Evolvability of the Genotype-Phenotype Relation in Populations of Self-Replicating Digital Organisms in a Tierra-like System

Attila Egri-Nagy and Chrystopher L. Nehaniv

University of Hertfordshire
Department of Computer Science, Faculty of Engineering and Information Sciences,
College Lane, Hatfield, Hertfordshire AL10 9AB, United Kingdom
{A.Nagy, C.L.Nehaniv}@herts.ac.uk

Abstract. In other Tierra-like systems the genotype is a sequence of instructions and the phenotype is the corresponding executed algorithm. This way the genotype-phenotype mapping is constrained by the structure of a creature's processor, and this structure was fixed for an evolutionary scenario in previous systems. Our approach here is to put the mapping under evolutionary control. We use a universal processor (analogous to a universal Turing-machine) and put the structural description of the creature's processor as well as the instruction set of the actual processor into the organism's genome. The life-cycle of an organism begins with building its actual processor, then the organism can start executing instructions in the rest of its genome with the newly built processor. Since the definitions of the processors and instruction sets are in the genome, they are subject to mutations and heritable variation enabling their evolution. In this work we investigate the evolutionary development of the processor structures. In evolving populations, changes in the components (registers, stacks, queues), variations in instruction-set size and the redefinition of the instructions can be observed during experiments.

1 Introduction

There is a trend in artificial life research that researchers try to put more freedom in their models. One of the big leaps was to implement ecological natural selection instead of artificial selection (i.e. optimizing an objective function). Using artificial selection in genetic algorithms may yield powerful optimization methods for very different problems, but the evolution of the system does not resemble biological evolution. In the best case the system converges to a solution satisfying externally imposed criteria. The evolutionary development is not open-ended. However, in Tierra-like systems [11, 1, 5, 2], digital organisms self-replicate *in silico*, and there is no global aim to reach; those entities which do not survive and replicate fast enough simply vanish from the population. But still the possible results are highly determined by the construction of the system: the evolved programs for example are written in the same assembly language as their progenitor. This language was designed by a human, not evolved. In this

paper we present a further improvement of the original idea of digital evolution. We give freedom to evolving populations of digital organisms (here called *archeans*) to change their 'genetic language' namely their instruction sets and the structures of their processors. This is done by putting the description of the processor and instruction set into the genome itself, thus the 'semantics' of the program is in the program itself (cf. [15]), and hence under evolutionary control.

2 Evolvability and Universal Processors

We handle the heritable information in digital organisms – a sequence of executable instructions or data cells or more simply a sequence of integers – as strings. Moreover descriptions of their processors and the instruction sets that run on them are parts of these strings. Variation in these structures thus becomes heritable. This results in new kinds of heritable variability and augments evolvability of the genotype-phenotype mapping [8, 17, 13, 6].

Among processors smooth evolvability means that in most cases we can apply transformations (delete, insert, replace) on these sequences without drastically changing their semantics with the hope that some of these minor changes will be favored by natural selection. This property does not appear in real processors where just a simple bit-flipping can cause the abnormal termination of execution. Therefore the design principles for processors will be different from the current industrial standards when the main aim is to enable open-ended evolution of machine-code programs driven by natural selection.

What are these design principles? By what means can we discover them? Should we try out various processors and check how they perform under evolution [10]? There are already significant achievements implemented in the previous systems (Tierra [11], Avida [1], Primordial Soup [5]). The most prominent examples are as:

- Labels and pattern-matching can be used instead of hardcoded addresses in transfer of flow of control, subroutine calls, etc. [11].
- Fault tolerance: there is no combination of instructions which makes the processor crash. Even in the worst cases violations result in nothing happening (like performing a `nop` operation) [11].
- Using different instruction sets or subsets of a bigger and redundant instruction set in order to test their evolvability [12, 1].

In these previous works of using a processor with fixed structure it simply executes the instructions and during execution a copy of the executed program is made. This is the replication process. Of course different fixed processors have been used but not in the same evolutionary run [12, 10]. One of our original goals of designing `Physis`, yet another digital evolution system, was to provide possibilities to implement several processor architectures (including Tierra, Avida, Primordial Soup) within the same experimental tool ensuring common metrics for evaluation [2]. But this has been superseded by the general idea of using evolvable architectures, so our approach here is now radically different. The idea

is if we don't know how to construct a processor which supports evolvability of self-replicating programs then we should *let it be evolved!* If we provide freedom for the evolutionary process in tinkering with the processor structure, evolvable processors may arise in the course of evolution. The guiding principle behind this: the evolutionary potential of a processor itself is an important property at lineage level and thus may be favoured by natural selection.

According to this we can outline a 'meta' design principle suggesting dynamic structures for processors. The internal structure (registers, stacks, queues and their sizes) and the instruction set may be varied over evolutionary time between different creatures in the population.

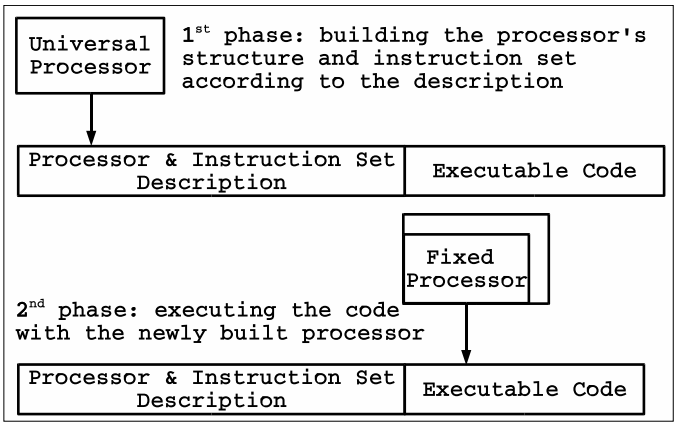

Fig. 1. Execution phases of the universal processor.

In the case of the universal processor the execution consists of 2 phases:

1. First the structure of the processor is built according to a description contained in the code of the digital organism itself. This structure remains constant for the whole lifespan of the organism.
2. Then the actual code is executed on the newly built processor. During replication the processor description is also copied and thus it is subject to mutation. So it may be changed.

The underlying theoretical construction is the universal Turing-machine [16]. Something similar happened in the hardware industry by introducing the `Crusoe` processors [9].

This idea is also motivated by biological analogies. In real life the decoding system, the mechanisms inside the cell, such as genetic code controlling transcription, translation, protein biosynthesis and genetic regulatory control (see, e.g. [13, Ch. 6]) evolved together with the genetic information, with DNA. How the genetic code for protein biosynthesis evolved is one of the biggest questions in today's evolutionary biology. *In silico* experiments may shed light on this mystery.

3 Processor Description Language

Obviously we need a genetic language to describe the inner structure of the processor and its instruction set. As mentioned before we treat a digital organism as a string of integers comprising a circular genome (as in prokaryotes or Avida). Unlike in other Tierra-like systems, the context of decoding determines whether an integer is an executable instruction or a component of the structural description: the processor treats an integer as an instruction in the instruction fetch phase while the same integer references a structural element (such as a register identifier) if an operand is needed. This is accomplished by using not directly integers $\mathbb{Z}$ but $\mathbb{Z}_m$, integers modulo m, where m is the number of components of the required type. This mapping of integers into either structural elements or instruction identifiers or positions in the genome constitutes an abstract analogue of DNA to RNA transcription (often said to be lacking in such systems, cf. [4]).

3.1 Genetic Description of the Processor Structure

Part of an archean's genome specifies its processor architecture. There are 3 basic building blocks – we call them structural elements or primitives representing basic data structures in the organism's specification of the processor that its code will run on. Each structural primitive has a corresponding symbol and we need a special mark `B` for separating structural elements:

`R` register
`S` stack
`Q` queue
`B` blank

For example, a `RRSSSBSS` string in the genome represents two registers and two stacks with sizes of 3 and 2. (A stack or queue of size 1 acts as a register, so it would be an equivalent possible choice to use `SB` instead of `R`.) For every structural element (stack, register or queue) a unique integer is assigned, by which it can then be referenced by executable instructions in the archean's genome. The structural element referenced by a simple zero is the instruction pointer by default.

3.2 Genetic Description of the Instruction Set

The instruction set of an archean is defined by using the universal processor's *basic instruction set*. Those instructions can be used as the smallest building blocks of a newly defined instructions. A digital organism cannot execute basic instructions directly but executes instructions defined in the genome itself with the syntax below. The descriptive part of the creature's genome is separated from the executable part by a special `SEPARATOR` instruction which marks where execution should begin. Instruction definition in an archean's genome specifies instructions that are comprised of basic instructions of the universal processor.

The basic instruction set was designed to fulfill the following requirements:

– each instruction should be as simple as possible:
  • an instruction as elementary building block represents a single action not a compound one
  • an instruction is independent from any addressing mode
– according to RISC philosophy only dedicated `load/store` instructions can access the memory [14]
– the instruction set should be complete (any more complex operation should be easily definable)

The basic instructions can be categorized in the following way (see [2] for more details and exact specifications):

Data transfer There are 3 types of data transfer: between structural elements, between a structural element and the environment, between structural elements and memory.
`in` reads data from environment,
`out` writes data into the environment,
`load` copies from memory to a structural element,
`store` copies from a structural elements to memory,
`move` general move between structural elements.

Control-flow Two instructions are enough to build any control structure.
`jump` unconditional jump,
`ifzero` skips the next instruction if operand not zero.

Arithmetic and Logic These are the usual instructions for mathematical and logical operations. The operands are structural elements and the result is stored in a structural element. These can be mixed arbitrarily. A defined instruction can even use one structural elements for operands and results, if it is for example a stack.
`compare, add, sub, neg, div, mod, shift-l, shift-r, and, or, xor, not, is_sep ...`

Biological These instructions are required for replication.
`cinc,cdec` cyclic increment/decrement for circular genomes,
`allocate` allocates memory,
`divide` splits the child and the parent program.

In order to enable inter-organism communication or parallelly executing organisms this basic instruction set could be extended by the appropriate additional basic operations.

The description of an instruction begins with an `I` symbol after which comes the code of the newly defined instruction – just like defining a subroutine in a higher level programming language. This technique enables evolution of modularization. The semantics of the instruction is defined by a microprogram written in the assembly language of the universal processor[1]. In short:

```
I basic_instr [operand...] [basic_instr [operand...]]...
```

[1] It can be said that a CISC processor is defined on a RISC architecture [14].

For example `I move 0 1` is an instruction that copies the content of the instruction pointer to the first structural element. The operands point to structural elements which are indexed by a nonnegative number (modulo the total number of structural elements). Also a number uniquely identifies an instruction (modulo the size of instruction set). Again it depends only on the context of execution whether an integer denotes a structural element or executable instruction or simply a data cell.

4 The Original Replicators

For starting the evolutionary process we need an original replicator. For the time being this should be written by a human. As such it consists of 3 distinct parts. (Fig. 2). In spite of this clear structure this organism is far from being optimal in terms of replication speed.

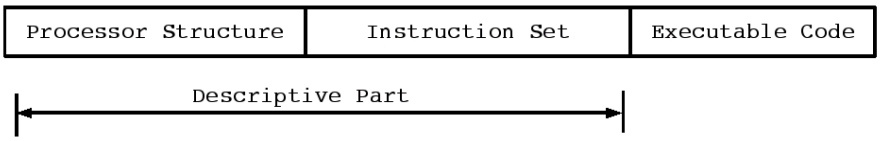

Fig. 2. The structure of an original archean replicator. The lengths of the different sections are not proportional to the number of cells. (length of structure description 8, definition of the instructionset 52, executable code for self-replication 18 instructions)

Two original replicators were designed having different processor architectures. One uses 4 registers (Fig. 3), and the other uses 2 registers and a stack of size 2. The instruction set reflects the difference but otherwise both have the same replication algorithm (count the instructions cyclically forward to the `SEPARATOR` in one loop, allocate memory, copy the instructions in another loop, divide).

5 Experimental Set-ups and Results

We are interested in changes of the processor structures over evolutionary time and in checking whether universal processors have at least as much evolutionary potential as their fixed counterparts or not. To assess the evolutionary potential of our system, we implemented two scenarios. In the first, no external or environmental rewards were provided and the archeans evolved only subject to the constraints of their biotic environment. In the second, archeans were rewarded with additional clock cycles for exploiting external environmental resources by performing certain operations.

R	I	I	I	0
B	move	load	load	1
R	0	2	1	2
B	2	4	4	3
R	I	is_sep	I	4
B	clear	4	rel-store	5
R	1	4	1	6
	I	ifzero	2	3
	move	4	4	7
	0	I	I	8
	3	jump 3	dec	2
	I	I	1	11
	inc	allocate	I	9
	1	1	ifnotzero	10
	I	I	1	12
	cinc	move	I	6
	2	1	divide	13
		2	SEPARATOR	

Fig. 3. The genome of the original archean replicator (read top-down, left-right). The first column is the structrual description (4 registers), the last column is the archean's actual executable code for self-replication and the three columns in the middle constitute the definition of the instruction set. For example the number 13 in the executable part refers to the last defined instruction containing only the `divide` basic instruction.

We carried out 5 different experiments for each setup. The maximum population size was 20000, the system runs for 200000 update cycles. This results in roughly a few thousands of generations of archeans.

We use an observational *fitness* f for measuring (but not guiding!) the evolutionary performance, defined as in Avida [3]:

$$f = \frac{m}{\gamma} \tag{1}$$

where m is merit, defined as the organism's effective length (the number of executed instructions in the genome) possibly multiplied by the bonus values earned by performing computational tasks, and γ is the gestation time (i.e. the number of processor cycles needed for self-replication).

5.1 The Simple Scenario

In this case fast replication is rewarded via natural selection: each organism receives constant size timeslice of 11 cycles and no computational task is rewarded. Evidently in the presence of these constraints the organisms try to decrease their gestation time either by shortening the genome or optimizing the replication algorithm. Most of the evolved organisms have less than half of the original replication time (Fig. 4). The handwritten progenitor is inefficient: it

executes extra instructions when measuring its size by scanning its genome from `SEPARATOR` cyclically, but the evolutionary process was able to improve this design in each run by one of the following mechanisms: instead of measuring its size in a loop the archean starts to execute the descriptive part of the genome resulting in faster determination of size, or it just produces its size by using the `cdec` instruction, which is decrementation modulo the length of the genome.

Organism	Genome length	Gestation times (1st & 2nd offspring)	Effective length
4regs original	78	857, 857	17
evolved	76 ± 3.7	498 ± 84.6, 414 ± 52.2	59.2 ± 26.2
2regs1stack	81	1369, 1369	17
evolved	80.4 ± 2.3	732.2 ± 43.7, 662.2 ± 34.9	49.6 ± 20.8

Fig. 4. Comparing the two different ancestral archeans and their evolved descendents.

As general trends the instruction set usually gets shorter by losing one or two instructions (but retains its size by defining empty instructions), and the use of operands read from the genome instead of hardcoded in the instruction set appears. The structure of the processors remains generally constant in each run although variants appear but never dominate in the population.

5.2 The Evolutionary Learning Scenario

In this case we would like to test whether universal processors have some drawbacks beyond being slower as compared to static processors. This scenario uses a task-handler subsystem, like in Avida [3], to model the exploitation of external environmental resources (other than time and space). In addition to replicating themselves digital organisms can get longer execution time by performing simple calculations i.e. reading integers from the environment, transforming and writing the result back. All activities of the organisms are monitored. The average timeslice given to an archean for one update is 11 processor cycles.

In this case changing the instruction set is more vital, as the instructions for communicating with the environment are not defined in the progenitor. The execution of the descriptive part had to evolve to provide the instructions for I/O operations necessary to perform the tasks.

The results of these experiments support the hypothesis that digital organisms with universal processors are able to learn the same set of computational tasks as the ones with fixed processors. There were no observed signs of the different evolutionary potential so far.

Fig. 5 shows the dynamics of the maximum and the average fitness through time. Leaps indicate the evolutionary innovations (i.e. the learning of some tasks). The evolutionary learning process shows the dynamics described by the theory of punctuated equilibrium [7].

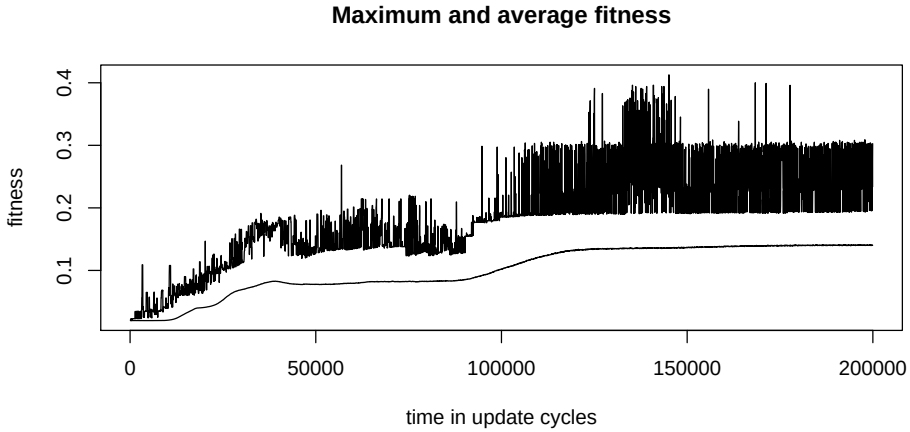

Fig. 5. The dynamics of average (below) and maximum fitness (above).

5.3 Evolution of Instruction Set and Processor Structure

The observations in terms of the evolution of processor structure can be summarized in two categories. One is for those attributes which did not change:

- Evolution is conservative in changing the processor's inner structure. Although there were several variants (e.g. with fewer or more registers, stacks or queues instead of a register, etc.), they could not gain domination over the population. The ratio of the original processor structure did not fall below 90% in the course of evolution observed.
- The number of defined instructions was also preserved during the evolutionary process similarly to the processor's structure. The rigidity of the mapping of integers to instructions might be a possible explanation for this observation.

Persistent change in the processor structure and instruction set size may require more time or some kind of macro-evolutionary steps in order to appear. Alternatively we may consider the problem of redesigning of the universal processor's structure in terms of a more flexible mapping of instruction codes to instructions.

The second category comprises attributes which were observed to vary:

- The instruction set was highly modified, especially when evolutionary learning was involved: instruction definitions were extended with new basic instructions or even completely changed.
- The sectioning present in the original handwritten organism vanished, i.e. the descriptive part and the executable code became interwoven. This way some parts of the organism's code were reused and had two or more different meanings depending on the context.

6 Conclusions

We have demonstrated that the evolutionary potential of universal processors is at least high as for the fixed processors in previous systems. The observed evolutionary changes in processor structures justify the need for further extending these experiments. While the number of instructions remained unchanged in our experiments, the instruction set itself did change radically during the course of evolution. The observed conservatism in number and type of processor components and instruction set size indicates a rigidity against changing the interpretation of instructions analogous to conservatism of DNA to RNA transcription in living organisms [13, Ch. 5-6]. By evolving the genotype-phenotype mapping by puting processor design and instruction set definition under evolutionary control, we may realize digital organisms with greater evolutionary potential.

References

1. Avida. `http://dllab.caltech.edu`. Digital Life Laboratory.
2. Physis. `http://physis.sourceforge.net`.
3. Chris Adami and C. Titus Brown. Evolutionary learning in the 2D artificial life system "Avida". *Proc. Artificial Life IV*, pages 377–381, 1994. MIT Press.
4. Christoph Adami and Claus O. Wilke. The biology of digital organisms. *Trends in Ecology & Evolution*, 17(11):528–532, November 2002.
5. Marc de Groot. Primordial soup. unpublished.
6. C.L. Nehaniv (ed.). *BioSystems*, special issue on evolvability. 69(2-3), 2003.
7. Stephen Jay Gould and Niles Eldredge. Punctuated equilibrium comes of age. *Nature*, 366:223–227, November 1993.
8. Marc Kirschner and John Gerhart. Evolvability. *PNAS*, 95:8420–8427, 1998.
9. Alexander Klaiber. The technology behind the Crusoe processors, 2000. `http://www.transmeta.com`.
10. Charles Ofria, Christoph Adami, and Travis C. Collier. Design of evolvable computer languages. *IEEE Transactions on Evolutionary Computation*, 6(4):420–424, 2002.
11. Thomas S. Ray. An approach to the synthesis of life. *Artificial Life II., Studies in the Sciences of Complexity*, IX:371–408, 1992. Addison Wesley.
12. Thomas S. Ray. Evolution, complexity, entropy, and artificial reality. *Physica D*, 75:239–263, 1994.
13. J. Maynard Smith and E. Szathmáry. *The Major Transitions in Evolution*. W.H. Freeman, 1995.
14. Daniel Tabak. *RISC systems and applications*. Research Studies Press, 1996.
15. Tim Taylor. Some representational and ecological aspects of evolvability. In Carlo C. Maley and Eilis Boudreau, editors, *Artificial Life 7 Workshop Proceedings*, pages 35–38. 2000. Available online at: `http://homepages.feis.herts.ac.uk/~nehaniv/al7ev/cnts.html`.
16. Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.
17. G.P. Wagner and L. Altenberg. Complex adaptations and the evolution of evolvability. *Evolution*, 50(3):967–976, June 1996.