Self-replication in a 2D von Neumann architecture

Florent de Dinechin
IRISA, campus de Beaulieu, Rennes, France
fdupont@irisa.fr

Abstract

This paper introduces new tools for the experimental study of the evolution of living systems. It borrows the main ideas developed by T. Ray in his evolution simulator Tierra, but addresses the main weakness of this system, the linear topology of its memory which is induced by *addressing by template*. We define an execution model in a *two dimensional* (2D) memory, in which sequential programs are stored on "threads" in the memory plane, and *jump* instruction are replaced with physical thread connection. We present an implementation of this 2D execution model and give examples of programs, including a self-replicating one. Then the use of this model for simulating evolution is discussed.

1 Introduction

In trying to define a science of life which is more universal than what we may observe on Earth, fundamental biologists rely more and more on abstract models and computer simulations. Such models, often termed *artificial life* [4], may address several research fields. The first is the study of the fundamental laws of *metabolism,* i.e. the energetic aspects of living systems [5]. The second one is to study the *emergence* of macroscopic behaviors from the interaction of microscopic entities [10]. A third aim of artificial life, which motivates this paper, is a better understanding of the *evolution* process: its purpose is to provide an artificial framework allowing us to carry out *experiments* about evolution, instead of the mere *observation* of our own terrestrial evolution process. Such experiments are needed to abstract the universal laws of evolution from the casual contingencies of the history of the earth.

The first steps in this direction were the cellular automata (CA) of von Neumann [14], who only studied self-replication. Among other, Langton [3] and Sipper [11] also addressed the question of evolution. Meanwhile Rasmussen et al. [7, 6] showed that a completely different model, inspired from computer architecture, could also be used in this domain. These ideas inspired T. Ray's Tierra simulator [8] providing the most spectacular simulations of evolution to date. In this model, the world is a computer memory, and the living beings are self-replicating computer programs, subject to mutation and death mechanisms inspired from biological life. Experiments using this system sometimes lead to the effective evolution of initial self-replicating programs towards more and more efficient and more and more complex ones. This simulator was also used and extended to study adaptation and learning in such "living" systems [1].

However we see weaknesses in Tierra as a tool for the study of artificial life. The first is the poor topology of the substrate – a linear memory – which prevents the apparition of complex interactions between more than two neighboring creatures. This topology is a consequence of one of the key features of the Tierra virtual computer, the fact that memory locations are addressed by their content *(by template)* instead of their address like in conventional computers. As a consequence, a *centralized* operating system is needed for memory allocation, to overcome this poor topology. Although it obviously doesn't prevent evolution from occurring, it is very different from what we know of Earth life, where all the interactions are local in a three dimensional space. As this question of locality is crucial in the study of self-organizing behavior, we fear that Tierra may be unable to simulate one of the major events in terrestrial evolution, the apparition of *multicellularity*.

A richer topology allowing one to truly exploit locality is thus needed. *Two dimensional* CA are much older than Tierra itself, however they are badly suited to the study of evolution: they are too *brittle*, which means that a small change in a "living" (able to self-replicate) organization has a very low probability of preserving this ability. The purpose of this paper is therefore to provide a bridge between cellular automata and the Tierra model, inheriting the topology of the first and the resistance to mutation of the latter.

This question has already been addressed, e.g. in the Computer Zoo [12] and Avida [2] systems. Both, however, separate a 1D space where the instructions are stored (à la von Neumann) and execute themselves, and a 2D "physical" space. Our system is both simpler and closer to the real world, as there is only one space where the programs both run and interact. This space is a memory shared by all the programs, as in Tierra, but this memory has a 2D topology. The von Neumann ex-

ecution model in 2D memory which we introduce is of little practical use in computer science, but we feel it addresses a need in the field of artificial life.

The remainder of this paper is organized as follows: the following section discusses the topology of memory accesses in conventional computers and in the Tierra virtual computer, then introduces the notion of 2D memory. Section 3 describes a virtual processor designed on top of this notion, with some example programs. Then section 4 discusses the use of this virtual processor to study the evolution of programs "living" in this 2D memory. Finally we draw conclusions from these initial experiments.

2 Memory topology

2.1 The von Neumann model

In computer science, the fact that the memory is monodimensional, i.e. that its address space is linear, has been a constant since the the very beginning: the Turing machine, an abstract model used to study the very foundations of computing, is based on a linear ribbon on which the data are written. The other example is the architectural model on which most general-purpose computers have been based, which is also due to von Neumann [13]. It consists basically of a processor communicating with a memory. A memory location is accessed by its address which is an integer. There are two operations possible on this memory: store a data at a given address, or read the data at a given address. We say that the memory is monodimensional because its address space (the set of the integers) is a monodimensional Euclidean space.

The processor contains a register usually called *program counter* or PC, and indefinitely executes the same cycle: read the program instruction stored at the address held by PC, decode it, execute it, add 1 to PC, and start again. The set of possible instructions may be very simple or very complex, but it always contains some instructions to read and write data at a given address in memory.

Now to define a topology of the memory we have to define a notion of *distance* between two memory locations. From a computational point of view, the relevant distance is is not the distance between their addresses, but rather the time it takes to access a memory location from a read or write instruction stored in another memory location. In the von Neumann model, this access time to a data is independent from both memory locations[1]. Therefore there is no need for more complex address spaces: if the logical topology of the memory is linear, the practical topology is such that each memory location is a direct neighbor to each other in terms of access time. This is how the linear memory may be interpreted for example as a two dimensional picture, without any complexity overhead.

[1]In current actual computers this is no longer true: there is a hierarchy in the memory access times which exploits a notion of locality. This doesn't affect our argument.

2.2 The Tierra virtual computer is not a von Neumann computer

Ray himself made a similar analysis [9], but this analysis doesn't hold for his own work: in the simplified computer model used in the Tierra simulator, there is no absolute address space. During memory operations (and also `jmp` instructions) the memory location to access isn't defined by its *address*, but by its *content* (called in this case a *template*). A local search must be performed, from the PC location, to find this template. Thus read/write/jump topology is actually 1D. Note that, in compensation, the instruction `mal` allocating the memory for a daughter program has no such restriction: the daughter is allocated anywhere in the soup. Thus mother/daughter topology is unrestricted, and therefore inconsistent with the read/write/jump topology.

In the 2D systems designed so far (to our knowledge Czoo [12] and Avida [2]) this inconsistency is even worse: read/write/jump operations happen in some 1D program space, while cell interactions happen partly in this program space, partly in some 2D "physical" space which is distinct from the previous. This works as far as evolution is concerned, and actually it is somehow similar to the 1D DNA program in the 3D living cells of the biological world. We intend, however, to simulate the most basic level of "chemistry": cells and cellularity may appear in the run of the simulation but we don't want to impose them by an external controller.

Our motivation is therefore to reconcile the program space and the physical space (as in cellular automata and real world chemistry) and build a model in which reads, writes, jumps, as well as daughter creation and other cell-cell interaction, obey the same Euclidean 2D topology. This unified space will be termed a *2D memory*.

2.3 Two dimensional memory

The definition of a 2D memory is very straightforward: the address of a memory location is a couple of integers instead of a single integer. This couple may be viewed as the coordinates of the location in a plane.

It is more difficult to define what a *program* is in such 2D memory: in the von Neumann model, a program is a set of consecutive memory locations holding instructions. The order of execution of the instructions in time reflects their order in the linear memory (only special `jump`/`goto`/`branch` instructions break this order). The execution of a program in 2D memory raises a new question: which, of the 4 neighbors of an instruction, is the instruction to be performed next? In other terms, how do we map the (linear) time onto our bidimensional memory?

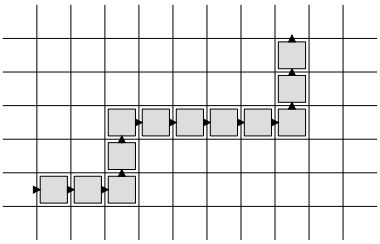

Figure 1: A program thread in 2D memory

We have studied two possible approaches. The first is to store, in each memory location and in addition to its data, an arrow pointing to the "next" memory location, which is one of its neighbours: up, right, down or left for instance. A program is then a sequence of instructions according to this succession relationship: when one looks at the memory, a program appears as a thread in this memory (Fig.1). The interesting thing is that the notion of *loops*, implemented in linear memory using a sequence of instruction including a *jump* instruction, actually appears as a loop in 2D. There is no need for a jump instruction, the thread simply drives to itself. Moreover, what programmers call "loop nests" actually appears as nests (Fig.2). In this figure, the instructions labelled `F` are *fork* instructions, which have two possible successors depending on the state of a flag.

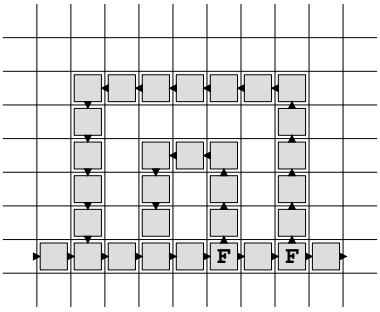

Figure 2: A loop nest in 2D memory

What we implemented – and will present in the remainder of this paper – is a slightly more complex model, such that the programs are *invariant by rotation*. To ensure this property, no absolute orientation is stored in the memory. Instead, the PC (within the processor) holds this absolute direction along with the address of the location it points. The memory locations only hold a *relative change* to this direction for the next instruction: keep the direction (`F` for "forward"), turn right (`R`), or turn left (`L`). The behavior of the PC is thus similar to the head of a Logo turtle, as shown by Fig 3.

To clarify things, here is the basic cycle corresponding to the von Neumann cycle: *read* the instruction pointed to by the PC, *execute* this instruction, *move the PC* to the next instruction (which is given by the PC orientation), *rotate* the PC according to the direction change (which is contained in this instruction), and start again. Figure 3 shows four of these basic cycles. The arrow is the PC orientation, and the instructions are not given, only their direction change. Notice that the head (the PC) first moves, then (possibly) rotates.

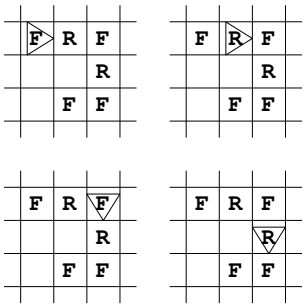

Figure 3: Execution of a 2D program

It is obvious that this execution model has the same expressive power as usual sequential computers: one may easily translate a sequential program into our model, by copying all the non-*jump* instructions on a thread and implementing the *jump* instructions as parallel threads of *no operation* instructions. Our first example will illustrate that.

We now present our implementation of this execution model, in a simple virtual processor whith a very restricted instruction set.

3 Ziemia

Our implementation is aimed at simulating evolution more than performing general-purpose computations. We assume that the reader is familiar with the Tierra virtual computer [8], from which we tried to keep the main features, in particular the small number of instructions. Another feature we borrowed from Tierra is the name, as Ziemia means "earth" in Polish.

One important difference between Ziemia and Tierra is that in our virtual computer, the only data manipulated are instructions: the Tierra language allows numerical computations, that is manipulating data (numbers) which are not present in the soup by themselves. Our model is closer to biology (if one admits that the only data manipulated by the DNA is amino-acids), but our main motivation was to keep the instruction set as small as possible. Performing computations is of course still possible, as our first example will show.

3.1 The Ziemia virtual processor

A virtual processor consists of three address registers called P (the PC), X and Y (each holding an address and a direction), two data register called A and B (each holding an instruction and a direction change), and a flag

used with some conditional instructions. In addition, the processor possesses two stacks, one for data and one for addresses.

A memory location holds a byte, in which two bits code the direction change, one bit is used for memory management, and five code the instructions. There are therefore at most 32 different instructions, out of which 6 are currently unused.

The following array briefly describes the current instruction set:

Np0	No Operation 0
Np1	No Operation 1
LdA	Load A: A ← (P)
A=B	A ← B
X=P	X ← P
X=Y	X ← Y
SXY	Swap X and Y
SAB	Swap A and B
RdA	Read A: A ← (X); flag set if (X) was Frk
WrA	Write A: (X) ← A
Flw	X follows the arrow it points
PsX	Push X on address stack; if full, set flag
PsA	Push A on instr. stack; if full, set flag
PpX	Pop X from addr stack; if empty, set flag
PpA	Pop A from instr. stack; if empty, set flag
APp	Discard top of addr. stack; if empty, set flag
IPp	Discard top of instr. stack; if empty, set flag
A?B	Set flag if A=B
X?Y	Set flag if X=Y
Run	Creates a new process whose P ← X
New	X = random addr. in the neighborhood of P
Frk	Fork: if flag set, move as usual, otherwise go forward
MvF	Move X forward
MvR	Move X right
MvB	Move X backward
MvL	Move X left

3.2 Implementation and example

We implemented a simulator for this virtual computer. We also had to write a 2D program editor which is very different from a text editor used to edit usual programs. This editor integrates a step-by-step debugger, which we used to write and test simple programs such as that of Fig.4. This figure is a screen dump of this editor: at each memory location it shows the instruction and the direction change stored there.

The program of Fig.4 performs the sum of two integers. It implements the usual binary addition, using the stack to store the carry bit. The threads composing this program are clearly visible. It is entered at the upmost `LdA` instruction (surrounded by two upwards arrows). The address register X must point to the least significant bit

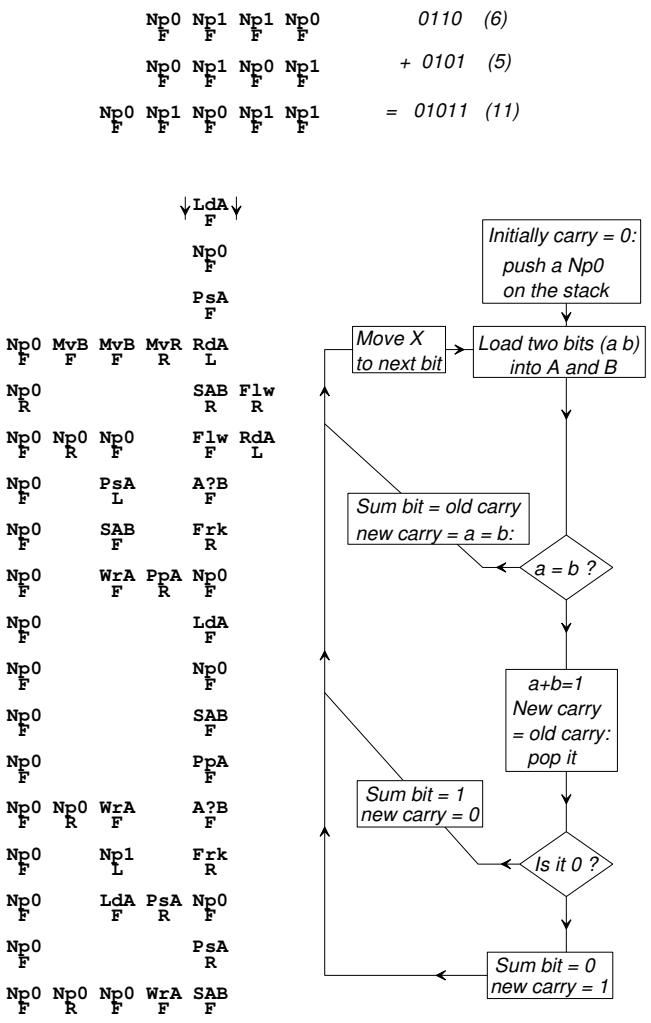

Figure 4: Binary addition in Ziemia

of one of the numbers (written in binary using `Np0` and `Np1`) as on the figure. The execution of this program mimics the flow chart given in the same figure. The interested reader should refer to Fig. 3 to step through the program.

4 Simulating evolution with Ziemia?

We programmed a complete system in the philosophy of the Tierra simulator: several processors share the memory and execute different programs. The operating system shares the simulation time between the processors, and maintains a count of the errors they make during their execution. Using this data, it removes the less successful processors from the simulation, leaving their code. In addition, several types of mutations may be applied to the system. We only implemented "cosmic ray" mutations, that is low-frequency random bit-flipping in the

memory[2].

4.1 Self-replication

We injected, in an initially blank memory, an ancestor which is the self-replicating program given in Fig.5. It consists of several threads (a thread starts with a `Frk` instruction), some of them form loops. The whole program loops on itself, starting bottom left (the boxed `X=P` instruction) with a smaller loop which copies a thread from the mother program (pointed to by register X) to the daughter program (pointed to by register Y). Each time this copy loop encounters a `Frk` instruction, it pushes on the address stack the corresponding location so as to come back to it later when it has finished copying the current thread. In this program we chose to end each thread with a `Np1` instruction: the copy loop thus stops when it encounters this instruction, and pops the next thread to copy from the address stack. If this stack is empty the copy is terminated, and a `Run` instruction is performed on the daughter program, creating a new processor for it.

In our experiments, a growing population of this ancestor program is easily observed, but to date no significant evolution occurred. Depending on the scheme by which programs are killed when the memory is saturated, it is possible to have a stable population of this program which resists mutations up to a certain rate. However, mutations do not seem to create new "living" programs: we observe new program loops with a sometimes complex behaviour (as the 2D memory is mapped on a computer screen we observe the apparition of various new repetitive patterns) but these mutants, being unable to self-replicate, are eventually killed by the system. Sometimes they invade the memory and make life impossible even for the ancestor, resulting in the extinction of the simulation, a phenomenon already described by Ray.

Very few experiments have been carried out yet, but they have shown that our system is too *brittle* to support an evolution process. The rest of this paper studies this brittleness issue.

[2]In Tierra experiments, other kinds of mutations (such as copy flaws) change little to the global evolution. We assume for a start the generality of this result.

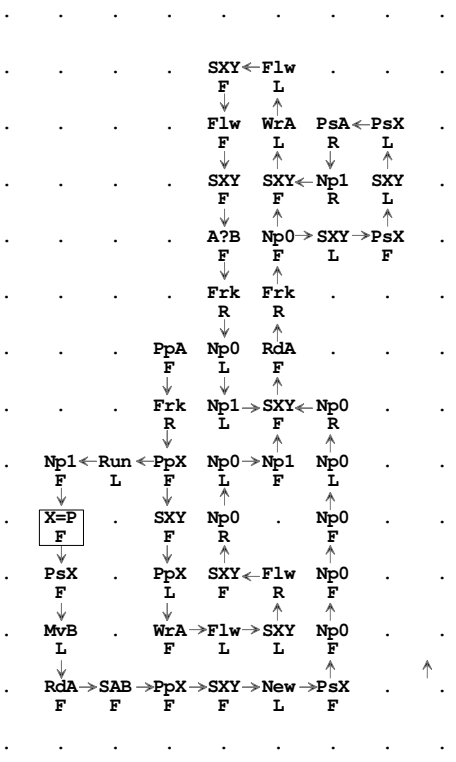

Figure 5: Self-replicating program

4.2 Addressing by template versus physical branching

A major difference between Tierra and Ziemia is the absence of addressing by template, replaced, as we already showed, with physical branching (see Fig.4 and 5). Addressing by template – a feature of Tierra borrowed from biology – is actually the basis of the evolution in the Tierra and Avida systems: the first significant mutations leading to new self-replicating creatures are modifications of the templates. For example, parasites, whose program don't contain any copy loop, are obtained by modifying the template structure of the Tierra ancestor to spare this copy loop.

We believe that physical branching could play the same role as addressing by template: in a memory saturated with dead code, a wandering PC due to some mutation has a high probability of encountering an "interesting" program thread, just the same way as in Tierra a mutated `jmp` instruction has a high probability of encountering the mutated template somewhere in the soup. In a 2D topology, however, this probability is (very roughly) squared, which increases brittleness. Besides, the probability that this PC comes back to its initial thread is very low, much lower than in the 1D case. This is the main reason why a mutant doesn't live.

We are considering various possible answers to this problem. The idea is to tie somehow a wandering PC to its initial program loop, for example by extending the instruction set with a *fork and push* instruction to be used in conjunction with a *return* instruction, in a way similar to the sequential subroutine `call` of Tierra. Another possibility is to introduce copy flaw mutations that preserve the threads.

4.3 Cellularity

The other fundamental difference between Ziemia and Tierra is the existence, in Tierra, of a *private* memory

space for each program, a segment of memory which is readable by all but on which only itself has the right to write. This private space is compared, in the Tierra metaphor, to the inside of a living cell protected by a semi-permeable membrane.

The drawback of this private space option is to rely on a centralized management system. Besides it is very difficult, for topological reasons, to design an equivalent in the 2D case. We therefore used a simpler scheme: each memory location contains one bit telling whether it is "alive" or "dead". All the processes have the right to write on "dead" locations, and none has the right to write to "alive" locations. A memory location is set "alive" each time it is accessed for reading or writing. The system periodically and randomly sets blocks of memory "dead". Thus, as long as a process is running, its code is kept "alive". Once the process is removed, its code remains flagged "alive" for a while, but is eventually set "dead" by the system.

The strong point of this approach is that there is no *centralized* cellularity: read/write access right is determined locally (another approach, more memory expensive, is to store at each memory location an identity number of the only processor which has the right to write there). To achieve our goals we will try and avoid a centralized cellularity mechanism, although we feel it is one of the key features making evolution possible in Tierra.

5 Conclusions

The work presented in this paper has two clearly distinct aspects. The first, concerning the domain of computer science, is the definition of a sequential execution model in bidimensional memory, validated by an implementation which allowed us to write several programs in this model. This work is interesting in itself, mostly because the absence of *jump* instruction makes structured programming mandatory and spatially explicit (see the loop nest and adder examples). This would be enough to try and use it as a programming model for parallel processing, but alas, to our knowledge, it is totally unrealistic from a technological point of view.

The other aspect is more specifically the use of this model to simulate evolution process in a world where, like in the real one, the physical space and the functional space are interdependent, and the interactions are local. So far we were able to exhibit a self-replicating program which is slightly more complicated than Tierra's ancestor, but much simpler – and hopefully less brittle – than a self-replicating 2D cellular automata. However evolvability is unsuccessful so far: preserving a high-level property such as self-replication is a known difficult problem. We have identified weaknesses of our model and proposed some solutions which remain to be explored. We are confident in the model : the 2D topology in itself shouldn't be an obstacle to evolution, since biological life evolved in a 3-D world, and Tierra in a 1-D one. Trying to solve the current problems will help us learn more about the underlying mechanisms and conditions of evolution.

References

[1] Chris Adami. Learning and complexity in genetic auto-adaptive systems. *Physica D*, 80:154, 1995.

[2] Chris Adami and C. Titus Brown. Evolutionary learning in the 2D artificial life system 'Avida'. In *Artificial Life IV*. MIT Press, July 1994.

[3] C.G. Langton. Self-reproduction in cellular automata. *Physica D*, 10(1-2):135–144, 1984.

[4] C.G. Langton and K. Kelley. Toward artificial life. *Whole Earth Review*, 58:74–79, 1988.

[5] F. Morán, A. Moreno, E. Minch, and F. Montero. Further steps towards a realistic description of the essence of life. In *Nara Fifth International Conference on Artificial Life*, May 1996.

[6] S. Rasmussen, C. Knudsen, and R. Feldberg. Dynamics of programmable matter. In *Artificial Life II*, Redwood City, 1992. Addison Wesley.

[7] S. Rasmussen, C. Knudsen, R. Feldberg, and M. Hindsholm. The Coreworld: emergence and evolution of cooperative structures in a computational chemistry. *Physica D*, 42:111–134, 1996.

[8] T.S Ray. An approach to the synthesis of life. In *Artificial Life II*, Redwood City, 1992. Addison Wesley. See also : *Evolution and Optimization of Digital Organisms,* included in the freeware Tierra distribution.

[9] T.S Ray. Artificial life. In W. Gilbert and G. T. Valentini, editors, *From Atoms To Mind*. In press, 1997.

[10] C.W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics:* Proceedings of SIGGRAPH '87, 21(4):25–34, July 1987.

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

[12] Jakob Skipper. The Computer Zoo – evolution in a box. In *European Conference on Artificial Life*, 1991.

[13] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.

[14] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, 1966.