Online Autonomous Evolware

Maxime Goeke, Moshe Sipper, Daniel Mange, Andre Stauffer, Eduardo Sanchez, and Marco Tomassini

Logic Systems Laboratory, Swiss Federal Institute of Technology, IN-Ecublens, CH-1015 Lausanne, Switzerland. E-mail: {Name.Surname}@di.epfl.ch

Abstract. We present the *cellular programming* approach, in which parallel cellular machines evolve to solve computational tasks, specifically demonstrating that high performance can be attained for the *synchronization* problem. We then described an FPGA-based implementation, demonstrating that 'evolving ware', *evolware*, can be attained; the implementation is facilitated by the cellular programming algorithm's local dynamics. The machine's only link to the outside world is an external power supply, thereby exhibiting online autonomous evolution.

1 Introduction

The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen an impressive growth in the past few years; usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming [1, 7, 8]. 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, an issue which usually remains in the background, is quite costly in many cases, since populations of solutions are involved coupled with computationally-intensive fitness evaluations. One possible solution is to parallelize the process, an idea which has been explored to some extent in recent years (see reviews by [3, 25]); 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 paper we consider the general issue of evolving machines; while this idea finds its origins in the cybernetics movement of the 1940s and the 1950s, it has recently resurged in the form of the nascent field of bio-inspired systems and evolvable hardware [14]. In what follows we present the *cellular programming* approach, in which parallel cellular machines evolve to solve computational tasks [17, 18, 19, 20, 21, 22]. We describe the algorithm and its hardware implementation, demonstrating that 'evolving ware', *evolware*, can be attained; while current evolware is hardware-based, future ware may include other forms, such as *bioware*. Our primary goal in this paper is to demonstrate that online autonomous evolware can be attained, which operates without any reference to an external device or computer; toward this end we shall concentrate on a specific, well-defined synchronization problem.

The machine model we employ is based on the cellular automata model. Cellular automata (CA) are dynamical systems in which space and time are discrete. They consist 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 current 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 [24, 26]. The cellular array (grid) is n-dimensional, where $n = 1, 2, 3$ is used in practice; in this work we shall concentrate on $n = 1$, i.e., one-dimensional grids.

CAs exhibit three notable features, namely massive parallelism, locality of cellular interactions, and simplicity of basic components (cells); thus, they present an excellent point of departure for our forays into the evolution of parallel cellular machines. The machine model we employ is an extension of the original CA model, termed *non-uniform cellular automata* [15]. 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.

The evolware implementation is based on FPGA (Field-Programmable Gate Array) technology. An FPGA circuit is an array of logic cells, laid out as an interconnected grid, with each cell capable of realizing a logic function [13]. The cells, as well as the interconnections, are programmable "on the fly", thus offering an attractive technological platform for realizing, among others, evolware.

In Section 2 we present previous work on evolving cellular machines; in particular, we present the synchronization problem, a non-trivial, global computational task. Section 3 delineates the cellular programming algorithm used to evolve non-uniform CAs; as opposed to the standard genetic algorithm, where a population of *independent* problem solutions *globally* evolves [8], our approach involves a grid of rules that *co-evolves locally*. In Section 4 we describe the FPGA-based evolware; evolution takes place within the machine itself, with no reference to or aid from any external device (e.g., a computer that carries out genetic operators) apart from a power supply, thus attaining *online autonomous evolware*. Finally, our conclusions are presented in Section 5.

2 Evolving parallel cellular machines

The application of genetic algorithms to the evolution of *uniform* cellular automata was initially studied by [12] and recently undertaken by the EVCA (evolving CA) group [4, 5, 6, 9, 10, 11]. They carried out experiments involving 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. A common method of examining the behavior of one-dimensional CAs is to display a two-dimensional space-time diagram, where the horizontal axis depicts the configuration at a certain time t and the vertical axis depicts successive time steps (e.g., Figure 1). The term 'configuration' refers to an assignment of 1 states to several cells, and 0s otherwise.

The EVCA group employed a genetic algorithm to evolve uniform CAs to perform two computational tasks, density and synchronization, the latter of which we shall consider in this paper. In the synchronization 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 noted by [5], this is perhaps the simplest, non-trivial synchronization task. 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, transfer of information must occur over large distances (i.e., $O(N)$, where N is the grid size) to remove these phase defects and produce a globally synchronous configuration. [5] reported that in 20% of the evolutionary runs the genetic algorithm discovered CAs that successfully solve the task.

It is interesting to point out that the phenomenon of synchronous oscillations occurs in nature, a striking example of which is exhibited by fireflies; thousands such creatures may flash on and off in unison, having started from totally uncoordinated flickerings [2]. Each insect has its own rhythm, which changes only through local interactions with its neighbors' lights. Another interesting case involves pendulum clocks; when several of these are placed near each other, they soon become synchronized by tiny coupling forces transmitted through the air or by vibrations in the wall to which they are attached (for a review on synchronous oscillations in nature see [23]).

The model investigated in this paper is that of non-uniform CAs, where cellular rules need not be identical for all cells. Thus, rather than seek a *single* rule that must be applied universally to all cells in the grid, we allow each cell to "choose" its own rule through evolution. As we shall see, the removal of the uniformity constraint from the original CA model lends itself to a novel algorithm which is more amenable to implementation as evolware, in comparison to standard evolutionary algorithms [17, 18, 19, 20, 21, 22].

Using the cellular programming algorithm, delineated in the next section, we have shown that non-uniform CAs can be evolved to perfectly[1] solve the synchronization task. This is achieved with minimal radius, $r = 1$ CAs (i.e., each cell is connected to its two immediate left and right neighbors), as opposed to the aforementioned uniform CAs, where $r = 3$. We have shown that the performance

[1] The term 'perfect' is used here in a stochastic sense since we cannot exhaustively test all 2^{149} possible initial configurations nor are we in possession to date of a formal proof; nonetheless, we have tested our best-performance CAs on numerous configurations, for all of which synchronization was attained.

level attained by evolved, non-uniform, $r = 1$ CAs is better than *any* possible *uniform*, $r = 1$ CA, none of which can solve the synchronization problem. The evolved systems were observed to be *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 [16, 17]. Figure 1 demonstrates the operation of two co-evolved CAs along with the corresponding rule maps; these maps depict the distribution of rules by assigning a unique color to each distinct rule.

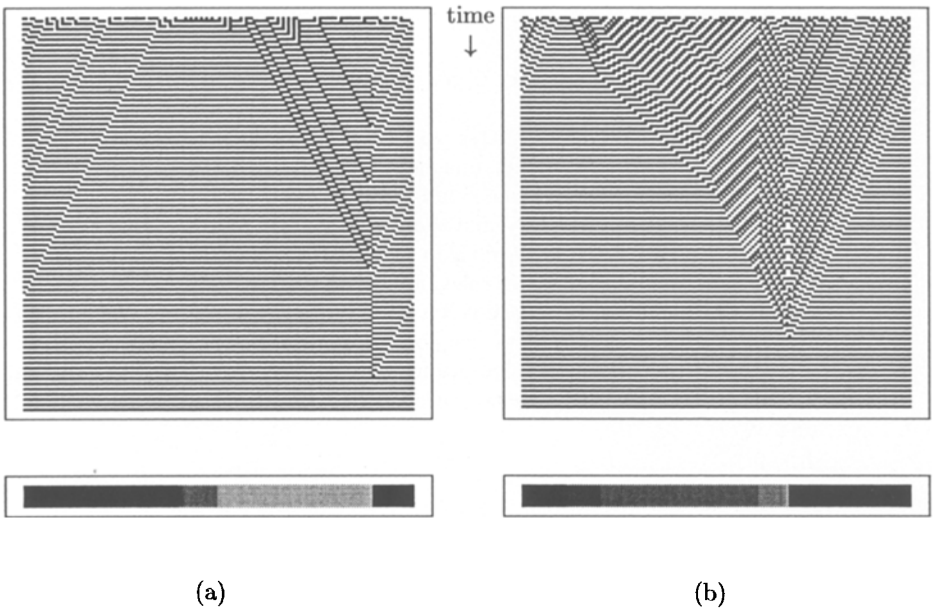

Fig. 1. The one-dimensional synchronization task: Operation of two co-evolved, non-uniform, 2-state CAs, with connectivity radius $r = 1$. Grid size is $N = 149$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). Top figures depict space-time diagrams, bottom figures depict rule maps.

3 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, listed in lexicographic order; e.g., for CAs with $r = 1$, the genome consists

```
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
```

Fig. 2. Pseudo-code of the cellular programming algorithm.

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, and for each one the CA is run for M time steps (in our simulations we used $M \approx N$ so that computation time is linear with grid size). 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 $M + 4$ iterations, and 0 otherwise. The (local) fitness score for the synchronization task is assigned to each cell by considering the last four time steps (i.e., $[M + 1..M + 4]$); if the sequence of states over these steps is precisely $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ (i.e., an alternation of 0s and 1s, starting from 0), the cell's fitness score is 1, otherwise this score is 0. 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 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.

As opposed to the standard genetic algorithm, where a population of *independent* problem solutions *globally* evolves [8], our approach involves a grid of rules that *co-evolves locally* [17]. As noted in Section 1, the CA performs computations in a completely local manner, each cell having access only to its immediate neighbors' states; in addition, the *evolutionary process* in our case is local since application of genetic operators as well as fitness assignment takes place locally. This renders our approach more amenable to implementation as evolware, in comparison to other approaches, e.g., the standard genetic algorithm.

4 Implementing evolware

The cellular programming algorithm presented in Section 3 was studied extensively through software simulation; in this section we present its online, autonomous hardware implementation, resulting in evolving ware, evolware. To facilitate implementation, the algorithm is slightly modified (with no loss in performance); the two genetic operators, one-point crossover and mutation, are replaced by a single operator, *uniform crossover*. Under this operation, a new rule, i.e., an "offspring" genome, is created from two "parent" genomes (bit strings) by choosing each offspring bit from one or the other parent, with a 50% probability for each parent [8, 25]. The changes to the algorithm are therefore as follows (refer to Figure 2):

```
else if $nf_i(c) = 1$ then replace rule $i$ with the fitter neighboring rule,
        without mutation
else if $nf_i(c) = 2$ then replace rule $i$ with the uniform crossover of the
        two fitter neighboring rules, without mutation
```

The evolutionary process ends following an arbitrary decision by an outside observer (the '**while** not done' loop of Figure 2).

The cellular programming evolware is implemented on a physical board whose only link to the "outside world" is an external power supply. The features distinguishing this implementation from previous ones [14] 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.

The active components of the evolware board comprise exclusively FPGA (Field-Programmable Gate Array) circuits, with no other commercial processor whatsoever. An LCD screen enables the display of information pertaining to the evolutionary process, including the current rule and fitness value of each cell. The parameters M (number of time steps a configuration is run) and C (number of configurations between evolutionary phases, see Section 3) are tunable through on-board knob selectors; in addition, their current values are displayed. The implemented grid size is $N = 56$ cells, each of which includes, apart from the logic component, a LED indicating its current state (on=1, off=0), and a switch by which its state can be manually set[2]. We have also implemented an on-board global synchronization detector circuit, for the sole purpose of facilitating the external observer's task; this circuit is *not* used by the CA in any of its operational phases. A schematic diagram of the board is depicted in Figure 3.

The architecture of a single cell is shown in Figure 4. The binary state is stored in a D-type flip-flop whose next state is determined either randomly, enabling the presentation of random initial configurations, or by the cell's rule table, in accordance with the current neighborhood of states. Each bit of the rule's bit string is stored in a D-type flip-flop whose inputs are channeled through a set of multiplexors according to the current operational phase of the system:

1. During the initialization phase of the evolutionary algorithm, the (eight) rule bits are loaded with random values; this is carried out once per evolutionary run.
2. During the execution phase of the CA, the rule bits remain unchanged. This phase lasts a total of $C * M$ time steps (C configurations, each one run for M time steps).
3. During the evolutionary phase, and depending on the number of fitter neighbors, $nf_i(c)$ (Section 3), the rule is either left unchanged ($nf_i(c) = 0$), replaced by the fitter left or right neighboring rule ($nf_i(c) = 1$), or replaced by the uniform crossover of the two fitter rules ($nf_i(c) = 2$).

To determine the cell's fitness score for a single initial configuration, i.e., after the CA has been run for $M + 4$ time steps (Section 3), a four-bit shift register is used (Figure 5); this register continuously stores the states of the cell over the last four time steps ($[t + 1..t + 4]$). An AND gate tests for occurrence of the "good" final sequence (i.e., $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$), producing the HIT signal, signifying whether the fitness score is 1 (HIT) or 0 (no HIT).

Each cell includes a fitness counter and two comparators for comparing the cell's fitness value with that of its two neighbors. Note that the cellular connections are entirely local, a characteristic enabled by the local operation of the cellular programming algorithm. In the interest of cost reduction, a number of resources have been implemented within a central control unit, including the

[2] This is used to test the evolved system after termination of the evolutionary process, by manually loading initial configurations.

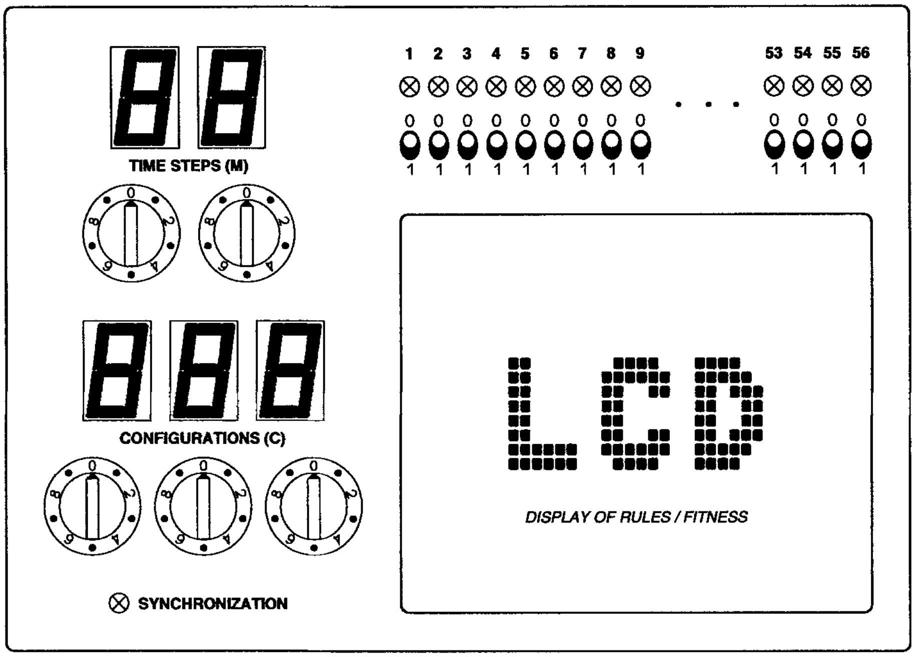

Fig. 3. Schematic diagram of evolware board: (1) LED indicators of cell states (upper right); (2) switches for manually setting the initial configuration (upper right, below LEDs); (3) display and knobs for controlling the M parameter (time steps) of the cellular programming algorithm (upper left); (4) display and knobs for controlling the C parameter (number of initial configurations between evolutionary phases) of the cellular programming algorithm (middle left); (5) synchronization indicator (lower left); (6) LCD display of evolved rules and fitness values (lower right).

random number generator and the M and C counters. Note that these are implemented *on-board* and do not comprise a breach in the machine's autonomous mode of operation.

The random number generator is implemented with a linear feedback shift register (LFSR), producing a random bit stream that cycles through $2^{32} - 1$ different values (the value 0 is excluded since it comprises an undesirable attractor). As a cell uses at most eight different random values at any given moment, it includes an 8-bit shift register through which the random bit stream propagates. The shift registers of all grid cells are concatenated to form one large stream of random bit values propagating through the entire CA. Cyclic behavior is eschewed due to the odd number of possible values produced by the random number generator ($2^{32} - 1$) and to the even number of random bits per cell.

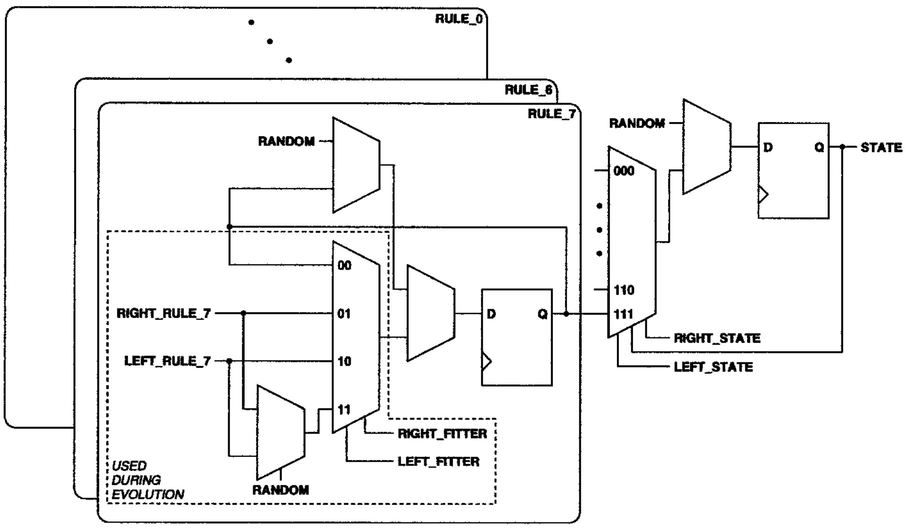

Fig. 4. Circuit design of a cell.

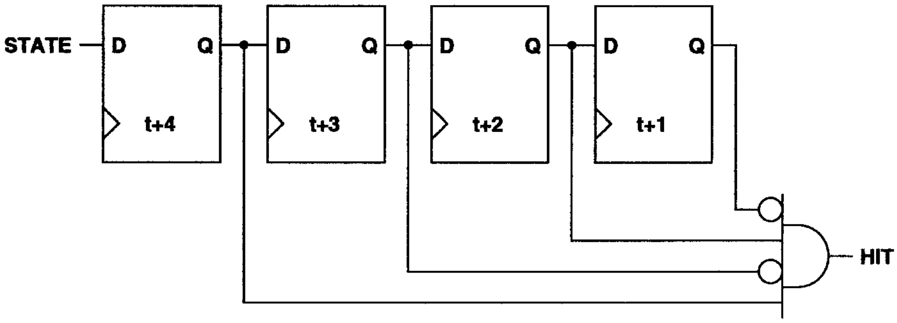

Fig. 5. Circuit used (in each cell) after execution of an initial configuration to detect whether a cell receives a fitness score of 1 (HIT) or 0 (no HIT).

5 Conclusions

In this paper we considered the general issue of evolving machines. We presented the cellular programming approach, in which parallel cellular machines evolve to solve computational tasks, specifically demonstrating that high performance can be attained for the synchronization problem. We then described an FPGA-based implementation, demonstrating that 'evolving ware', *evolware*, can be attained;

the implementation was facilitated by the cellular programming algorithm's local dynamics. The machine's only link to the outside world is an external power supply, thereby exhibiting online autonomous evolution.

Evolving, cellular machines hold potential both scientifically, as vehicles for studying phenomena of interest in areas such as complex adaptive systems and artificial life, as well as practically, showing a range of potential future applications ensuing the construction of adaptive systems. Our primary goal in this paper was to demonstrate that online autonomous evolware can be attained, toward which end we concentrated on a specific, well-defined problem. The success of our system raises the possibility of constructing more complex evolware, able to tackle real-world problems, that call for adaptive behavior.

References

1. T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, New York, 1996.
2. J. Buck. Synchronous rhythmic flashing of fireflies II. *The Quarterly Review of Biology*, 63(3):265–289, September 1988.
3. E. Cantú-Paz. A summary of research on parallel genetic algorithms. Technical Report 95007, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, July 1995.
4. J. P. Crutchfield and M. Mitchell. The evolution of emergent computation. *Proceedings of the National Academy of Sciences USA*, 92(23):10742–10746, 1995.
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. R. Das, M. Mitchell, and J. P. Crutchfield. A genetic algorithm discovers particle-based computation in cellular automata. In Y. Davidor, H. -P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature- PPSN III*, volume 866 of *Lecture Notes in Computer Science*, pages 344–353, Berlin, 1994. Springer-Verlag.
7. Z. Michalewicz. *Genetic algorithms + data structures = evolution programs.* Springer, Berlin, third edition, 1996.
8. M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.
9. M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Dynamics, computation, and the "edge of chaos": A re-examination. In G. Cowan, D. Pines, and D. Melzner, editors, *Complexity: Metaphors, Models and Reality*, pages 491–513. Addison-Wesley, Reading, MA, 1994.
10. M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, 75:361–391, 1994.
11. M. Mitchell, P. T. Hraber, and J. P. Crutchfield. Revisiting the edge of chaos: Evolving cellular automata to perform computations. *Complex Systems*, 7:89–130, 1993.

12. 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.
13. E. Sanchez. Field programmable gate array (FPGA) circuits. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Berlin, 1996.
14. E. Sanchez and M. Tomassini, editors. *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.
15. 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.
16. 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, Berlin, 1995. Springer-Verlag.
17. M. Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D*, 92:193–208, 1996.
18. M. Sipper. Designing evolware by cellular programming. In *Proceedings of The First International Conference on Evolvable Systems: from Biology to Hardware (ICES96)*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 1996.
19. M. Sipper and E. Ruppin. Co-evolving architectures for cellular machines. *Physica D*, 1996. (to appear).
20. 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.
21. 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.
22. M. Sipper and M. Tomassini. Generating parallel random number generators by cellular programming. *International Journal of Modern Physics C*, 7(2):181–190, 1996.
23. S. H. Strogatz and I. Stewart. Coupled oscillators and biological synchronization. *Scientific American*, pages 102–109, December 1993.
24. T. Toffoli and N. Margolus. *Cellular Automata Machines*. The MIT Press, Cambridge, Massachusetts, 1987.
25. M. Tomassini. Evolutionary algorithms. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 19–47. Springer-Verlag, Berlin, 1996.
26. S. Wolfram. Universality and complexity in cellular automata. *Physica D*, 10:1–35, 1984.