Proceedings of IEEE Fourth International Conference on Evolutionary Computation (ICEC'97).

The Firefly Machine: Online Evolware

Moshe Sipper, Maxime Goeke, Daniel Mange, Andre Stauffer, Eduardo Sanchez, and Marco Tomassini[1]

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 firefly machine, an evolving hardware system, demonstrating that "evolving ware," *evolware*, can be attained. The system is based on the *cellular programming* approach, in which parallel cellular machines evolve to solve computational tasks. The firefly system operates with no reference to an external device, such as a computer that carries out genetic operators, thereby exhibiting online autonomous evolution.**

I. Introduction

The idea of evolving machines, whose origins can be traced to the cybernetics movement of the 1940s and the 1950s, has recently resurged in the form of the nascent field of bio-inspired systems and evolvable hardware [1]. Most work carried out to date under this heading involves the application of evolutionary algorithms to the synthesis of digital or analog systems. From this perspective, evolvable hardware is simply a sub-domain of artificial evolution, where the final goal is the synthesis of an electronic circuit [2, 3]. However, several researchers have set more far-reaching goals for the field as a whole.

Current and (possible) future evolving hardware systems can be classified according to two distinguishing characteristics. The first involves the distinction between *offline* genetic operations, carried out in software, and *online* ones which take place on an actual circuit. The second characteristic concerns *open-endedness*. When the fitness criterion is imposed by the user in accordance with the task to be solved (currently the rule with artificial-evolution techniques), one attains a form of *guided evolution*. This is to be contrasted with *open-ended* evolution occurring in nature, which admits no externally-imposed fitness criterion, but rather an implicit, emergent, dynamical one (that could arguably be summed up as survivability). In view of these two characteristics, one can define the following four categories of evolvable hardware [2, 3]:

- The first category can be described as *evolutionary circuit design*, where the entire evolutionary process takes place in software, with the resulting solution possibly loaded onto a real circuit. Though a potentially useful design methodology, this falls completely within the realm of traditional evolutionary techniques, as noted above. As examples one can cite the works of Refs. [4–7].
- The second category involves systems in which a real circuit is used during the evolutionary process, though most operations are still carried out offline, in software. As examples one can cite Refs. [8–11], where fitness calculation is carried out on a real circuit.
- In the third category one finds systems in which *all* operations (selection, crossover, mutation), as well as fitness evaluation, are carried out *online*, in hardware. The major aspect missing concerns the fact that evolution is not open ended, i.e., there is a predefined goal and no dynamic environment to speak of. An example is the firefly machine described in this paper [12, 13].
- The last category involves a *population* of hardware entities evolving in an *open-ended* environment.

It has been argued that only systems within the last category can be truly considered evolvable hardware, a goal which still eludes us at present [2, 3]. We point out that a more correct term would probably be *evolving* hardware. A natural application area for such systems is within the field of autonomous robots, which involves machines capable of operating in unknown environments without human intervention [14]. A related application domain is that of controllers for noisy, changing environments. Another interesting example would be what has been dubbed by Ref. [2] "Hard-Tierra." This involves the hardware implementation of the Tierra "world," which consists of an open-ended environment of evolving computer programs [15]. A small-scale experiment along this line was undertaken by Ref. [16]. The idea of Hard-Tierra is interesting since it leads us to the observation that open-endedness does not necessarily imply a real, biological environment. The firefly machine, belonging to the third category, demonstrates that complete online evolution can be attained, though not in an open-ended environment. This latter goal remains a prime target for future research.

In this paper we present the firefly machine, an online, evolving hardware system, thus demonstrating that "evolving ware," *evolware*, can be attained [12, 13, 17, 18]. Section II presents cellular automata (CA), the machine model used in our project, as well as the synchronization problem, which we set out to solve via evolution. Section III delineates the cellular programming algorithm, by which CAs are evolved to perform computational tasks. In Section IV we briefly present large-scale programmable circuits, specifically concentrating on Field-Programmable Gate Arrays (FPGA). An FPGA can be programmed "on the fly," thus offering an attractive technological platform for realizing, among others, evolware. In Section V we describe the FPGA-based firefly machine. Evolution takes

[1] M. Tomassini is also with the Computer Science Institute, University of Lausanne, Switzerland.

place on-board, with no reference to or aid from any external device (such as a computer that carries out genetic operators), thus attaining online autonomous evolware. Finally, some concluding remarks are presented in Section VI.

II. Cellular automata and the synchronization task

The machine model we employ is based on the cellular automata (CA) model. CAs 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 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 [19, 20]. The cellular array (grid) is n-dimensional, where $n = 1, 2, 3$ is used in practice. In this work we shall concentrate on one-dimensional grids, with two possible states per cell, denoted 0 and 1. In such CAs each cell is connected to r local neighbors (cells) on either side, as well as to itself, where r is a parameter referred to as the radius (thus, each cell has $2r + 1$ neighbors).

CAs exhibit three notable features, namely, massive parallelism, locality of cellular interactions, and simplicity of basic components (cells), rendering them ideal for our studies. The machine model we employ is an extension of the original CA model, termed *non-uniform cellular automata* [21]. Such automata function in the same way as uniform ones, the only difference being in the local cellular interaction rules that need not be identical for all cells. A major problem common to such local, parallel systems, is the painstaking task one is faced with in designing them to exhibit a specific behavior or solve a particular problem. This results from the local dynamics of the system, which renders the design of local interaction rules to perform global computational tasks extremely arduous. In recent years evolutionary techniques have been employed to evolve CAs.

The application of genetic algorithms to the evolution of *uniform* cellular automata was studied by Refs. [22–25]. We have applied *cellular programming*, described in the next section, to the evolution of *non-uniform* CAs [12, 13, 17, 18, 26–31]. One of the computational tasks studied is known as synchronization: given any initial configuration, the CA must reach a final configuration, within M time steps, that oscillates between all 0s and all 1s on successive time steps (Figure 1). The term *configuration* refers to an assignment of states to cells in the grid. As noted by Ref. [24], this is perhaps the simplest, non-trivial synchronization task, since oscillation is a global property of a configuration, whereas a small-radius CA employs only local interactions.

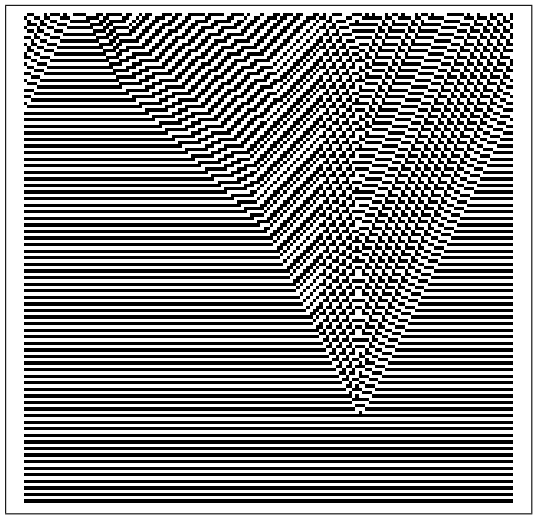

Fig. 1. The one-dimensional synchronization task: Operation of a coevolved, non-uniform, 2-state CA. The connectivity radius is $r = 1$, meaning that each cell has two neighbors, one to its immediate left and one to its immediate right. Grid size is $N = 149$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). The initial configuration was randomly generated.

III. 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 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.[2] 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 $0 \to 1 \to 0 \to 1$ (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

[2]Note that our algorithm is not necessarily restricted to a single, non-uniform CA since an ensemble of distinct grids can evolve independently in parallel.

in Figure 2.

```
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 r > 1)
          end if
          f_i = 0
      end parallel for
    end if
end while
```

Fig. 2. Cellular programming pseudo-code.

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 [32, 33], our approach involves a grid of rules that *coevolves locally* [12, 26]. 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, such as the standard genetic algorithm.

Using the cellular programming algorithm we have shown that non-uniform, $r = 1$ CAs can be evolved to successfully solve the synchronization task. Furthermore, the performance 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 [12, 18]. Figure 1 demonstrates the operation of a coevolved CA.

IV. Large-scale programmable circuits

An integrated circuit is called programmable when the user can configure its function by programming. The circuit is delivered after manufacturing in a generic state and the user can adapt it by programming a particular function. The programmed function is coded as a string of bits, representing the configuration of the circuit. In this paper we consider solely programmable *logic* circuits, where the programmable function is a logic one, ranging from simple boolean functions to complex state machines.

The first programmable circuits allowed the implementation of logic circuits that were expressed as a logic sum of products. These are the PLDs (Programmable Logic Devices), whose most popular version is the PAL (Programmable Array Logic). More recently, a novel technology has emerged, affording higher flexibility and more complex functionality: the Field-Programmable Gate Array, or FPGA [34]. An FPGA is an array of logic cells placed in an infrastructure of interconnections, which can be programmed at three distinct levels (Figure 3): (1) the function of the logic cells, (2) the interconnections between cells, and (3) the inputs and outputs. All three levels are configured via a string of bits that is loaded from an external source, either once or several times. In the latter case the FPGA is considered *reconfigurable*.

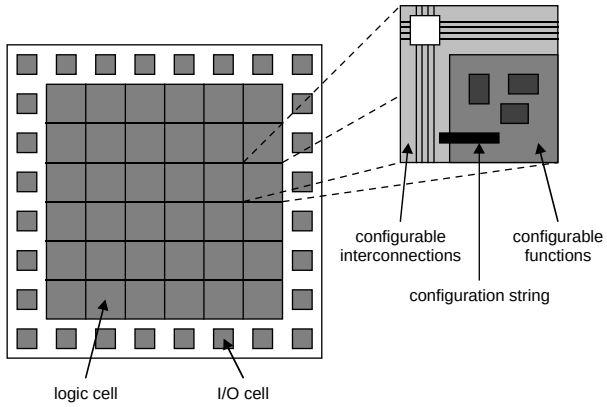

Fig. 3. A schematic diagram of a Field-Programmable Gate Array (FPGA). An FPGA is an array of logic cells placed in an infrastructure of interconnections, which can be programmed at three distinct levels: (1) the function of the logic cells, (2) the interconnections between cells, and (3) the inputs and outputs. All three levels are configured via a configuration bit string that is loaded from an external source, either once or several times.

FPGAs are highly versatile devices that offer the designer a wide range of design choices. However, this potential power necessitates a suite of tools in order to design a system. Essentially, these generate the configuration bit string upon given such inputs as a logic diagram or a high-level functional description.

V. Implementing evolware

In this section we describe the firefly evolware machine, an online implementation of the cellular programming algorithm (see Refs. [12, 13] for the relationship between the synchronization problem and fireflies in nature). 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

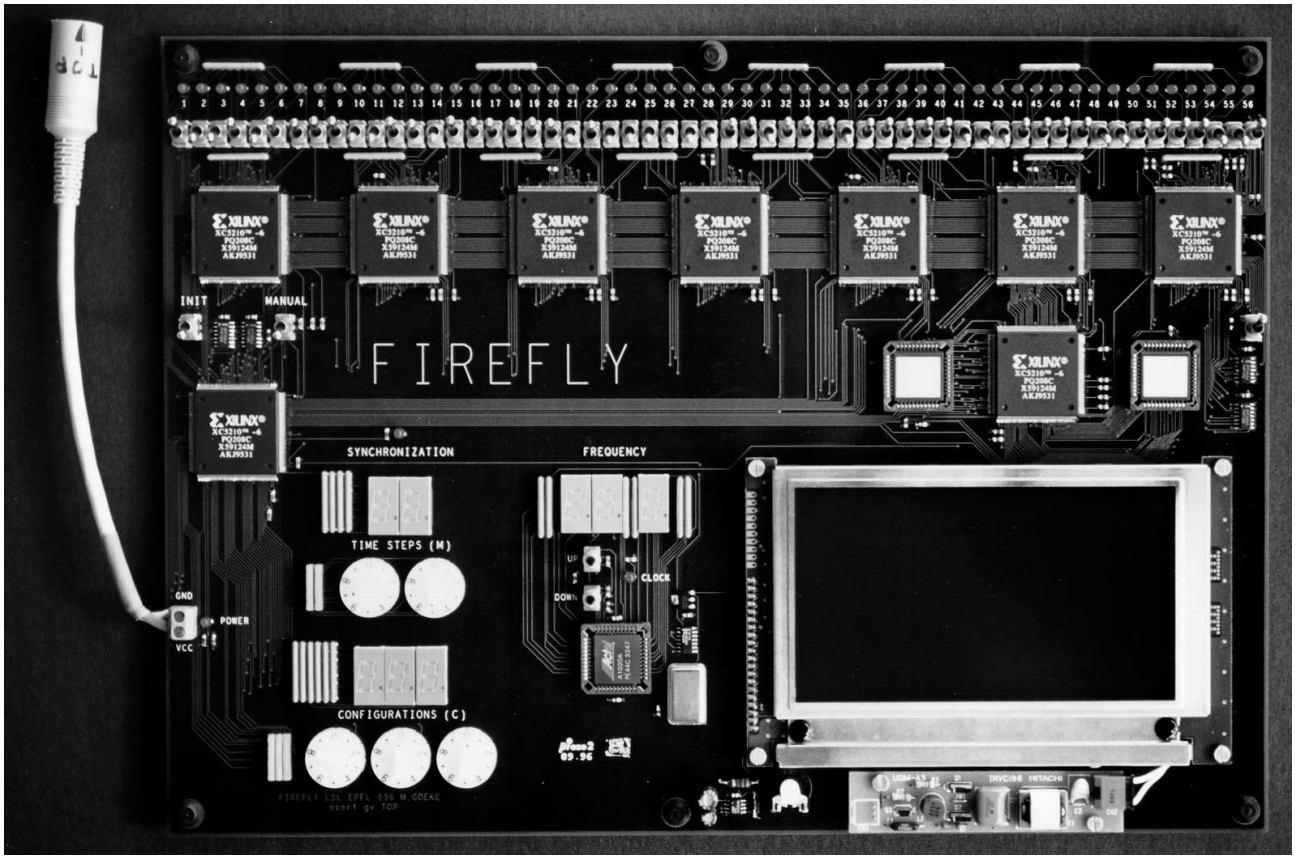

Fig. 4. The firefly evolware machine. The system is a one-dimensional, non-uniform, $r = 1$ cellular automaton that evolves via execution of the cellular programming algorithm. Each of the 56 cells contains the genome representing its rule table; these genomes are randomly initialized, after which evolution takes place. 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.)

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 [32, 33]. 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 "external world" is the power-supply cable. The features distinguishing this implementation from previous ones (described in Ref. [1]) are: (1) an ensemble of individuals (cells) is at work rather than a single one, (2) genetic operators are all executed on-board, rather than on a remote, offline computer, and (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 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 III) 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 (this latter is used to test the evolved system after termination of the evolutionary process, by manually loading initial configurations). 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. The machine is depicted in Figure 4.

The architecture of a single cell is shown in Figure 5. 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 III), 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 ($nf_i(c) = 2$).

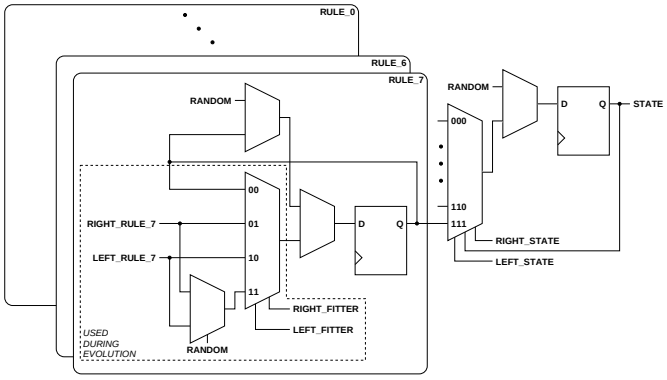

Fig. 5. Circuit design of a cell. The binary state is stored in a D-type flip-flop whose next state is determined either randomly, enabling the presentation of random initial configurations, or by the cell's rule table, in accordance with the current neighborhood of states. Each bit of the rule's bit 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 (initialization, execution, or evolution).

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, a four-bit shift register is used. 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., $0 \to 1 \to 0 \to 1$), 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 those 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 random number generator and the M and C counters. 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.

VI. Concluding remarks

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 focusing on the synchronization problem. We described an FPGA-based implementation of the cellular programming algorithm, the firefly machine, that exhibits complete online evolution, all operators carried out in hardware, with no reference to an external computer. Firefly thus belongs to the third category of evolving hardware, described in Section I. The major aspect missing concerns the fact that evolution is not open ended, i.e., there is a predefined goal and no dynamic environment to speak of. Open-endedness remains a prime target for future research in the field. We note that the machine's construction was facilitated by the cellular programming algorithm's local dynamics, highlighting a major advantage of such local evolutionary processes.

Evolware systems such as firefly enable enormous gains in execution speed. The cellular programming algorithm, when run on a high-performance workstation, executes 60 initial configurations per second.[3] In comparison, the firefly machine executes 13,000 initial configurations per second.[4]

The evolware machine was implemented using FPGA circuits, configured such that each cell within the system behaves in a certain general manner, after which evolution is used to "find" the cell's specific behavior, i.e., its rule table. Thus, the system consists of a fixed part and an evolving part, both specified via FPGA configuration strings (Figure 6). An interesting outlook on this setup is to consider the evolutionary process as one where an organism evolves within a given species, the former specified by the FPGA's evolving part, the latter specified by the fixed part. This raises the interesting issue of evolving the species itself.

[3]This was measured using a grid of size $N = 56$, each initial configuration being run for $M = 75$ time steps, with the number of configurations between evolutionary phases $C = 300$.

[4]This is achieved when the machine operates at the current maximal frequency of 1 MHz. In fact, this can easily be increased to 6 MHz, thereby attaining 78,000 configurations per second.

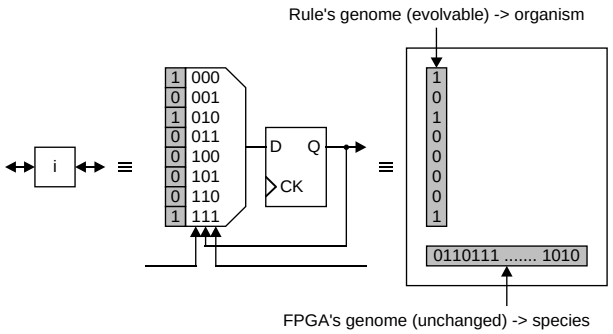

Fig. 6. The firefly cell is hierarchically organized, consisting of two parts: (1) the "organismic" level, comprising an evolving configuration string that specifies the cell's rule table, and (2) the "species" level, a fixed (non-evolved) configuration string that defines the underlying FPGA's behavior.

References

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

[2] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Pérez-Uribe, and A. Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. 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 35–54. Springer-Verlag, Heidelberg, 1997.

[3] 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.

[4] J. R. Koza, F. H Bennett III, D. Andre, and M. A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 123–131, Cambridge, MA, 1996. The MIT Press.

[5] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development and evolution of hardware behaviors. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, Heidelberg, 1996.

[6] H. Kitano. Morphogenesis for evolvable systems. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 99–117. Springer-Verlag, Heidelberg, 1996.

[7] T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, T. Furuya, and B. Manderick. Evolvable hardware and its application to pattern recognition and fault-tolerant systems. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, Heidelberg, 1996.

[8] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. 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 62–71. Springer-Verlag, Heidelberg, 1996.

[9] M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi. A pattern recognition system using evolvable hardware. 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 761–770. Springer-Verlag, Heidelberg, 1996.

[10] A. Thompson, I. Harvey, and P. Husbands. Unconstrained evolution and hard consequences. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 136–165. Springer-Verlag, Heidelberg, 1996.

[11] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. 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 390–405. Springer-Verlag, Heidelberg, 1997.

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

[13] 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.

[14] R. A. Brooks. New approaches to robotics. *Science*, 253(5025):1227–1232, September 1991.

[15] T. S. Ray. An approach to the synthesis of life. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, volume X of *SFI Studies in the Sciences of Complexity*, pages 371–408, Redwood City, CA, 1992. Addison-Wesley.

[16] P. Galley and E. Sanchez. A hardware implementation of a Tierra processor. Unpublished internal report (in French), Logic Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, 1996.

[17] 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.

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

[19] T. Toffoli and N. Margolus. *Cellular Automata Machines*. The MIT Press, Cambridge, Massachusetts, 1987.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] J. P. Crutchfield and M. Mitchell. The evolution of emergent computation. *Proceedings of the National Academy of Sciences USA*, 92(23):10742–10746, 1995.

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

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

[28] 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.

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

[30] 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.

[31] 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.

[32] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press,

Cambridge, MA, 1996.

[33] 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, Heidelberg, 1996.

[34] 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, Heidelberg, 1996.