ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **16**, Numbers 2–3, 2013, 177–191

Control Global Loops in Self-Organizing Systems

Mihaela MALIŢA

Saint Anselm College, NH
E-mail: mmalita@anselm.edu

Gheorghe ŞTEFAN

Politehnica University of Bucharest, Romania
E-mail: gstefan@arh.pub.ro

Abstract. Self-organizing systems are used to model complex or at least apparent complex behaviors. The idea currently in use is to study global behaviors based only on fixed and local rules. The emergence of a specific global behavior depends too strongly on the initial state of the system. This chaotic aspect can be avoided adding, besides the local loops, a global control loop to balance the too big "local freedom". The global loop can be used to tune the local behavior so as the entire system becomes more stable. Using various reduction functions some global parameters are extracted from the system and sent back locally to the elements of the system. The information received locally through the global loop is used to "tune" the local behavior. So the self-organized system becomes "less self-organized", but more flexible in maintaining its global state in prescribed limits. In order to illustrate the idea, the paper presents simple examples of cellular automata with global loops.

1. Introduction

In a previous paper [1] we concluded that in order to catch the entire complexity of the behavior of a social system, the self-organizing mechanism is not the only one to be considered. The real behavior of a complex system cannot be modeled taking into account only fix and local rules. The rules must change slightly according to the nonlocal events. A more accurate model will consider a more extended context for

each element of the model. The strict local environment must be expanded up to the global level if needed. A general model can consider a hierarchy of "regional" loops ended with a global loop. For this initial approach of the problem let us consider the simplest situation with a global loop added over a self-organizing system.

We chose the cellular automata model as the environment to prove the symbiosis between local loops and global loops, between local behaviors and global behaviors. The model was successfully used by J. M. Epstein and R. Axtell to study artificial societies [2]. They considered only fix and local rules. Their approach is a successful one, but a lot of aspects remained to be considered in order to catch all the behaviors characteristic to a real community. A real community evolves taking into account both, local and global events. Global behaviors can emerge from local and fix rules, but the surviving conditions for a complex community assume a global feedback also.

The aim of this paper is to proof that by extracting, from each cell of a cellular automaton (CA), a simple information, which is globally analyzed and sent back to each cell, we are able to induce additional behaviors. For the sake of simplicity, in this paper only one-dimension and two-dimension CA, with one-bit state, are used to proof the effects of a global feedback introduced in a self-organizing system. For more complex CA, such as *The Sugarscape Model* presented in [2], are suggested few types of loops to control the surviving process of the actors.

The second section of the paper is a short introduction in CA, with emphasis of the behaviors initiated in a random state. We will prove that, under the local control of the fixed rules, the complexity of the activity attenuates too fast. The third uses the CA introduced in the previous section (a linear 1-bit cell CA and Conway's Game of Life CA) to exemplify the effect of the control global loop. In the fourth section few control global loops are proposed for *The Sugarscape Model*.

2. The Cellular Automaton Model

A one-dimension cellular automaton, CA1, is described by the state vector V, the transition function $F(V)$, and the initial state of V. For a n-cell CA1 the state vector is:

$$V = (a_1\ a_2 \dots a_i \dots a_n)$$

where the state of each cell a_i is a list containing all the characteristics used to model the investigated behavior. The transition function provides the next state for each cell. Each cell switches its state according to the following function:

$$a_i^+ = f(a_{i-1}, a_i, a_{i+1})$$

A two-dimension cellular automaton, $CA2$, is described by the state matrix M, the transition function $F(M)$, and the initial state of M. For a $n \times n$ $CA2$ the state matrix is:

$$M = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nj} & \dots & a_{nn} \end{pmatrix}$$

where the state of each cell a_{ij} is a list containing all the characteristics used to model the investigated behavior. The transition function provides the next state for each cell. For example, if the smallest von Neumann neighborhood is considered, each cell switches its state according to the following function:

$$a_{ij}^{+} = f(a_{ij}, a_{i+1,j}, a_{i-1,j}, a_{i,j+1}, a_{i,j-1})$$

For example, in *The Sugarscape Model*

$$a_{ij} = (e_{ij} p_{ij})$$

where the list has two components: one for the environment, the list e_{ij}, and one for people, the list p_{ij}. The first describes the local resources for surviving (the amount of sugar, the regeneration rate,), while the second describes the actor living in the cell (its existence, the amount of surviving resources it has, its metabolism, ...).

Modeling a social behavior means to maintain a behavior or to maintain an emergent behavior starting from a randomly established state. In most cases a CA, starting from a random state, ends in a state which is far from the initial state and shows usually a simple behavior. Two examples are provided in the following.

Example 1. Let be a simple one-dimension CA of 1-bit 64 cells. The transition rule is 120 (`8'b0111_1000`). The simulator presented in *Appendix 1* is used with:

```
initFunc = 8'b0111_1000;
loop = 1'b0; // no loop closed; rule = initFunc
initState = 64'b00101100_10100110_10111100_00000110_10111111_10100110_
                  10111100_10100110;
```

The resulting sequence of the state of CA is:

```
#1  out=0010110010100110101111000000011010111111101001101011110010100110
#3  out=0011110011100111111001000000011111100000111001111110010011100110
#5  out=0010010010100100001001000000010000100000101001000010010010100110
#7  out=0010010011100100001001000000010000100000111001000010010011100110
#9  out=0010010010100100001001000000010000100000101001000010010010100110
#11 out=0010010011100100001001000000010000100000111001000010010011100110

#29 out=0010010010100100001001000000010000100000101001000010010010100110
#31 out=0010010011100100001001000000010000100000111001000010010011100110
```

After only two cycles the system enters in a stable loop of two states, thus showing a completely uninteresting behavior. This is a typical way to evolve starting from almost any randomly selected initial state. The code presented in *Appendix 1* can be used to repeat the experiment with other initial states.

◇

Example 2. Conway's Game of Life is an example of two-dimension CA. Each cell has two states: alive, coded by 1, and dead, coded by 0. Always from a randomly established distribution of 1s and 0s is: the population of 1s dies or loops with a small amount of living cells. Using the simulator built in Scheme (see Appendix 2) we obtained the results summarized in Table 1, where:

Size : is the number of line and columns of the CA

Initial Population : the number of the initial alive cells randomly established

Average Population : the average population of the CA when they enter in a steady state operation with no extinct population

Average Population Lost : the average population lost when the CA enters in a steady state operation with no extinct population

Died % : the percentage of CA died when enter in a steady state operation

Table 1. The behavior of the $Size \times Size$ CA starting from a random initial state. Average Population is measured only for the CA with living population at the end of test.

Size	Initial Population	Average Population	Average Population Lost	Died %
10	50	10.69	39.31	30%
12	72	11.48	60.52	20%
14	98	14.67	83.33	15%
16	128	15.93	112.07	9%
18	162	19.48	142.52	8%
20	200	21.63	178.37	4%

For each type of CA were performed 100 experiments. No one was "able" to keep alive its entire population. Some have lost their populations, and where the population is not extinct no more than 20% of the population was maintained alive. The living population has by the rule a simple cyclic behavior. For example, the 16×16 CA initialized randomly after 26 cycles is found stuck in a state with only 10 active cells, as follows:

```
0 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 0 1 1 1 0 1 0 0 1          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 1 1 0 0 0 1 0 1 1 1 1 1          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 1 0 1 1 1 0 0 0 1 0 1          0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 0          0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 1 1 0 0 1 0 1 0 1 1 1 0 0 1 1  26 x => 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 1 1 0 0 1 1 1 1 1 1 0 1          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 1 0 0 1 0 0 0 0 0 0 1 0          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 1 1 0 1 0 1 0 1 1 1          0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 0 1 1 0 1 1 0 1          0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 0 1 0 1 1 1 0 0 0 1 0          0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 0          0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 1          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

◇

In order to use these CA and other more complex CA for modeling and simulating complex processes initiated in a complex random state, something must be done at least for maintaining "alive" their population.

3. The Control Global Loop

The *local & fix* functions are very good at *conserving* the *libera*l local behavior of the actors which populates the cells of a CA. But, global ("social") behaviors could emerge in such a system. Then, at the level of the system global control mechanisms are needed in order to direct the overall behavior, because, as we have seen in the previous section, the global behaviors tended toward simple insignificant behaviors. We propose to add a global loop closed over the entire system in order to provide a mechanism which can be used to induce *social*l "politics" inside the system which is now too *conservative* (because the rules are fix) and *liberal* (because the rules are only locally applied).

Let be a CA, linear or two-dimension, defined by the array A (which is V or M), and the transition function $F(A)$. Introducing the global loop means to define a new function $G(A)$ and to modify the transition function to $F(A, G(A))$. The new $G(A)$ function is a reduction function. It is used to characterize globally the state of the CA. The resulting value is used to "adjust" the way the transition function of the system is applied.

For example, the simplest version of a global loop, closed over the Conway's Game of Life CA, uses the reduction function $G(A)$ defined to count in each cycle the alive cells.

The first attempt to use a global loop closed over a CA is published in [3]. The loop was closed over a 255-cell linear CA with one-bit cells. The function $G(A$ was a reduction add which provided in each cycle the local transition function.

The two examples from the previous section will be revisited. The same types of CAs will be investigated how they behave with a global loop.

Example 3. The linear CA described in *Appendix 1* is used now with the global loop closed (`loop = 1`). The output `active` of the module `progAC` provides in each cycle the number of cells in state 1. In the top module `noiseGenAC` the value `active` is compared with $n/2 + h$ and $n/2 - h$. If it is in the interval, then the initial function is applied. If it is below the interval the function 121 (`8'b0111_1001`), which increases the number of 1s, is applied. If it is above the interval the function 104 (`8'b0110_1000`), which increases the number of 0s, is applied. The parameter `h` defines the *hysteresis* of the process. For the same initial state as in *Example 1*, now the CA, with the global loop closed and the hysteresis `h=6`, provides a non-periodic behavior of 585 cycles (one cycle is #2 unit times), as follows:

```
#1    00101100101001101011110000000110101111110100110101111001010011 0 rule=120
#3    00111100111001111110010000000111111000001110011111100100111001 10 rule=121
#5    10100100101001000010010111110100001011101010010000100100101001 10 rule=120
#7    11100100111001000010011100011100001110111110010000100100111001 11 rule=120
#9    00100100101001000010010100010100001011100010010000100100101001 00 rule=120
#11   00100100111001000010011100011100001110100010010000100100111001 00 rule=120
#13   00100100101001000010010100010100001011100010010000100100101001 00 rule=121
#15   10100100111001011010011101011101101110101010010110100100111001 01 rule=121
```

```
#17   11100100101001111110010111110111111011111110011111100100101001 11 rule=121
#19   00100100111001000010011100011100001110000010010000100100111001 00 rule=120
#21   00100100101001000010010100010100001010000010010000100100101001 00 rule=104
#23   00000000010000000000000100000100000010000000000000000000001000 00 rule=121
                                        ...
#1163 11110000000111111001110000011111100111000111001111110000011100 11 rule=121
#1165 00010111110100001001010111010000100101010101001000010111010010 10 rule=104
#1167 00001100011000000000101101100000000001010101000000000011011010 00 rule=120
#1169 00001100011000000000111111100000000001111111000000000011111110 00 rule=120
#1171 00001100011000000000100000100000000001000001000000000010000010 00 rule=121
                                        ...
```

then enters in the 11-step cyclic evolution:

```
#1173 11101101011011111110101110101111111010111010111111101011101011 11 rule=121
#1175 00111111111110000011111011111000001111101111100000111110111110 00 rule=121
#1177 10100000000010111010001110001011101000111000101110100011100010 11 rule=104
#1179 11000000000001101100001010000110110000101000011011000010100001 10 rule=104
#1181 11000000000001111100000100000111110000010000011111000001000001 11 rule=120
#1183 01000000000001000100000100000010001000001000001000100000100000 00 rule=121
#1185 01011111111101010101110101110101010111010111010101011101011101 01 rule=121
#1187 11110000000111111111011111011111111101111101111111111011111011 11 rule=121
#1189 00010111110100000001110001110000000111000111000000011100011100 00 rule=104
#1191 00001100011000000000101000101000000010100010100000000101000101 00 rule=104
#1193 00001100011000000000100000100000000001000001000000000010000010 00 rule=121
```

Now, instead of applying in each cycle the same transition rule (`rule=120`), the rule changes according to the value computed by the function $G(A)$which provides, in each cycle, the number of cells in the state 1. The resulting number is compared with $32 + 6$ and $32 - 6$ to decide if the current rule (initially 120) is maintained or is substituted with 121 or 104. If the number of 1s decreases under 26, then the rule 121 is applied; if the number of 1s increases over 38, then the rule 104 is enforced; if the number of 1s is maintained in [26,38] interval the rule used to switch the state of the CA is the initial one, 120.

If the hysteresis is more tight, $h = 3$, then the complexity of the behavior increases. The non-periodic initial sequence has 772 cycles and the periodic sequences which follow have 1225 cycles.

There is a very big difference compared with the simple behavior of the CA in *Example 1* where in few cycles the system enters in a very short cycle of states. The code from *Appendix 1* can be used to make more experiments.

◇

Example 4. The Game of Life can be also modified so that the surviving rate is maintained higher, near the initial value. The function $G(A)$ is similar. It counts the number of living cells. The result is used to increase the birth rate by modifying the local transition rule if the population decreases under the accepted threshold `T`, where:

```
T = initValue - Difference
```

with:

- `initValue`: the number of living cells at the random initialization

- `Difference`: the maximum negative variation of the population related to the initial population which is accepted before taking action to increase birth

Table 2. The average population in *Game of Life* cellular automaton (For each pair `(Difference, Step i)` 20 experiments were run.)

Difference	Step 1	Step 4	Step 8	Step 16
0	104.14	103.03	102.02	100.02
5	103.73	100.41	99.23	96.15
10	102.12	96.70	96.68	93.94
15	99.07	96.39	92.41	85.28
20	96.74	93.79	87.09	84.14
25	94.31	89.79	85.98	83.69
30	93.43	86.58	86.58	77.86
35	91.59	85.78	79.91	76.42
40	89.10	84.29	80.13	75.31
45	87.48	82.44	79.82	74.61
50	85.90	81.71	78.03	74.25

The local transition rule is modified as follows:

- Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.

- Any live cell with two or three live neighbors lives on to the next generation.

- Any live cell with more than three live neighbors dies, as if by overcrowding.

- **If** `(G(A)> T)` **then** any dead cell with exactly three live neighbors becomes a live cell **else** any dead cell with two or three live neighbors becomes a live cell.

The condition can be tested in each step or periodically at each 4, 8, 16, ... steps. The simulator described in *Appendix 2* was used to measure the behavior of the Game of Life for a 16×16 CA for different values of `Difference` and of the step used to test the threshold condition. The results are summarized in Table 2 and Fig. 1.

Because the initial population is established randomly, the average number of initial living cells is 128. The worst case in the investigated scenarios shows around 60% of population survives in ***any*** experiment (the accepted deviation from the initial population is 50 and the test is performed at each 16 cycles only). For a deviation no bigger than 15 people and a test performed once at no more than 8 cycles the population is maintained in ***any*** experiment at more than 72% from the initial population.

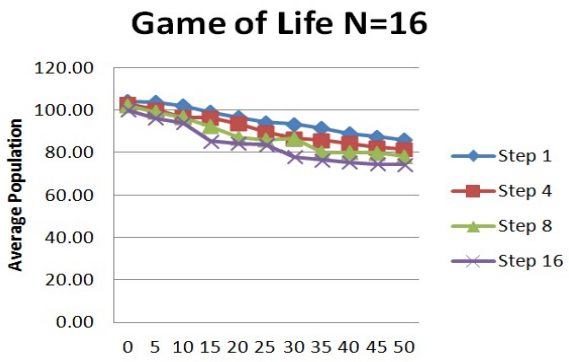

Fig. 1. The graphic representation of the Table 1.

If the control is "vigilant" and "harsh", i.e. `Step 1` and `Difference=5`, more than 80% of the population is maintained alive.

If the loop-controlled Game of Life CA is compared with the no loop version we see a big improvement in the ability to keep the population alive. The no loop version with the same size, 16×16, is able to keep alive no more than 12.5% of the initial population when not all the population is extinct (which is the case in 9% of the experiments)

We conclude that the ability to keep the population alive can be increased more than 5 times by adding the global control.

◇

At the end of this section we are in the position to assert that the global loop increases the possibility to maintain the randomly established distribution at a reasonable level of complexity. The no loop versions of the simple CA we investigated tend to be attracted, after a small number of cycles, to simple, uninteresting behaviors. In the next section are offered few suggestions about how to add the global loop for more complex CA.

4. Control Global Loops in Artificial Societies

In what follows we shall describe first the agent-based model of social processes presented by Epstein & Axtell as Artificial Societies (AS) in [2]. Then, the way a global loop can be applied is presented as the foundation for future researches in the AS domain.

Epstein & Axtell viewed AS as laboratories, where they attempted to grow certain social structures with the aim to discover fundamental rules that are sufficient to generate macroscopic social structures and collective behaviors of interest.

For their computer experiments the authors involved the model of CA with three basic ingredients: *agents*, *environment*, and *rules*. Agents and environment are associated to each cell, while the rules specify locally the way the system evolves.

4.1. Agents

The "people" of AS are called agents. Each agent is characterized by its internal states and behavioral rules. Some states are fixed, while others change through interaction with the neighbor agents or with the local environment. For example agent's sex, metabolic rate, and vision are all fixed. However, wealth and health change as agents move around and interact locally.

4.2. Environment

Life in an artificial society unfolds in the environment which is characterized by parameters associated to each cell. It is naturally modeled as a lattice of resource-bearing sites located in each cell. The idea is that the environment is a medium, separated from the agents, on which the agents operate and with which they interact.

4.3. Rules

The behavior for the agents and for sites of the environment is specified by rules. There are three kinds of rules:

- *agent-environment* rules: "look around as far as you can, find the site richest in food, go there and eat the food"
- *environment-environment* rules: the rate of resource growth at a site could be a function of the resource levels at neighboring sites
- *agent-agent* rules: mating rules, combat rules, trade rules,

4.4. How the model works?

An initial population of agents is released into the environment. Then, after a number of cycles we look for organizations into recognizable macroscopic "social" patterns. The formation of social groups or the emergence of certain "wealth" distributions would be meaningful. Thus, if we are both, lucky and enough optimistic, we will see how the collective structures "grow" from bottom up, and we will infer that the rules embedded in our model approximate the rules governing real societies.

4.5. The Sugarscape Model

The environment consists of variously shaped regions, some rich in *sugar* agents like to eat, some relatively impoverished. Agents are born onto the *sugarscape* with a vision (how far they can see the sugar), and a metabolism (how much sugar they consume lasting in their cells or moving in a neighbor cell). Every time an agent moves around looking for more sugar to refill their sugar reserve. But, meantime they metabolize, consuming from their sugar reserve. Agents die if and when they eat all their sugar. Since agents are trying to accumulate sugar at all times, there is always a

distribution of wealth, measured in sugar. Thus, the wealth distribution is mimicked in the model of AS.

Each cell of the CA used to build *The Sugarscape Model* is described by the list: $a_{ji} = (e_{ji}p_{ji})$, where the two components of the list are structured as lists containing the description of the environment and of the people, if any, in the cell.

4.6. Loop-Controlled Sugarscape

In *Sugarscape* all is local. The agent is able to look around at constant distance only. The rule takes into account only a limited neighborhood. The rules are fixed, or can change according to the information gathered from a constant extended neighborhood. Global behaviors emerge, but the global parameters cannot influence the overall evolution of the system. The system looks like a pure self-organizing system, unable to follow a desired evolution.

A real social system is a complex combination of *fix*, *local* and *global* issues. A good balance between these three tendencies is the key that opens the ways for the most valuable evolutions. It is good to have fixed rules, but not fixer than is needed. It is efficient to solve locally most of issues, but not all can be solved locally. It is fair to take into account global concerns, but not in a totalitarian mode. All kind of "politics" must be accommodated in our approach. "Conservative", "liberal" and "social" politics, applied with an appropriate weight in time and space, have the chance to keep our *Sugarscape* in the best shape. Thus, besides the*fix & local* rules used in the original model of Epstein & Axtell, we propose to add one or few global loops of the same type presented in the previous section.

The first step is to add reduction functions in system. From the description of each cell, $a_{ij} = (e_i p_{ij})$, meaningful information is sent to a *log*-deep reduction network having in its nodes specific functions, such as `add, max, min, XOR, OR, ...`. The results obtained from these reduction networks provide a global evaluation of the system's state. It is used to establish the global decisions to be enforced back into the system. Possible parameters computed globally are: the number of actors, the average value of the food reserve each actor owns, the total amount of food distributed over the landscape, the average amount of food in each cell, how many cells are empty, how many cells are empty of food, but populated with an actor, ...

The second step is to add in each $a_{ij} = (e_i p_{ij})$ appropriate local dependencies by the global parameters computed with the reduction functions. Thus, the metabolism of each actor can be made dependent on the total amount of food in the system (shortage of food induces changes in diet), or the regeneration rate of the food in the system must be dependent on the average value of the food reserve each actor owns, and so on.

The third step is to find the best timing for closing the control loops. The social regulations must be applied with an appropriate delay. If global regulations change to fast the system is unable to follow them, while they introduced with a too big delay the control is lost. There are situations when a non appropriate rate of social intervention produces fatal instabilities in the system. The good news is that we work in a simulated environment where experiments and corrections are possible.

5. Concluding Remarks

The self-organized systems were studied in this paper from a new perspective. Instead of looking, as usual, for spectacular emergent behaviors we considered another approach. Our goal is to maintain and to gently direct the behavior of a complex system in order to fulfill surviving goals for randomly instantiated processes.

We consider that *fix & local* rules must be complemented with *global control loops* in order to "keep alive" a complex randomly instantiated system. We exemplified our proposal with simple cellular automata.

The aim of this paper was also to expand ideas like those introduced in [2] to global loop controlled self-organizing systems. We consider this kind of approach as promising for studying complex relations in multiethnic communities.

References

[1] Maliţa M., Gheorghe M. Ş., *Real complexity vs. apparent complexity in modeling social processes*, in this volume.

[2] Epstein M. Epstein R.A., *Growing Artificial Societies: Social Science from the Bottom Up*, MIT Press, 1996.

[3] Gheorghe M. Ş., *Looking for the Lost Noise*, CAS '98 Proceedings, Oct. 6–10, 1998, Sinaia, Romania, pp. 579–582.

Appendixes

The two instruments used to exemplify and prove the opportunity of the control global loop in a self-organized system are annexed. The electronic forms are downloadable at: `http://www.anselm.edu/internet/compsci/Faculty_Staff/mmalita/`

A. Appendix 1

The CA used in Example 1 and 3 is described here as circuit.

```
module noiseGenAC #(parameter n = 64          , // number of cells
                              m = 6           , // active paramenter size
                              h = 4'b110      , // the histerezis
                              P = 8'b0111_1001, // function to plus
                              M = 8'b0110_1000) // function to minus
       (output [n-1:0] out      ,
        input          init     ,
        input  [n-1:0] initState,
        input  [7:0]   initFunc ,
        input          loop     ,// 1: loop closed
        input          clock    );

    reg  [7:0]  rule   ;
```

```
    reg  [31:0] counter;
    wire [m:0]  active ;
    always @(posedge clock)
        if (init)                                rule <= initFunc;
         else if (~loop)                         rule <= rule    ;
               else if (active > (n/2+h))        rule <= M       ;
                     else if (active < (n/2-h)) rule <= P       ;
                           else                 rule <= initFunc;

    always @(posedge clock)
        if (init) counter <= 0               ;
         else     counter <= counter + active;

    ruleAC progAC(out      ,
                  active   ,
                  init     ,
                  initState,
                  rule     ,
                  clock    );
endmodule

module progAC #(parameter n = 64, m = 6)
       (output reg [n-1:0] out      ,
        output reg [m:0]   active   ,
        input              init     ,
        input      [n-1:0] initState,
        input      [7:0]   rule     ,
        input              clock    );
    reg [n-1:0] nextOut;
    reg [m:0]   sum    ;

    integer i;

    always @(*) begin
      if init)n extOut = initState  ;
       else for (i=0; i<n; i=i+1)
             nextOut[i] = rule[{out[i], out[(n+i-1)%n], out[(i+1)%n]}];
       sum = 0;
       for (i=0; i<n; i=i+1)
        sum = sum + out[i];
    end

    always @(posedge clock) begin
      out <= nextOut ;
      if (init) active <= 0  ;
       else     active <= sum;
    end
endmodule
```

B. Appendix 2

The Game of Life CA used in Examples 2 and 4 is described here in Scheme.

```
; Game of Life in Scheme (Racket) with a global loop
(load "matrices.rkt"); functions for working with matrices
;; GLOBAL variables
(define A ())  ; our initial array of cells DIM x DIM
(define newA ()) ; the New Array (next generation)
(define DIM 8)   ; size Matrix
(define STEP 5) ; after how many generations the System checks itself
                ;( when the Global loop is effective)
(define clock 200) ; how many generations we let them run
(define manyGames 10);how many games
(define CYCLES 0) ; how many cycles does a population live
(define countDied 0); how many games died before the max number of
                    ; cycles given
(define DIFF 1000); difference  where it looks (MARGIN)
(define aveAlive 0) ; average alive during the game
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (start)
(let((countDied 0)
     (totalAlive 0)
 (aveAliveEnd 0)
 (alive 0)
 (totalAveAlive 0)
)
  (do ((k 0 (inc k))) ((= k manyGames))
    (game)
    (set! alive (countAll A)) ; count last alive in last game
    (set! totalAlive (+ totalAlive alive)); total alive in all GAMES
    (set! totalAveAlive(+ totalAveAlive aveAlive)); average alive at last
    (display `(Game: ,k Size: ,DIM GlobalLoop: ,STEP Cycles: ,CYCLES ))
               (display `(Max: ,clock Alive: ,alive AveAlive: ,aveAlive))
    (if (> CYCLES 0) (set! countDied (inc countDied)))
  ); end do
  (set! aveAliveEnd (/ totalAlive manyGames))
  (set! aveAlive     (/ totalAveAlive manyGames))
  (display `(REPORT Games: ,manyGames Size: ,DIM GlobalLoop: ,STEP ))
  (display `(DifferenceSee: ,DIFF
             Max: ,clock
 Died: ,countDied
 aveAlive: ,aveAlive)
)
  (display `(aveAliveEnd ,aveAliveEnd))
))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (game); matrice A is DIM x DIM in our first example
(set! A (randMatrix DIM 0 2))  ; make initial array DIM x DIM
(writeMatrix A)
```

```
(set! newA A) (set! CYCLES 0)
(let ((aliveStart(countAll A))(aliveNow 0) (rule "") (totalGameAlive 0))
  (call/cc (lambda (myexit)
    (do ((k 0 (inc k))) ((= k clock))
     (set! aliveNow (countAll A));count the cells to decide the rule
     (set! totalGameAlive (+ totalGameAlive aliveNow))
     (if (= aliveNow 0) (begin (set! CYCLES k)(myexit k)))
     ; Global View decides the rules to be applied.
     (cond ((= (remainder k STEP) 0)
              (if (< aliveNow (- aliveStart DIFF)); check the decrease
                (begin (set! rule   "r23")(rule2or3born)); more born
                (begin (set! rule   "r3") (rule3born)); old rule
            ))
            (#t (if (eq? rule "r3") (rule3born)
                     (rule2or3born) ));continue with the same last rule
      )
      (set! aveAlive (/ totalGameAlive (+ k 1))); average in last game
      (display
   `(,k ".AveAlive=" ,aveAlive " Alive=" ,aliveNow " Rule=" ,rule)
  )
      (writeMatrix A)
      (set! A newA) ; A becomes NewA
    ); end do - change clock with a new matrix (generation)
  )) ; close call/cc
))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (rule3born) ; old rule
 (do ((i 0 (inc i))) ((= i DIM)) ; for all cells (i j)
   (do ((j 0 (inc j))) ((= j DIM))
    (let ((sum (countAlive A i j)) ) ; count neighbors alive for (i j)
         (cond ((= (val A i j) 1) ; if cell is alive
                (if (member sum '(2 3))
  #t (set! newA (changeMatrix newA i j 0)))
)
               ((= (val A i j) 0) ; check if cell is dead
                 (if (= sum 3) ; when exactly 3 are alive get alive
                  (set! newA (changeMatrix newA i j 1))
 )
               ); becomes alive
         )
))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (rule2or3born) ; new rule - more conditions for born
(do ((i 0 (inc i))) ((= i DIM))
   (do ((j 0 (inc j))) ((= j DIM))
    (let ((sum (countAlive A i j)) ) ; count neighbors alive for (i j)
         (cond ((= (val A i j) 1) ; is alive
                 (if (member sum '(2 3))
   #t (set! newA (changeMatrix newA i j 0)))
```

```
               )
               ((= (val A i j) 0) ; check if dead
                 (if (member sum '(2 3));when 2 or 3 are alive get alive
                   (set! newA (changeMatrix newA i j 1)))
               ); becomes alive
         )
))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (countAll A) ; count all alives in matrix A
  (apply + (map (lambda(z)(apply + z)) A)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (countAlive A i j); sum of all neighbors alive for (A i j)
(let ((n (dec DIM)))
(cond ((and (> i 0)(> j 0) (< i n) (< j n)); cells with 8 neighbors
        (+ (val A (dec i) (dec j))(val A i (dec j))(val A (dec i) j)
           (val A (inc i) (inc j)) (val A (inc i) j) (val A i (inc j))
           (val A (inc i) (dec j)) (val A (dec i)(inc j))  ))
       ((and (= i 0)(= j 0)); corner up-left: 3 neighbors
            (+(val A (inc i)(inc j))(val A i (inc j))(val A (inc i) j)))
       ((and (= i 0)(= j n)); corner up-right: 3 neighbors
            (+(val A i (dec j))(val A (inc i)(dec j))(val A (inc i) j)))
   ((and (= i n) (= j 0)); corner down-left: 3 neighbors
            (+(val A (dec i) j)(val A i (inc j))(val A (dec i)(inc j))))
       ((and (= i n) (= j n)); corner right-down: 3 neighbors
            (+(val A (dec i)(dec j))(val A i (dec j))(val A (dec i) j)))
        ((= i 0) ; cells first line : 5 neighbor
            (+ (val A i (dec j)) (val A (inc i) (inc j))(val A i (inc j))
               (val A (inc i) j) (val A (inc i) (dec j)) ))
       ((= i n); cells last line: 5 neighbors
            (+(val A i (dec j))(val A  i (inc j))(val A (dec i) (dec j))
                  (val A (dec i) j) (val A (dec i)(inc j))))
       ((= j 0); cells first column:5 neighbors
            (+(val A (dec i) j)(val A (inc i) (inc j)) (val A i (inc j))
               (val A i (inc j)) (val A (dec i)(inc j)) ))
       ((= j n); cells last column: 5 neighbors
            (+(val A (dec i) (dec j))(val A i (dec j))(val A (dec i) j)
               (val A (inc i)  j) (val A (inc i) (dec j))))
)))
```