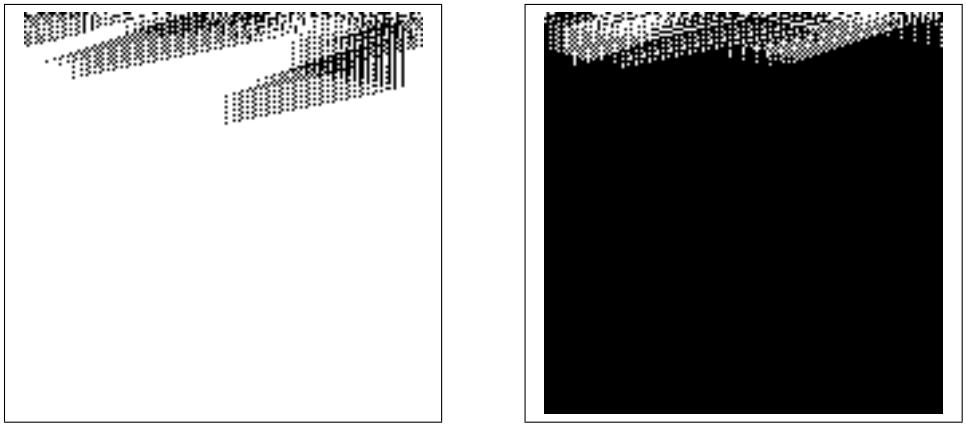
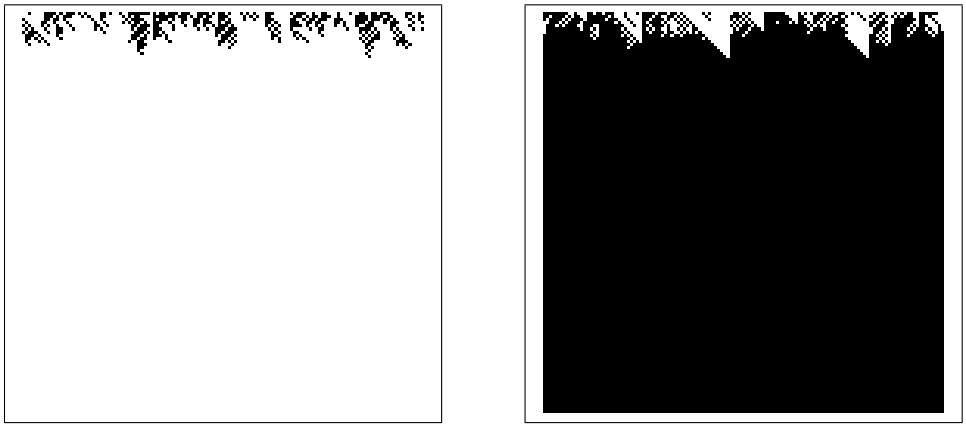


Figure 1.10: A $C_8(2,3)$ circulant graph. Each node is connected to four neighbors, with connection lengths of 2 and 3.



(a) (b)

Figure 1.11: The density task: Operation of a coevolved, non-uniform, $C_{149}(3,5)$ CA. (a) Initial density of 1s is 0.48. (b) Initial density of 1s is 0.51. Note that computation time, i.e., the number of time steps until convergence to the correct final pattern, is shorter than that of the CA of Figure 1.3. Furthermore, it can be qualitatively observed that the computational "behavior" is different, as is to be expected due to the different connectivity architecture.



(a) (b)

Figure 1.12: The density task: Operation of a coevolved, non-uniform, $C_{129}(1, d_i)$ CA. (a) Initial density of 1s is 0.496. (b) Initial density of 1s is 0.504. Note that computation time is shorter than that of the fixed-architecture CA and markedly shorter than that of the CA of Figure 1.3.

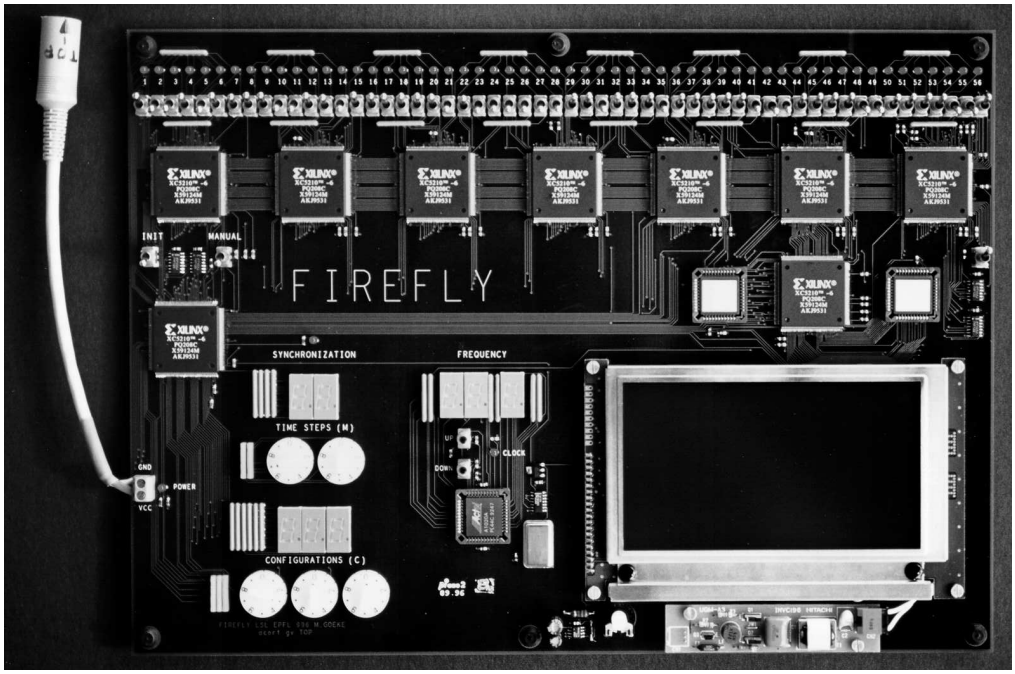
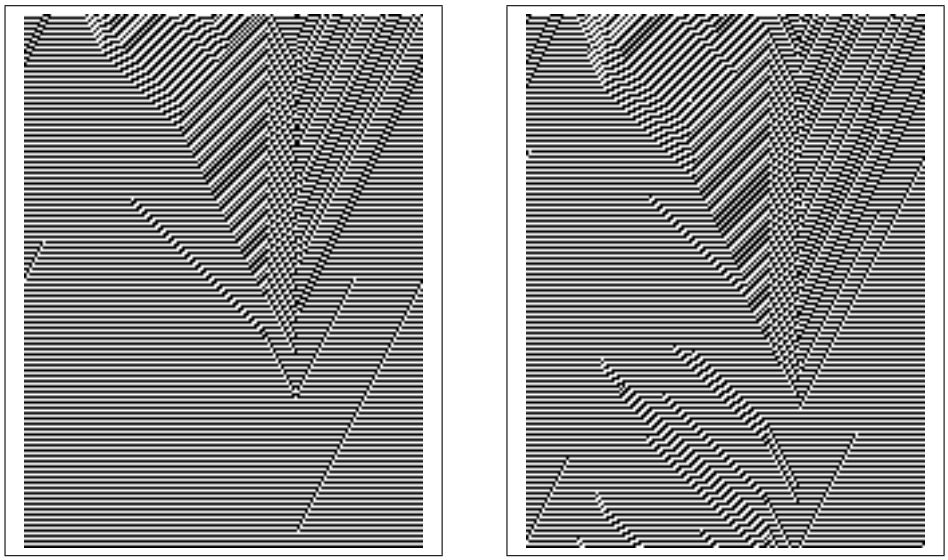


Figure 1.13: The firefly evolware board. The system is an evolving, one-dimensional, non-uniform cellular automaton. Each of the 56 cells contains a genome that represents its rule table; these genomes are randomly initialized, thereupon to be subjected to evolution. The board contains the following components: (1) LED indicators of cell states (top), (2) switches for manually setting the initial states of cells (top, below LEDs), (3) Xilinx FPGA chips (below switches), (4) display and knobs for controlling two parameters ('time steps' and 'configurations') of the cellular programming algorithm (bottom left), (5) a synchronization indicator (middle left), (6) a clock pulse generator with a manually adjustable frequency from 0.1 Hz to 1 MHz (bottom middle), (7) an LCD display of evolved rule tables and fitness values obtained during evolution (bottom right), and (8) a power-supply cable (extreme left). (Note that this is the system's sole external connection.)



(a) (b)

Figure 1.14: One-dimensional synchronization task: Operation of a coevolved, non-uniform, $r = 1$ CA, with probability of fault $p_f > 0$. Grid size is $N = 149$. Initial configurations were generated at random. (a) $p_f = 0.0001$. (b) $p_f = 0.001$.

Bibliography

[1] A. Broggi, V. D'Andrea, and G. Destri. Cellular automata as a computational model for low-level vision. *International Journal of Modern Physics C*, 4(1):5–16, 1993.

[2] F. Buckley and F. Harary. *Distance in Graphs*. Addison-Wesley, Redwood City, CA, 1990.

[3] M. S. Capcarrère, M. Sipper, and M. Tomassini. Two-state, r=1 cellular automaton that classifies density. *Physical Review Letters*, 77(24):4969–4971, December 1996.

[4] J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. Richards. Punctuated equilibria: A parallel genetic algorithm. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, page 148. Lawrence Erlbaum Associates, 1987.

[5] R. Das, J. P. Crutchfield, M. Mitchell, and J. E. Hanson. Evolving globally synchronized cellular automata. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 336–343, San Francisco, CA, 1995. Morgan Kaufmann.

[6] K. E. Drexler. *Nanosystems: Molecular Machinery, Manufacturing and Computation*. John Wiley, New York, 1992.

[7] M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. Online autonomous evolware. In T. Higuchi, M. Iwata, and W. Liu, editors, *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 96–106. Springer-Verlag, Heidelberg, 1997.

[8] Z. Guo and R. W. Hall. Parallel thinning with two-subiteration algorithms. *Communications of the ACM*, 32(3):359–373, March 1989.

[9] G. Hernandez and H. J. Herrmann. Cellular-automata for elementary image-enhancement. *CVGIP: Graphical Models and Image Processing*, 58(1):82–89, January 1996.

[10] P. D. Hortensius, R. D. McLeod, and H. C. Card. Parallel random number generation for VLSI systems using cellular automata. *IEEE Transactions on Computers*, 38(10):1466–1473, October 1989.

[11] S. A. Kauffman. *The Origins of Order*. Oxford University Press, New York, 1993.

[12] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.

[13] M. Land and R. K. Belew. No perfect two-state cellular automata for density classification exists. *Physical Review Letters*, 74(25):5148–5150, June 1995.

[14] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, page 428. Morgan Kaufmann, 1989.

[15] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[16] M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, 75:361–391, 1994.

[17] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[18] K. Preston, Jr. and M. J. B. Duff. *Modern Cellular Automata: Theory and Applications*. Plenum Press, New York, 1984.

[19] E. Sanchez and M. Tomassini, editors. *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1996.

[20] M. Sipper. Non-uniform cellular automata: Evolution in rule space and formation of complex structures. In R. A. Brooks and P. Maes, editors, *Artificial Life IV*, pages 394–399, Cambridge, Massachusetts, 1994. The MIT Press.

[21] M. Sipper. Quasi-uniform computation-universal cellular automata. In F. Morán, A. Moreno, J. J. Merelo, and P. Chacón, editors, *ECAL'95: Third European Conference on Artificial Life*, volume 929 of *Lecture Notes in Computer Science*, pages 544–554, Heidelberg, 1995. Springer-Verlag.

[22] M. Sipper. Studying artificial life using a simple, general cellular model. *Artificial Life*, 2(1):1–35, 1995. The MIT Press, Cambridge, MA.

[23] M. Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D*, 92:193–208, 1996.

[24] M. Sipper. Designing evolware by cellular programming. In T. Higuchi, M. Iwata, and W. Liu, editors, *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 81–95. Springer-Verlag, Heidelberg, 1997.

[25] M. Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag, Heidelberg, 1997.

[26] M. Sipper. The evolution of parallel cellular machines: Toward evolware. *BioSystems*, 42:29–43, 1997.

[27] M. Sipper. Evolving uniform and non-uniform cellular automata networks. In D. Stauffer, editor, *Annual Reviews of Computational Physics*, volume V, pages 243–285. World Scientific, Singapore, 1997.

[28] M. Sipper, M. Goeke, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. The firefly machine: Online evolware. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*, pages 181–186, 1997.

[29] M. Sipper and E. Ruppin. Co-evolving cellular architectures by cellular programming. In *Proceedings of IEEE Third International Conference on Evolutionary Computation (ICEC'96)*, pages 306–311, 1996.

[30] M. Sipper and E. Ruppin. Co-evolving architectures for cellular machines. *Physica D*, 99:428–441, 1997.

[31] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Transactions on Evolutionary Computation*, 1(1):83–97, April 1997.

[32] M. Sipper and M. Tomassini. Co-evolving parallel random number generators. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 950–959. Springer-Verlag, Heidelberg, 1996.

[33] M. Sipper and M. Tomassini. Generating parallel random number generators by cellular programming. *International Journal of Modern Physics C*, 7(2):181–190, 1996.

[34] M. Sipper, M. Tomassini, and O. Beuret. Studying probabilistic faults in evolved non-uniform cellular automata. *International Journal of Modern Physics C*, 7(6):923–939, 1996.

[35] M. Sipper, M. Tomassini, and M. S. Capcarrère. Designing cellular automata using a parallel evolutionary algorithm. *Nuclear Instruments & Methods in Physics Research, Section A*, 389(1-2):278–283, 1997.

[36] M. Sipper, M. Tomassini, and M. S. Capcarrère. Evolving asynchronous and scalable non-uniform cellular automata. In G. D. Smith, N. C. Steele, and R. F. Albrecht, editors, *Proceedings of International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA97)*, pages 66–70. Springer-Verlag, Vienna, 1997.

[37] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, page 176, Heidelberg, 1991. Springer-Verlag.

[38] R. Tanese. Parallel genetic algorithms for a hypercube. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, page 177. Lawrence Erlbaum Associates, 1987.

[39] M. Tomassini. The parallel genetic cellular automata: Application to global function optimization. In R. F. Albrecht, C. R. Reeves, and N. C. Steele, editors, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 385–391. Springer-Verlag, 1993.

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

[41] S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123–169, June 1986.

[42] S. Wolfram. *Cellular Automata and Complexity*. Addison-Wesley, Reading, MA, 1994.