Temporal Resolution in Coevolution

Towards Creating the Molecules of Life in Silico

Hossein Mobahi, Caro Lucas
Department of Electrical and Computer Engineering, University of Tehran
Tehran, Iran
hmobahi@acm.org , lucas@ipm.ir

Abstract— **In order to have an artificial life system with high evolvability, we should not only look for life-like organisms, but also the molecules that are able to create them. From this point of view, a new trend has appeared that coevolves machine instruction set and programs together, hoping to reach a more complex life and a higher diversity. However, we argue that simple coevolution of programs and instruction set can become problematic. We suggest using different temporal resolutions between the two evolutions to alleviate the problem. Our simulation results indicate that this approach and improve coevolution's performance.**

Keywords- ***Assembler Automata, Coevolution, Tierra, Artificial Life.***

1 Introduction

Our understanding of the evolution of living organisms, and attempts to derive general theories of evolution, are hampered by the fact that we only have one example of life to study, life on the Earth. Therefore, there are many situations where we are uncertain whether a particular feature of life general to any comparable system of evolving self-replicators, or whether it is specific to the biological organisms on our planet.

Recently, a number of researchers have started investigating the idea of creating alternative examples of evolution, in software, to assist in the formulation of generalized theories. Such work involves the creation of a computer environment in which large numbers of programs compete for the resources (i.e. CPU time and memory) required to make copies of themselves. The pioneer of this type of research is Tom Ray, who designed the Tierra system [13].

In the development of an Artificial Life (Alife) system, it is important that the system be designed so that it is evolvable, i.e. sustain a spontaneous and apparently open-ended growth of complexity. Therefore, it is necessary for us to prepare a good artificial environment during system construction, since the possibility of evolution happening in the system usually depends on the quality of the environment prepared by the designer

The actual biological system provides a good example for deciding what is needed in an Alife system. Today, it is widely accepted that an appropriate state of ancient earth permitted life to emerge and a variety of complex life forms to evolve. If we could identify the conditions necessary for this evolution [14], it would greatly assist in designing of a "good' Alife system.

In order to have an artificial life system with high evolvability, we should not only look for life-like organisms, but also the molecules that are able to create them. From this point of view, a new trend in has appeared that coevolves machine instruction set and programs together hoping to reach more complex life and higher diversity. The first steps of this approach have already been taken by Matsuzaki et. al [7, 8, 9]. However, they could not find any change in the instruction set; despite the fact that such capacity of instruction evolution was provided.

We argue that one obstacle could be the nature of coevolution itself. In fact, it is not fair to evaluate an instruction set before it is sufficiently explored by the program space. We believe that providing different temporal resolutions between these two populations can improve the performance of evolution. This is the problem that we will explore in this article.

This paper is organized as follows. In section 2 we will review early works on evolution of life using assembler automata and their recent extension. Section 3 discusses about the main issue, coevolutionary of molecules and organisms together. In its subsections, first cooperative coevolution will be reviewed. Next, we will introduce temporal resolution approach in coevolution. Eventually, we will discuss about credit assignment method for evaluating species. Section 4 is about simulations. In its first subsection we will review the architecture of our simulator. Next, we will show simulations results and obtained improvements. Finally in 5, we will have a conclusion and state future topics of this work.

2 Related Works

This work follows a line of research initiated by Dewdney in 1984 [4], who suggested an assembler automaton as a world of "Core War" where assembler programs can compete and may fight for computer resources. The system is in fact a multi-processor, shared-memory system. All processors operate on the same linear and cyclic memory called core. The machine code (called redcode) consists of only 10 instructions with one or two arguments. Memory cells can be addressed only direct-relative or indirect relative. Thus, the execution of a program is independent from its absolute position in the memory.

Rasmussen et al. have used Core Wars to build Coreworld [11,12]. They introduced random fluctuations in two different ways: random start of new processes with randomly initialized program counters and possibility of mutation when copying data using MOV instruction. Ray has designed Tierra to model the origin of diversity of life (e.g., the Cambrian explosion) not its origin [13]. Tierra is similar to Core War with some modifications and extensions: small instruction set without numeric operands, addressing by templates, memory allocation and protection, and artificial fitness pressure.

Later, these ideas were extended by other researchers for achieving a higher diversity. An extension was introducing a two-dimensional lattice space where lattice sites hold whole (linear) organisms. For instance Adami's work on Avida [1] and Dittrich's CoreSys [5] are in this path. These systems allow the study of pattern formation on population level and have showed a higher diversity.

In the original experiment of Tierra [13], the instruction set, a set of machine codes that species the operation was pre-programmed and thus could not able to be modified. If we were able to equip creatures with different instruction sets, the evolution of the instruction set in addition to the usual evolution of Tierra might take place. Therefore, another and a newer extension has appeared with a focus on coevolution of molecules and organisms of life together. So instead of designing ad-hoc machine instructions, suitable instructions for life formation can spontaneously emerge throughout the coevolution.

The first steps on this path was taken by Matsuzaki et al. [7, 8, 9] by following von Neumann's self-reproducing machine. Their proposed structure is divided into two parts: "Machine" and "Description". In the machine, there is a control memory and a set of registers. All operations are propelled by the interpretation and execution of "micro-codes" in the control memory. In the process of self-replication, the codes are initially written in the core memory as a part of the description tape.

3 Coevolutionary Approach

3.1 *Cooperative Coevolution*

Evolution is termed to the situation where a single population of species is evolved based on the fitness of species. The fitness is solely determined from individual species, regardless of the progress in others. On the other hand, in coevolution a set of populations interact with each other (directly or indirectly) throughout the evolution. This interaction may couple fitness of populations such that the fitness of individuals in one population depends on the evolutionary progress of the others. In fact, change in one individual will change the fitness landscape in others [3]; this has been referred to as "coupled fitness landscapes" [2].

3.2 *Temporal Resuoltion*

In the context of "Life in Silico", machine instructions are molecules in the environment that construct organisms. These molecules play an important role in emergence of the life in silico and they have been the focus of attention and a factor of extension of previous works. Inappropriate definition of these molecules can result in very brittle life. This life may vanish throughout generations due to mutations, or it may have a limited diversity. In fact, this was the reason that Ray changed Red Code to Tierra Code.

Nevertheless, there is no promise that Ray's ad-hoc instruction set is the optimum for emergence of life. Actually, current life on the earth exists thanks to the right origin for its emergence. Therefore, to recreate life, we should not only seek for the right organization of molecules, but also the molecules themselves which are capable of constructing life. Adapting instruction set as well as programs for achieving digital life has recently attracted attention of researchers. Matsuzaki et al. employed a coevolutionary approach to this problem [9]. Despite of adding mutability of instruction set, only programs evolved in their experiment and no mutation happened to instruction set.

We argue that one obstacle could be the nature of coevolution itself. In fact, it is not fair to evaluate an instruction set before it is sufficiently explored by the program space. This is because mutations of instructions have a broader effect (all occurrences of that instruction in the program code). Moreover, coevolution reduces the ratio of solutions to the search space and makes searching process harder. When only the program is evolved, the concern is just the right order of instructions, but when instructions are evolved too, we need the right instructions in the right arrangement.

Perhaps considering different time scales in generations of the two populations helps. This can delay selection of instructions so that programs have enough time for evolution and exploring the capacities of the mutated instruction. To assess this presumption, we designed a coevolutionary system with adjustable temporal resolution (Fig. 1).

At the beginning populations are initialized with manually designed instructions and programs. Evolution steps (evaluation, selection and mutation) occur faster in programs population, through the tight loop shown in the figure. Fitness is computed according to the success of programs in the population. The time resolution in instruction side is lower, i.e. it takes an evolutionary step after passing several generations in programs side. Fitness of programs is transmitted to instruction population for evaluation and selection purpose. The fitter instruction sets are then copied (possibly with mutation) to the next generation. These molecules are again used for evolving programs.

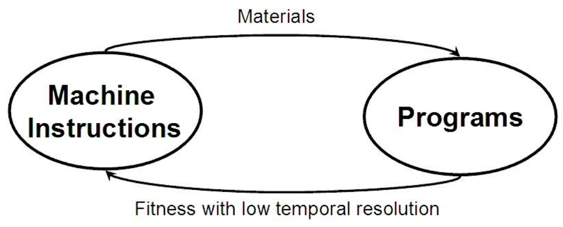

Figure 1 Cooperative Interaction between populations

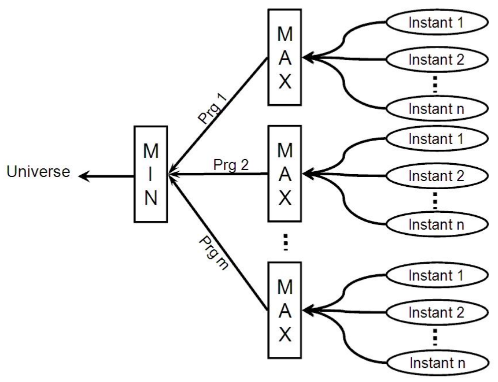

Figure 2 Tree-Like Fitness Propagation

3.3 *Credit Assignment*

A well-known issue associated with the fitness evaluation of multi-agent systems is credit assignment. When the system performs well or poorly which of the individual agents gets the credit or blame. For competitive coevolution systems with only two agents, this is not a problem since one can use a complement. However, when multiple agents are cooperating to achieve a solution, this is a more serious issue.

In general cooperative coevolution problems, there is no hierarchy among populations and no constraint on fitness transmission flow [10]. However, in our problem, there is a fitness tree where fitness values propagate in a bottom-up fashion (Fig. 2). Each node contains certain information.

The universe, which is the root of the tree, provides the molecules of programs, i.e. a table of machine instruction set. In the middle level there are programs. We generally need more than one program to evaluate the instruction set accurately; otherwise the instructions may be biased toward a single program and become too specific. For instance, if we want to evolve self-replicating programs, the programs should all contain the self-replication code, but possibly with different algorithms that can be used for this purpose.

Program nodes contain the information of its expected behavior (e.g. competing for CPU time and memory or doing a simple calculation). Instances are at the bottom of the tree and they hold different codes that realize the parent program. Initially all instances are the same (the code that is manually prepared by designer), but evolution may diversify them later. This is needed in order for the program population to explore newly created instructions (by mutation in instruction level).

Fitness evaluation is feasible only in instance level, by simulating the code and examining its behavior. Given a program, its winning instance indicates that that program with that instruction set can reach such a fitness (the code of the winning instance shows how to do this). In other words, even a good instance can label the population as good (it can be instance_1 or instance_2 or … instance_n). This is similar to the logical *OR* operation. When instead of binary variables, continuous ones are used, other equivalents of binary *OR* operation can be used. Our choice was *MAX* operator which is a popular equivalent for binary *OR* in Fuzzy Logic domain [15].

In Universe level, the story is reversed; we need all programs to work fine because they do different tasks (though may have similar behaviors). Even if one of the programs cannot work correctly, we should consider a low fitness for that universe (which contains instruction set). This is because we have given enough time for programs to evolve before evaluating instruction set. Therefore, the instruction set probably lacks required instructions for one of the program with low fitness, and the instruction set must be label as low fitness too. When we need all programs to work (program_1, and program_2 and … program_m), we can use logical *AND*. Complementarily, we use *MIN* operation as our variables are not binary but continuous.

This structure is completely different from Matsuzaki's work [9] where each program self-contained its instruction set. Since the universe holds only one instruction set, for efficiently evolving instructions, we consider a population of universes, each of which contains its own program instances.

4 Experments

4.1 *Simulator*

Although our ultimate goal is studying coevolution for emergence of digital life, this paper has focused on the general problem of coevolvability of instruction set and programs. As the first step, we used a simple computational framework, where programs were supposed to do certain calculations. In the future, we are to examine the same concept with self-replicating programs, which are obviously more complex than our current experimental case.

In computer architecture, each machine instruction should be mapped to a sequence of very low-level micro-operations that the actual hardware can execute. These micro-operations are generally simple operations on registers. A popular method for realizing this translation is called micro-programming [6], which is also the way we used for evolving instruction set.

In micro-programming technique, there is a memory called "control memory" that holds the information of micro-operations sequence. Each machine instruction is mapped to an address in control memory. Then control unit executes corresponding micro-operations and then this process is repeated for the next instruction.

Instruction codes, in our experiment, are represented by 4 bits that allow 16 instructions. We also assumed that each instruction at most requires 8 micro-operations. Therefore, the control memory has 128 cells. Obviously, instruction code is mapped to control memory address by multiplying the code by 8. In our simulated architecture, the core memory is 256 byte long and there are five 8-bit registers namely X, Y, Z, PC and SP. The first three registers are general purpose, but the last two ones correspond to Program Counter and Stack Pointer. There are 22 micro-operations that operate on these registers and each of them can be conditional or non-conditional as shown in Table 1. Conditional micro-operations execute depending on the zero or non-zero status of register X.

For evaluating a program instance, it is executed by the simulator. If it does not show the desired behavior (e.g. producing wrong answers for a computational problem), a high value will be assigned to its cost, otherwise its costs becomes equal to the number of micro-operations that it has executed. Counting the number of micro-operations instead of number instructions encourages programs to use light-weight instructions (which generally execute faster).

The execution of incorrect programs may trap the simulator in an infinite loop. Infinite loops are detected when the simulator does not reach *HALT* instruction after executing a certain number of instructions (100 in our case). If an infinite loop is detected, the simulator will automatically stops and assigns a high value to the cost of that program instance.

Eventually, cost values are converted to fitness so that a roulette wheel scheme can be used for selection. This conversion is achieved by finding the maximum cost in the population of that program, and subtracting it from all costs of that population.

Table 1 Available Micro-Operations

Micro Operation Code	Micro Operation Symbol
0	fetch
1	halt
2	X ← X + 1
3	X ← X – 1
4	X ← X + Y
5	X ← X – Y
6	X ← Y
7	Y ← X
8	Y ← Z
9	Z ← X
10	PC ← X
11	X ← PC
12	Z ← PC
13	Z ← SP
14	PC ← PC + 1
15	SP ← SP + 1
16	SP ← SP – 1
17	X ← MEM[Z]
18	MEM[Z] ← X
19	Shift_left X
20	Shift_right X
21	X ← 0

4.2 *Simulation Results*

Initially we manually designed 15 instructions based on the available micro-operations. These instructions are shown in Table 2. Two programs were used to compute the following algebraic expressions:

i) x := 4x+3

ii) x := 8x+1

Correctness of programs is checked by some test input/output pairs. We used three test cases for each of the programs: (0,1), (1,9), and (3,25) for expression (i) and (0,3), (1,7), and (3,15) for expression (ii). The cost of each instance is the sum of costs obtained by its execution with all test cases.

Ancestor programs were manually written. Since there was no multiplication instruction in ancestor universe, iterative addition was used in these programs. The initial code of the first program (that computes the first expression) is shown below. For better understanding, the micro-operations that compose each instruction are written as comments in its right hand side.

```
EXCH  X,Y ; Z←X, X←Y, Y←Z
MOV   X,1 ; Z←PC, PC←PC+1, X←MEM[Z]
ADD   X,Y ; X←X+Y
ADD   X,Y ; X←X+Y
ADD   X,Y ; X←X+Y
ADD   X,Y ; X←X+Y
HLT       ; halt
```

Table 2 Initial Instructions

Instruction Code	Assembly Symbol
0	ADD X,Y
1	SUB X,Y
2	HLT
3	NOP
4	JMP ADDR
5	CLR X
6	JNZ ADDR
7	INC X
8	DEC X
9	EXCH X,Y
10	MOV X,VAL
11	MOV [ADR],X
12	PUSH X
13	POP X
14	CALL ADDR
15	RET

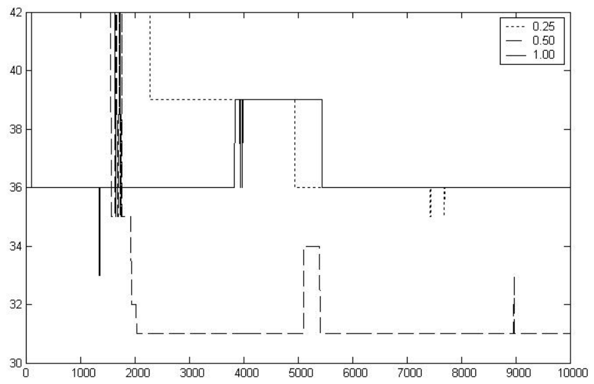

Figure 3 Cost versus Generation

In all simulations the instance population of each program contained 20 species. The universe population also had 20 universes. When copying bytes of codes from one generation to another, each byte was subject to mutation with the probability of 0.005. Similarly, the probability of universe mutations, which changes instructions, was considered to be 0.005. However, note that there are different time scales between universe and program evolution. Although program instances evolve continually, evolution of instruction set (evaluation, selection and finally mutation) occurs with a certain probability.

To study the effect of the probability for evolution of universe, we repeated the simulation with different values of this probability. Figure 3 shows the cost values versus program generation steps. It can be seen that when the temporal resolutions of universe and programs are not, i.e. probability of universe evolution differ, coevolution proceeds better. However, increasing the difference between these temporal resolutions too much degrades coevolution performance again. According to our simulation results, the best result is obtained when evolution of universe occurs half time slower than evolution of programs. Below is shown the code of the first program with the university probability of 0.5.

```
MOVNZ Y,X ; (x==TRUE) Y←X
MOVC  X,3 ; Z←PC, (x==T)PC←PC+1,
          ; X←MEM[Z]
HLT       ; halt
ADD   X,Y ; X←X+Y
ADD   X,Y ; X←X+Y
ADD   X,Y ; X←X+Y
ADD   X,Y ; X←X+Y
HLT       ; halt
```

Coevolution has invented two new instructions. From the effect of their micro-operations, we chose two assembly symbols for these instructions namely MONZ and MOVC. In addition, in the original instruction set that is shown in Table 2, HLT instruction's code is 2. However, coevolution found a better code for this instruction, it used 3. This means that instead of 8th cell of control memory, 16th cell contains the code of "halt" micro-operation now.

Now let's investigate what is the improvement. The changes are very intellectual and elegant. The main optimization occurs when X is equal to zero. In this case, micro-operations of MONZ are not executed. MOVC reads the next byte and puts it into X. Since X is zero, program counter is not increased, so the next byte will be executed as well. As the next byte contains HALT instruction whose code is 3, X becomes equal to 3 and when MOVC is finished, the next instruction is executed, that is HALT and X contain the correct result, which is 3. It did not enter to the repetitive additions part, it also optimized EXCH to MOVNZ for faster execution.

When X is not zero, the program acts similar to the original code. This is because MOVZ becomes equivalent to MOV. Note that MOV Y,X did not exist in original instruction set (Table 2). This is itself a novel instruction appeared during coevolution. Next, MOVC X,3 acts as MOV X,3 because X is non-zero and HLT is threatened as data, increment of PC bypasses execution of HLT. The rest of program is executed normally. So the gain in non zero cases is only creating the novel instruction MOV Y,X instead of EXCH X,Y.

Perhaps you think zero X is a special case that rarely happens and most of times the code is executed for non-zero values of X. By this view point, it seems the coevolution has achieved insignificant improvement. However, this is not true in algorithm's viewpoint, because it had used only three test cases, one of which was zero input. So considering what was given to the algorithm, its achievement is considerable. Perhaps if more test cases were given to the program, a different optimization would arise.

5 Conclusion And Future Works

A new trend in artificial life research has emerged which focuses on coevolutionary approach for creating life in silico. This is a new method comparing with previous works where programs had to be evolved within a fixed and ad-hoc machine instruction set. Ad-hoc instructions sets are brittle; they reduce the complexity of arisen digital life.

In this paper we postulated that naive coevolution of programs and instruction set can become problematic too. However, by considering different temporal resolutions for the two evolutionary processes, we showed that the performance of coevolution is improved. Our experiments indicate that choosing the rate of instruction set evolution half of the rate of programs evolution gives the best performance.

An important work that we want to pursue in the future is returning to the main motivation of this work, creating digital life. We plan to examine the effect of temporal resolution on the complexity and diversity of the emerged life in silico. We expect to achieve more evolvable system using this approach comparing with simple coevolution or bare evolution.

6 References

[1] Adami, C. and Brown, C.T, "Evolutionary learning in the 2D artificial life system avida", In Brooks, R.A. and Maes, P., editors, Artificial Life IV, pp. 377-381, Cambridge, MA, MIT Press, 1994.

[2] Kauffman, S.A. and Johnsen, "Coevolution to the edge of chaos: Coupled fitness, landscapes, poised states, and co-evolutionary avalanches", In C.G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen (Eds.), Artificial Life II, SFI Studies in the Sciences of Complexity, 10: 325-369. Addison-Wesley, 1991.

[3] DeJong, K. and Spears, W., "On The State of Evolutionary Computation", In Proceedings of the Fifth International Conference on Genetic Algorithms Mining, pp. 618-623, San Mateo, CA: Morgan Kaufmann, 1993.

[4] Dewdney, A. K., "In the game called core war hostile programs engage in a battle of bits", Scientific American, 250: 14-22, 1984.

[5] P.Dittrich, M. Wulf, and W. Banzhaf, "A vital two-dimensional assembler automaton", In C. C. Maley and E. Boudreau, eds., Artificial Life VII Workshop Proceedings, 2000.

[6] Mano, M.M., "Digital logic and computer design", Prentice-Hall, Inc., NJ, 1979.

[7] Matsuzaki, S., Suzuki, H., Osano, M. "Universal constructor to build a Tierran machine structure" In: Sugisaka, M., Tanaka, H. (eds.): Proceedings of the Eighth International Symposium on Artificial Life and Robotics (AROB 8th '03) Vol. 1 (2003) 259-262.

[8] Matsuzaki, S., Suzuki, H., Osano, M. "Digital creatures in a core replicated with microoperations" In: Proceedings of the Sixth International Conference on Humans and Computers (HC-2003) (2003) 188-193l

[9] Matsuzaki, S., Suzuki, H., Osano, M. "An approach to describe the Tierra instruction set using microoperations: the first result" In: Banzhaf, W., Christaller, T., Dittrich, P., Kim, J.T., Ziegler, J. (eds.): Advances in Artificial Life (7th European Conference on Artificial Life Proceedings), Springer-Verlag, Berlin (2003) 357-366.

[10] Potter, M.A. and DeJong, K, "Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents", Evolutionary Computation 8(1): pp. 1-29, 2000.

[11] Rasmussen, S., Knudsen, C., Feldberg, R., and Hindsholm, M., "The coreworld: Emergence and evolution of cooperative structures in a computational chemistery", Physica D, 42: 111-134, , 1984.

[12] S. Rasmussen, C. Knudsen, and R. Feldberg, "Dynamics of programmable matter", In C. G. Langton, C. Taylor, J. Doyne Farmer, and S. Rasmussen, eds. , Artificial Life II, pp. 211-291, Redwood City, CA, 1992, Addison-Wesley.

[13] Ray, T. S. "An approach to the synthesis of life", In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, Eds., Artificial Life II: proceedings of the second arifiticial life workshop, pp. 371-408, Redwood City, CA, Addison-Wesley.

[14] Suzuki, H., Ono, N., and Yuta, K., "Several necessary conditions for the evolution of complex forms of life in an artificial environment", Artificial Life V.9, N.2, pp. 537-558, 2003.

[15] Zadeh, L.A., "Fuzzy sets. Information and Control", Vol. 8, pp. 338-353, 1965.