Evolvable Cellular Machines

Moshe Sipper
Logic Systems Laboratory, Swiss Federal Institute of Technology,
1015 Lausanne, Switzerland. E-mail: Moshe.Sipper@di.epfl.ch

Marco Tomassini
Institute of Computer Science, University of Lausanne, and
Logic Systems Laboratory, Swiss Federal Institute of Technology,
1015 Lausanne, Switzerland. E-mail: Marco.Tomassini@di.epfl.ch

Abstract

A major impediment preventing ubiquitous computing with cellular automata (CA) stems from the difficulty of utilizing their complex behavior to perform useful computations. In this paper *non-uniform* CAs are studied, presenting the *cellular programming* algorithm for co-evolving such CAs to perform computations. The algorithm's efficacy is demonstrated on two non-trivial computational tasks, namely synchronization and random number generation; furthermore, we present initial results demonstrating the robustness of our evolved systems. We believe that cellular programming holds potential for attaining 'evolving ware', *evolware*, which can be implemented in software, hardware, or other possible forms, such as bioware.

1 Introduction

Cellular automata (CA) are dynamical systems in which space and time are discrete, exhibiting three notable features: massive parallelism, locality of cellular interactions, and simplicity of basic components (cells). A major impediment preventing ubiquitous computing with CAs stems from the difficulty of utilizing their complex behavior to perform useful computations. Designing CAs to have a specific behavior or perform a particular task is highly complicated, thus severely limiting their applications; automating the design (programming) process would greatly enhance the viability of CAs [Mitchell *et al.*, 1994]. A prime motivation for studying CAs stems from the observation that 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 [Gacs, 1985].

Recent studies have shown that CAs can be evolved, using genetic-algorithm based methods, to perform non-trivial computational tasks. The model investigated in this paper is an extension of the CA model, termed *non-uniform cellular automata* [Sipper, 1994, Vichniac *et al.*, 1986]. 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, employing a local,

co-evolutionary algorithm, an approach referred to as *cellular programming*. In this paper we present our approach and demonstrate its application to two non-trivial computational problems: synchronization and random number generation. We believe that cellular programming holds potential for attaining 'evolving ware', *evolware*, which can be implemented in software, hardware, or other possible forms. Of particular interest is the issue of evolving hardware, which has recently made its appearance on the artificial evolution scene [Sanchez and Tomassini, 1996].

The application of genetic algorithms to the *evolution* of *uniform* cellular automata was initially studied by [Packard, 1988] and recently undertaken by the EVCA (evolving CA) group [Mitchell *et al.*, 1994, Das *et al.*, 1995], demonstrating that CAs can be evolved to perform computational tasks. They carried out experiments involving uniform, one-dimensional CAs with $k = 2$ and $r = 3$, where k denotes the number of possible states per cell and r denotes the radius of a cell, i.e., the number of neighbors on either side (thus each cell has $2r + 1$ neighbors, including itself). Spatially periodic boundary conditions are used, resulting in a circular grid. We had first studied non-uniform CAs in [Sipper, 1994, Sipper, 1995b] and demonstrated in [Sipper, 1995a] that universal computation can be attained in such CAs. The universal systems we presented are simpler than previous ones and are *quasi*-uniform, meaning that 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. The co-evolution of non-uniform, one-dimensional CAs to perform computations was undertaken in [Sipper, 1996], where the cellular programming algorithm was presented; we showed that high performance, non-uniform CAs can be co-evolved not only with radius $r = 3$, as previously studied, but also for smaller radiuses, most notably for minimal $r = 1$. It was also found that evolved systems exhibiting high performance are quasi-uniform.

The cellular programming algorithm is delineated in the next section. In Section 3, we demonstrate its application to two computational tasks, namely synchronization and random number generation. Our conclusions are presented in Section 4.

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, known as the "genome", containing the next-state (output) bits for all possible neighborhood configurations,[1] 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. Cell rules are initialized at random, uniformly distributed among

[1] The term 'configuration' refers to an assignment of states to grid cells.

different fractions of output 1 bits. Initial configurations are then generated at random, in accordance with the task at hand. For each initial configuration 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 our algorithm is delineated in Figure 1.

```
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: Pseudo-code of the cellular programming algorithm.

The genetic operators of crossover and mutation are those used in genetic algorithms [Mitchell, 1996]. 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 our algorithm and the standard genetic algorithm: (a) A standard genetic algorithm 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, our 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; each CA is run independently, after which genetic operators are applied to produce a new population. In contrast, our CA *co-evolves* since each cell's fitness depends upon its evolving neighbors.

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 evolutionary algorithms, thereby decreasing computation time and enhancing performance [Tomassini, 1995]. A number of models have been suggested, among them coarse-grained, island models [Starkweather *et al.*, 1991, Cohoon *et al.*, 1987, Tanese, 1987], and fine-grained, grid models [Tomassini, 1993, Manderick and Spiessens, 1989]. The latter resemble our system in that they are massively parallel and local; however, the co-evolutionary 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 co-evolve, thereby comprising a "tightly-coupled" system.

3 Results

In this section we demonstrate the application of our algorithm to two non-trivial computational problems, namely synchronization and random number generation. The cellular space used is minimal, with $k = 2$ and $r = 1$. Performance values reported hereafter represent the average fitness of all grid cells after C configurations, normalized to the range $[0, 1]$.

3.1 The synchronization task

The one-dimensional synchronization task was introduced by [Das *et al.*, 1995] and studied by us in [Sipper, 1997] 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. It belongs to a class of problems studied in other domains, such as distributed computing, known as firing squad problems [Lamport and Lynch, 1990].

The task is non-trivial since synchronous oscillation is a global property of a configuration, whereas a small radius CA employs only local interactions. Thus, while local regions of synchrony can be directly attained, it is more difficult to design CAs in which spatially distant regions are in phase. Since out-of-phase regions can be distributed throughout the lattice, propagation of information must occur over large space-time distances (i.e., $O(N)$) to remove these phase defects and produce a globally synchronous configuration [Das *et al.*, 1995].

In [Sipper, 1997] we studied non-uniform, one-dimensional, minimal radius $r = 1$ CAs of size $N = 149$. The size of *uniform*, $r = 1$ CA rule space is small, consisting of only $2^8 = 256$ rules. This enabled us to check each and every one of these rules on the synchronization task, a feat not possible for larger values of r. Our results show that the maximal performance for uniform, $r = 1$ CAs is 0.84. For the cellular programming algorithm we used randomly generated initial configurations, with the CA being run for $M = 150$ time steps. We found that quasi-uniform CAs had co-evolved that exhibit near-perfect performance, thereby surpassing any possible uniform CA. Figure 2a depicts the operation of a co-evolved CA, along with a rules map, depicting the distribution of rules by assigning a unique color to each distinct rule. A detailed investigation of the one-dimensional synchronization task can be found in [Sipper, 1997].

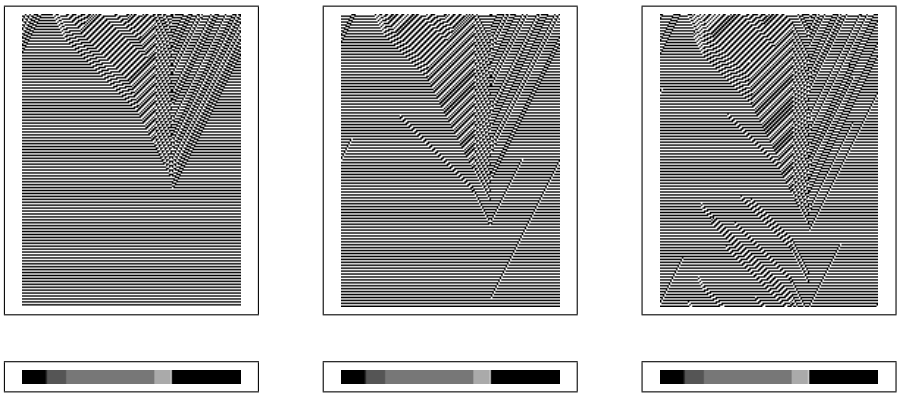
We have recently begun an investigation of the robustness of the solutions discovered by evolution. Toward this end we consider the effects of faults on the CA's behavior with the intention of studying the recovery capabilities of the system. We focus on one type of error where a cell updates its state 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. Figures 2b and 2c demonstrate the effects of different p_f values on the CA's behavior. We note that for small p_f values quick recovery is possible, while for larger values of p_f behavior becomes more erratic. We are currently conducting a quantitative study of the fault-tolerance issue.

3.2 Random number generation

Random numbers are needed in a variety of applications, yet finding good random number generators, or randomizers, is a difficult task [Park and Miller, 1988]. 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 order to demonstrate the efficiency of a proposed generator, it is usually subjected to a battery of empirical and theoretical tests, among which the most well known are those described in [Knuth, 1981].

In the last decade CAs have been used to generate random numbers. The first such work is that of [Wolfram, 1986], in which rule 30 is extensively studied for its ability to produce random, temporal bit sequences.[2] Such sequences are obtained by sampling the values that a particular cell attains as a function of time. In [Wolfram, 1986] the uniform, two-state, $r = 1$, rule 30 CA is initialized with a configuration consisting of a single cell in state 1, with all other cells

[2]Rule numbers are given in accordance with Wolfram's convention [Wolfram, 1983], representing the decimal equivalent of the binary number encoding the rule table.



(a) (b) (c)

Figure 2: The one-dimensional synchronization task: Operation of a co-evolved, 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) $p_f = 0$. (b) $p_f = 0.0001$. (c) $p_f = 0.001$.

being in state 0; the initially non-zero cell is the site at which the random temporal sequence is generated. Wolfram studied this particular rule extensively, demonstrating its suitability as a high-performance randomizer which can be efficiently implemented in parallel; indeed, this CA is one of the standard generators of the massively parallel Connection Machine CM2 [Connection, 1991]. A non-uniform CA randomizer was presented by [Hortensius *et al.*, 1989a, Hortensius *et al.*, 1989b] (based on the work of [Pries *et al.*, 1986]), consisting of two rules, 90 and 150, arranged in a specific order in the grid. The performance of this CA in terms of random number generation was found to be at least as good as that of rule 30, with the added benefit of less costly hardware implementation. It is interesting in that it combines two rules, both of which are simple linear rules that do not comprise good randomizers, to form an efficient, high-performance generator. An example application of such CA randomizers has recently been demonstrated by [Chowdhury *et al.*, 1995] who designed a low-cost, high-capacity associative memory.

An evolutionary approach for obtaining random number generators was taken by [Koza, 1992], who applied genetic programming to the evolution of a symbolic LISP expression that acts as a rule for a uniform CA (i.e., the expression is inserted into each CA cell, thereby comprising the function according to which the cell's next state is computed). He demonstrated evolved expressions that are equivalent to Wolfram's rule 30. The fitness measure used by Koza is the *entropy* E_h: let k be the number of possible values per sequence position

(in our case CA states) and h a subsequence length. E_h (measured in bits) for the set of k^h probabilities of the k^h possible subsequences of length h is given by:

$$E_h = -\sum_{j=1}^{k^h} p_{h_j} \log_2 p_{h_j}$$

where $h_1, h_2, \ldots, h_{k^h}$ are all the possible subsequences of length h (by convention, $\log_2 0 = 0$ when computing entropy). The entropy attains its maximal value when the probabilities of all k^h possible subsequences of length h are equal to $1/k^h$; in our case $k = 2$ and the maximal entropy is $E_h = h$. Koza evolved LISP expressions which act as rules for uniform, one-dimensional CAs. The CAs were run for 4096 time steps and the entropy of the resulting temporal sequence of a designated cell (usually the central one) was taken as the fitness of the particular rule (i.e., LISP expression). In his experiments Koza used a subsequence length of $h = 4$, obtaining rules with an entropy of 3.996. The best rule of each run was re-tested over 65536 time steps, some of which exhibited the maximal entropy value of 4.0.

For the cellular programming algorithm the cell's fitness score for a single configuration is defined as the entropy E_h of the temporal sequence, after the CA has been run for M time steps; f_i is then updated as follows (refer to Figure 1):

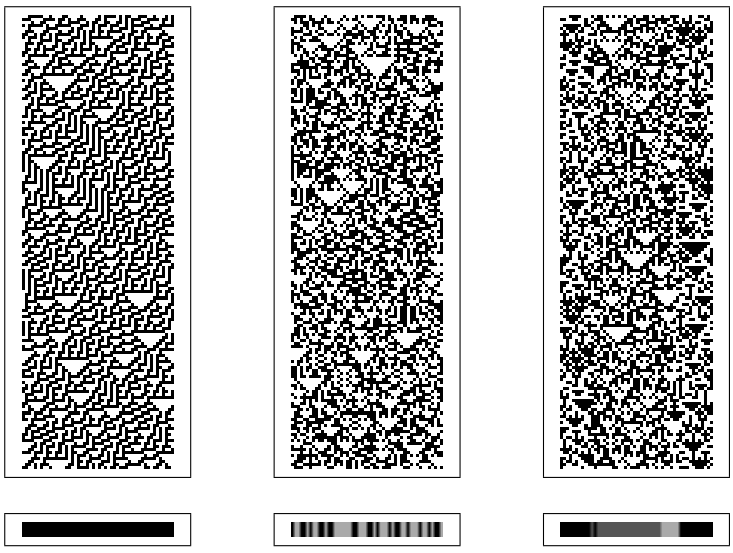
for each cell i **do in parallel**
    $f_i = f_i +$ entropy E_h of the temporal sequence of cell i
end parallel for

Rather than restrict ourselves to one designated cell, we consider all grid cells, thus obtaining N random sequences in parallel, rather than a single one. Initial configurations for our evolving, non-uniform CA are selected at random,[3] after which the CA is run for $M = 4096$ time steps. In our simulations (using grids of sizes $N = 50$ and $N = 150$), we observed that the average cellular entropy taken over all grid cells is initially low (usually in the range $[0.2, 0.5]$), ultimately evolving to a maximum of 3.997, when using a subsequence size of $h = 4$ (i.e., entropy is computed by considering the occurrence probabilities of 16 possible subsequences, using a "sliding window" of length 4).

We performed several such experiments using $h = 4$ and $h = 7$; the evolved, non-uniform CAs attained average fitness values (entropy) of 3.997 and 6.978, respectively. We then re-tested our best CAs over $M = 65536$ times steps (as in [Koza, 1992]), obtaining entropy values of 3.9998 and 6.999, respectively. Interestingly, when we performed this test with $h = 7$ for CAs which were evolved using $h = 4$, high entropy was displayed as for CAs which were originally evolved with $h = 7$. The entropy results are comparable to those of [Koza, 1992] as well as to the rule 30 CA of [Wolfram, 1986] and the non-uniform, rules $\{90, 150\}$ CA of [Hortensius *et al.*, 1989a, Hortensius *et al.*, 1989b]. Note that while our fitness measure is local, the evolved entropy results reported above

[3] A standard, 48-bit, linear congruential algorithm proved sufficient for the generation of initial configurations.

represent the average of *all* grid cells; thus, we obtain N random sequences in parallel, rather than a single one. Figure 3 demonstrates the operation of three CAs discussed above: rule 30, rules $\{90, 150\}$, and a co-evolved CA. A more detailed investigation has been carried out in [Sipper and Tomassini, 1996b, Sipper and Tomassini, 1996a], using tests described in [Knuth, 1981], suggesting that good randomizers can be evolved; these exhibit behavior at least as good as that of previously described CA generators, with notable advantages arising from the existence of a "tunable" algorithm for the generation of randomizers.



(a) (b) (c)

Figure 3: One-dimensional random number generators: Operation of three CAs. Grid size is $N = 50$, radius is $r = 1$. Top figures depict space-time diagrams, bottom figures depict rule maps. (a) Rule 30 CA. (b) Rules $\{90, 150\}$ CA. (c) A co-evolved, non-uniform CA, consisting of three rules: rule 165 (22 cells), rule 90 (22 cells), rule 150 (6 cells).

4 Conclusions

A major impediment preventing ubiquitous computing with CAs stems from the difficulty of utilizing their complex behavior to perform useful computations. We presented the cellular programming algorithm for co-evolving computation in non-uniform CAs, demonstrating that high performance systems can be evolved for non-trivial computational tasks. Several possible avenues of research suggest themselves; one of these concerns a detailed investigation

into the robustness of our systems, as described in Section 3.1. Another study which we have undertaken involves a modified model, in which the concomitant evolution of cellular rules and cellular connections takes place. We found that performance can be markedly increased for global computational tasks by such co-evolving architectures [Sipper and Ruppin, 1997, Sipper and Ruppin, 1996].

Evolving, non-uniform CAs hold potential for studying phenomena of interest in areas such as complex systems, artificial life and parallel computation. This work has shed light on the possibility of computing with such CAs, and demonstrated the feasibility of their programming by means of co-evolution. We believe that cellular programming holds potential for attaining 'evolving ware', evolware, which can be implemented in software, hardware, or other possible forms, such as bioware.

References

[Chowdhury *et al.*, 1995] D. R. Chowdhury, I. S. Gupta, and P. P. Chaudhuri. A low-cost high-capacity associative memory design using cellular automata. *IEEE Transactions on Computers*, 44(10):1260–1264, October 1995.

[Cohoon *et al.*, 1987] 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.

[Connection, 1991] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine: CM-200 Series Technical Summary*, June 1991.

[Das *et al.*, 1995] 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.

[Gacs, 1985] P. Gacs. Nonergodic one-dimensional media and reliable computation. *Contemporary Mathematics*, 41:125, 1985.

[Hortensius *et al.*, 1989a] 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.

[Hortensius *et al.*, 1989b] P. D. Hortensius, R. D. McLeod, W. Pries, D. M. Miller, and H. C. Card. Cellular automata-based pseudorandom number generators for built-in self-test. *IEEE Transactions on Computer-Aided Design*, 8(8):842–859, August 1989.

[Knuth, 1981] D. E. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, second edition, 1981.

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

[Lamport and Lynch, 1990] L. Lamport and N. Lynch. Distributed computing: Models and methods. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1159–1199. Elsevier, Amsterdam, 1990.

[Manderick and Spiessens, 1989] 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.

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

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

[Packard, 1988] N. H. Packard. Adaptation toward the edge of chaos. In J. A. S. Kelso, A. J. Mandell, and M. F. Shlesinger, editors, *Dynamic Patterns in Complex Systems*, pages 293–301. World Scientific, Singapore, 1988.

[Park and Miller, 1988] 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.

[Pries *et al.*, 1986] W. Pries, A. Thanailakis, and H. C. Card. Group properties of cellular automata and VLSI applications. *IEEE Transactions on Computers*, C-35(12):1013–1024, December 1986.

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

[Sipper and Ruppin, 1996] 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.

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

[Sipper and Tomassini, 1996a] 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.

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

[Sipper, 1994] 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.

[Sipper, 1995a] 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.

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

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

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

[Starkweather *et al.*, 1991] 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.

[Tanese, 1987] 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.

[Tomassini, 1993] 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.

[Tomassini, 1995] M. Tomassini. A survey of genetic algorithms. In D. Stauffer, editor, *Annual Reviews of Computational Physics*, volume III, pages 87–118. World Scientific, Singapore, 1995.

[Vichniac *et al.*, 1986] G. Y. Vichniac, P. Tamayo, and H. Hartman. Annealed and quenched inhomogeneous cellular automata. *Journal of Statistical Physics*, 45:875–883, 1986.

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

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