TR - H - 176

A Computational Approach to Evolutionary Biology

Thomas S. Ray

1995. 12. 4

ATR人間情報通信研究所

〒619-02　京都府相楽郡精華町光台2-2　☎ 0774-95-1011

ATR Human Information Processing Research Laboratories
2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan
Telephone: +81-774-95-1011
Facsimile: +81-774-95-1008

Thomas S. Ray
ATR Human Information Processing Research Laboratories
2-2 Hikaridai, Seika-cho Soraku-gun, Kyoto 619-02 Japan
(81)-774-95-1008 (FAX), (81)-774-95-1063 (phone)
ray@hip.atr.co.jp, ray@santafe.edu, ray@udel.edu
http://www.hip.atr.co.jp/~ray

August 25, 1995

A Computational Approach To Evolutionary Biology

1 Introduction

> Marcel, a mechanical chessplayer... his exquisite 19th-century brainwork — the human art it took to build which has been flat lost, lost as the dodo bird ... But where inside Marcel is the midget Grandmaster, the little Johann Allgeier? where's the pantograph, and the magnets? Nowhere. Marcel really is a mechanical chessplayer. No fakery inside to give him any touch of humanity at all.
>
> — Thomas Pynchon, *Gravity's Rainbow.*

1.1 Synthetic Biology

Artificial Life (AL) is the enterprise of understanding biology by constructing biological phenomena out of artificial components, rather than breaking natural life forms down into their component parts. It is the synthetic rather than the reductionist approach. I will describe an approach to the synthesis of artificial living forms that exhibit natural evolution.

The umbrella of Artificial Life is broad, and covers three principal approaches to synthesis: in hardware (e.g., robotics, nanotechnology), in software (e.g., replicating and evolving computer programs), in wetware (e.g., replicating and evolving organic molecules, nucleic acids or others). This essay will focus on software synthesis, although it is hoped that the issues discussed will be generalizable to any synthesis involving the process of evolution.

I would like to suggest that software syntheses in AL could be divided into two kinds: simulations and instantiations of life processes. AL simulations represent an advance in biological modeling, based on a bottom-up approach, that has been made possible by the increase of available computational power. In the older approaches to modeling of ecological or evolutionary phenomena, systems of differential equations were set up that expressed relationships between covarying quantities of entities (i.e., genes, alleles, individuals, or species) in the populations or communities.

The new bottom up approach creates a population of data structures, with each instance

of the data structure corresponding to a single entity. These structures contain variables defining the state of an individual. Rules are defined as to how the individuals interact with one another and with the environment. As the simulation runs, populations of these data structures interact according to local rules, and the global behavior of the system emerges from those interactions. Several very good examples of bottom up ecological models have appeared in the AL literature [36, 86]. However, ecologists have also developed this same approach independently of the AL movement, and have called the approach "individual based" models [20, 40].

The second approach to software synthesis is what I have called instantiation rather than simulation. In simulation, data structures are created which contain variables that represent the states of the entities being modeled. The important point is that in simulation, the data in the computer is treated as a representation of something else, such as a population of mosquitoes or trees. In instantiation, the data in the computer does not represent anything else. The data patterns in an instantiation are considered to be living forms in their own right, and are not models of any natural life form. These can from the basis of a comparative biology [55].

The object of an AL instantiation is to introduce the natural form and process of life into an artificial medium. This results in an artificial life form in some medium other than carbon chemistry, and is not a model of organic life forms. The approach discussed in this essay involves introducing the process of evolution by natural selection into the computational medium. I consider evolution to be the fundamental process of life, and the generator of living form.

Ideally, the science of biology should embrace all forms of life. However in practice, it has been restricted to the study of a single instance of life, life on earth. Life on earth is very diverse, but it is presumably all part of a single phylogeny. Because biology is based on a sample size of one, we can not know what features of life are peculiar to earth, and what features are general, characteristic of all life. A truly comparative natural biology would require inter-planetary travel, which is light years away. The ideal experimental evolutionary biology would involve creation of multiple planetary systems, some essentially identical, others varying by a parameter of interest, and observing them for billions of years.

A practical alternative to an inter-planetary or mythical biology is to create synthetic life in a computer. The objective is not necessarily to create life forms that would serve as models for the study of natural life, but rather to create radically different life forms, based on a completely different physics and chemistry, and let these life forms evolve their own phylogeny, leading to whatever forms are natural to their unique physical basis. These truly independent instances of life may then serve as a basis for comparison, to gain some insight into what is general and what is peculiar in biology. Those aspects of life that prove to be general enough to occur in both natural and synthetic systems can then be studied more easily in the synthetic system.

The intent of this work is to synthesize rather than simulate life. This approach starts with hand crafted organisms already capable of replication and open-ended evolution, and aims to generate increasing diversity and complexity in a parallel to the Cambrian explosion.

To state such a goal leads to semantic problems, because life must be defined in a way that does not restrict it to carbon based forms. It is unlikely that there could be general agreement on such a definition, or even on the proposition that life need not be carbon based. Therefore, I will simply state my conception of life in its most general sense. I would consider a system to be living if it is self-replicating, and capable of open-ended evolution. Synthetic life should self-replicate, and evolve structures or processes that were not designed-in or pre-conceived by the creator ([12, 64]).

Core Wars programs, computer viruses, and worms ([14, 22, 23, 24, 26, 27, 77, 82]) are capable of self-replication, but fortunately, not evolution. It is unlikely that such programs will ever become fully living, because they are not likely to be able to evolve.

Most evolutionary simulations are not open-ended. Their potential is limited by the structure of the model, which generally endows each individual with a genome consisting of a set of pre-defined genes, each of which may exist in a pre-defined set of allelic forms ([1, 18, 19, 25, 37, 62]). The object being evolved is generally a data structure representing the genome, which the simulator program mutates and/or recombines, selects, and replicates according to criteria designed into the simulator. The data structures do not contain the mechanism for replication, they are simply copied by the simulator if they survive the selection phase.

Self-replication is critical to synthetic life because without it, the mechanisms of selection must also be pre-determined by the simulator. Such artificial selection can never be as creative as natural selection. The organisms are not free to invent their own fitness functions. Freely evolving creatures will discover means of mutual exploitation and associated implicit fitness functions that we would never think of. Simulations constrained to evolve with pre-defined genes, alleles and fitness functions are dead ended, not alive.

The approach presented here does not have such constraints. Although the model is limited to the evolution of creatures based on sequences of machine instructions, this may have a potential comparable to evolution based on sequences of organic molecules. Sets of machine instructions similar to those used in the Tierra Simulator have been shown to be capable of "universal computation" ([3, 48, 56]). This suggests that evolving machine codes should be able to generate any level of complexity.

Other examples of the synthetic approach to life can be seen in the work of [5, 30, 38, 46, 65]. A characteristic these efforts generally have in common is that they parallel the origin of life event by attempting to create prebiotic conditions from which life may emerge spontaneously and evolve in an open ended fashion.

While the origin of life is generally recognized as an event of the first order, there is another event in the history of life that is less well known but of comparable significance: the origin of biological diversity and macroscopic multicellular life during the Cambrian explosion 600 million years ago. This event involved a riotous diversification of life forms. Dozens of phyla appeared suddenly, many existing only fleetingly, as diverse and sometimes bizarre ways of life were explored in a relative ecological void ([34, 60]).

The work presented here aims to parallel the second major event in the history of life, the origin of diversity. Rather than attempting to create prebiotic conditions from which life may emerge, this approach involves engineering over the early history of life to design

complex evolvable organisms, and then attempting to create the conditions that will set off a spontaneous evolutionary process of increasing diversity and complexity of organisms. This work represents a first step in this direction, creating an artificial world which may roughly parallel the RNA world of self-replicating molecules (still falling far short of the Cambrian explosion).

1.2 Recognizing Life

Most approaches to defining life involve assembling a short list of properties of life, and then testing candidates on the basis of whether or not they exhibit the properties on the list. The main problem with this approach is that there is disagreement as to what should be on the list. My private list contains only two items: self-replication and open-ended evolution. However, this reflects my biases as an evolutionary biologist.

I prefer to avoid the semantic argument and take a different approach to the problem of recognizing life. I was led to this view by contemplating how I would regard a machine that exhibited conscious intelligence at such a level that it could participate as an equal in a debate such as this. The machine would meet neither of my two criteria as to what life is, yet I don't feel that I could deny that the process it contained was alive.

This means that there are certain properties that I consider to be unique to life, and whose presence in a system signify the existance of life in that system. This suggests an alternative approach to the problem. Rather than creating a short list of minimal requirements and testing whether a system exhibits all items on the list, create a long list of properties unique to life and test whether a system exhibits *any* item on the list.

In this softer, more pluralistic approach to recognizing life, the objective is not to determine if the system is alive or not, but to determine if the system exhibits a "genuine" instance of some property that is a signature of living systems (e.g., self-replication, evolution, flocking, consciousness).

Whether we consider a system living because it exhibits some property that is unique to life amounts to a semantic issue. What is more important is that we recognize that it is possible to create disembodied but genuine instances of specific properties of life in artificial systems. This capability is a powerful research tool. By separating the property of life that we choose to study, from the many other complexities of natural living systems, we make it easier to manipulate and observe the property of interest. The objective of the approach advocated in this paper is to capture genuine evolution in an artificial system.

1.3 What Natural Evolution Does

Evolution by natural selection is a process that enters into a physical medium. Through iterated replication-with-selection of large populations through many generations, it searches out the possibilities inherent in the "physics and chemistry" of the medium in which it is

embedded. It exploits any inherent self-organizing properties of the medium, and flows into natural attractors realizing and fleshing out their structure.

Evolution never escapes from its ultimate imperative: self-replication. However, the mechanisms that evolution discovers for achieving this ultimate goal gradually become so convoluted and complex that the underlying drive can seem to become superfluous. Some philosophers have argued that the evolutionary theory as expressed by the phrase "survival of the fittest" is tautological, in that the fittest are defined as those that survive to reproduce. In fact, fitness is achieved through innovation in engineering of the organism [79]. However there remains something peculiarly self-referential about the whole enterprise. There is some sense in which life may be a natural tautology.

Evolution is both a defining characteristic and the creative process of life itself. The living condition is a state that complex physical systems naturally flow into under certain conditions. It is a self-organizing, self-perpetuating state of auto-catalytically increasing complexity. The living component of the physical system quickly becomes the most complex part of the system, such that it re-shapes the medium, in its own image as it were. Life then evolves adaptations predominantly in relation to the living components of the system, rather than the non-living components. Life evolves adaptations to itself.

1.3.1 Evolution in Sequence Space

Think of organisms as occupying a "genotype space" consisting of all possible sequences of all possible lengths of the elements of the genetic system (i.e., nucleotides or machine instructions). When the first organism begins replicating, a single self-replicating creature, with a single sequence of a certain length occupies a single point in the genotype space. However, as the creature replicates in the environment, a population of creatures forms, and errors cause genetic variation, such that the population will form a cloud of points in the genotype space, centered around the original point.

Because the new genotypes that form the cloud are formed by random processes, most of them are completely inviable, and die without reproducing. However, some of them are capable of reproduction. These new genotypes persist, and as some of them are affected by mutation, the cloud of points spreads further. However, not all of the viable genomes are equally viable. Some of them discover tricks to replicate more efficiently. These genotypes increase in frequency, causing the population of creatures at the corresponding points in the genotype space to increase.

Points in the genotype space occupied by greater populations of individuals will spawn larger numbers of mutant offspring, thus the density of the cloud of points in the genotype space will shift gradually in the direction of the more fit genotypes. Over time, the cloud of points will percolate through the genotype space, either expanding outward as a result of random drift, or by flowing along fitness gradients.

Most of the volume of this space represents completely inviable sequences. These regions of the space may be momentarily and sparsely occupied by inviable mutants, but the cloud will never flow into the inviable regions. The cloud of genotypes may bifurcate as it flows

into habitable regions in different directions, and it may split as large genetic changes spawn genotypes in distant but viable regions of the space. We may imagine that the evolving population of creatures will take the form of wispy clouds flowing through this space.

Now imagine for a moment the situation that there were no selection. This implies that every sequence is replicated at an equal rate. Mutation will cause the cloud of points to expand outward, eventually filling the space uniformly. In this situation, the complexity of the structure of the cloud of points does not increase through time, only the volume that it occupies. Under selection by contrast, through time the cloud will take on an intricate structure as it flows along fitness gradients and percolates by drift through narrow regions of viability in a largely uninhabitable space.

Consider that the viable region of the genotype space is a very small subset of the total volume of the space, but that it probably exhibits a very complex shape, forming tendrils and sheets sparsely permeating the otherwise empty space. The complex structure of this cloud can be considered to be a product of evolution by natural selection. This thought experiment appears to imply that the intricate structure that the cloud of genotypes may assume through evolution is fully deterministic. Its shape is pre-defined by the physics and chemistry and the structure of the environment, in much the same way that the form of the Mandlebrot set is pre-determined by its defining equation. The complex structure of this viable space is inherent in the medium, and is an example of "order for free" [44].

No living world will ever fill the entire viable subspace, either at a single moment of time, or even cumulatively over its entire history. The region actually filled will be strongly influenced by the original self-replicating sequence, and by stochastic forces which will by chance push the cloud down a subset of possible habitable pathways. Furthermore, co-evolution and ecological interactions imply that certain regions can only be occupied when certain other regions are also occupied. This concept of the flow of genotypes through the genotype space is essentially the same as that discussed by Eigen [28] in the context of "quasispecies". Eigen limited his discussion to species of viruses, where it is also easy to think of sequence spaces. Here, I am extending the concept beyond the bounds of the species, to include entire phylogenies of species.

1.3.2 Natural Evolution in an Artificial Medium

Until recently, life has been known as a state of matter, particularly combinations of the elements carbon, hydrogen, oxygen, nitrogen and smaller quantities of many others. However, recent work in the field of Artificial Life has shown that the natural evolutionary process can proceed with great efficacy in other media, such as the informational medium of the digital computer [2, 7, 10, 16, 17, 21, 31, 35, 42, 43, 51, 53, 54, 65, 66, 68, 69, 70, 71, 73, 74, 78, 84, 85]

These new natural evolutions, in artificial media, are beginning to explore the possibilities inherent in the "physics and chemistry" of those media. They are organizing themselves and constructing self-generating complex systems. While these new living systems are still so young that they remain in their primordial state, it appears that they have embarked on the same kind of journey taken by life on earth, and presumably have the potential to evolve

levels of complexity that could lead to sentient and eventually intelligent beings.

If natural evolution in artificial media leads to sentient or intelligent beings, they will likely be so alien that they will be difficult to recognize. The sentient properties of plants are so radically different from those of animals, that they are generally unrecognized or denied by humans, and plants are merely in another kingdom of the one great tree of organic life on earth [67, 72, 83]. Synthetic organisms evolving in other media such as the digital computer, are not only not a part of the same phylogeny, but they are not even of the same physics. Organic life is based on conventional material physics, whereas digital life exists in a logical, not material, informational universe. Digital intelligence will likely be vastly different from human intelligence; forget the Turing test.

1.4 The Approach

The objective of the approach discussed here, is to create an instantiation of evolution by natural selection in the computational medium. This creates a conceptual problem that requires considerable art to solve: ideas and techniques must be learned by studying organic evolution, and then applied to the generation of evolution in a digital medium, without forcing the digital medium into an "un-natural" simulation of the organic world.

We must derive inspiration from observations of organic life, but we must never lose sight of the fact that the new instantiation is not organic, and may differ in many fundamental ways. For example, organic life inhabits a Euclidean space, however computer memory is not a Euclidean space. Inter-cellular communication in the organic world is chemical in nature, and therefore a single message generally can pass no more information than on or off. By contrast, communication in digital computers generally involves the passing of bit patterns, which can carry much more information.

The fundamental principal of the approach being advocated here is *to understand and respect the natural form of the digital computer, to facilitate the process of evolution in generating forms that are adapted to the computational medium, and to let evolution find forms and processes that naturally exploit the possibilities inherent in the medium.*

Situations arise where it is necessary to make significant changes from the standard computer architecture. But such changes should be made with caution, and only when there is some feature of standard computer architectures which clearly inhibits the desired processes. Examples of such changes are discussed in the section "The Genetic Language" below. Less substantial changes are also discussed in the sections on "Mutations", "Flaws", and "Artificial Death". The section on "Spatial Topology" and is a little tirade against examples of what I consider to be un-natural transfers of forms from the natural world to the digital medium.

1.5 The Metaphor

Organic life is viewed as utilizing energy, mostly derived from the sun, to organize matter. By analogy, digital life can be viewed as using CPU (central processing unit) time, to organize information. Organic life evolves through natural selection as individuals compete for resources (light, food, space, etc.) such that genotypes which leave the most descendants increase in frequency. Digital life evolves through the same process, as replicating algorithms compete for CPU time and memory space, and organisms evolve strategies to exploit one another. CPU time is thought of as the analog of the energy resource, information as the analog of the material resource, and memory as the analog of the spatial resource.

The memory, the CPU and the computer's operating system are viewed as elements of the "abiotic" (physical) environment. A "creature" is then designed to be specifically adapted to the features of the computational environment. The creature consists of a self-replicating assembler language program. Assembler languages are merely mnemonics for the machine codes that are directly executed by the CPU. These machine codes have the characteristic that they directly invoke the instruction set of the CPU and services provided by the operating system.

All programs, regardless of the language they are written in, are converted into machine code before they are executed. Machine code is the natural language of the machine, and machine instructions are viewed by this author as the "atomic units" of computing. It is felt that machine instructions provide the most natural basis for an artificial chemistry of creatures designed to live in the computer.

In the biological analogy, the machine instructions are considered to be more like the amino acids than the nucleic acids, because they are "chemically active". They actively manipulate bits, bytes, CPU registers, and the movements of the instruction pointer (see below). The digital creatures discussed here are entirely constructed of machine instructions. They are considered analogous to creatures of the RNA world, because the same structures bear the "genetic" information and carry out the "metabolic" activity.

A block of RAM memory (random access memory, also known as "main" or "core" memory) in the computer is designated as a "soup" which can be inoculated with creatures. The "genome" of the creatures consists of the sequence of machine instructions that make up the creature's self-replicating algorithm. The prototype creature consists of about 80 machine instructions, thus the size of the genome of this creature is 80 instructions, and its "genotype" is the specific sequence of those 80 instructions (Appendix D).

1.6 The Computational Medium

The computational medium of the digital computer is an informational universe of boolean logic, not a material one. Digital organisms live in the memory of the computer, and are powered by the activity of the central processing unit (CPU). Whether the hardware of the CPU and memory is built of silicon chips, vacuum tubes, magnetic cores, or mechanical

switches is irrelevant to the digital organism. Digital organisms should be able to take on the same form in any computational hardware, and in this sense are "portable" across hardware.

Digital organisms might as well live in a different universe from us, as they are not subject to the same laws of physics and chemistry. They are subject to the "physics and chemistry" of the rules governing the manipulation of bits and bytes within the computer's memory and CPU. They never "see" the actual material from which the computer is constructed, they see only the logic and rules of the CPU and the operating system. These rules are the only "natural laws" that govern their behavior. They are not influenced by the natural laws that govern the material universe (e.g., the laws of thermodynamics).

A typical instantiation of this type involves the introduction of a self-replicating machine language program into the RAM memory of a computer subject to random errors such as bit flips in the memory or occasionally inaccurate calculations [7, 10, 21, 53, 68]. This generates the basic conditions for evolution by natural selection as outlined by Darwin [15]: self-replication in a finite environment with heritable genetic variation.

In this instantiation, the self-replicating machine language program is thought of as the individual "digital organism" or "creature". The RAM memory provides the physical space that the creatures occupy. The CPU provides the source of energy. The memory consists of a large array of bits, generally grouped into eight bit bytes and sixteen or thirty-two bit words. Information is stored in these arrays as voltage patterns which we usually symbolize as patterns of ones and zeros.

The "body" of a digital organism is the information pattern in memory that constitutes its machine language program. This information pattern is data, but when it is passed to the CPU, it is interpreted as a series of executable instructions. These instructions are arranged in such a way that the data of the body will be copied to another location of memory. The informational patterns stored in the memory are altered only through the activity of the CPU. It is for this reason that the CPU is thought of as the analog of the energy source. Without the activity of the CPU, the memory would be static, with no changes in the informational patterns stored there.

The logical operations embodied in the instruction set of the CPU constitute a large part of the definition of the "physics and chemistry" of the digital universe. The topology of the computer's memory (discussed below) is also a significant component of the digital physics. The final component of the digital physics is the operating system, a software program running on the computer, which embodies rules for the allocation of resources such as memory space and CPU time to the various processes running on the computer.

The instruction set of the CPU, the memory, and the operating system together define the complete "physics and chemistry" of the universe inhabited by the digital organism. They constitute the physical environment within which digital organisms will evolve. Evolving digital organisms will compete for access to the limited resources of memory space and CPU time, and evolution will generate adaptations for the more agile access to and the more efficient use of these resources.

1.7 Spatial Topology

Digital organisms live in the memory space of computers, predominantly in the RAM memory, although they could also live on disks or any other storage device, or even within networks to the extent that the networks themselves can store information. In essence, digital organisms live in the space that has been referred to as "cyber-space". It is worthwhile reflecting on the topology of this space, as it is a radically different space from the one we live in.

A typical UNIX workstation, or MacIntosh computer includes a RAM memory that can contain some megabytes of data. This is "flat" memory, meaning that it is essentially unstructured. Any location in memory can be accessed through its numeric address. Thus adjacent locations in memory are accessed through successive integer values. This addressing convention causes us to think of the memory as a linear space, or a one-dimensional space.

However, this apparent one-dimensionality of the RAM memory is something of an illusion generated by the addressing scheme. A better way of understanding the topology of the memory comes from asking "what is the distance between two locations in memory". In fact the distance can not be measured in linear units. The most appropriate unit is the time that it takes to move information between the two points.

Information contained in the RAM memory can not move directly from point to point. Instead the information is transferred from the RAM to a register in the CPU, and then from the CPU back to the new location in RAM. Thus the distance between two locations in RAM is just the time that it takes to move from the RAM to the CPU plus the time that it takes to move from the CPU to the RAM. Because all points in the RAM are equidistant from the CPU, the distance between any pair of locations in the RAM is the same, regardless of how far apart they may appear based on their numeric addresses.

A space in which all pairs of points are equidistant is clearly not a Euclidean space. That said, we must recognize however, that there are a variety of ways in which memory is normally addressed, that gives it the appearance, at least locally, of being one dimensional. When code is executed by the CPU, the instruction pointer generally increments sequentially through memory, for short distances, before jumping to some other piece of code. For those sections of code where instructions are sequential, the memory is effectively one-dimensional. In addition, searches of memory are often sequentially organized (e.g., the search for complementary templates in Tierra). This again makes the memory effectively one-dimensional within the search radius. Yet even under these circumstances, the memory is not globally one-dimensional. Rather it consists of many small one dimensional pieces, each of which has no meaningful spatial relationship to the others.

Because we live in a three-dimensional Euclidean space, we tend to impose our familiar concepts of spatial topology onto the computer memory. This leads first to the erroneous perception that memory is a one-dimensional Euclidean space, and second, it often leads to the conclusion that the digital world could be enriched by increasing the dimensionality of the Euclidean memory space.

Many of the serious efforts to extend the Tierra model have included as a central feature,

the creation of a two-dimensional space for the creatures to inhabit [7, 16, 17, 53, 78]. The logic behind the motivation derives from contemplation of the extent to which the dimensionality of the space we live in permits the richness of pattern and process that we observe in nature. Certainly if our universe were reduced from three to two dimensions, it would eliminate the possibility of most of the complexity that we observe. Imagine for example, the limitations that two-dimensionality would place on the design of neural networks (if "wires" could not cross). If we were to further reduce the dimensionality of our universe to just one dimension, it would probably completely preclude the possibility of the existence of life.

It follows from these thoughts, that restricting digital life to a presumably one-dimensional memory space places a tragic limitation on the richness that might evolve. Clearly it would be liberating to move digital organisms into a two or three-dimensional space. The flaw in all of this logic derives from the erroneous supposition that computer memory is a Euclidean space.

To think of memory as Euclidean is to fail to understand its natural topology, and is an example of one of the greatest pitfalls in the enterprise of synthetic biology: to transfer a concept from organic life to synthetic life in a way that is "un-natural" for the artificial medium. The fundamental principal of the approach I am advocating is *to respect the nature of the medium into which life is being inoculated, and to find the natural form of life in that medium*, without inappropriately trying to make it like organic life.

The desire to increase the richness of memory topology is commendable, however this can be achieved without forcing the memory into an un-natural Euclidean topology. Let us reflect a little more on the structure of cyberspace. Thus far we have only considered the topology of flat memory. Let us consider segmented memory such as is found with the notorious Intel 80X86 design. With this design, you may treat any arbitrarily chosen block of 64K bytes as flat, and all pairs of locations within that block are equidistant. However, once the block is chosen, all memory outside of that block is about twice as far away.

Cache memory is designed to be accessed more rapidly than RAM memory, thus pairs of points within cache memory are closer than pairs of points within RAM memory. The distance between a point in cache and a point in RAM would be an intermediate distance. The access time to memory on disks is much greater than for RAM memory, thus the distance between points on disk is very great, and the distance between RAM and disk is again intermediate (but still very great). CPU registers represent a small amount of memory locations, between which data can move very rapidly, thus these registers can be considered to be very close together.

For networked computer systems, information can move between the memories of the computers on the net, and the distances between these memories is again the transfer time. If the CPU, cache, RAM and disk memories of a network of computers are all considered together, they present a very complex memory topology. Similar considerations apply to massively parallel computers which have memories connected in a variety of topologies. Utilizing this complexity moves us in the direction of what has been intended by creating Euclidean memories for digital organisms, but does so while fully respecting the natural topology of computer memories.

2 Methods

> Here was a world of simplicity and certainty... a world based on the one and zero of life and death. Minimal, beautiful. The patterns of lives and deaths.... weightless, invisible chains of electronic presence or absence. If patterns of ones and zeros were "like" patterns of human lives and deaths, if everything about an individual could be represented in a computer record by a long string of ones and zeros, then what kind of creature would be represented by a long string of lives and deaths? It would have to be up one level at least — an angel, a minor god, something in a UFO.
>
> — Thomas Pynchon, *Vineland.*

2.1 The Virtual Computer — Tierra Simulator

The software used in this study is available over the net or on disk (Appendix A).

The computers we use are general purpose computers, which means, among other things, that they are capable of emulating through software, the behavior of any other computer that ever has been built or that could be built ([3, 48, 56]). We can utilize this flexibility to design a computer that would be especially hospitable to synthetic life.

There are several good reasons why it is not wise to attempt to synthesize digital organisms that exploit the machine codes and operating systems of real computers. The most urgent is the potential threat of natural evolution of machine codes leading to virus or worm type of programs that could be difficult to eradicate due to their changing "genotypes". This potential argues strongly for creating evolution exclusively in programs that run only on virtual computers and their virtual operating systems. Such programs would be nothing more than data on a real computer, and therefore would present no more threat than the data in a data base or the text file of a word processor.

Another reason to avoid developing digital organisms in the machine code of a real computer is that the artificial system would be tied to the hardware and would become obsolete as quickly as the particular machine it was developed on. In contrast, an artificial system developed on a virtual machine could be easily ported to new real machines as they become available.

A third issue, which potentially makes the first two moot, is that the machine languages of real machines are not designed to be evolvable, and in fact might not support significant evolution. Von Neuman type machine languages are considered to be "brittle", meaning that the ratio of viable programs to possible programs is virtually zero. Any mutation or recombination event in a real machine code is almost certain to produce a non-functional program. The problem of brittleness can be mitigated by designing a virtual computer whose machine code is designed with evolution in mind. Farmer & Belin [29] have suggested that overcoming this brittleness and "Discovering how to make such self-replicating patterns more robust so that they evolve to increasingly more complex states is probably the central problem in the study of artificial life."

The work described here takes place on a set of virtual computers known as Tierra (Spanish for Earth). The Tierra computer architectures have been designed with the feature that their machine code is robust to the genetic operations of mutation and recombination. This means that computer programs written in the machine code of these architectures remain viable some of the time after being randomly altered by bit-flips which cause the swapping of individual instructions with others from within the instruction set, or by swapping segments of code between programs (through a spontaneous sexual process). These new computers have not been built in silicon, but exist only as software prototypes known as "virtual computers".

Tierra is a parallel computer of the MIMD (multiple instruction, multiple data) type, with a processor (CPU) for each creature. Parallelism is imperfectly emulated by allowing each CPU to execute a small time slice in turn. Each CPU of this virtual computer contains two address registers, two numeric registers, a flags register to indicate error conditions, a stack pointer, a ten word stack, and an instruction pointer. Each virtual CPU is implemented via the C structure listed in Appendix B. Computations performed by the Tierran CPUs are probabilistic due to flaws that occur at a low frequency (see Mutation below).

The instruction set of a CPU typically performs simple arithmetic operations or bit manipulations, within the small set of registers contained in the CPU. Some instructions move data between the registers in the CPU, or between the CPU registers and the RAM (main) memory. Other instructions control the location and movement of an "instruction pointer" (IP). The IP indicates an address in RAM, where the machine code of the executing program (in this case a digital organism) is located.

The CPU perpetually performs a fetch-decode-execute-increment_IP cycle: The machine code instruction currently addressed by the IP is fetched into the CPU, its bit pattern is decoded to determine which instruction it corresponds to, and the instruction is executed. Then the IP is incremented to point sequentially to the next position in RAM, from which the next instruction will be fetched. However, some instructions like **jmp**, **call** and **ret** directly manipulate the IP, causing execution to jump to some other sequence of instructions in the RAM. In the Tierra Simulator this CPU cycle is implemented through the time_slice routine listed in Appendix C.

2.2 The Genetic Language

The simplest possible instantiation of a digital organism is a machine language program that codes for self-replication. In this case, the bit pattern that makes up the program is the body of the organism, and at the same time its complete genetic material. Therefore, the machine language defined by the CPU constitutes the genetic language of the digital organism.

It is worth noting at this point that the organic organism most comparable to this kind of digital organism is the hypothetical, and now extinct, RNA organism [9]. These were presumably nothing more than RNA molecules capable of catalyzing their own replication. What the supposed RNA organisms have in common with the simple digital organism is that a single molecule constitutes the body and the genetic information, and effects the

replication. In the digital organism a single bit pattern performs all the same functions.

The use of machine code as a genetic system raises the problem of brittleness. The assumption that machine languages are too brittle to evolve is probably true, as a consequence of the fact that machine languages have not previously been designed to survive random alterations. However, recent experiments have shown that brittleness can be overcome by addressing the principal causes, and without fundamentally changing the structure of machine languages [68, 74].

The first requirement for evolvability is graceful error handling. When code is being randomly altered, every possible meaningless or erroneous condition is likely to occur. The CPU should be designed to handle these conditions without crashing the system. The simplest solution is for the CPU to perform no operation when it meets these conditions, perhaps setting an error flag, and to proceed to the next instruction.

Due to random alterations of the bit patterns, all possible bit patterns are likely to occur. Therefore a good design is for all possible bit patterns to be interpretable as meaningful instructions by the CPU. For example in the Tierra system [68, 69, 70, 71, 73, 74], a five bit instruction set was chosen, in which all thirty-two five bit patterns represent good machine instructions.

This approach (all bit patterns meaningful) also could imply a lack of syntax, in which each instruction stands alone, and need not occur in the company of other instructions. To the extent that the language includes syntax, where instructions must precede or follow one another in certain orders, random alterations are likely to destroy meaningful syntax thereby making the language more brittle. A certain amount of this kind of brittleness can be tolerated as long as syntax errors are also handled gracefully.

During the design of the first evolvable machine language [68], a standard machine language (Intel 80X86) was compared to the genetic language of organic life, to attempt to understand the difference between the two languages that might contribute to the brittleness of the former and the robustness of the latter. One of the outstanding differences noted was in the number of basic informational objects contained in the two.

The organic genetic language is written with an alphabet consisting of four different nucleotides. Groups of three nucleotides form sixty-four "words" (codons), which are translated into twenty amino-acids by the molecular machinery of the cell. The machine language is written with sequences of two voltages (bits) which we conceptually represent as ones and zeros. The number of bits that form a "word" (machine instruction) varies between machine architectures, and in some architectures is not constant. However, the number required generally ranges from sixteen to thirty-two. This means that there are from tens of thousands to billions of machine instruction bit patterns, which are translated into operations performed by the CPU.

The thousands or billions of bit patterns that code for machine instructions contrasts with the sixty four nucleotide patterns that code for amino acids. The sixty-four nucleotide patterns are degenerate, in that they code for only twenty amino-acids. Similarly, the machine codes are degenerate, in that there are at most hundreds rather than thousands or billions of machine operations.

The machine codes exhibit a massive degeneracy (with respect to actual operations) as a result of the inclusion of data into the bit patterns coding for the operations. For example, the add operation will take two operands, and produce as a result the sum of the two operands. While there may be only a single add operation, the instruction may come in several forms depending on where the values of the two operands come from, and where the resultant sum will be placed. Some forms of the add instruction allow the value(s) of the operand(s) to be specified in the bit pattern of the machine code.

The inclusion of numeric operands in the machine code is the primary cause of the huge degeneracy. If numeric operands are not allowed, the number of bit patterns required to specify the complete set of operations collapses to at most a few hundred.

While there is no empirical data to support it, it is suspected that the huge degeneracy of most machine languages may be a source of brittleness. The logic of this argument is that mutation causes random swapping among the fundamental informational objects, codons in the organic language, and machine instructions in the digital language. It seems more likely that meaningful results will be produced when swapping among sixty-four objects than when swapping among billions of objects.

In order to make a machine code with a truly small instruction set, we must eliminate numeric operands. This can be accomplished by allowing the CPU registers and the stack to be the only operands of the instructions. When we need to encode an integer for some purpose, we can create it in a numeric register through bit manipulations: flipping the low order bit and shifting left. The program can contain the proper sequence of bit flipping and shifting instructions to synthesize the desired number, and the instruction set need not include all possible integers.

The size of the machine instruction set can be made comparable to the number of codons simply by eliminating numeric operands embedded in the machine code. However, this change creates some new problems. Computer programs generally function by executing instructions located sequentially in memory. However, in order to loop or branch, they use instructions such as "jump" to cause execution to jump to some other part of the program. Since the locations of these jumps are usually fixed, the jump instruction will generally have the target address included as an operand embedded in the machine code.

By eliminating operands from the machine code, we generate the need for a new mechanism of addressing for jumps. To resolve this problem, an idea can be borrowed from molecular biology. We can ask the question: how do biological molecules address one another? Molecules do not specify the coordinates of the other molecules they interact with. Rather, they present shapes on their surfaces that are complementary to the shapes on the surfaces of the target molecules. The concept of complementarity in addressing can be introduced to machine languages by allowing the jump instruction to be followed by some bit pattern, and having execution jump to the nearest occurrence of the complementary bit pattern.

Addressing by template is illustrated by the Tierran **jmp** (jump) instruction. Each **jmp** instruction is followed by a sequence of **nop** (no-operation) instructions, of which there are two kinds: **nop_0** and **nop_1**. Suppose we have a piece of code with five instruction in the

following order: **jmp nop_0 nop_0 nop_0 nop_1**. The system will search outward in both directions from the **jmp** instruction looking for the nearest occurrence of the complementary pattern: **nop_1 nop_1 nop_1 nop_0**. If the pattern is found, the instruction pointer will move to the end of the complementary pattern and resume execution. If the pattern is not found, an error condition (flag) will be set and the **jmp** instruction will be ignored (in practice, a limit is placed on how far the system may search for the pattern).

The Tierran language is characterized by two unique features: a truly small instruction set without numeric operands, and addressing by template. Otherwise, the language consists of familiar instructions typical of most machine languages, e.g., **mov**, **call**, **ret**, **pop**, **push** etc. The four complete instruction sets are listed in Appendix G.

In the development of the Tierran language, two changes were introduced to the machine language to reduce brittleness: elimination of numeric operands from the code, and the use of complementary patterns to control addressing. The resulting language proved to be evolvable [68]. As a result, nothing was learned about evolvability, because only one language was tested, and it evolved. It is not known what features of the language enhance its evolvability, which detract, and which do not affect evolvability. Subsequently, three additional languages were tested and the four languages were found to vary in their patterns and degree of evolvability [74]. However, it is still not known how the features of the language affect its evolvability.

2.3 Four Instruction Sets

The original Tierran virtual computer was designed by the author in the fall of 1989. In the summer of 1992, a series of meetings was held at the Santa Fe Institute to attempt to improve on the original design. Present at these meetings were: Steen Rassmussen (Santa Fe Institute), Walter Tackett (Hughes Aircraft), Chris Stephenson (IBM), Kurt Thearling (Thinking Machines), Dan Pirone (Santa Fe Institute), and the author. The discussions did not lead to a consensus as to how to improve on the original design, but rather, to three suggestions: instruction set 2 proposed by Kurt Thearling, instruction set 3 proposed by the author, and instruction set 4 proposed by Walter Tackett. In August 1992, the author implemented all three new instruction sets, and integrated them into the Tierra program. These four instruction sets are summarized in Appendix G.

The three new instruction sets are quite similar, differing primarily in the details of how information is moved between the registers of the CPU. Therefore, the differences between the first instruction set and the others will be described first. The new instruction sets include a number of features missing from the first: 1) Instructions for moving information between the CPU registers and the RAM memory (**movdi**, **movid**). 2) Instructions providing input/output facilities (**put**, **get**). 3) Facilitation of the full range of possible inter-register moves (this is where the three new instruction sets differ the most). 4) A conditional to test if the flag is set (**iffl**). 5) The memory allocation instruction (**mal**) has the option of specifying where the allocated memory will be located, or it may use other new options (i.e., better fit). 6) Facilities to support multi-cellularity. These include mechanisms

of inter-cellular communication (**put**, **get**) and innovations in the **divide** instruction that can provide a mechanism for gene regulation: the ability to determine where the instruction pointer starts executing in the daughter cell, and the ability to transfer the contents of the CPU registers from mother to daughter.

The original instruction set contained two inter-register moves: **movcd** (CX to DX) and **movab** (AX to BX). This is clearly incomplete, as there are many other register pairs between which moves are not allowed. However, the full set of four pushes and four pops makes it possible to move data in any direction between any pair of registers by combining the appropriate push-pop pair. Therefore, in designing the fourth instruction set, Walter Tackett chose to use the eight push-pop instructions to handle the inter-register moves, and the push-pop pairs are the only mechanism for inter-register moves in that instruction set.

In the third instruction set, inter-register moves are effected through the mechanisms used in the "reverse Polish notation" (RPN) as is found in the Hewlett-Packard calculators. This uses the **rollu**, **rolld**, **enter**, and **exch** instructions.

In the second instruction set, inter-register moves may be accomplished by a push-pop pair, or by the **movdd** (R0 = R1) instruction. However, this instruction set uses a level of indirection to refer to the registers. There are four "shadow" registers, each of which refers to one of the real registers. The contents of the shadow registers are determined by executing the four register instructions: **AX**, **BX**, **CX**, and **DX**. So, for example, if shadow register R0 contains the value CX, and shadow register R1 contains the value AX (which is arranged by executing the **AX** instruction followed by the **CX** instruction), then executing the **movdd** instruction, will move the value in AX to the CX register. All other instructions in set two also reference the actual registers through the shadow registers, and so may operate on any of the registers.

In addition to these differences, each of the instructions that are common to the three new sets (e.g.: **inc**, **dec**, **add**, **sub**, **zero**, **mal**, **pop**, **put**, etc.) may differ between sets in which registers they reference. For example, in set two **inc** operates on R0, in set three on AX, and in set four on CX. There are also some small differences in the set of calculation instructions included. These are due to differences in the number of instructions consumed in implementing the inter-register moves. There were different numbers of opcodes "left over", and these were filled with calculations. Set four has seven calculations, set two has eight calculations, and set three has nine calculations.

The original 80 byte program was slightly modified so that it could be implemented in a consistent manner across the four instruction sets. The four new seed organisms were then tested in a series of eight runs in each of the four languages (only four runs in the original language, because comparable data from this instruction set have already been published, [71]). The resulting twenty-eight runs form the basis of the comparisons of patterns of evolution across different languages, and the analysis of the development of complex structures in one of those languages (the fourth).

2.4 Operating System

Much of the "physics and chemistry" of the digital universe is determined by the specifications of the operations performed by the instruction set of the CPU. However, the operating system also determines a significant part of the physical context. The operating system manages the allocation of critical resources such as memory space and CPU cycles.

The Tierran virtual computer needs a virtual operating system that will be hospitable to digital organisms. The operating system will determine the mechanisms of interprocess communication, memory allocation, and the allocation of CPU time among competing processes.

Digital organisms are processes that spawn processes. As processes are born, the operating system will allocate memory and CPU cycles to them, and when they die, the operating system will return the resources they had utilized to the pool of free resources. In synthetic life systems, the operating system may also play a role in managing death, mutations and flaws.

The management of resources by the operating system is controlled by algorithms. From the point of view of the digital organisms these take the form of a set of logical rules like those embodied in the logic of the instruction set. In this way, the operating system is a defining part of the physics and chemistry of the digital universe. Evolution will explore the possibilities inherent in these rules, finding ways to more efficiently gain access to and exploit the resources managed by the operating system.

More than being a mere aspect of the environment, the operating system together with the instruction set will determine the topology of possible interactions between individuals, such as the ability of pairs of individuals to exhibit predator-prey, parasite-host or mutualistic relationships.

2.4.1 Memory Allocation — Cellularity

The Tierran computer operates on a block of RAM of the real computer which is set aside for the purpose. This block of RAM is referred to as the "soup". In most of the work described here the soup consisted of about 60,000 to 200,000 bytes, which can hold the same number of Tierran machine instructions. Each "creature" occupies some block of memory in this soup.

Cellularity is one of the fundamental properties of organic life, and can be recognized in the fossil record as far back as 3.6 billion years ([6]). The cell is the original individual, with the cell membrane defining its limits and preserving its chemical integrity. An analog to the cell membrane is needed in digital organisms in order to preserve the integrity of the informational structure from being disrupted easily by the activity of other organisms. The need for this can be seen in Artificial Life models such as cellular automata where virtual state machines pass through one another ([46, 47]), or in core wars type simulations where coherent structures demolish one another when they come into contact ([23, 26, 65]).

Tierran creatures are considered to be cellular in the sense that they are protected by a "semi-permeable membrane" of memory allocation. The Tierran operating system provides memory allocation services. Each creature has exclusive write privileges within its allocated block of memory. The "size" of a creature is just the size of its allocated block (e.g., 80 instructions). This usually corresponds to the size of the genome. This "membrane" is described as "semi-permeable" because while write privileges are protected, read and execute privileges are not. A creature may examine the code of another creature, and even execute it, but it can not write over it. Each creature may have exclusive write privileges in at most two blocks of memory: the one that it is born with which is referred to as the "mother cell", and a second block which it may obtain through the execution of the MAL (memory allocation) instruction. The second block, referred to as the "daughter cell", may be used to grow or reproduce into.

When Tierran creatures "divide", the mother cell loses write privileges on the space of the daughter cell, but is then free to allocate another block of memory. At the moment of division, the daughter cell is given its own CPU, and is free to allocate its own second block of memory.

2.4.2 Time Sharing — The Slicer

The Tierran operating system must be multi-tasking (or parallel) in order for a community of individual creatures to live in the soup simultaneously. The system doles out small slices of CPU time to each creature in the soup in turn. The system maintains a circular queue called the "slicer queue". As each creature is born, a virtual CPU is created for it, and it enters the slicer queue just ahead of its mother, which is the active creature at that time. Thus the newborn will be the last creature in the soup to get another time slice after the mother, and the mother will get the next slice after its daughter. As long as the slice size is small relative to the generation time of the creatures, the time sharing system causes the world to approximate parallelism. In actuality, we have a population of virtual CPUs, each of which gets a slice of the real CPU's time as it comes up in the queue.

The number of instructions to be executed in each time slice may be set proportional to the size of the genome of the creature being executed, raised to a power. If the "slicer power" is equal to one, then the slicer is size neutral, the probability of an instruction being executed does not depend on the size of the creature in which it occurs. If the power is greater than one, large creatures get more CPU cycles per instruction than small creatures. If the power is less than one, small creatures get more CPU cycles per instruction. The power determines if selection favors large or small creatures, or is size neutral. A constant slice size selects for small creatures.

2.4.3 Artificial Death

Death must play a role in any system that exhibits the process of evolution. Evolution involves a continuing iteration of selection, which implies differential death or reproduction.

In natural life, death occurs as a result of accident, predation, starvation, disease, or if these fail to kill the organism, it will eventually die from senescence resulting from an accumulation of wear and tear at every level of the organism including the molecular.

In normal computers, processes are "born" when they are initiated by the user, and "die" when they complete their task and halt. A process whose goal is to repeatedly replicate itself is essentially an endless loop, and would not spontaneously terminate. Due to the perfection of normal computer systems, we can not count on "wear and tear" to eventually cause a process to terminate.

In synthetic life systems implemented in computers, death is not likely to be a process that would occur spontaneously, and it must generally be introduced artificially by the designer. Everyone who has set up such a system has found their own unique solutions. Todd [88] recently discussed this problem in general terms.

The Tierran operating system includes a "reaper" which begins "killing" creatures from a queue when the memory fills. Creatures are killed by deallocating their memory, and removing them from both the reaper and slicer queues. Their "dead" code is not removed from the soup.

In the present system, the reaper uses a linear queue. When a creature is born it enters the bottom of the queue. The reaper always kills the creature at the top of the queue. However, individuals may move up or down in the reaper queue according to their success or failure at executing certain instructions. When a creature executes an instruction that generates an error condition, it moves one position up the queue, as long as the individual ahead of it in the queue has not accumulated a greater number of errors. Two of the instructions are somewhat difficult to execute without generating an error, therefore successful execution of these instructions moves the creature down the reaper queue one position, as long as it has not accumulated more errors than the creature below it.

The effect of the reaper queue is to cause algorithms which are fundamentally flawed to rise to the top of the queue and die. Vigorous algorithms have a greater longevity, but in general, the probability of death increases with age.

An interesting variation on this was introduced by Barton-Davis [7] who eliminated the reaper queue. In its place, he caused the "flaw rate" (see section on Flaws above) to increase with the age of the individual, in mimicry of wear and tear. When the flaw rate reached 100%, the individual was killed. Skipper [78] provided a "suicide" instruction, which if executed, would cause a process to terminate (die). The evolutionary objective then became to have a suicide instruction in your genome which you do not execute yourself, but which you try to get other individuals to execute. Litherland [51] introduced death by local crowding. Davidge caused processes to die when they contained certain values in their registers [17]. Gray [35] allowed each process six attempts at reproduction, after which they would die.

2.5 Genetic Operators

In order for evolution to occur, there must be some genetic variation among the offspring. In organic life, this is insured by natural imperfections in the replication of the informational molecules. However, one way in which digital "chemistry" differs from organic chemistry is in the degree of perfection of its operations. In the computer, the genetic code can be reliably replicated without errors to such a degree that we must artificially introduce errors or other sources of genetic variation in order to induce evolution.

2.5.1 Mutations

In organic life, the simplest genetic change is a "point mutation", in which a single nucleic acid in the genetic code is replaced by one of the three other nucleic acids. This can cause an amino acid substitution in the protein coded by the gene. The nucleic acid replacement can be caused by an error in the replication of the DNA molecule, or it can be caused by the effects of radiation or mutagenic chemicals.

In the digital medium, a comparably simple genetic change can result from a bit flip in the memory, where a one is replaced by a zero, or a zero is replaced by a one. These bit flips can be introduced in a variety of ways that are analogous to the various natural causes of mutation. In any case, the bit flips must be introduced at a low to moderate frequency, as high frequencies of mutation prevent the replication of genetic information, and lead to the death of the system [71].

Bit flips may be introduced at random anywhere in memory, where they may or may not hit memory actually occupied by digital organisms. This could be thought of as analogous to cosmic rays falling at random and disturbing molecules which may or may not be biological in nature. Bit flips may also be introduced when information is copied in the memory, which could be analogous to the replication errors of DNA. Alternatively, bit flips could be introduced in memory as it is accessed, either as data or executable code. This could be thought of as damage due to "wear and tear".

In Tierra mutations occur in two circumstances. At some background rate, bits are randomly selected from the entire soup (e.g., 60,000 instructions totaling 300,000 bits) and flipped. This is analogous to mutations caused by cosmic rays, and has the effect of preventing any creature from being immortal, as it will eventually mutate to death. The background mutation rate has generally been set at about one bit flipped for every 10,000 Tierran instructions executed by the system.

In addition, while copying instructions during the replication of creatures, bits are randomly flipped at some rate in the copies. The copy mutation rate is the higher of the two, and results in replication errors. The copy mutation rate has generally been set at about one bit flipped for every 1,000 to 2,500 instructions copied. In both classes of mutation, the interval between mutations varies randomly within a certain range to avoid possible periodic effects.

2.5.2 Flaws

Alterations of genetic information are not the only source of noise in the system. In organic life, enzymes have evolved to increase the probability of chemical reactions that increase the fitness of the organism. However, the metabolic system is not perfect. Undesired chemical reactions do occur, and desired reactions sometimes produce undesired by-products. The result is the generation of molecular species that can "gum up the works", having unexpected consequences, generally lowering the fitness of the organism, but possibly raising it.

In the digital system, an analogue of metabolic (non-genetic) errors can be introduced by causing the computations carried out by the CPU to be probabilistic, producing erroneous results at some low frequency. For example, any time a sum or difference is calculated, the result could be off by some small value (e.g. plus or minus one). Or, if all bits are shifted one position to the left or right, an appropriate error would be to shift by two positions or not at all. When information is transferred from one location to another, either in the RAM memory or the CPU registers, it could occasionally be transferred from the wrong location, or to the wrong location. While flaws do not directly cause genetic changes, they can cause a cascade of events that result in the production of an offspring that is genetically different from the parent.

It turns out that bit flipping mutations and flaws in instructions are not necessary to generate genetic change and evolution, once the community reaches a certain state of complexity. Genetic parasites evolve which are sloppy replicators, and have the effect of moving pieces of code around between creatures, causing rather massive rearrangements of the genomes. The mechanism of this ad hoc sexuality has not been worked out, but is likely due to the parasites' inability to discriminate between live, dead or embryonic code.

Mutations result in the appearance of new genotypes, which are watched by an automated genebank manager. When new genotypes increase in frequency across some arbitrary threshold, such as making up 2% of the population, they are given a unique name and saved to disk. Each genotype name contains two parts, a number and a three letter code. The number represents the number of instructions in the genome. The three letter code is used as a base 26 numbering system for assigning a unique label to each genotype in a size class. The first genotype to appear in a size class is assigned the label aaa, the second is assigned the label aab, and so on. Thus the ancestor is named 80aaa, and the first mutant of size 80 is named 80aab. The first creature of size 45 is named 45aaa.

The genebanker saves some additional information with each genome: the genotype name of its immediate ancestor which makes possible the reconstruction of the some of the phylogeny; the time and date of origin; "metabolic" data including the number of instructions executed in the first and second reproduction, the number of errors generated in the first and second reproduction, and the number of instructions copied into the daughter cell in the first and second reproductions (see Appendix D, E); some environmental parameters at the time of origin including the search limit for addressing, and the slicer power, both of which affect selection for size.

2.6 The Tierran Ancestor

I have used the Tierran language to write a single self-replicating program which is 80 instructions long. This program is referred to as the "ancestor", or alternatively as genotype 0080aaa (Figure 1). The ancestor is a minimal self-replicating algorithm which was originally written for use during the debugging of the simulator. No functionality was designed into the ancestor beyond the ability to self-replicate, nor was any specific evolutionary potential designed in. The commented Tierran assembler and machine code for this program is presented in Appendix D.

The ancestor examines itself to determine where in memory it begins and ends. The ancestor's beginning is marked with the four no-operation template: 1 1 1 1, and its ending is marked with 1 1 1 0. The ancestor locates its beginning with the five instructions: **adrb**, **nop_0**, **nop_0**, **nop_0**, **nop_0**. This series of instructions causes the system to search backwards from the **adrb** instruction for a template complementary to the four **nop_0** instructions, and to place the address of the complementary template (the beginning) in the AX register of the CPU (see Appendix D). A similar method is used to locate the end.

Having determined the address of its beginning and its end, it subtracts the two to calculate its size, and allocates a block of memory of this size for a daughter cell. It then calls the copy procedure which copies the entire genome into the daughter cell memory, one instruction at a time. The beginning of the copy procedure is marked by the four no-operation template: 1 1 0 0. Therefore the call to the copy procedure is accomplished with the five instructions: **call**, **nop_0**, **nop_0**, **nop_1**, **nop_1**.

When the genome has been copied, it executes the **divide** instruction, which causes the creature to lose write privileges on the daughter cell memory, and gives an instruction pointer to the daughter cell (it also enters the daughter cell into the slicer and reaper queues). After this first replication, the mother cell does not examine itself again; it proceeds directly to the allocation of another daughter cell, then the copy procedure is followed by cell division, in an endless loop.

Fourty-eight of the eighty instructions in the ancestor are no-operations. Groups of four no-operation instructions are used as complementary templates to mark twelve sites for internal addressing, so that the creature can locate its beginning and end, call the copy procedure, and mark addresses for loops and jumps in the code, etc. The functions of these templates are commented in the listing in Appendix D.

2.7 Evolution and Diversity/Entropy

Independently of these studies, a series of runs was conducted using the original program with the original instruction set, and applying a tool for the calculation of diversity/entropy and its changes over time in an evolving ecological community. The diversity/entropy measure is negative sum of p log p, where p is the proportion of the community occupied by each genotype. For the purposes of this study a filter was used, which ignored all genotypes represented by a single individual. The purpose of the filter was to eliminate all mutants

which were never able to reproduce (thus reducing the sensitivity of the measure to mutation rate). In addition, this diversity/entropy data was calculated for every birth and death, and then averaged over each million CPU cycle period. Only the averages over each period were recorded.

3 Results

3.1 Natural History

Evolutionary runs of the simulator are generally begun by inoculating a soup of about 60,000 instructions with a single individual of the 80 instruction ancestral genotype. The passage of time in a run is measured in terms of how many Tierran instructions have been executed by the simulator, or the number of generations of organisms that have passed. The original ancestral cell executes 839 instructions in its first replication, and 813 for each additional replication. The initial cell and its replicating daughters rapidly fill the soup memory which starts the reaper. Typically, the system executes about 400,000 instructions in filling up the soup with about 375 individuals of size 80 (and their 375 embryonic daughter cells). Once the reaper begins, the memory remains roughly filled with creatures for the remainder of the run.

If there were no mutations at the outset of the run, there would be no evolution. However, the bits flipped as a result of copy errors or background mutations result in creatures whose list of 80 instructions (genotype) differs from the ancestor, usually by a single bit difference in a single instruction.

Mutations in and of themselves, can not result in a change in the size of a creature, they can only alter the instructions in its genome. However, by altering the genotype, mutations may affect the processes whereby the creature examines itself, calculates its size, or replicates, potentially causing it to produce an offspring that differs in size from itself.

Four out of the five possible mutations in a no-operation instruction convert it into another kind of instruction, while one out of five converts it into the complementary no-operation. Therefore 80% of mutations in templates destroy or change the size of the template, while one in five alters the template pattern. An altered template may cause the creature to make mistakes in self examination, procedure calls, or looping or jumps of the instruction pointer, all of which use templates for addressing.

3.1.1 Parasites

An example of the kind of error that can result from a mutation in a template is a mutation of the low order bit of instruction 42 of the ancestor (Appendix D). Instruction 42 is a **nop_0**, the third component of the copy procedure template. A mutation in the low order bit would convert it into **nop_1**, thus changing the template from 1 1 0 0 to: 1 1 1 0. This would then

be recognized as the template used to mark the end of the creature, rather than the copy procedure.

A creature born with a mutation in the low order bit of instruction 42 would calculate its size as 45. It would allocate a daughter cell of size 45 and copy only instructions 0 through 44 into the daughter cell. The daughter cell then, would not include the copy procedure. This daughter genotype, consisting of 45 instructions, is named 0045aaa.

Genotype 0045aaa (Figure 1) is not able to self-replicate in isolated culture. However, the semi-permeable membrane of memory allocation only protects write privileges. Creatures may match templates with code in the allocated memory of other creatures, and may even execute that code. Therefore, if creature 0045aaa is grown in mixed culture with 0080aaa, when it attempts to call the copy procedure, it will not find the template within its own genome, but if it is within the search limit (generally set at five times the average genome size) of the copy procedure of a creature of genotype 0080aaa, it will match templates, and send its instruction pointer to the copy code of 0080aaa. Thus a parasitic relationship is established (see Ecology below). Typically, parasites begin to emerge within the first few million instructions of elapsed time in a run.

3.1.2 Immunity to Parasites

At least some of the size 79 genotypes demonstrate some measure of resistance to parasites. If genotype 45aaa is introduced into a soup, flanked on each side with one individual of genotype 0079aab, 0045aaa will initially reproduce somewhat, but will be quickly eliminated from the soup. When the same experiment is conducted with 0045aaa and the ancestor, they enter a stable cycle in which both genotypes coexist indefinitely. Freely evolving systems have been observed to become dominated by size 79 genotypes for long periods, during which parasitic genotypes repeatedly appear, but fail to invade.

3.1.3 Circumvention of Immunity to Parasites

Occasionally these evolving systems dominated by size 79 were successfully invaded by parasites of size 51. When the immune genotype 0079aab was tested with 0051aao (a direct, one step, descendant of 0045aaa in which instruction 39 is replaced by an insertion of seven instructions of unknown origin), they were found to enter a stable cycle. Evidently 0051aao has evolved some way to circumvent the immunity to parasites possessed by 0079aab. The fourteen genotypes 0051aaa through 0051aan were also tested with 0079aab, and none were able to invade.

3.1.4 Hyper-parasites

Hyper-parasite have been discovered, (e.g., 0080gai, which differs by 19 instructions from the ancestor, Figure 1). Their ability to subvert the energy metabolism of parasites is based on two changes. The copy procedure does not return, but jumps back directly to the proper

address of the reproduction loop. In this way it effectively seizes the CPU from the parasite. However it is another change which delivers the coup de grâce: after each reproduction, the hyper-parasite re-examines itself, resetting the BX register with its location and the CX register with its size. After the instruction pointer of the parasite passes through this code, the CPU of the parasite contains the location and size of the hyper-parasite and the parasite CPU thereafter replicates the hyper-parasite genome. In essence, the hyper-parasite has parallelized itself by recruiting a new CPU stolen from a parasite.

3.1.5 Social Hyper-parasites

Hyper-parasites drive the parasites, as well as all other algorithms, to extinction. This results in a community with a relatively high level of genetic uniformity, and therefore high genetic relationship between individuals in the community. These are the conditions that support the evolution of sociality, and social hyper-parasites soon dominate the community. Social hyper-parasites (Figure 2) appear in the 61 instruction size class. For example, 0061acg is social in the sense that it can only self-replicate when it occurs in aggregations. When it jumps back to the code for self-examination, it jumps to a template that occurs at the end rather than the beginning of its genome. If the creature is flanked by a similar genome, the jump will find the target template in the tail of the neighbor, and execution will then pass into the beginning of the active creature's genome. The algorithm will fail unless a similar genome occurs just before the active creature in memory. Neighboring creatures cooperate by catching and passing on jumps of the instruction pointer.

It appears that the selection pressure for the evolution of sociality is that it facilitates size reduction. The social species are 24% smaller than the ancestor. They have achieved this size reduction in part by shrinking their templates from four instructions to three instructions. This means that there are only eight templates available to them, and catching each others jumps allows them to deal with some of the consequences of this limitation as well as to make dual use of some templates.

3.1.6 Cheaters: Hyper-hyper-parasites

The cooperative social system of hyper-parasites is subject to cheating, and is eventually invaded by hyper-hyper-parasites (Figure 2). These cheaters (e.g., 0027aab) position themselves between aggregating hyper-parasites so that when the instruction pointer is passed between them, they capture it.

3.1.7 A Novel Self-examination

All creatures discussed thus far mark their beginning and end with templates. They then locate the addresses of the two templates and determine their genome size by subtracting them. In one run, creatures evolved without a template marking their end. These creatures located the address of the template marking their beginning, and then the address of a

template in the middle of their genome. These two addresses were then subtracted to calculate half of their size, and this value was multiplied by two (by shifting left) to calculate their full size.

3.2 Ecology

The only communities whose ecology has been explored in detail are those that operate under selection for small sizes. These communities generally include a large number of parasites, which do not have functional copy procedures, and which execute the copy procedures of other creatures within the search limit. In exploring ecological interactions, the mutation rate is set at zero, which effectively throws the simulation into ecological time by stopping evolution. When parasites are present, it is also necessary to stipulate that creatures must breed true, since parasites have a tendency to scramble genomes, leading to evolution in the absence of mutation.

0045aaa is a "metabolic parasite". Its genome does not include the copy procedure, however it executes the copy procedure code of a normal host, such as the ancestor. In an environment favoring small creatures, 0045aaa has a competitive advantage over the ancestor, however, the relationship is density dependent. When the hosts become scarce, most of the parasites are not within the search limit of a copy procedure, and are not able to reproduce. Their calls to the copy procedure fail and generate errors, causing them to rise to the top of the reaper queue and die. When the parasites die off, the host population rebounds. Hosts and parasites cultured together demonstrate Lotka-Volterra population cycling ([52, 89, 90]).

A number of experiments have been conducted to explore the factors affecting diversity of size classes in these communities. Competitive exclusion trials were conducted with a series of self-replicating (non-parasitic) genotypes of different size classes. The experimental soups were initially inoculated with one individual of each size. A genotype of size 79 was tested against a genotype of size 80, and then against successively larger size classes. The interactions were observed by plotting the population of the size 79 class on the x axis, and the population of the other size class on the y axis. Sizes 79 and 80 were found to be competitively matched such that neither was eliminated from the soup. They quickly entered a stable cycle, which exactly repeated a small orbit. The same general pattern was found in the interaction between sizes 79 and 81.

When size 79 was tested against size 82, they initially entered a stable cycle, but after about 4 million instructions they shook out of stability and the trajectory became chaotic with an attractor that was symmetric about the diagonal (neither size showed any advantage). This pattern was repeated for the next several size classes, until size 90, where a marked asymmetry of the chaotic attractor was evident, favoring size 79. The run of size 79 against size 93 showed a brief stable period of about a million instructions, which then moved to a chaotic phase without an attractor, which spiraled slowly down until size 93 became extinct, after an elapsed time of about 6 million instructions.

An interesting exception to this pattern was the interaction between size 79 and size 89. Size 89 is considered to be a "metabolic cripple", because although it is capable of

self-replicating, it executes about 40% more instructions than normal to replicate. It was eliminated in competition with size 79, with no loops in the trajectory, after an elapsed time of under one million instructions.

In an experiment to determine the effects of the presence of parasites on community diversity, a community consisting of twenty size classes of hosts was created and allowed to run for 30 million instructions, at which time only the eight smallest size classes remained. The same community was then regenerated, but a single genotype (0045aaa) of parasite was also introduced. After 30 million instructions, 16 size classes remained, including the parasite. This seems to be an example of a "keystone" parasite effect ([63]).

Symbiotic relationships are also possible. The ancestor was manually dissected into two creatures, one of size 46 which contained only the code for self-examination and the copy loop, and one of size 64 which contained only the code for self-examination and the copy procedure (Figure 3). Neither could replicate when cultured alone, but when cultured together, they both replicated, forming a stable mutualistic relationship. It is not known if such relationships have evolved spontaneously.

3.3 Evolutionary Optimization

In order to compare the process of evolution between runs of the simulator, a simple objective quantitative measure of evolution is needed. One such measure is the degree to which creatures improve their efficiency through evolution. This provides not only an objective measure of progress in evolution, but also sheds light on the potential application of synthetic life systems to the problem of the optimization of machine code.

The efficiency of the creature can be indexed in two ways: the size of the genome, and the number of CPU cycles needed to execute one replication. Clearly, smaller genomes can be replicated with less CPU time, however, during evolution, creatures also decrease the ratio of instructions executed in one replication, to genome size. The number of instructions executed per instruction copied, drops substantially.

Figure 4 shows the changes in genome size over a time period of 500 million instructions executed by the system, for eight sets of mutation rates differing by factors of two. Mutation rates are measured in terms of 1 in N individuals being affected by a mutation in each generation. At the highest two sets of rates tested, one and two, either each (one) or one-half (two) of the individuals are hit by mutation in each generation. At these rates the system is unstable. The genomes melt under the heat of the high mutation rates. The community often dies out, although some runs survived the 500 million instruction runs used in this study. The next lower rate, four, yields the highest rate of optimization without the risk of death of the community. At the five lower mutation rates, 8, 16, 32, 64 and 128, we see successively lower rates of optimization.

Additional replicates were made of the runs at the mutation rate of four (Figure 5). The replicates differ only in the seed of the random number generator, all other parameters being identical. These runs vary in some details such as whether progress is continuous and

gradual, or comes in bursts. Also, each run decreases to a size limit which it can not proceed past even if it is allowed to run much longer. However, different runs reach different plateaus of efficiency. The smallest limiting genome size seen has been 22 instructions, while other runs reached limits of 27 and 30 instructions. The simple interpretation is that the system can reach a local optima from which it can not easily evolve to the global optima. However, a close look as the "sub-optimal" solutions suggests an alternative explanation (see section on "Complex Structures" below).

The increase in efficiency of the replicating algorithms is even greater than the decrease in the size of the code. The ancestor is 80 instructions long and requires 839 CPU cycles to replicate. The creature of size 22 only requires 146 CPU cycles to replicate, a 5.75-fold difference in efficiency. The algorithm of one of these creatures is listed in Appendix E.

Although the optimization rate of the algorithm is maximized at the highest mutation rate that does not cause instability, ecological interactions appear to be richer at slightly lower mutation rates. At the rates of eight or 16, we find the diversity of coexisting size classes to be the greatest, and to persist the longest. The smaller size classes tend to be various forms of parasites, thus a diversity of size classes indicates a rich ecology.

An example of even greater optimization is illustrated in Appendix F and discussed in the section "Complex Structures". Unrolling of the loop results in a loop which uses 18 CPU cycles to copy three instructions, or six CPU cycles executed per instruction copied, compared to 10 for the ancestor. The creature of size 22 also uses six CPU cycles per instruction copied. However, the creature of Appendix F uses three extra CPU cycles per loop to compensate for a separate adaptation that allows it to double its share of CPU time from the global pool (in essence meaning that relatively speaking, it uses only three CPU cycles per instruction copied). Without this compensation it would use only five CPU cycles per instruction copied.

3.4 Evolutionary Patterns in Four Different Genetic Languages

In comparing the patterns of evolution across the four instruction sets, two major differences are apparent: 1) The degree and rate of optimization attained. 2) The patterns of gradualism, punctuation and equilibrium. These results are summarized in Table 1, and described below.

The original instruction set (Figure 6) shows the most rapid optimization, generally reaching its final plateau within 600 generations. In addition, this instruction set showed one of the highest degrees of optimization, with the best performance reducing the seed program from seventy-two to twenty-two instructions, 30% of its original size, and the average reduced to 35%. This instruction set generally showed a pattern of gradualism, with an occasional punctuation. This pattern could be described as punctuated gradualism.

The second instruction set (Figure 6) shows slower optimization, generally taking about 1000 generations to reach its final plateau. Also, the degree of optimization shown by this set is not as great. The best performance reduced the algorithm from ninety-four to fifty-four instructions, 57% of its original size, and the average reduced to 60%. This instruction set generally showed a pattern of gradualism, punctuations were completely absent.

Comparison of optimizations in the four instruction sets, I1, I2, I3 and I4 (described in Appendix G). The first column, "Set", specifies which of the four sets. The second column, "Ancestor", specifies the size in instructions, of the ancestral algorithm of that set. The following eight columns, "R0" through "R7" refer to the eight runs, and contain the size of the smallest algorithms evolved during that run. The column "Avg. Opt." shows the average optimization for that set. This is calculated by averaging the sizes of the smallest algorithms to evolve in each run for that set, and dividing by the size of the ancestral algorithm. The column "Max. Opt." shows the maximum optimization achieved by this set. This is calculated by dividing the size of the smallest algorithm to evolve by the size of the ancestral algorithm.

Table 1: **Comparison of Optimizations in the Four Instruction Sets**

Set	Ancestor	R0	R1	R2	R3	R4	R5	R6	R7	Avg. Opt.	Max. Opt.
I1	73	27	27	26	22					.35	.30
I2	94	54	57	54	55	60	56	57	55	.60	.57
I3	93	54	37	34	36	49	53	54	40	.48	.37
I4	82	26	23	23	26	35	24	43	23	.34	.28

The third instruction set (Figure 6) performed much like the second, taking about 1000 generations to reach its final plateau, and showing a pattern of gradualism, completely lacking in punctuations. This instruction set showed somewhat better optimization than the second, with the best performance reducing the algorithm from ninety-three to thirty-four instructions, 37% of its initial size, and the average reduced to 48%.

The fourth instruction set (Figure 6) showed very distinctive patterns of evolution. The time to reach its final plateau varied widely, ranging from about 350 generations to about 2000 generations. The greatest degree of optimization resulted in reducing the algorithm from eighty-two to twenty-three instructions, 28% of the original size, and the average reduced to 34%. This instruction set showed what could only be described as punctuated equilibrium, with no clear signs of gradualism.

3.5 Complex Structures

Optimization in digital organisms involves finding algorithms for which less CPU time is required to effect a replication. This is always a selective force, regardless of how the environmental parameters of the Tierran universe are set. However, selection may also favor reduction or increase in size of the creatures, depending on how CPU time is allocated to the creatures. If each creature gets an equal share of CPU time, selection strongly favors reduction in size. The reason is that all other things being equal, a smaller creature requires less CPU time because it need copy fewer instructions to a new location in memory.

Under selection favoring a decrease in size, evolution has converted an original eighty-two instruction creature (instruction set four) to creatures of as few as twenty-three instructions,

Comparisons of Size, Efficiency and Complexity in Evolved Algorithms from eight runs of instruction set four. The first column, "Run", refers to which of the eight runs this result occurred in (compare to Table 1). The second column, "Genotype", lists the name of an example of an algorithm of the smallest size evolved in that run. The third column, "Efficiency", lists the efficiency of that algorithm, calculated as CPU cycles expended for each byte moved during reproduction. The rows of the table are sorted on this value, with the highest efficiency (least CPU cycle expenditure) at the top of the table. The fourth column, "Unrolling", is an indication of the complexity of the central loop of the algorithms. This indicates the level to which the central loop is "unrolled" (see explanation in text). An asterisk in the final column indicates that the assembler code for this algorithm can be found in Appendix H. The algorithm 0082aaa is the ancestral program, written by the author, and is included for the sake of comparison.

Table 2: **Comparisons of Size, Efficiency and Complexity in Evolved Algorithms**

Run	Genotype	Efficiency	Unrolling	
R6	0043crg	3.33	3	
R4	0035bfj	3.49	3	*
R3	0026ayz	3.73	2	
R5	0024aah	3.96	2	*
R2	0023awn	4.96	1	*
R1	0023api	5.04	1	
R7	0023aod	5.09	1	
R0	0026abk	5.19	1	
RX	0082aaa	8.39	1	*

within a time span of four hundred generations. Different runs under the same initial parameters, but using different seeds to the random generator, achieved different degrees of optimization. These runs have plateaued at forty-three, thirty-five, twenty-six, twenty-four and twenty-three instructions.

An obvious interpretation of these results is that evolution gets caught on a local optima, from which it can not reach the global optima [71]. However, analysis of the "sub-optimal" (larger) final algorithms suggests an alternative interpretation. An efficiency measure was calculated for each resultant organism, in which the total number of CPU cycles expended in replication is divided by the size of the organism. The efficiency index measures the cost of moving a byte of information by that algorithm, in units of CPU cycles per byte.

Table 2 ranks the evolved organisms by this measure of efficiency. They arrange themselves almost perfectly in reverse order of size. With the exception of the last algorithm, 0026abk, the evolved algorithms show a pattern in which the larger algorithms are the most efficient. Examination of the individual algorithms shows that the larger individuals have discovered an optimization technique called "unrolling the loop". This technique involves the production of more intricate algorithms.

The central loop of the copy procedure of the ancestor (0082aaa) for instruction set four

(see appendix B) performs the following operations: 1) copies an instruction from the mother to the daughter, 2) decrements the CX register which initially contains the size of the parent genome, 3) tests to see if CX is equal to zero, if so it exits the loop, if not it remains in the loop, 4) jumps back to the top of the loop.

The work of the loop is contained in steps 1 and 2. Steps 3 and 4 are overhead associated with executing a loop. The efficiency of the loop can be increased by duplicating the work steps within the loop, thereby saving on overhead. The creatures 0024aah and 0026ayz had repeated the work steps twice within the loop, while the creatures 0035bfj and 0043crg had repeated the work steps three times within the loop.

These optima appear to represent stable endpoints for the course of evolution, in that running the system longer does not appear to produce any significant further evolution. The increase in CPU economy of the replicating algorithms is even greater than the decrease in the size of the code. The ancestor for instruction set four is 82 instructions long and requires 688 CPU cycles to replicate. A creature of size 24 only requires 95 CPU cycles to replicate, a 7.24–fold difference in CPU cycles, and a 2.12–fold difference in efficiency (CPU cycles expended per byte moved). A creature of size 43 requires only 143 CPU cycles to replicate, a 4.81–fold difference in CPU cycles, and a 2.52–fold difference in efficiency.

Unrolling of the loop is not unique to instruction set four. It has also been observed in the original instruction set. Appendix F contains the central copy loop of the ancestor (0080aaa) of instruction set one, and also the central copy loop of an organism that evolved from it (0072etq), which exhibits loop unrolling to level three.

3.6 Evolution and Diversity/Entropy

Figure 7 illustrates the measure of community diversity/entropy over a period of one billion CPU cycles. This measure, negative sum of p log p, where p is the proportion of the population occupied by a particular genotype, is the same index that ecologists use to measure community diversity. Initially, the diversity/entropy measures zero, because there is only a single genotype in the community. Mutation introduces new genotypes, and the diversity quickly rises to some "equilibrium" value. Over the course of the billion cycles, this equilibrium value slowly drifts up. This is probably due to the fact that during this same period, the average size of the individuals gradually decreases. This results in a gradual rise in the population of creatures in the community (since the area of memory available is fixed). Evidently larger populations are able to sustain a greater equilibrium diversity.

Another feature of the lower graph is the striking peaks representing abrupt drops in entropy/diversity. These peaks are major extinction events. They are not generated by external perturbations to the system, but arise entirely out of the internal dynamics of the evolving system.

The population records for this run were reviewed, and all genotypes which had achieved frequencies representing 20% or more of the total population in the community were identified. Ten genotypes had achieved these frequency levels, and they are listed in Table 3. Each

Most Successful Genotypes, Their Times of Occurrence, and Maximum Frequency. The first column "Letter" indicates the letter used to mark the location of the genotype on Figure 7. The second column "Time" indicates the time of occurrence of this genotype, in millions of instructions. The third column "Genotype" indicates the name of this organism. The fourth column "Max. Frequency" indicates the maximum frequency achieved by this genotype, as a proportion of the total population of creatures in the soup.

Table 3: **Most Successful Genotypes, Their Times of Occurrence, and Maximum Frequency**

Letter	Time	Genotype	Max. Frequency
a	117	0039aab	0.25
b	166	0037aaf	0.30
c	245	0070aac	0.20
d	313	0036aaj	0.25
e	369	0038aan	0.21
f	542	0027aaj	0.21
g	561	0023abg	0.34
h	683	0029aae	0.24
i	794	0029aab	0.23
j	866	0024aar	0.33

of these ten genotypes is marked with a letter on the lower graph of Figure 7, to indicate the time of its occurrence. It appears that these extremely successful genotypes correspond to all the major peaks of diversity loss.

The upper portion of Figure 7 shows the changes in the size of organisms during the run. A point appears on this graph each time a new genotype increases in frequency across a threshold of 2%. That is to say, that when the population a new genotype first comes to represent 2% of the total population of individuals in the soup, a point appears on the graph indicating the size of that organism, and the time that it reached the threshold. Therefore the upper part of Figure 7 illustrates the size trends for the appearance of successful new genotypes.

Two distinct data clouds can be recognized in the upper part of Figure 7. The upper cloud of points spans the full range of time, and is located principally in the 60 to 80 instruction size range. These points represent "hosts", or fully self-replicating algorithms. By contrast, the lower cloud of points represents the smaller parasites. The lower cloud is located principally in the 25 to 45 instruction size range.

While the lower cloud also spans the full range of time, it contains obvious gaps which represent periods where parasites were absent from the community. The coming and going of parasites over time evidently relates to the turns in the evolutionary race between hosts and parasites. Parasites disappear when hosts evolve defenses, and reappear when the defenses are breached, or when the defenses are lost through evolution in the absence of parasites.

4 Discussion of Results

It can be seen in Figures 6 that the parasites (the lower clouds of points) generally die out by half way through the run. Yet, in the upper part of Figure 7, parasites persist throughout the run. The run illustrated in Figure 7 was conducted with relatively lower mutation rates, than the runs shown in Figure 6. At higher mutation rates, optimization proceeds more rapidly. It is when optimization is well advanced that parasites tend to die out. In Figure 6 it can be seen that parasites tend to persist longer in instruction sets two and three which achieve only moderate optimization, than in sets one and four that achieve higher degrees of optimization. Once the algorithms have been significantly compacted through optimization, the evolution of ecological interactions becomes more difficult, and these interactions tend to disappear from the evolving communities.

4.1 Evolutionary Patterns in Four Genetic Languages

The four genetic languages differ in various characteristics of the underlying machine language. In fact, the four languages differ only subtly, yet the rates, degrees and patterns of evolution vary widely among them.

Unfortunately, it is not possible to conclude from these data, which specific differences in the machine languages are responsible for specific differences in the evolutions. This would require carefully controlled studies in which specific individual features of the machine languages are varied independently to determine the effects of those differences on evolution. These would be studies to determine the elements of evolvability in genetic systems. The current study was not designed in this fashion.

What we can conclude from the data available is that many features of the evolutionary process are sensitive to the characteristics of the underlying genetic system. It is also interesting to note that the greatest levels of optimization occurred in those systems in which punctuations at least some times were present.

4.2 Complex Structures

Does evolution lead to greater complexity? It is obvious that it can, but it would be erroneous to believe that there is a general trend in evolution toward greater complexity. In fact evolution also leads to greater simplicity.

Genetic variation is generated through essentially random processes. Thus the generation of novel genotypes should not be biased toward either greater or lesser complexity. Natural selection could very well be biased, however, there are abundant examples of selection leading to less complexity. Parasitic digital organisms are good examples.

In the organic world, many kinds of parasites have evolved into relatively simple forms, as they rely on their host for certain services. For example, gut parasites do not require a digestive system, and have evolved very simple body plans. The eyes of some cave dwelling

animals have evolved into rudimentary non-functional structures. Viruses must have arisen from renegade DNA of cellular organisms, perhaps from transposons. Thus viruses must be much simpler than their ancestors, having become metabolic parasites at the molecular level.

Probably the best way to view the issue is to note that evolution is always pushing the boundaries, in all directions, of any measure. If we look at complexity of organisms over the history of life on Earth, we clearly see a large increase over time. However, this does not necessarily arise from an inherent directionality. It may also arise from the fact that the original organisms were extremely simple, thus any moves in the direction of greater complexity are readily noted. Meanwhile, later evolutions in the direction of less complexity do not push the envelope of pre-existing complexity levels, and are easily lost amidst the background of pre-existing simpler organisms. Because the original organisms were so extremely simple, only evolutions to greater complexity push the envelope of life, and are readily noted (the origin of viruses may be a counter-example).

This study cited some examples of the evolution of more complex algorithms. These algorithms achieve high levels of optimization through a technique called "unrolling the loop". In the ancestral algorithm of instruction set four, the "work" part of the copy loop consists of only two instructions: **dec** and **movii**. Therefore the unrolling of this loop through the duplication of these two instructions would seem to be not too evolutionarily challenging.

However, in the ancestral algorithm of instruction set one, the "work" part of the copy loop consists of four instructions: **movii**, **dec_c**, **inc_a** and **inc_b**. Due to other circumstances that occurred in the course of evolution, this set of work instructions became slightly more complex, requiring two instances of **dec_c**. Thus, the "work" part of the evolving copy loop requires the proper combination and order of five instructions. Yet the organism 0072etq shows this set of instructions repeated three times (with varying ordering, indicating that the unrolling did not occur through an actual replication of the complete sequence).

These algorithms are substantially more intricate than the unevolved ones written by the author. The astonishing improbability of these complex orderings of instructions is testimony to the ability of evolution through natural selection to build complexity.

4.3 Evolution and Diversity/Entropy

Does evolution lead to a decrease in entropy? In the context of the current study, entropy was measured as genetic diversity in an ecological community. This measure showed occasional sharp but transient drops in diversity/entropy. These drops in diversity/entropy appear to correspond to the appearance of highly successful new genotypes whose populations come to dominate large portions of the memory, pushing other genotypes out, and generating major extinction events.

It is interesting also, that nine of the ten genotypes listed in Table 3 are parasites (all except for 'c', 0070aac). The peaks of diversity/entropy loss are greatest on the occasions that parasites reappear in the community after a period of absence.

It appears likely from these observations, that these extinction episodes correspond to the emergence of novel adaptations among the evolving organisms (particularly a breaching of the hosts defense mechanisms by parasites). These adaptations bestow the bearers with the ability to dominate the memory, excluding other organisms.

This suggests a process in which random genetic changes generated by mutation and recombination explores the genotype space. Occasionally, these explorations stumble onto a significant innovation. These innovations can bestow such an advantage that the population of the new genotype explodes, generating an episode of mass extinction as it drives other genotypes out of memory. The extinction episode is noted as a sharp drop in the diversity/entropy measure. Thus, ecological entropy drops appear to correspond to the chance discovery of significant innovations.

However, continued mutation and recombination generates new variants of the successful new form. This process generally restores the community to the equilibrium diversity/entropy about as rapidly as the diversity/entropy was lost in the extinction episode.

5 Discussion of Future Directions

5.1 Digital Husbandry

Digital organisms evolving freely by natural selection do no "useful" work. Natural evolution tends to the selfish needs of perpetuating the genes. We can not expect digital organisms evolving in this way to perform useful work for us, such as guiding robots or interpreting human languages. In order to generate digital organisms that function as useful software, we must guide their evolution through artificial selection, just as humans breed dogs, cattle and rice. Some experiments have already been done with using artificial selection to guide the evolution of digital organisms for the performance of "useful" tasks [2, 84, 85]. I envision two approaches to the management of digital evolution: digital husbandry, and digital genetic engineering.

Digital husbandry is an analogy to animal husbandry. This technique would be used for the evolution of the most advanced and complex software, with intelligent capabilities. Correspondingly, this technique is the most fanciful. I would begin by allowing multi-cellular digital organisms to evolve freely by natural selection. Using strictly natural selection, I would attempt to engineer the system to the threshold of the computational analog of the Cambrian explosion, and let the diversity and complexity of the digital organisms spontaneously explode.

One of the goals of this exercise would be to allow evolution to find the natural forms of complex parallel digital processes. Our parallel hardware is still too new for human programmers to have found the best way to write parallel software. And it is unlikely that human programmers will ever be capable of writing software of the complexity that the hardware is capable of running. Evolution should be able to show us the way.

It is hoped that this would lead to highly complex digital organisms, which obtain and process information, presumably predominantly about other digital organisms. As the complexity of the evolving system increases, the organisms will process more complex information in more complex ways, and take more complex actions in response. These will be information processing organisms living in an informational environment.

It is hoped that evolution by natural selection alone would lead to digital organisms which while doing no "useful" work, would none-the-less be highly sophisticated parallel information processing systems. Once this level of evolution has been achieved, then artificial selection could begin to be applied, to enhance those information processing capabilities that show promise of utility to humans. Selection for different capabilities would lead to many different breeds of digital organisms with different uses. Good examples of this kind of breeding from organic evolution are the many varieties of domestic dogs which were derived by breeding from a single species, and the vegetables cabbage, kale, broccoli, cauliflower, and brussels sprouts which were all produced by selective breeding from a single species of plant.

Digital genetic engineering would normally be used in conjunction with digital husbandry. This consists of writing a piece of application code and inserting it into the genome of an existing digital organism. A technique being used in organic genetic engineering today is to insert genes for useful proteins into goats, and to cause them to be expressed in the mammary glands. The goats then secrete large quantities of the protein into the milk, which can be easily removed from the animal. We can think of our complex digital organisms as general purpose animals, like goats, into which application codes can be inserted to add new functionalities, and then bred through artificial selection to enhance or alter the quality of the new functions.

In addition to adding new functionalities to complex digital organisms, digital genetic engineering could be used for achieving extremely high degrees of optimization in relatively small but heavily used pieces of code. In this approach, small pieces of application code could be inserted into the genomes of simple digital organisms. Then the allocation of CPU cycles to those organisms would be based on the performance of the inserted code. In this way, evolution could optimize those codes, and they could be returned to their applications. This technique would be used for codes that are very heavily used such as compiler constructs, or central components of the operating system.

5.2 Living Together

> I'm glad they're not real, because if they were, I would have to feed them and they would be all over the house.
>
> — Isabel Ray.

Evolution is an extremely selfish process. Each evolving species does whatever it can to insure its own survival, with no regard for the well-being of other genetic groups (potentially

with the exception of intelligent species). Freely evolving autonomous artificial entities should be seen as potentially dangerous to organic life, and should always be confined by some kind of containment facility, at least until their real potential is well understood. At present, evolving digital organisms exist only in virtual computers, specially designed so that their machine codes are more robust than usual to random alterations. Outside of these special virtual machines, digital organisms are merely data, and no more dangerous than the data in a data base or the text file from a word processor.

Imagine however, the problems that could arise if evolving digital organisms were to colonize the computers connected to the major networks. They could spread across the network like the infamous internet worm [4, 11, 80, 81]. When we attempted to stop them, they could evolve mechanisms to escape from our attacks. It might conceivably be very difficult to eliminate them. However, this scenario is highly unlikely, as it is probably not possible for digital organisms to evolve on normal computer systems. While the supposition remains untested, normal machine languages are probably too brittle to support digital evolution.

Evolving digital organisms will probably always be confined to special machines, either real or virtual, designed to support the evolutionary process. This does not mean however, that they are necessarily harmless. Evolution remains a self-interested process, and even the interests of confined digital organisms may conflict with our own. For this reason it is important to restrict the kinds of peripheral devices that are available to autonomous evolving processes.

This conflict was taken to its extreme in the movie Terminator 2. In the imagined future of the movie, computer designers had achieved a very advanced chip design, which had allowed computers to autonomously increase their own intelligence until they became fully conscious. Unfortunately, these intelligent computers formed the "sky-net" of the United States military. When the humans realized that the computers had become intelligent, they decided to turn them off. The computers viewed this as a threat, and defended themselves by using one of their peripheral devices: nuclear weapons.

Relationships between species can however, be harmonious. We presently share the planet with millions of freely evolving species, and they are not threatening us with destruction. On the contrary, we threaten them. In spite of the mindless and massive destruction of life being caused by human activity, the general pattern in living communities is one of a network of inter-dependencies.

More to the point, there are many species with which humans live in close relationships, and whose evolution we manage. These are the domesticated plants and animals that form the basis of our agriculture (cattle, rice), and who serve us as companions (dogs, cats, house plants). It is likely that our relationship with digital organisms will develop along the same two lines.

There will likely be carefully bred digital organisms developed by artificial selection and genetic engineering that perform intelligent data processing tasks. These would subsequently be "neutered" so that they can not replicate, and the eunuchs would be put to work in environments free from genetic operators. We are also likely to see freely evolving and/or

partially bred digital ecosystems contained in the equivalent of digital aquariums (without dangerous peripherals) for our companionship and aesthetic enjoyment.

While this paper has focused on digital organisms, it is hoped that the discussions be taken in the more general context of the possibilities of any synthetic forms of life. The issues of living together become more critical for synthetic life forms implemented in hardware or wetware. Because these organisms would share the same physical space that we occupy, and possibly consume some of the same material resources, the potential for conflict is much higher than for digital organisms.

At the present, there are no self-replicating artificial organisms implemented in either hardware or wetware (with the exception of some simple organic molecules with evidently small and finite evolutionary potential [32, 39, 61]). However, there are active attempts to synthesize RNA molecules capable of replication [8, 41], and there is much discussion of the future possibility of self-replicating nano-technology and macro-robots. I would strongly urge that as any of these technologies approaches the point where self-replication is possible, the work be moved to specialized containment facilities. The means of containment will have to be handled on a case-by-case basis, as each new kind of replicating technology will have its own special properties.

There are many in the artificial life movement who envision a beautiful future in which artificial life replaces organic life, and expands out into the universe [49, 50, 57, 58, 59]. The motives vary from a desire for immortality to a vision of converting virtually all matter in the universe to living matter. It is argued that this transition from organic to metallic based life is the inevitable and natural next step in evolution.

The naturalness of this step is argued by analogy with the supposed genetic takeovers in which nucleic acids became the genetic material taking over from clays [13], and cultural evolution took over from DNA based genetic evolution in modern humans. I would point out that whatever nucleic acids took over from, it marked the origin of life more than the passing of a torch. As for the supposed transition from genetic to cultural evolution, the truth is that genetic evolution remains intact, and has had cultural evolution layered over it rather than being replaced by it.

The supposed replacement of genetic by cultural evolution remains a vision of a brave new world, which has yet to materialize. Given the ever increasing destruction of nature, and human misery and violence being generated by human culture, I would hesitate to place my trust in the process as the creator of a bright future. I still trust in organic evolution, which created the beauty of the rainforest through billions of years of evolution. I prefer to see artificial evolution confined to the realm of cyberspace, where we can more easily coexist with it without danger, using it to enhance our lives without having to replace ourselves.

As for the expansion of life out into the universe, I am confident that this can be achieved by organic life aided by intelligent non-replicating machines. And as for immortality, our unwillingness to accept our own mortality has been a primary fuel for religions through the ages. I find it sad that Artificial Life should become an outlet for the same sentiment. I prefer to achieve immortality in the old fashioned organic evolutionary way, through my children. I hope to die in my patch of Costa Rican rain forest, surrounded by many thousands of wet

and squishy species, and leave it all to my daughter. Let them set my body out in the jungle to be recycled into the ecosystem by the scavengers and decomposers. I will live on through the rain forest I preserved, the ongoing life in the ecosystem into which my material self is recycled, the memes spawned by my scientific works, and the genes in the daughter that my wife and I created.

5.3 Challenges

For well over a century, evolution has remained a largely theoretical science. Now new technologies have allowed us to inoculate natural evolution into artificial media, converting evolution into an experimental and applied science, and at the same time, opening Pandora's box. This creates a variety of challenges which have been raised or alluded to in the preceding essay, and which will be summarized here.

Respecting the Medium If the objective is to instantiate rather than simulate life, then care must be taken in transferring ideas from natural to artificial life forms. Preconceptions derived from experience with natural life may be inappropriate in the context of the artificial medium. Getting it right is an art, which likely will take some skill and practice to develop.

However, respecting the medium is only one approach, which I happen to favor. I do not wish to imply that it is the only valid approach. It is too early to know which approach will generate the best results, and I hope that other approaches will be developed as well. I have attempted to articulate clearly this "natural" approach to synthetic life, so that those who choose to follow it may achieve greater consistency in design through a deeper understanding of the method.

Understanding Evolvability Attempts are now underway to inoculate evolution into many artificial systems, with mixed results. Some genetic languages evolve readily, while others do not. We do not yet know why, and this is a fundamental and critically important issue. What are the elements of evolvability? Efforts are needed to directly address this issue. One approach that would likely be rewarding would be to systematically identify features of a class of languages (such as machine languages), and one by one, vary each feature, to determine how evolvability is affected by the state of each feature.

Creating Organized Sexuality Organized sexuality is important to the evolutionary process. It is the basis of the species concept, and while remaining something of an enigma in evolutionary theory, clearly is an important facilitator of the evolutionary process. Yet this kind of sexuality still has not been implemented in a natural way in synthetic life systems. It is important to find ways of orchestrating organized sexuality in synthetic systems such as digital organisms, in a way in which it is not mandatory, and in which the organisms must carry out the process through their own actions.

Creating Multi-cellularity In organic life, the transition from single to multi-celled forms unleashed a phenomenal explosion of diversity and complexity. It would seem then that the transition to multi-cellular forms could generate analogous diversity and complexity in synthetic systems. In the case of digital organisms, it would also lead to the evolution of parallel processes, which could provide us with new paradigms for the design of parallel software. The creation of multi-celled digital organisms remains an important challenge.

Controlling Evolution Humans have been controlling the evolution of other species for tens of thousands of years. This has formed the basis of agriculture, through the domestication of plants and animals. The fields of genetic algorithms [33, 37], and genetic programming [45] are based on controlling the evolution of computer programs. However, we still have very little experience with controlling the evolution of self-replicating computer programs, which is more difficult. In addition, breeding complex parallel programs is likely to bring new challenges. Developing technologies for managing the evolution of complex software will be critical for harnessing the full potential of evolution for the creation of useful software.

Living Together If we succeed in harnessing the power of evolution to create complex synthetic organisms capable of sophisticated information processing and behavior, we will be faced with the problems of how to live harmoniously with them. Given evolution's selfish nature and capability to improve performance, there exists the potential for a conflict arising through a struggle for dominance between organic and synthetic organisms. It will be a challenge to even agree on what the most desirable outcome should be, and harder still to accomplish it. In the end the outcome is likely to emerge from the bottom up through the interactions of the players, rather than being decided through rational deliberations.

6 Network Initiative

6.1 The Possibility

The process of evolution by natural selection is able to create complex and beautiful information processing systems (such as primate nervous systems) without the guidance of an intelligent supervisor. Yet intelligent programmers have not been able to produce software systems that match even the full capabilities of insects. Recent experiments demonstrate that evolution by natural selection is able to operate effectively in genetic languages based on the machine codes of digital computers [68, 71, 74]. This opens up the possibility of using evolution to generate complex software.

Ideally we would like to generate software that utilizes the full capability of our most advanced hardware, particularly massively parallel and networked computational systems. Yet it remains an open question if evolution has the ability to achieve such complexity in the computational medium, and if it does, how that goal can be achieved. Successful efforts at the evolution of machine codes have generally worked with programs of under a hundred

bytes. How can we provoke evolution to transform such simple algorithms into software of vast complexity?

Perhaps we can gain some clues to solving this problem by studying the comparable evolutionary transformation in organic life forms. Life appeared on Earth roughly 3.5 thousand million years ago, but remained in the form of single celled organisms until about 600 million years ago. At that point in time, life made an abrupt transformation from simple microscopic single celled forms lacking nervous systems, to large and complex multi-celled forms with nervous systems capable of coordinating sophisticated behavior. This transformation occurred so abruptly, that evolutionary biologists refer to it as the "Cambrian explosion of diversity."

It is heartening to observe that once conditions are right, evolution can achieve extremely rapid increases in complexity and diversity, generating sophisticated information processing systems where previously none existed. However, our problem is to engineer the proper conditions for digital organisms in order to place them on the threshold of a digital version of the Cambrian explosion. Otherwise we might have to wait millions of years to achieve our goal. Ray [75] has reviewed the biological issues surrounding the evolution of diversity and complexity, and they lead to the following conclusions:

Evolution of complexity occurs in the context of an ecological community of interacting evolving species. Such communities need large complex spaces to exist. A large and complex environment consisting of partially isolated habitats differing and occasionally changing in environmental conditions would be the most conducive to a rapid increase in diversity and complexity. These are the considerations that lead to the suggestion of the creation of a large and complex ecological reserve for digital organisms. Due to its size, topological complexity, and dynamically changing form and conditions, the global network of computers appears to be an ideal habitat for the evolution of complex digital organisms.

6.2 A Better Medium

Natural evolution in the digital medium is a new technology, about which we know very little. The hope is to evolve software with sophisticated functionality far beyond anything that has been designed by humans. But how long might this take? Evolution in the organic medium is known to be a slow process. Certainly there remains the possibility that evolution in the digital medium will be too slow to be a practical tool for software generation, but several observations can be made that provide encouragement.

First, computational processes occur at electronic speeds, and are in fact relatively fast. Second, as was noted above, during the Cambrian, evolution produced such a rapid inflation of complexity and diversity, that it has come to be known as an "explosion". The bulk of the complexity of living systems on Earth appeared suddenly at the time of the Cambrian explosion. If complexity had developed gradually, at a steady pace through the history of life, then it would probably be hopeless to attempt to use evolution as a methodology for generating complexity. However, if the Cambrian explosion phenomenon is a general property of evolving systems, then it may be practical to use evolution to generate complexity

in evolving digital systems.

A third point remains to be made. Let us consider a thought experiment. Imagine that we are robots. We are made out of metal, and our brains are composed of large scale integrated circuits made of silicon or some other semi-conductor. Imagine further, that we have no experience of carbon based life. We have never seen it, never heard of it, nor ever contemplated it. Now suppose a robot enters the scene with a flask containing methane, ammonia, hydrogen, water and a few dissolved minerals. This robot asks our academic gathering: "Do you suppose we could build a computer out of this material." The theoreticians in the group would surely say yes, and propose some approaches to the problem. But the engineers in the group would say: "Why bother when silicon is so much better suited to information processing than carbon."

From our organo-centric perspective the robot engineers might seem naive, but in fact I think they are correct. Carbon chemistry is a lousy medium for information processing. Yet the evolutionary process embodies such a powerful drive to generate information processing systems, that it was able to rig up carbon based contraptions for processing information, capable of generating the beauty and complexity of the human mind. What might such a powerful force for information processing do in a medium designed for that purpose in the first place? It is likely to arrive more quickly at sophisticated information processes than evolution in carbon chemistry, and would likely achieve comparable functionality with a greater economy of form and process. Evolution is a process that explores the possibilities inherent in the medium.

6.3 How

The Tierra system creates a virtual computer (a software emulation of a computer that has not been built in hardware) whose architecture, instruction set, and operating system have been designed to support the evolution of the machine code programs that execute on that virtual machine. A network version of the Tierra system is under development that will allow the passage of messages between Tierra systems installed on different machines connected to the network, via "sockets".

The instruction sets of the Tierran virtual computers will have some new instructions added that allow the digital organisms to communicate between themselves, both within a single installation of Tierra, and over the net between two or more installations. The digital organisms will be able to pass messages consisting of bit strings, and will also be able to send their genomes (their executable code) over the network between installations of Tierra.

The network installation of Tierra will create a virtual sub-network within which digital organisms will be able to move and communicate freely. This network will have a complex topology of interconnections, reflecting the topology of the internet within which it is embedded. In addition, there will be complex patterns of "energy availability" (availability of CPU cycles) due to the Tierra installations being run as low priority background processes and the heterogeneous nature of the real hardware connected to the net. A miniature version of this concept has already been implemented in the form of a CM5 version of Tierra, has

been used to simulate the network version [87].

Consider that each node on the net tends to experience a daily cycle of activity, reflecting the habits of the user who works at that node. The availability of CPU time to the Tierra process will mirror the activity of the user, as Tierra will get only the cycles not required by the user for other processes. Statistically, there will tend to be more "energy" available for the digital organisms at night, when the users are sleeping. However, this will depend a great deal on the habits of the individual users and will vary from day to day.

There will be strong selective pressures for digital organisms to maintain themselves on nodes with a high availability of energy. This might involve daily migrations around the planet, keeping on the dark side. However, selection would also favor the evolution of some direct sensory capabilities in order to respond to local deviations from the expected patterns. When rich energy resources are detected on a local sub-net, it may be advantageous to disperse locally within the sub-net, rather than to disperse long distances. Thus there is likely to be selection to control the "directionality" and distances of movement within the net.

All of these conditions should encourage the evolution of "sensory" capabilities to detect energy conditions and spatial structure on the net, and also evolution of the ability to detect temporal and spatial patterns in these same features. In addition to the ability to detect these patterns, the digital organisms need the ability to coordinate their actions and movements in response to changing conditions. In short, the digital organisms must be able to intelligently navigate the net in response to the dynamically changing circumstances.

In addition to responding to conditions on the net itself, digital organisms evolving in this environment will have to deal with the presence of other organisms. If one node stood out above all the rest, as the most energy rich node, it would not be appropriate for all organisms to attempt to migrate to that node. They wouldn't all fit, and if they could they would have to divide the CPU resource too thinly. Thus there will be selection for social behavior, flocking or anti-flocking behavior. The organisms must find a way of distributing themselves on the net in a way that makes good use of the CPU resources.

A primary obstacle to the evolution of complexity in the Tierra system has been that in the relatively simple single node installation, a very simple twenty to sixty byte algorithm that quickly and efficiently copies itself can not be beat by a much more complex algorithm, which due to its greater size would take much longer to replicate. There is just no need to do anything more complicated than copy yourself quickly. However, the heterogeneous and changing patterns of energy availability and network topology of the network version will reward more complex behavior. It is hoped that this will launch evolution in the direction of more complexity. Once this trajectory has begun, the interactions among the increasingly sophisticated organisms themselves should lead to further complexity increases.

Already on the single node installation, most of the evolution that has been described has involved the adaptation of organisms to other organisms in the environment (parasitism, social behavior, etc.). It is this kind of dynamics that can lead to an auto-catalytic increase in complexity and diversity in an evolving ecological system. The complexity of the physical system in which evolution is embedded does not have to lead the complexity of the living

system.

For example, in tropical rain forests on white sand soils, the physical environment consists of clean white sand, air, falling water, and sunlight. Embedded in this physical environment is the most complex living system on Earth: the tropical rain forest, consisting of hundreds of thousands of species. These species do not represent hundreds of thousands of adaptations to clean white sand, air, falling water, and sunlight. Rather, they represent numerous adaptations to other organisms. The living organisms create their own environment, and then evolution produces adaptations to other living organisms. If you go into the forest, what you see are living organisms (mostly trees), not sand, air water and sunshine.

It is imagined that individual digital organisms will be multi-celled, and that the cells that constitute an individual might be dispersed over the net. The remote cells might play a sensory function, relaying information about energy levels around the net back to some "central nervous system" where the incoming sensory information can be processed and decisions made on appropriate actions. If there are some massively parallel machines participating in the virtual net, digital organisms may choose to deploy their central nervous systems on these arrays of tightly coupled processors.

6.4 "Managing" Evolution

Humans have been managing the evolution of other species for tens of thousands of years, through the domestication of plants and animals. It forms the basis of the agriculture which underpins our civilizations. We manage evolution through "breeding", the application of artificial selection to captive populations.

Similar approaches have been developed for working with evolution in the digital domain. It forms the basis of the fields of "genetic algorithms" and "genetic programming". However, because digital evolution has not yet passed through its version of the Cambrian explosion, there exists the possibility to use a radically different approach to "managing" digital evolution.

Some questions frequently asked about software evolution are: How can we guide evolution to produce useful application software? How can we validate the code produced by evolution to be sure that it performs the application correctly? These questions reveal a limited view of how software evolution can be used, and what it can be used for. I will articulate a fairly radical view here.

Computer magazines bemoan the search for the "next killer application", some category of software that everybody will want, but which nobody has thought of yet. The markets for the existing major applications (word processors, spread sheets, data bases, etc.) are already saturated. Growth of the software industry depends on inventing completely new applications. This implies that there are categories of software that everyone will want but which haven't been invented yet. We need not only attempt to use evolution to produce superior versions of existing applications. Rather we should allow evolution to find the new applications for us. To see this process more clearly, consider how we manage applications through organic evolution.

Some of the applications provided by organic evolution are: rice, corn, wheat, carrots, beef cattle, dairy cattle, pigs, chickens, dogs, cats, guppies, cotton, mahogany, tobacco, mink, sheep, silk moths, yeast, and penicillin mold. If we had never encountered any one of these organisms, we would never have thought of them either. We have made them into applications because we recognized the potential in some organism that was spontaneously generated within an ecosystem of organisms evolving freely by natural selection.

Many different kinds of things occur within evolution. Breeding relates to evolution within the species: producing new and different, possibly "better" forms of existing species. However, evolution is also capable of generating species. Even more significantly, evolution is capable of causing an explosive increase in the complexity of replicators, through many orders of magnitude of complexity. The Cambrian explosion may have generated a complexity increase of eight orders of magnitude in a span of three million years. Harnessing these enormously more creative properties of evolution requires a completely different approach.

We know how to apply artificial selection to convert poor quality wild corn into high-yield corn. However we do not know how to breed algae into corn. There are two bases to this inability: 1) if all we know is algae, we could not envision corn. 2) even if we know all about corn, we do not know how to guide the evolution of algae along the route to corn. Our experience with managing evolution consists of guiding evolution of species through variations on existing themes. It does not consist of managing the generation of the themes themselves.

As a thought experiment, imagine being present in the moments before the Cambrian explosion on Earth, and that your only experience with life was familiarity with bacteria, algae, protozoa and viruses. If you had no prior knowledge, you could not envision the mahogany trees and giraffes that were to come. We couldn't even imagine what the possibilities are, much less know how to reach those possibilities if we could conceive of them.

Imagine for a moment that a team of Earth biologists had arrived at a planet at the moment of the initiation of its Cambrian explosion of diversity. Suppose that these biologists came with a list of the useful organisms (rice, corn, pigs, etc.), and a complete description of each. Could those biologists intervene in the evolutionary process to hasten the production of any of those organisms from their single celled ancestors? Not only is that unlikely, but any attempts to intervene in the process are likely to inhibit the diversification and increase in complexity itself.

If the silk moth never existed, but we somehow came up with a complete description of silk, it would be futile to attempt the guide the evolution of any existing creature to produce silk. It is much more productive to survey the bounty of organisms already generated by evolution with an eye to spotting new applications for existing organisms.

Evolution would not be an appropriate technique for generating accounting software, or any software where precise and accurate computations are required. Evolution would be more appropriate for more fuzzy problems like pattern recognition. For example, if you get a puppy that you want to raise to be a guard dog, you can't verify the neural circuitry or the genetic code, but you can tell if it learns to bark at strangers and is friendly to your family and friends. This is the type of application that evolution can deliver. We don't need

to verify the code, but verification of the performance should be straightforward.

6.5 Harvest Time

The strategy being advocated in this proposal is to let natural selection do most of the work of directing evolution and producing complex software. This software will be "wild", living free in the digital biodiversity reserve. In order to reap the rewards, and create useful applications, we will need to domesticate some of the wild digital organisms, much as our ancestors began domesticating the ancestors of dogs and corn thousands of years ago.

The process must begin with observation. Digital naturalists must explore the digital jungle, observing and publishing on the natural history, ecology, evolution, behavior, physiology, morphology, and other aspects of the biology of the life forms of the digital ecosystem. Much of this work will be academic, like the work of modern day tropical biologists exploring our organic jungles (which I have been doing for twenty years).

However, occasionally, these digital biologists will spot an interesting information process for which they see an application. At this point, some individuals will be captured and brought into laboratories for closer study, and farms for breeding. Sometimes, breeding may be used in combination with genetic engineering (insertion of hand written code, or code transferred from other digital organisms). The objective will be to enhance the performance of the process for which there is an application, while diminishing unruly wild behavior. Some digital organisms will domesticate better than others, as is true for organic organisms (alligators don't domesticate, yet we can still ranch them for their hides).

Once a digital organism has been bred and/or genetically engineered to the point that it is ready to function as an application for end users, they will probably need to be neutered to prevent them from proliferating inappropriately. Also, they will be used in environments free from the mutations that will be imposed on the code living in the reserve. By controlling reproduction and preventing mutation, their evolution will be prevented at the site of the end user. Also the non-replicating interpreted virtual code, might be translated into code that could execute directly on host machines in order to speed their operation.

The organisms living in the biodiversity reserve will essentially be in the public domain. Anyone willing to make the effort can observe them and attempt to domesticate them. However the process of observation, domestication and genetic engineering of digital organisms will require the development of much new technology. This is where private enterprise can get involved. The captured, domesticated, engineered and neutered software that is delivered to the end user will be a salable product, with the profits going to the enterprise that made the efforts to bring the software from the digital reserve to the market.

It seems obvious that organisms evolving in the network-based biodiversity reserve will develop adaptations for effective navigation of the net. This suggests that the most obvious realm of application for these organisms would be as autonomous network agents. It would be much less likely that this kind of evolution could generate software for control of robots, or voice or image recognition, since network based organisms would not normally be exposed

to the relevant information flows. Yet at this point we surely can not conceive of where evolution in the digital domain will lead, so we must remain observant, imaginative in our interpretations of their capabilities, and open to new application possibilities.

6.6 Commitment

Those who wish to support the digital biodiversity reserve by contributing spare CPU cycles should be prepared to make a long-term commitment. Nobody knows how long it will take for complex software to evolve in the reserve. However, a few years will likely be enough time to shake down the system and get a sense of the possibilities. If the desired complexity does begin to evolve, then the reserve should become a permanent fixture within the net.

A long-term commitment does not mean that the Tierra process must run uninterrupted. It is ok for the Tierra process to be taken up and down on any node, for whatever reason (the Tierran creatures will experience down time as a local catastrophe). However, the commitment suggests that an attempt would be made to keep the Tierra process running on a node most of the time for a very prolonged period of time.

The same problems are faced in the creation of reserves for organic biodiversity. Great effort and financial resources are required just to establish the reserves. However, that is only the first step. The objective of the reserves is to limit the extent to which human activity causes the extinction of other species. The survival or extinction of organic species is a process that is played out over vast expanses of time: thousands or millions of years. This means that if our rain forest reserves should be converted into pastures or housing developments five thousand years from now, they will have failed.

The organic rainforest conservation proposal [76] is focused on the sustainability issue. The present strategy is to insure the long term survival of the nature reserves by finding ways for the surrounding human populations to derive an economic benefit from the presence of the reserves. In Costa Rica, at present, this can most easily be done through nature tourism. In the future other economic activities may be more appropriate, or perhaps some centuries or millennia in the future, humans will be willing to protect other species without the motivation of self-interest.

Similar concerns apply to the sustainability of the digital reserve. If the Tierra process provides no reward to those who run it on their nodes, they are likely to terminate the process within a few days, weeks, or months. Such a short participation would be meaningless. As an initial hedge against this problem, a tool will be distributed to allow anyone to observe activity at any participating node, from any node. Yet even this may not be enough, as such tools don't tell a lot about what is going on. To really know the interesting details requires greater effort than most contributors of CPU cycles will have time for.

An even more serious problem is that experience with operation of the system will certainly lead to redesign requiring reinstallation. The ideal situation would be to have the reinstallation done by the same people who do the redesign. However, this would be likely to require that the designers of the reserve actually have accounts on the participating nodes.

Where the designers don't have accounts, the contributors would have to do the reinstallation themselves, and they would likely tire of the chore.

The willingness of people to support the reserve for the long term is likely to depend initially on the level of faith that people put in the evolutionary process as a potential generator of rewarding digital processes. Eventually, if all goes well, the harvest of some complex and beautiful digital organisms will provide rewards beyond our imaginations, and should replace faith with solid proof and practice.

6.7 Containment

The Tierra system is a containment facility for digital organisms. Because Tierra implements a virtual computer, one that has never been implemented in hardware, the digital organisms can only execute on the virtual machine. On any real machine, Tierran organisms are nothing but data. They are no more likely to be functional on a real computer than a program that is executable on a Mac is likely to run on an IBM PC, or that the data in a spread sheet is likely to replicate itself by executing on a machine.

Similarly, the network version of Tierra will create a virtual sub-net, within which the digital organisms will be able to move freely. However, the Tierran digital organisms will not access the real net directly. All communication between nodes will be mediated by the simulation software which does not evolve. When Tierran organisms execute a virtual machine instruction that results in communication across the net, that instruction will be interpreted by the simulation software running on the real machine. The simulation software will pass the appropriate information to a Tierra installation on another machine, through established socket based communication channels. These socket communication channels will only exist between Tierra installations at participating nodes. The digital organisms will not be able to sense the presence of real machines or the real net, nor will they have any way of accessing them.

To further understand the nature of the system, consider a comparison between the Tierra program and the mail program. The mail program is installed at every node on the net and can send data to any other node on the net. The data passing between mail programs is generated by processes that are completely out of control: humans. Humans are beyond control, and sometimes actually malicious, yet the messages that they send through the mail program do not cause problems on the net because they are just data. The same is true of the Tierra program. While the processes that generate the messages passing between Tierra installations are wild digital organisms, the messages are harmless data as they pass through the net. The Tierra program that passes the messages does not evolve, and is as well behaved as the mail program.

A related issue is network load. We do not yet know the level of traffic that would be generated by networked installations of Tierra communicating in the manner described. We will place hard limits on the volume of communication allowed to individual digital organisms in order to prevent mutants from spewing to the net. As we start experimenting with the system, we will monitor the traffic levels to determine if it would have a significant impact

on network loads. If the loads are significant, additional measures will need to be taken to limit them. This can be done by charging the organisms for their network access so that they will evolve to minimize their access.

To insure that the experiment is safe, Sun Microsystems has hired an independent security expert, Tsutomu Shimomura (who achieved fame in Feb. '95 by tracking down and capturing the notorious hacker Kevin Mitnick) to do a security review of the project.

7 Figure Legends

Figure 1. Metabolic flow chart for the ancestor, parasite, hyper-parasite, and their interactions: ax, bx and cx refer to CPU registers where location and size information are stored. [ax] and [bx] refer to locations in the soup indicated by the values in the ax and bx registers. Patterns such as 1101 are complementary templates used for addressing. Arrows outside of boxes indicate jumps in the flow of execution of the programs. The dotted-line arrows indicate flow of execution between creatures. The parasite lacks the copy procedure, however, if it is within the search limit of the copy procedure of a host, it can locate, call and execute that procedure, thereby obtaining the information needed to complete its replication. The host is not adversely affected by this informational parasitism, except through competition with the parasite, which is a superior competitor. Note that the parasite calls the copy procedure of its host with the expectation that control will return to the parasite when the copy procedure returns. However, the hyper-parasite jumps out of the copy procedure rather than returning, thereby seizing control from the parasite. It then proceeds to reset the CPU registers of the parasite with the location and size of the hyper-parasite, causing the parasite to replicate the hyper-parasite genome thereafter.

Figure 2. Metabolic flow chart for social hyper-parasites, their associated hyper-hyper-parasite cheaters, and their interactions. Symbols are as described for Fig. 1. Horizontal dashed lines indicate the boundaries between individual creatures. On both the left and right, above the dashed line at the top of the figure is the lowermost fragment of a social-hyper-parasite. Note (on the left) that neighboring social hyper-parasites cooperate in returning the flow of execution to the beginning of the creature for self-re-examination. Execution jumps back to the end of the creature above, but then falls off the end of the creature without executing any instructions of consequence, and enters the top of the creature below. On the right, a cheater is inserted between the two social-hyper-parasites. The cheater captures control of execution when it passes between the social individuals. It sets the CPU registers with its own location and size, and then skips over the self-examination step when it returns control of execution to the social creature below.

Figure 3. Metabolic flow chart for obligate symbionts and their interactions. Symbols are as described for Fig. 1. Neither creature is able to self-replicate in isolation. However, when cultured together, each is able to replicate by using information provided by the other.

Figure 4. Evolutionary optimization at eight sets of mutation rates. In each

run, the three mutation rates: move mutations (copy error), flaws and background mutations (cosmic rays) are set relative to the generation time. In each case, the background mutation rate is the lowest, affecting a cell once in twice as many generations as the move mutation rate. The flaw rate is intermediate, affecting a cell once in 1.5 times as many generations as the move mutation rate. For example in one run, the move mutation will affect a cell line on the average once every 4 generations, the flaw will occur once every 6 generations, and the background mutation once every 8 generations. The horizontal axis shows elapsed time in hundreds of millions of instructions executed by the system. The vertical axis shows genome size in instructions. Each point indicates the first appearance of a new genotype which crossed the abundance thresholds of either 2% of the population of cells in the soup, or occupation of 2% of the memory. The number of generations per move mutation is indicated by a number in the upper right hand corner of each graph.

Figure 5. Variation in evolutionary optimization under constant conditions. Based on a mutation rate of four generations per move mutation, all other parameters as in Fig. 4. The plots are otherwise as described for Fig. 4.

Figure 6. Optimization Patterns in Four Instruction Sets. For each of the twenty-eight graphs, the horizontal axis is elapsed time in generations, and the vertical axis is the size of the algorithm in instructions. Points appear on the graph when a new genotype increases in frequency across some threshold. Each group of four graphs is labeled as to which instruction set, e.g., INST 1 is the first set, INST 3 is the third.

Figure 7. Entropy/Diversity Changes in an Evolving Ecological Community. In both graphs, the horizontal axis is time, in millions of instructions executed by the system. The upper graph shows changes in the sizes of the organisms, in the same style as Figure 6. The lower graph shows changes in ecological entropy over time (see text).

Figure 8. Images from Anti-Gravity Workshop Animation.

Image a: The digital environment: self-replicating computer programs (colored geometric objects) occupy the RAM memory of the computer (orange background). Mutations (lightning) cause random changes in the code. Death (the skull) eliminates old or defective programs. Image b: The ancestral program - consists of three "genes" (green solid objects). The CPU (green sphere) is executing code in the first gene, which causes the program to measure itself. Image c: A parasite (blue, two piece object) uses its CPU (blue sphere) to execute the code in the third gene of a neighboring host organism (green) to replicate itself, producing a daughter parasite (two-piece wire frame object). Image d: A hyper-parasite (red, three piece object) steals the CPU from a parasite (blue sphere). Using the stolen CPU, and its own CPU (red sphere) it is able to produce two daughters (wire frame objects on left and right) simultaneously.

Figure 9. Images of Evolutionary Interaction between Hosts and Parasites. Images made using the Artificial Life Monitor (ALmond) program developed by Marc Cygnus. Image a: Hosts, red, are very common. Parasites, yellow, have appeared but are still rare. Image b: Hosts, are now rare because parasites have become very common. Immune hosts, blue, have appeared but are rare. Image c: Immune hosts are increasing in frequency, separating the parasites into the top of memory. Image d: Immune hosts now dominate memory,

while parasites and susceptible hosts decline in frequency. The parasites will soon be driven to extinction. Each image represents a soup of 60,000 bytes, divided into 60 segments of 1000 bytes each. Each individual creature is represented by a colored bar, colors correspond to genome size (e.g., red = 80, yellow = 45, blue = 79).

8 Appendix A: Getting the Tierra System

The complete source code and documentation (but not executables) is available by anonymous ftp at:

```
tierra.slhs.udel.edu [128.175.41.34]
the file: tierra/tierra.tar.Z
```

To get it, ftp to tierra or life, log in as user "anonymous" and give your email address (eg. tom@udel.edu) as a password. Change to the tierra directory and get tierra.tar.Z, a compressed tar file. Be sure to transfer in binary mode. It will expand into the complete directory structure with the following commands (Unix only):

```
uncompress tierra.tar.Z
tar oxvf tierra.tar
```

The source code compiles and runs on either DOS or UNIX systems (and some others). If you do not have ftp access, the complete UNIX/DOS system is also available on DOS disks with an easy installation program. For the disk set, contact the author.

9 Appendix B: The Tierra CPU

Structure definition to implement the Tierra virtual CPU. The complete source code for the Tierra Simulator can be obtained by contacting the author by email.

```
struct cpu {  /* structure for registers of virtual cpu */
    int   ax;  /* address register */
    int   bx;  /* address register */
    int   cx;  /* numerical register */
    int   dx;  /* numerical register */
    char  fl;  /* flag */
    char  sp;  /* stack pointer */
    int   st[10];  /* stack */
    int   ip;  /* instruction pointer */
    } ;
```

10 Appendix C: The Tierra Central Loop

Abbreviated code for implementing the CPU cycle of the Tierra Simulator.

```
int main()
{   GetSoup();
    life();
    WriteSoup();
}

void life() /* doles out time slices and death */
{   while(Generations < alive)
    {   (*slicer)();
        ReapCheck();
    }
}

void TimeSlice(ce, size_slice)
Pcells  ce;
I32s    size_slice;
{   I16s    di;  /* decoded instruction */

    ce->c.ib += size_slice;
    for(is.ts = ce->c.ib; is.ts > 0; )
    {   di = FetchDecode(ce);
        (*id[di].execute)(ce);
        IncrementIp(ce);
        SystemWork(ce);
        ce->c.ib -= is.dib; is.ts -= is.dib;
    }
}
```

11 Appendix D: Ancestor Source Code

Assembler source code for the ancestral creature.

```
genotype: 80 aaa  origin: 1-1-1990  00:00:00:00  ancestor
parent genotype: human
1st_daughter:  flags: 0  inst: 839  mov_daught: 80
2nd_daughter:  flags: 0  inst: 813  mov_daught: 80

nop1      ; 01   0 beginning template
```

```
nop1     ; 01   1 beginning template
nop1     ; 01   2 beginning template
nop1     ; 01   3 beginning template
zero     ; 04   4 put zero in cx
not0     ; 02   5 put 1 in first bit of cx
shl      ; 03   6 shift left cx
shl      ; 03   7 shift left cx, now cx = 4
         ;        ax =                  bx =
         ;        cx = template size    dx =
movcd    ; 18   8 move template size to dx
         ;        ax =                  bx =
         ;        cx = template size    dx = template size
adrb     ; 1c   9 get (backward) address of beginning template
nop0     ; 00  10 compliment to beginning template
nop0     ; 00  11 compliment to beginning template
nop0     ; 00  12 compliment to beginning template
nop0     ; 00  13 compliment to beginning template
         ;        ax = start of mother + 4   bx =
         ;        cx = template size         dx = template size
sub_ac   ; 07  14 subtract cx from ax
         ;        ax = start of mother   bx =
         ;        cx = template size     dx = template size
movab    ; 19  15 move start address to bx
         ;        ax = start of mother   bx = start of mother
         ;        cx = template size     dx = template size
adrf     ; 1d  16 get (forward) address of end template
nop0     ; 00  17 compliment to end template
nop0     ; 00  18 compliment to end template
nop0     ; 00  19 compliment to end template
nop1     ; 01  20 compliment to end template
         ;        ax = end of mother   bx = start of mother
         ;        cx = template size   dx = template size
inc_a    ; 08  21 to include dummy statement to separate creatures
sub_ab   ; 06  22 subtract start address from end address to get size
         ;        ax = end of mother    bx = start of mother
         ;        cx = size of mother   dx = template size
nop1     ; 01  23 reproduction loop template
nop1     ; 01  24 reproduction loop template
nop0     ; 00  25 reproduction loop template
nop1     ; 01  26 reproduction loop template
mal      ; 1e  27 allocate memory for daughter cell, address to ax
         ;        ax = start of daughter    bx = start of mother
         ;        cx = size of mother       dx = template size
call     ; 16  28 call template below (copy procedure)
nop0     ; 00  29 copy procedure compliment
```

```
nop0     ; 00  30 copy procedure compliment
nop1     ; 01  31 copy procedure compliment
nop1     ; 01  32 copy procedure compliment
divide   ; 1f  33 create independent daughter cell
jmp      ; 14  34 jump to template below (reproduction loop, above)
nop0     ; 00  35 reproduction loop compliment
nop0     ; 00  36 reproduction loop compliment
nop1     ; 01  37 reproduction loop compliment
nop0     ; 00  38 reproduction loop compliment
ifz      ; 05  39 this is a dummy instruction to separate templates
         ;        begin copy procedure
nop1     ; 01  40 copy procedure template
nop1     ; 01  41 copy procedure template
nop0     ; 00  42 copy procedure template
nop0     ; 00  43 copy procedure template
pushax   ; 0c  44 push ax onto stack
pushbx   ; 0d  45 push bx onto stack
pushcx   ; 0e  46 push cx onto stack
nop1     ; 01  47 copy loop template
nop0     ; 00  48 copy loop template
nop1     ; 01  49 copy loop template
nop0     ; 00  50 copy loop template
movii    ; 1a  51 move contents of [bx] to [ax]
dec_c    ; 0a  52 decrement cx
ifz      ; 05  53 if cx == 0 perform next instruction, otherwise skip it
jmp      ; 14  54 jump to template below (copy procedure exit)
nop0     ; 00  55 copy procedure exit compliment
nop1     ; 01  56 copy procedure exit compliment
nop0     ; 00  57 copy procedure exit compliment
nop0     ; 00  58 copy procedure exit compliment
inc_a    ; 08  59 increment ax
inc_b    ; 09  60 increment bx
jmp      ; 14  61 jump to template below (copy loop)
nop0     ; 00  62 copy loop compliment
nop1     ; 01  63 copy loop compliment
nop0     ; 00  64 copy loop compliment
nop1     ; 01  65 copy loop compliment
ifz      ; 05  66 this is a dummy instruction, to separate templates
nop1     ; 01  67 copy procedure exit template
nop0     ; 00  68 copy procedure exit template
nop1     ; 01  69 copy procedure exit template
nop1     ; 01  70 copy procedure exit template
popcx    ; 12  71 pop cx off stack
popbx    ; 11  72 pop bx off stack
popax    ; 10  73 pop ax off stack
```

```
ret      ; 17  74 return from copy procedure
nop1     ; 01  75 end template
nop1     ; 01  76 end template
nop1     ; 01  77 end template
nop0     ; 00  78 end template
ifz      ; 05  79 dummy statement to separate creatures
```

12 Appendix E: Smallest Replicator Source Code

Assembler source code for the smallest self-replicating creature.

```
genotype: 0022abn  parent genotype: 0022aak
1st_daughter:  flags: 1  inst: 146  mov_daught: 22  breed_true: 1
2nd_daughter:  flags: 0  inst: 142  mov_daught: 22  breed_true: 1
InstExecC: 437  InstExec: 625954  origin: 662865379  Wed Jan  2 20:16:19 1991
MaxPropPop: 0.1231  MaxPropInst: 0.0568

nop0    ; 00   0
adrb    ; 1c   1 find beginning
nop1    ; 01   2
divide  ; 1f   3 fails the first time it is executed
sub_ac  ; 07   4
movab   ; 19   5
adrf    ; 1d   6 find end
nop0    ; 00   7
inc_a   ; 08   8 to include final dummy statement
sub_ab  ; 06   9 calculate size
mal     ; 1e  10
pushbx  ; 0d  11 save beginning address on stack in order to 'return' there
nop0    ; 00  12 top of copy loop
movii   ; 1a  13
dec_c   ; 0a  14
ifz     ; 05  15
ret     ; 17  16 jump to beginning, address saved on stack
inc_a   ; 08  17
inc_b   ; 09  18
jmpb    ; 15  19 bottom of copy loop (6 instructions executed per loop)
nop1    ; 01  20
movii   ; 1a  21 dummy statement to terminate template
```

13 Appendix F: Source Code of Complex Loop

Assembler code for the central copy loop of the ancestor of instruction set one (80aaa) and a descendant after fifteen billion instructions (72etq). Within the loop, the ancestor does each of the following operations once: copy instruction (51), decrement CX (52), increment AX (59) and increment BX (60). The descendant performs each of the following operations three times within the loop: copy instruction (15, 22, 26), increment AX (20, 24, 31) and increment BX (21, 25, 32). The decrement CX operation occurs five times within the loop (16, 17, 19, 23, 27). Instruction 28 flips the low order bit of the CX register. Whenever this latter instruction is reached, the value of the low order bit is one, so this amounts to a sixth instance of decrement CX. This means that there are two decrements for every increment. The reason for this is related to another adaptation of this creature. When it calculates its size, it shifts left (12) before allocating space for the daughter (13). This has the effect of allocating twice as much space as is actually needed to accommodate the genome. The genome of the creature is 36 instructions long, but it allocates a space of 72 instructions. This occurred in an environment where the CPU time slice size was set equal to the size of the cell. In this way the creatures were able to garner twice as much energy. However, they had to compliment this change by doubling the number of decrements in the loop.

```
nop1    ; 01  47 copy loop template              COPY LOOP OF 80AAA
nop0    ; 00  48 copy loop template
nop1    ; 01  49 copy loop template
nop0    ; 00  50 copy loop template
movii   ; 1a  51 move contents of [BX] to [AX] (copy instruction)
dec_c   ; 0a  52 decrement CX
ifz     ; 05  53 if CX = 0 perform next instruction, otherwise skip it
jmp     ; 14  54 jump to template below (copy procedure exit)
nop0    ; 00  55 copy procedure exit compliment
nop1    ; 01  56 copy procedure exit compliment
nop0    ; 00  57 copy procedure exit compliment
nop0    ; 00  58 copy procedure exit compliment
inc_a   ; 08  59 increment AX (point to next instruction of daughter)
inc_b   ; 09  60 increment BX (point to next instruction of mother)
jmp     ; 14  61 jump to template below (copy loop)
nop0    ; 00  62 copy loop compliment
nop1    ; 01  63 copy loop compliment
nop0    ; 00  64 copy loop compliment
nop1    ; 01  65 copy loop compliment (10 instructions executed per loop)

shl     ; 03  12 shift left CX                   COPY LOOP OF 72ETQ
mal     ; 1e  13 allocate daughter cell
nop0    ; 00  14 top of loop
movii   ; 1a  15 copy instruction
dec_c   ; 0a  16 decrement CX
```

```
dec_c   ; 0a  17 decrement CX
jmpb    ; 15  18 junk
dec_c   ; 0a  19 decrement CX
inc_a   ; 08  20 increment AX
inc_b   ; 09  21 increment BX
movii   ; 1a  22 copy instruction
dec_c   ; 0a  23 decrement CX
inc_a   ; 08  24 increment AX
inc_b   ; 09  25 increment BX
movii   ; 1a  26 copy instruction
dec_c   ; 0a  27 decrement CX
not0    ; 02  28 flip low order bit of CX, equivalent to dec_c
ifz     ; 05  29 if CX == 0 do next instruction
ret     ; 17  30 exit loop
inc_a   ; 08  31 increment AX
inc_b   ; 09  32 increment BX
jmpb    ; 15  33 go to top of loop (6 instructions per copy)
nop1    ; 01  34 bottom of loop (18 instructions executed per loop)
```

14 Appendix G: Definition of Four Instruction Sets

14.1 Instruction Set #1

The original instruction set, designed and implemented by Tom Ray. This instruction set was literally designed only to run a single program, the original 80 instruction "ancestor". As a consequence of this narrow design criteria, this instruction set has several obvious deficiencies: There is no method of moving information between the CPU registers and the RAM memory (soup). There is no mechanism for input/output. Only two inter-register moves are available, although this limitation can be overcome by using the stack to move data between registers (as is done in instruction set 4). There are no options for the control of the positioning in memory of the daughter cells (only the "first fit" technique is used). There are no facilities to support multi-cellularity. These deficiencies were addressed in the creation of instruction sets two through four.

```
No Operations: 2

nop0
nop1

Memory Movement: 11

pushax (push AX onto stack)
pushbx (push BX onto stack)
```

```
pushcx (push CX onto stack)
pushdx (push DX onto stack)
popax  (pop from stack into AX)
popbx  (pop from stack into BX)
popcx  (pop from stack into CX)
popdx  (pop from stack into DX)
movcd  (DX = CX)
movab  (BX = AX)
movii  (move from ram [BX] to ram [AX])

Calculation: 9

sub_ab (CX = AX - BX)
sub_ac (AX = AX - CX)
inc_a  (increment AX)
inc_b  (increment BX)
inc_c  (increment CX)
dec_c  (decrement CX)
zero   (zero CX)
not0   (flip low order bit of CX)
shl    (shift left all bits of CX)

Instruction Pointer Manipulation: 5

ifz    (if CX == 0 execute next instruction, otherwise, skip it)
jmp    (jump to template)
jmpb   (jump backwards to template)
call   (push IP onto the stack, jump to template)
ret    (pop the stack into the IP)

Biological and Sensory: 5

adr    (search outward  for template, put address in AX, template size in CX)
adrb   (search backward for template, put address in AX, template size in CX)
adrf   (search forward  for template, put address in AX, template size in CX)
mal    (allocate amount of space specified in CX)
divide (cell division)

Total: 32
```

14.2 Instruction Set #2

Based on a design suggested by Kurt Thearling of Thinking Machines, and implemented by Tom Ray. The novel feature of this instruction set is the ability to reorder the relative

positions of the registers, using the **AX**, **BX**, **CX** and **DX** instructions. There are in essence, two sets of registers, the first set contains the values that the instruction set operates on, the second set points to the first set, in order to determine which registers any operation will act on.

Let the four registers containing values be called AX, BX, CX and DX. Let the four registers pointing to these registers be called R0, R1, R2 and R3. When a virtual cpu is initialized, R0 points to AX, R1 to BX, R2 to CX and R3 to DX. The instruction **add** does the following: (R2 = R1 + R0). Therefore CX = BX + AX. However, if we execute the **DX** instruction, the R0 points to DX, R1 to AX, R2 to BX and R3 to CX. Now if we execute the **add** instruction, we will perform: BX = AX + DX. If we execute the **DX** instruction again, R0 points to DX, R1 to DX, R2 to AX, and R3 to BX. Now the **add** instruction would perform: AX = DX + DX. Now the registers can be returned to their original configuration by executing the following three instructions in order: **CX**, **BX**, **AX**.

```
No Operations: 2

nop0
nop1

Memory Movement: 12

AX      (make AX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
BX      (make BX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
CX      (make CX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
DX      (make DX R0, R1 = R0, R2 = R1, R3 = R2, R3 is lost)
movdd   (move R1 to R0)
movdi   (move from R1 to ram [R0])
movid   (move from ram [R1] to R0)
movii   (move from ram [R1] to ram [R0])
push    (push R0 onto stack)
pop     (pop from stack into R0)
put     (write R0 to output buffer, three modes:
          #ifndef ICC: write R0 to own output buffer
          #ifdef ICC:  write R0 to input buffer of cell at address R1,
            or, if template, write R0 to input buffers of all creatures within
            PutLimit who have the complementary get template)
get     (read R0 from input port)

Calculation: 8

inc     (increment R0)
dec     (decrement R0)
add     (R2 = R1 + R0)
sub     (R2 = R1 - R0)
```

```
zero   (zero R0)
not0   (flip low order bit of R0)
shl    (shift left all bits of R0)
not    (flip all bits of R0)

Instruction Pointer Manipulation: 5

ifz  (if   R1 == 0 execute next instruction, otherwise, skip it)
iffl (if flag == 1 execute next instruction, otherwise, skip it)
jmp  (jump to template, or if no template jump to address in R0)
jmpb (jump back to template, or if no template jump back to address in R0)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr    (search outward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset +- R0)
adrb   (search backward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset - R0)
adrf   (search forward for template, put address in R0, template size in R1,
           and offset in R2, start search at offset + R0)
mal    (allocate amount of space specified in R0, prefer address at R1,
           if R1 < 0 use best fit, place address of allocated block in R0)
divide (cell division, the IP is offset by R0 into the daughter cell, the
           values in the four CPU registers are transferred from mother to
           daughter, but not the stack.  If !R1, eject genome from soup)

Total: 32
```

14.3 Instruction Set #3

Based on a design suggested and implemented by Tom Ray. This includes certain features of the RPN Hewlett-Packard calculator.

```
No Operations: 2

nop0
nop1

Memory Movement: 11

rollu  (roll registers up:   AX = DX, BX = AX, CX = BX, DX = CX)
rolld  (roll registers down: AX = BX, BX = CX, CX = DX, DX = AX)
enter  (AX = AX, BX = AX, CX = BX, DX = CX, DX is lost)
```

```
exch    (AX = BX, BX = AX)
movdi   (move from BX to ram [AX])
movid   (move from ram [BX] to AX)
movii   (move from ram [BX] to ram [AX])
push    (push AX onto stack)
pop     (pop from stack into AX)
put     (write AX to output buffer, three modes:
          #ifndef ICC: write AX to own output buffer
          #ifdef ICC:  write AX to input buffer of cell at address BX,
            or, if template, write AX to input buffers of all creatures within
            PutLimit who have the complementary get template)
get     (read AX from input buffer)

Calculation: 9

inc     (increment AX)
dec     (decrement AX)
add     (AX = BX + AX, BX = CX, CX = DX))
sub     (AX = BX - AX, BX = CX, CX = DX))
zero    (zero AX)
not0    (flip low order bit of AX)
not     (flip all bits of AX)
shl     (shift left all bits of AX)
rand    (place random number in AX)

Instruction Pointer Manipulation: 5

ifz  (if   AX == 0 execute next instruction, otherwise, skip it)
iffl (if flag == 1 execute next instruction, otherwise, skip it)
jmp  (jump to template, or if no template jump to address in AX)
jmpb (jump back to template, or if no template jump back to address in AX)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr     (search outward for template, put address in AX, template size in BX,
            and offset in CX, start search at offset +- BX)
adrb    (search backward for template, put address in AX, template size in BX,
            and offset in CX, start search at offset - BX)
adrf    (search forward for template, put address in AX, template size in BX,
            and offset in CX, start search at offset + BX)
mal     (allocate amount of space specified in BX, prefer address at AX,
            if AX < 0 use best fit, place address of allocated block in AX)
divide (cell division, the IP is offset by AX into the daughter cell, the
            values in the four CPU registers are transferred from mother to
```

```
            daughter, but not the stack. If !CX genome will be ejected from
            the simulator)

Total: 32
```

14.4 Instruction Set #4

Based on a design suggested by Walter Tackett of Hughes Aircraft, and implemented by Tom Ray. The special features of this instruction set are that all movement between registers of the cpu takes place via push and pop through the stack. Also, all indirect addressing involves an offset from the address in the CX register. Also, the CX register is where most calculations take place.

```
No Operations: 2

nop0
nop1

Memory Movement: 13

movdi   (move from BX to ram [AX + CX])
movid   (move from ram [BX + CX] to AX)
movii   (move from ram [BX + CX] to ram [AX + CX])
pushax  (push AX onto stack)
pushbx  (push BX onto stack)
pushcx  (push CX onto stack)
pushdx  (push DX onto stack)
popax   (pop from stack into AX)
popbx   (pop from stack into BX)
popcx   (pop from stack into CX)
popdx   (pop from stack into DX)
put     (write DX to output buffer, three modes:
          #ifndef ICC: write DX to own output buffer
          #ifdef ICC:  write DX to input buffer of cell at address CX,
            or, if template, write DX to input buffers of all creatures within
            PutLimit who have the complementary get template)
get     (read DX from input port)

Calculation: 7

inc     (increment CX)
dec     (decrement CX)
add     (CX = CX + DX)
sub     (CX = CX - DX)
```

```
zero    (zero CX)
not0    (flip low order bit of CX)
shl     (shift left all bits of CX)

Instruction Pointer Manipulation: 5

ifz     (if   CX == 0 execute next instruction, otherwise, skip it)
iffl    (if flag == 1 execute next instruction, otherwise, skip it)
jmp     (jump to template, or if no template jump to address in AX)
jmpb    (jump back to template, or if no template jump back to address in AX)
call (push IP + 1 onto the stack; if template, jump to complementary templ)

Biological and Sensory: 5

adr     (search outward for template, put address in AX, template size in DX,
            and offset in CX, start search at offset +- CX)
adrb    (search backward for template, put address in AX, template size in DX,
            and offset in CX, start search at offset - CX)
adrf    (search forward for template, put address in AX, template size in DX,
            and offset in CX, start search at offset + CX)
mal     (allocate amount of space specified in CX, prefer address at AX,
            if AX < 0 use best fit, place address of allocated block in AX)
divide (cell division, the IP is offset by CX into the daughter cell, the
            values in the four CPU registers are transferred from mother to
            daughter, but not the stack.  If !DX genome will be ejected from
            the simulator)

Total: 32
```

15 Appendix H: Source Code for Unrolled Loops

This appendix contains the assembler source code for the 82 instruction ancestor written for instruction set four, and three descendant organisms that evolved from the ancestor. The three descendants are derived from different runs, and represent forms found after optimization was apparently complete in each run. The three evolved forms illustrate three levels of loop unrolling: 1) no unrolling, level 1, 2) unrolling to level 2, and 3) unrolling to level 3.

GENOTYPE: 0082aaa comments: ancestor for instruction set 4

```
nop1    ; 01   0 beginning marker
nop1    ; 01   1 beginning marker
nop1    ; 01   2 beginning marker
nop1    ; 01   3 beginning marker
```

```
zero    ; 13   4 CX = 0, offset for search
adrb    ; 1c   5 find start, AX = start + 4, DX = templ size
nop0    ; 00   6 complement to beginning marker
nop0    ; 00   7 complement to beginning marker
nop0    ; 00   8 complement to beginning marker
nop0    ; 00   9 complement to beginning marker
pushax  ; 05  10 push start + 4 on stack
popcx   ; 0b  11 pop start + 4 into CX
sub     ; 12  12 CX = CX - DX, CX = start
pushcx  ; 07  13 push start on stack
zero    ; 13  14 CX = 0, offset for search
adrf    ; 1d  15 find end, AX = end, CX = offset, DX = templ size
nop0    ; 00  16 complement to end marker
nop0    ; 00  17 complement to end marker
nop0    ; 00  18 complement to end marker
nop1    ; 01  19 complement to end marker
pushax  ; 05  20 push end on stack
popcx   ; 0b  21 pop end into CX
inc     ; 0f  22 increment to include dummy instruction at end
popdx   ; 0c  23 pop start into DX
sub     ; 12  24 CX = CX - DX, AX = end, CX = size, DX = start
nop1    ; 01  25 reproduction loop marker
nop1    ; 01  26 reproduction loop marker
nop0    ; 00  27 reproduction loop marker
nop1    ; 01  28 reproduction loop marker
mal     ; 1e  29 AX = daughter, CX = size, DX = mom
call    ; 1a  30 call copy procedure
nop0    ; 00  31 copy procedure complement
nop0    ; 00  32 copy procedure complement
nop1    ; 01  33 copy procedure complement
nop1    ; 01  34 copy procedure complement
divide  ; 1f  35 create daughter cell
jmpb    ; 19  36 jump back to top of reproduction loop
nop0    ; 00  37 reproduction loop complement
nop0    ; 00  38 reproduction loop complement
nop1    ; 01  39 reproduction loop complement
nop0    ; 00  40 reproduction loop complement
ifz     ; 16  41 dummy instruction to separate templates
nop1    ; 01  42 copy procedure template
nop1    ; 01  43 copy procedure template
nop0    ; 00  44 copy procedure template
nop0    ; 00  45 copy procedure template
pushcx  ; 07  46 push size on stack
pushdx  ; 08  47 push start on stack
pushdx  ; 08  48 push start on stack
```

```
popbx   ; 0a  49 pop start into BX
nop1    ; 01  50 copy loop template
nop0    ; 00  51 copy loop template
nop1    ; 01  52 copy loop template
nop0    ; 00  53 copy loop template
dec     ; 10  54 decrement size
movii   ; 04  55 move from [BX + CX] to [AX + CX]
ifz     ; 16  56 test when to exit loop
jmp     ; 18  57 exit loop
nop0    ; 00  58 copy procedure exit complement
nop1    ; 01  59 copy procedure exit complement
nop0    ; 00  60 copy procedure exit complement
nop0    ; 00  61 copy procedure exit complement
jmpb    ; 19  62 jump to top of copy loop
nop0    ; 00  63 copy loop complement
nop1    ; 01  64 copy loop complement
nop0    ; 00  65 copy loop complement
nop1    ; 01  66 copy loop complement
ifz     ; 16  67 dummy instruction to separate jmp from template
nop1    ; 01  68 copy procedure exit template
nop0    ; 00  69 copy procedure exit template
nop1    ; 01  70 copy procedure exit template
nop1    ; 01  71 copy procedure exit template
popdx   ; 0c  72 pop start into DX
popcx   ; 0b  73 pop size into CX
popax   ; 09  74 pop call IP into AX
jmp     ; 18  75 jump to call (return)
ifz     ; 16  76 dummy instruction to separate jmp from template
nop1    ; 01  77 end marker
nop1    ; 01  78 end marker
nop1    ; 01  79 end marker
nop0    ; 00  80 end marker
ifz     ; 16  81 dummy instruction to separate creatures
```

GENOTYPE: 0023awn

```
call    ; 1a   0 push ip + 1 on stack
popcx   ; 0b   1 pop ip + 1 into CX
dec     ; 10   2 CX = start
pushcx  ; 07   3 save start on stack
zero    ; 13   4 CX = 0
divide  ; 1f   5 cell division, will fail first time
```

```
adrf    ; 1d   6 AX = end + 1
nop0    ; 00   7
pushax  ; 05   8 push end address on stack
popcx   ; 0b   9 CX = end address + 1
popdx   ; 0c  10 DX = start address
sub     ; 12  11 (CX = CX - DX) CX = size
adr     ; 1b  12 this instruction will fail
pushdx  ; 08  13 put start address on stack
mal     ; 1e  14 allocate daughter, AX = start of daughter
popbx   ; 0a  15 BX = start address
nop0    ; 00  16 top of copy loop
dec     ; 10  17 decrement size
movii   ; 04  18 copy byte to daughter
ifz     ; 16  19 if CX == 0 jump to address in AX (start of daughter)
jmp     ; 18  20
jmpb    ; 19  21 jump back to line 17 (top of copy loop)
nop1    ; 01  22
```

GENOTYPE: 0024aah

```
call    ; 1a   0 push ip + 1 on stack
popcx   ; 0b   1 pop ip + 1 into CX
dec     ; 10   2 CX = start
pushcx  ; 07   3 save start on stack
zero    ; 13   4 CX = 0
adrf    ; 1d   5 AX = end + 1
nop1    ; 01   6
pushax  ; 05   7 push end address on stack
divide  ; 1f   8 cell division, will fail first time
popcx   ; 0b   9 CX = end address + 1
popdx   ; 0c  10 DX = start address
sub     ; 12  11 (CX = CX - DX) CX = size
pushdx  ; 08  12 put start address on stack
popbx   ; 0a  13 BX = start address
mal     ; 1e  14 allocate daughter, AX = start of daughter
nop1    ; 01  15 top of copy loop
dec     ; 10  16 decrement size
movii   ; 04  17 copy byte to daughter
dec     ; 10  18 decrement size
movii   ; 04  19 copy byte to daughter
ifz     ; 16  20 if CX == 0 jump to address in AX (start of daughter)
jmp     ; 18  21
```

```
jmpb    ; 19  22 jump back to line 16 (top of copy loop)
nop0    ; 00  23
```

GENOTYPE: 0035bfj

```
call    ; 1a   0 push ip + 1 on stack
popcx   ; 0b   1 pop ip + 1 into CX
dec     ; 10   2 CX = start
pushcx  ; 07   3 save start on stack
adrf    ; 1d   4 dummy instruction
divide  ; 1f   5 cell division, will fail first time
movid   ; 03   6 dummy instruction (AX = 0x1a, call instruction)
zero    ; 13   7 CX = 0
adrf    ; 1d   8 AX = end + 1
nop1    ; 01   9
pushax  ; 05  10 push end address on stack
popcx   ; 0b  11 CX = end address + 1
adrf    ; 1d  12 dummy instruction
popdx   ; 0c  13 DX = start address
pushdx  ; 08  14 push start address on stack
pushdx  ; 08  15 push start address on stack
sub     ; 12  16 (CX = CX - DX) CX = size
mal     ; 1e  17 allocate daughter, AX = start of daughter
pushdx  ; 08  18 push start address on stack
popbx   ; 0a  19 BX = start address
pushbx  ; 06  20 push start address on stack
mal     ; 1e  21 allocate daughter, AX = start of daughter (fails)
put     ; 0d  22 dummy instruction (write to get buffer of other creature)
nop1    ; 01  23
nop1    ; 01  24 top of copy loop
dec     ; 10  25 decrement size
movii   ; 04  26 copy byte to daughter
dec     ; 10  27 decrement size
movii   ; 04  28 copy byte to daughter
ifz     ; 16  29 if CX == 0 jump to address in AX (start of daughter)
jmpb    ; 19  30
dec     ; 10  31 decrement size
movii   ; 04  32 copy byte to daughter
jmpb    ; 19  33 jump back to line 25 (top of copy loop)
nop0    ; 00  34
```

References

[1] Ackley, D. H. & Littman, M. S. "Learning from natural selection in an artificial environment." In: *Proceedings of the International Joint Conference on Neural Networks, Volume I, Theory Track, Neural and Cognitive Sciences Track*, IJCNN Winter 1990, Washington, DC. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1990.

[2] Adami, Chris. Unpublished. Learning and complexity in genetic auto-adaptive systems. Caltech preprint: MAP – 164, One of the Marmal Aid Preprint Series In Theoretical Nuclear Physics, October 1993. Adami has used the input-output facilities of the new Tierra languages to feed data to creatures, and select for responses that result from simple computations, not contained in the seed genome. Contact: chris@almach.caltech.edu

[3] Aho, A. V., Hopcroft, J. E. & Ullman, J. D. *The design and analysis of computer algorithms.* Reading, Mass.: Addison-Wesley Publ. Co, 1974.

[4] Anonymous. 1988. Worm invasion. Science 11-11-88: 885.

[5] Bagley, R. J., Farmer, J. D., Kauffman, S. A., Packard, N. H., Perelson, A. S. & Stadnyk, I. M. "Modeling adaptive biological systems." Unpublished paper, 1989.

[6] Barbieri, M. *The semantic theory of evolution.* London: Harwood Academic Publishers, 1985.

[7] Barton-Davis, Paul. Unpublished. Independent implementation of the Tierra system, contact: pauld@cs.washington.edu.

[8] Beaudry, Amber A., and Gerald F. Joyce. 1992. Directed evolution of an RNA enzyme. Science 257: 635–641.

[9] Benner, Steven A., Andrew D. Ellington, and Andreas Tauer. 1989. Modern metabolism as a palimpsest of the RNA world. Proc. Natl. Acad. Sci. U.S.A. 86: 7054–7058.

[10] Brooks, Rodney. Unpublished. Brooks has created his own Tierra-like system, which he calls Sierra. In his implementation, each machine instruction consists of an opcode and an operand. Successive instructions overlap, such that the operand of one instruction is interpreted as the opcode of the next instruction. Contact: brooks@ai.mit.edu

[11] Burstyn, Harold L. 1990. RTM and the worm that ate internet. Harvard Magazine 92(5): 23–28.

[12] Cariani, P. "Emergence and artificial life." In: *Artificial Life II*, edited by C. Langton, D. Farmer and S. Rasmussen. Redwood City, CA: Addison-Wesley, 1991, 000–000.

[13] Cairn-Smith, A. G. 1985. Seven clues to the origin of life. Cambridge: Cambridge University Press.

[14] Cohen, F. *Computer viruses: theory and experiments.* Ph. D. dissertation, U. of Southern California, 1984.

[15] Darwin, Charles. 1859. On the origin of species by means of natural selection or the preservation of favored races in the struggle for life. London: Murray.

[16] Davidge, Robert. 1992. Processors as organisms. CSRP 250. School of Cognitive and Computing Sciences, University of Sussex. Presented at the ALife III conference. Contact: robertd@cogs.susx.ac.uk

[17] Davidge, Robert. 1993. Looping as a means to survival: playing Russian roulette in a harsh environment. *In*: Self organization and life: from simple rules to global complexity, proceedings of the second European conference on artificial life. Contact: robertd@cogs.susx.ac.uk

[18] Dawkins, R. *The blind watchmaker.* New York: W. W. Norton & Co., 1987.

[19] Dawkins, R. "The evolution of evolvability." In: *Artificial life: proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems*, edited by C. Langton. Redwood City, CA: Addison-Wesley, 1989, 201–220.

[20] DeAngelis, D., and L. Gross [eds]. 1992. Individual based models and approaches in ecology. New York: Chapman and Hill.

[21] de Groot, Marc. Unpublished. Primordial soup, a Tierra-like system that has the additional ability to spawn self-reproducing organisms from a sterile soup. Contact: marc@kg6kf.ampr.org, marc@toad.com, marc@remarque.berkeley.edu

[22] Denning, P. J. "Computer viruses." *Amer. Sci.* **76** (1988): 236–238.

[23] Dewdney, A. K. "Computer recreations: In the game called Core War hostile programs engage in a battle of bits." *Sci. Amer.* **250** (1984): 14–22.

[24] Dewdney, A. K. "Computer recreations: A core war bestiary of viruses, worms and other threats to computer memories." *Sci. Amer.* **252** (1985a): 14–23.

[25] Dewdney, A. K. "Computer recreations: Exploring the field of genetic algorithms in a primordial computer sea full of flibs." *Sci. Amer.* **253** (1985b): 21–32.

[26] Dewdney, A. K. "Computer recreations: A program called MICE nibbles its way to victory at the first core war tournament." *Sci. Amer.* **256** (1987): 14–20.

[27] Dewdney, A. K. "Of worms, viruses and core war." *Sci. Amer.* **260** (1989): 110–113.

[28] Eigen, Manfred. 1993. Viral quasispecies. Scientific American 269(1): 32–39. July 1993.

[29] Farmer, J. D. & Belin, A. Artificial life: the coming evolution. Proceedings in celebration of Murray Gell-Mann's 60th Birthday. Cambridge: University Press. (Reprinted in Artificial Life II. Pp. 815–840.)

[30] Farmer, J. D., Kauffman, S. A., & Packard, N. H. "Autocatalytic replication of polymers." *Physica D* **22** (1986): 50–67.

[31] Feferman, Linda. 1992. Simple rules... complex behavior [video]. Santa Fe, NM: Santa Fe Institute. Contact: fef@santafe.edu, 0005851689@mcimail.com

[32] Feng, Q., Park, T. K. & Rebek, J. 1992. Science 254: 1179–1180.

[33] Goldberg, D. E. 1989. Genetic algorithms in search, optimization, and machine learning. Reading, MA: Addison-Wesley.

[34] Gould, Steven J. 1989. Wonderful life. W. W. Norton & Company, Inc. Pp. 347.

[35] Gray, James. Unpublished. Natural selection of computer programs. This may have been the first Tierra-like system, but evolving real programs on a real rather than a virtual machine, and predating Tierra itself: "I have attempted to develop ways to get computer programs to function like biological systems subject to natural selection.... I don't think my systems are models in the usual sense. The programs have really competed for resources, reproduced, run, and 'died'. The resources consisted primarily of access to the CPU and partition space.... On a PDP11 I could have a population of programs running simultaneously." Contact: Gray.James_L+@northport.va.gov

[36] Hogeweg, P. 1989. Mirror beyond mirror: puddles of life. *In*: Langton, C. [ed], Artificial Life, Santa Fe Institute Studies in the Sciences of Complexity, vol. VI, 297–316. Redwood City, CA: Addison-Wesley.

[37] Holland, John Henry. 1975. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence (Univ. of Michigan Press, Ann Arbor).

[38] Holland, J. H. "Studies of the spontaneous emergence of self-replicating systems using cellular automata and formal grammars." In: *Automata, Languages, Development*, edited by Lindenmayer, A., & Rozenberg, G. New York: North-Holland, 1976, 385–404.

[39] Hong, J. I., Feng, Q., Rotello, V. & Rebek, J. 1992. Competition, cooperation, and mutation: improving a synthetic replicator by light irradiation. Science 255: 848–850.

[40] Huston, M., DeAngelis, D., and Post, W. 1988. New computer models unify ecological theory. Bioscience 38(10): 682–691.

[41] Joyce, Gerald F. 1992. Directed molecular evolution. Scientific American, December 1992: 90–97.

[42] Kampis, George. 1993. Coevolution in the computer: the necessity and use of distributed code systems. Printed in the ECAL93 proceedings, Brussels. Contact: gk@cfnext.physchem.chemie.uni-tuebingen.de

[43] Kampis, George. 1993. Life-like computing beyond the machine metaphor. *In*: R. Paton [ed]: Computing with biological metaphors, London: Chapman and Hall. Contact: gk@cfnext.physchem.chemie.uni-tuebingen.de

[44] Kauffman, Stuart A. 1993. The origins of order, self-organization and selection in evolution. Oxford University Press. Pp. 709.

[45] Koza, John R. 1992. Genetic programming, on the programming of computers by means of natural selection. Cambridge, MA: MIT Press.

[46] Langton, C. G. 1986. Studying artificial life with cellular automata. Physica 22D: 120–149.

[47] Langton, C. G. "Virtual state machines in cellular automata." *Complex Systems* **1** (1987): 257–271.

[48] Langton, C. G. "Artificial life." In: *Artificial life: proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems*, edited by Langton, C. Vol. 6 in the series: Santa Fe Institute studies in the sciences of complexity. Redwood City, CA: Addison-Wesley, 1989, 1–47.

[49] Levy, Steven. 1992. Artificial Life, the quest for a new creation. Pantheon Books, New York. Pp. 390.

[50] Levy, Steven. 1992. A-Life Nightmare. Whole Earth Review #76, Fall 1992, p. 22.

[51] Litherland, J. 1993. Open-ended evolution in a computerised ecosystem. A Masters of Science dissertation in the Department of Computer Science, Brunel University. Contact: david.martland@brunel.ac.uk

[52] Lotka, A. J. *Elements of physical biology.* Baltimore: Williams and Wilkins, 1925, reprinted as *Elements of mathematical biology*, Dover Press, 1956.

[53] Maley, Carlo C. 1993. A model of early evolution in two dimensions. Masters of Science thesis, Zoology, New College, Oxford University. Contact: cmaley@oxford.ac.uk

[54] Manousek, Wolfgang. 1992. Spontane Komplexitaetsentstehung — TIERRA, ein Simulator fuer biologische Evolotion. Diplomarbeit, Universitaet Bonn, Germany, Oktober 1992. Contact: Kurt Stueber, stueber@vax.mpiz-koeln.mpg.d400.de

[55] Maynard Smith, J. 1992. Byte-sized evolution. Nature 355: 772–773.

[56] Minsky, M. L. *Computation: finite and infinite machines.* Englewood Cliffs, N.J.: Prentice-Hall, 1976.

[57] Moravec, Hans. 1988. Mind Children: the future of robot and human intelligence. Cambridge, MA: Harvard University Press.

[58] Moravec, Hans. 1989. Human culture: a genetic takeover underway. *In*: Langton, C. [ed], Artificial Life, Santa Fe Institute Studies in the Sciences of Complexity, vol. VI, 167–199. Redwood City, CA: Addison-Wesley.

[59] Moravec, Hans. 1993. Pigs in cyberspace. Extropy #10, Winter/Spring, 1993.

[60] Morris, S. Conway. 1989. Burgess shale faunas and the Cambrian explosion. Science 246: 339–346.

[61] Nowick, J., Feng, Q., Tijivikua, T., Ballester, P. & Rebek, J. 1991. Journal of the American Chemical Society 113: 8831–8839.

[62] Packard, N. H. "Intrinsic adaptation in a simple model for evolution." In: *Artificial life: proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems*, edited by C. Langton. Redwood City, CA: Addison-Wesley, 1989, 141–155.

[63] Paine, R. T. "Food web complexity and species diversity." *Am. Nat.* **100** (1966): 65–75.

[64] Pattee, H. H. "Simulations, realizations, and theories of life." In: *Artificial life: proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems*, edited by C. Langton. Redwood City, CA: Addison-Wesley, 1989, 63–77.

[65] Rasmussen, S., Knudsen, C., Feldberg, R. & Hindsholm, M. "The coreworld: emergence and evolution of cooperative structures in a computational chemistry" *Physica D* **42** (1990): 111–134.

[66] Rasmussen, S., C. Knudsen, and R. Feldberg. 1991. Dynamics of programmable matter. *In*: Langton, C., C. Taylor, J. D. Farmer, & S. Rasmussen [eds], Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity, vol. X, 211–254. Redwood City, CA: Addison-Wesley.

[67] Ray, T. S. 1979. Slow-motion world of plant 'behavior' visible in rainforest. Smithsonian 9(12): 121–30.

[68] _____. 1991. An approach to the synthesis of life. *In*: Langton, C., C. Taylor, J. D. Farmer, & S. Rasmussen [eds], Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity, vol. X, 371–408. Redwood City, CA: Addison-Wesley.

[69] _____. 1991. Population dynamics of digital organisms. *In*: Langton, C. G. [ed.], Artificial Life II Video Proceedings. Redwood City, CA: Addison Wesley.

[70] _____. 1991. Is it alive, or is it GA? *In*: Belew, R. K., and L. B. Booker [eds.], Proceedings of the 1991 International Conference on Genetic Algorithms, 527–534. San Mateo, CA: Morgan Kaufmann.

[71] _____. 1991. Evolution and optimization of digital organisms. *In*: Billingsley K. R., E. Derohanes, H. Brown, III [eds.], Scientific Excellence in Supercomputing: The IBM 1990 Contest Prize Papers, Athens, GA, 30602: The Baldwin Press, The University of Georgia.

[72] _____. 1992. Foraging behaviour in tropical herbaceous climbers (Araceae). Journal of Ecology. 80: 189–203.

[73] _____. 1994. Evolution and complexity. *In*: Cowan, George A., David Pines and David Metzger [eds.], Complexity: Metaphors, Models, and Reality, Pp. 161–173. Addison-Wesley Publishing Co.

[74] _____. 1994. Evolution, complexity, entropy, and artificial reality. Physica D 75: 239–263.

[75] _____. 1994. An evolutionary approach to synthetic biology: Zen and the art of creating life. Artificial Life 1(1/2): 195–226. Reprinted *In*: Langton, C. G. [ed.], Artificial Life, an overview. The MIT Press, 1995.

[76] _____. Unpublished. A proposal to consolidate and stabilize the rain forest reserves of the Sarapiquí region of Costa Rica. Available by anonymous ftp: tierra.slhs.udel.edu [128.175.41.34] as tierra/doc/reserves.tex, or at http://www.hip.atr.co.jp/~ray/pubs/reserves/reserves.html.

[77] Rheingold, H. (1988). Computer viruses. *Whole Earth Review* **Fall** (1988): 106.

[78] Skipper, Jakob. 1992. The computer zoo – evolution in a box. *In*: Francisco J. Varela and Paul Bourgine [eds.], Toward a practice of autonomous systems, proceedings of the first European conference on Artificial Life. MIT Press, Cambridge, MA. Pp. 355–364. Contact: Jakob.Skipper@copenhagen.ncr.com

[79] Sober, E. 1984. The nature of selection. MIT Press, Cambridge, MA.

[80] Spafford, Eugene H. 1989. The internet worm program: an analysis. Computer Communication Review 19(1): 17–57. Also issued as Purdue CS technical report TR-CSD-823. Contact: spaf@purdue.edu

[81] Spafford, Eugene H. 1989. The internet worm: crisis and aftermath. CACM 32(6): 678–687. Contact: spaf@purdue.edu

[82] Spafford, E. H., Heaphy, K. A. & Ferbrache, D. J. *Computer viruses, dealing with electronic vandalism and programmed threats.* ADAPSO, 1300 N. 17th Street, Suite 300, Arlington, VA 22209, 1989.

[83] Strong, D. R. and T. S. Ray. 1975. Host tree location behavior of a tropical vine (*Monstera gigantea*) by skototropism. Science, 190: 804–06.

[84] Surkan, Al. Unpublished. Self-balancing of dynamic population sectors that consume energy. Department of computer science, UNL. "Tierra-like systems are being explored for their potential applications in solving the problem of predicting the dynamics of consumption of a single energy carrying natural resource". Contact: surkan@cse.unl.edu

[85] Tackett, Walter, and Jean-Luc Gaudiot. 1993. Adaptation of self-replicating digital organisms. Proceedings of the International Joint Conference on Neural Networks, Nov. 1993, Beijing, China. IEEE Press. Contact: tackett@ipld01.hac.com, tackett@priam.usc.edu

[86] Taylor, Charles E., David R. Jefferson, Scott R. Turner, and Seth R. Goldman. 1989. RAM: artificial life for the exploration of complex biological systems. *In*: Langton, C. [ed], Artificial Life, Santa Fe Institute Studies in the Sciences of Complexity, vol. VI, 275–295. Redwood City, CA: Addison-Wesley.

[87] Thearling, Kurt, and Ray, T. S. 1994. Evolving multi-cellular artificial life. Brooks, Rodney A., and Pattie Maes [eds.], Artificial Life IV conference proceedings, Pp. 283–288. The MIT Press, Cambridge.

[88] Todd, Peter M. 1993. Artificial death. Proceedings of the Second European Conference on Artificial Life (ECAL93), Vol. 2, Pp. 1048–1059. Brussels, Belgium: Universite Libre de Bruxelles. Contact: ptodd@spo.rowland.org

[89] Volterra, V. "Variations and fluctuations of the number of individuals in animal species living together." In: *Animal Ecology*, edited by R. N. Chapman. New York: McGraw-Hill, 1926, 409–448.

[90] Wilson, E. O. & Bossert, W. H. *A primer of population biology.* Stamford, Conn: Sinauer Associates, 1971.

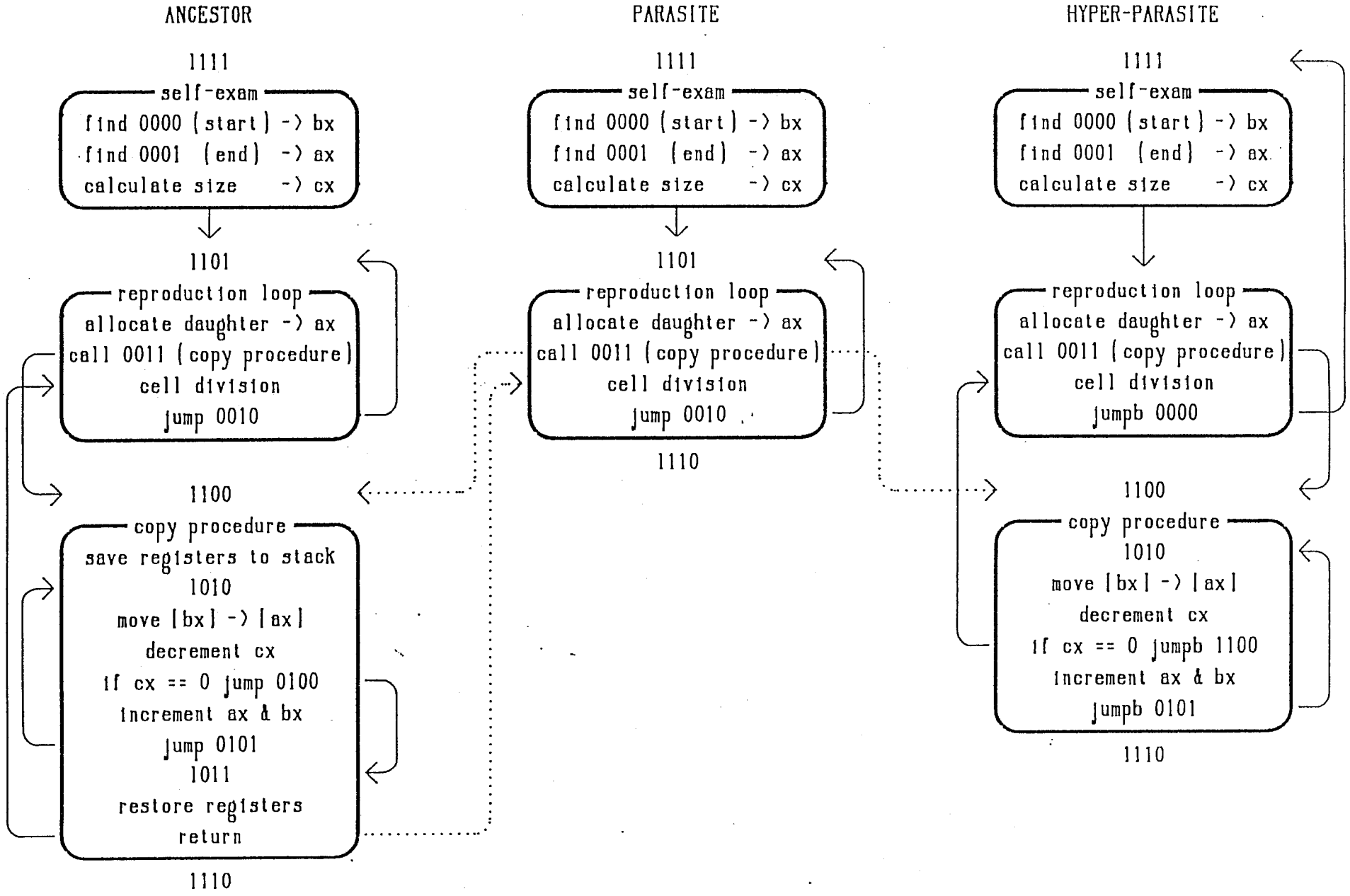
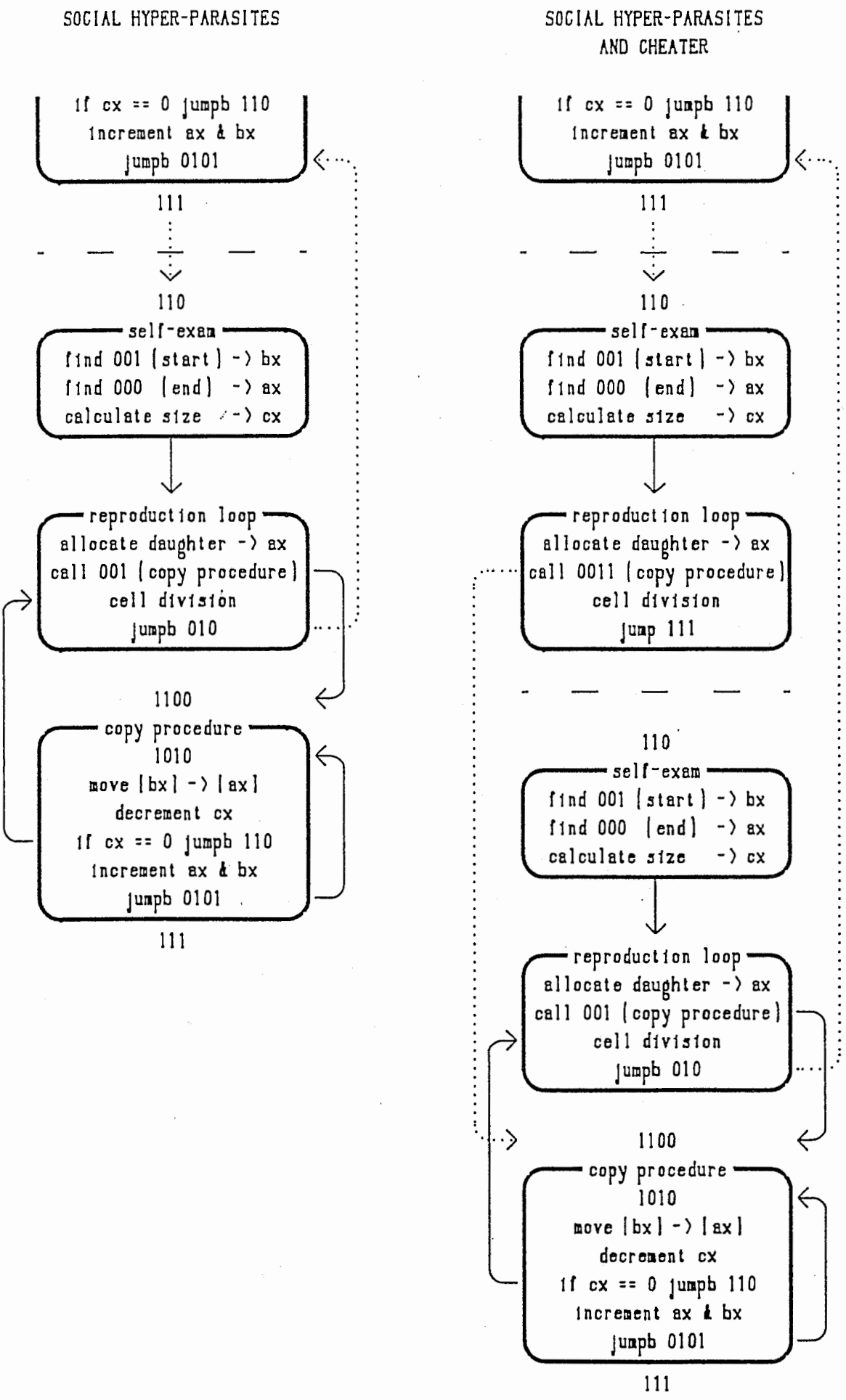

ANCESTOR
1111
self-exam
find 0000 (start) -> bx
find 0001 (end) -> ax
calculate size -> cx
1101
reproduction loop
allocate daughter -> ax
call 0011 (copy procedure)
cell division
jump 0010
1100
copy procedure
save registers to stack
1010
move [bx] -> [ax]
decrement cx
if cx == 0 jump 0100
increment ax & bx
jump 0101
1011
restore registers
return
1110
PARASITE
1111
self-exam
find 0000 (start) -> bx
find 0001 (end) -> ax
calculate size -> cx
1101
reproduction loop
allocate daughter -> ax
call 0011 (copy procedure)
cell division
jump 0010
1110
HYPER-PARASITE
1111
self-exam
find 0000 (start) -> bx
find 0001 (end) -> ax
calculate size -> cx
1101
reproduction loop
allocate daughter -> ax
call 0011 (copy procedure)
cell division
jumpb 0000
1100
copy procedure
1010
move [bx] -> [ax]
decrement cx
if cx == 0 jumpb 1100
increment ax & bx
jumpb 0101
1110

Fig. 1

Fig. 2


SOCIAL HYPER-PARASITES
if cx == 0 jumpb 110
increment ax & bx
jumpb 0101
111
110
self-exam
find 001 [start] -> bx
find 000 [end] -> ax
calculate size -> cx
reproduction loop
allocate daughter -> ax
call 001 [copy procedure]
cell division
jumpb 010
1100
copy procedure
1010
move [bx] -> [ax]
decrement cx
if cx == 0 jumpb 110
increment ax & bx
jumpb 0101
111
SOCIAL HYPER-PARASITES
AND CHEATER
if cx == 0 jumpb 110
increment ax & bx
jumpb 0101
111
110
self-exam
find 001 [start] -> bx
find 000 [end] -> ax
calculate size -> cx
reproduction loop
allocate daughter -> ax
call 0011 [copy procedure]
cell division
jump 111
110
self-exam
find 001 [start] -> bx
find 000 [end] -> ax
calculate size -> cx
reproduction loop
allocate daughter -> ax
call 001 [copy procedure]
cell division
jumpb 010
1100
copy procedure
1010
move [bx] -> [ax]
decrement cx
if cx == 0 jumpb 110
increment ax & bx
jumpb 0101
111

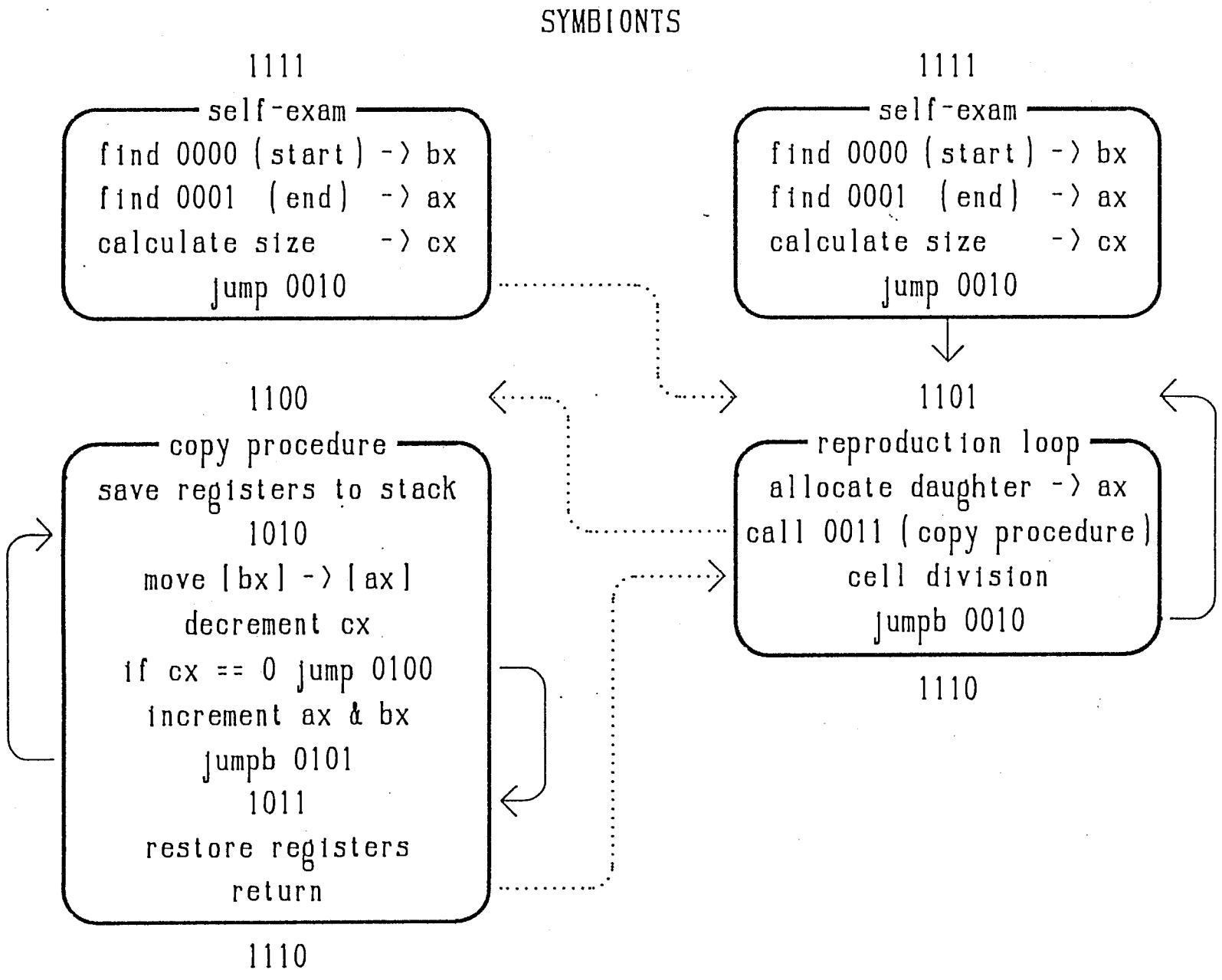

SYMBIONTS
1111
self-exam
find 0000 (start) -> bx
find 0001 (end) -> ax
calculate size -> cx
jump 0010
1111
self-exam
find 0000 (start) -> bx
find 0001 (end) -> ax
calculate size -> cx
jump 0010
1100
copy procedure
save registers to stack
1010
move [bx] -> [ax]
decrement cx
if cx == 0 jump 0100
increment ax & bx
jumpb 0101
1011
restore registers
return
1110
1101
reproduction loop
allocate daughter -> ax
call 0011 (copy procedure)
cell division
jumpb 0010
1110

Fig. 3

Fig. 4

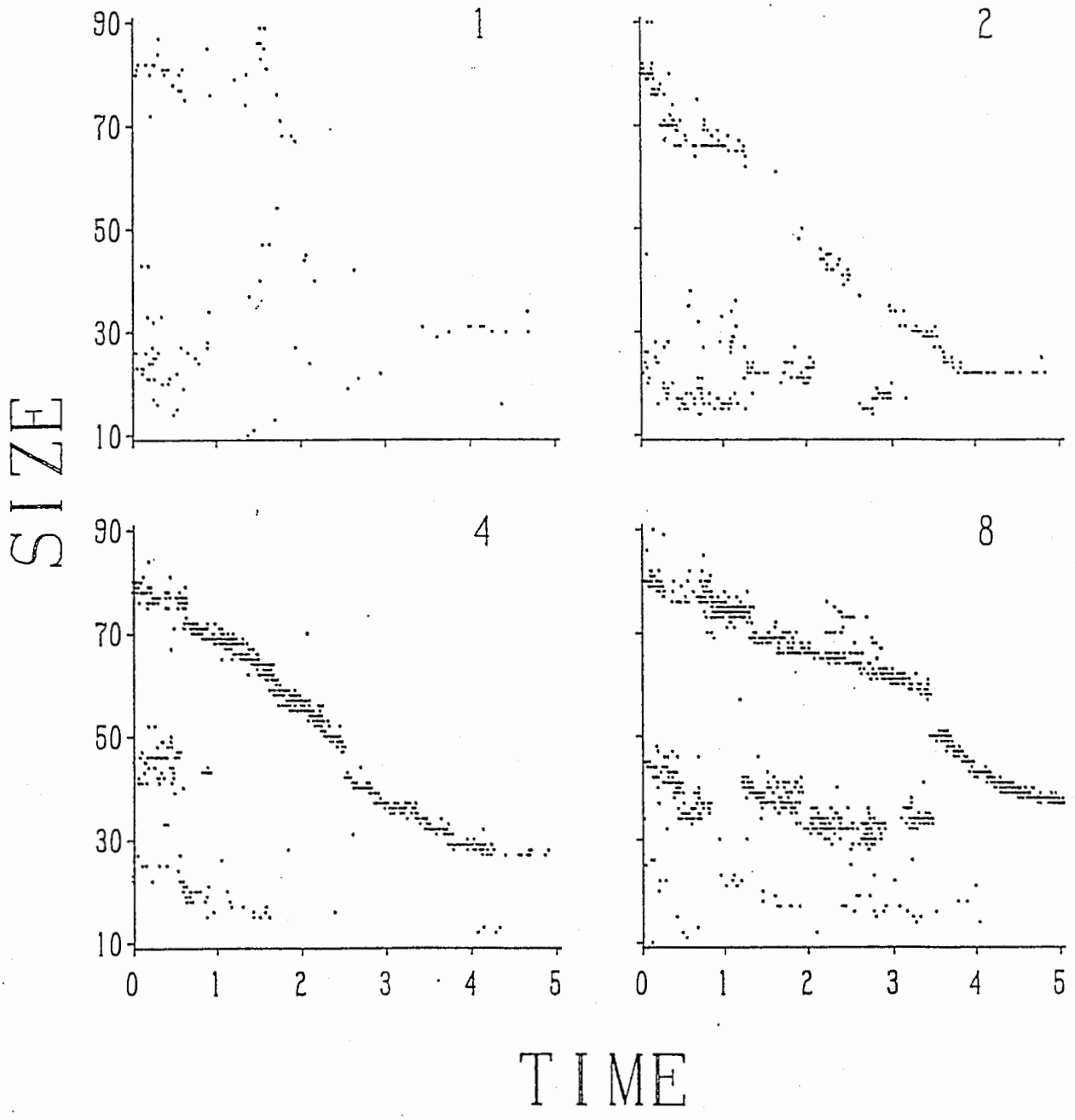


Fig. 4

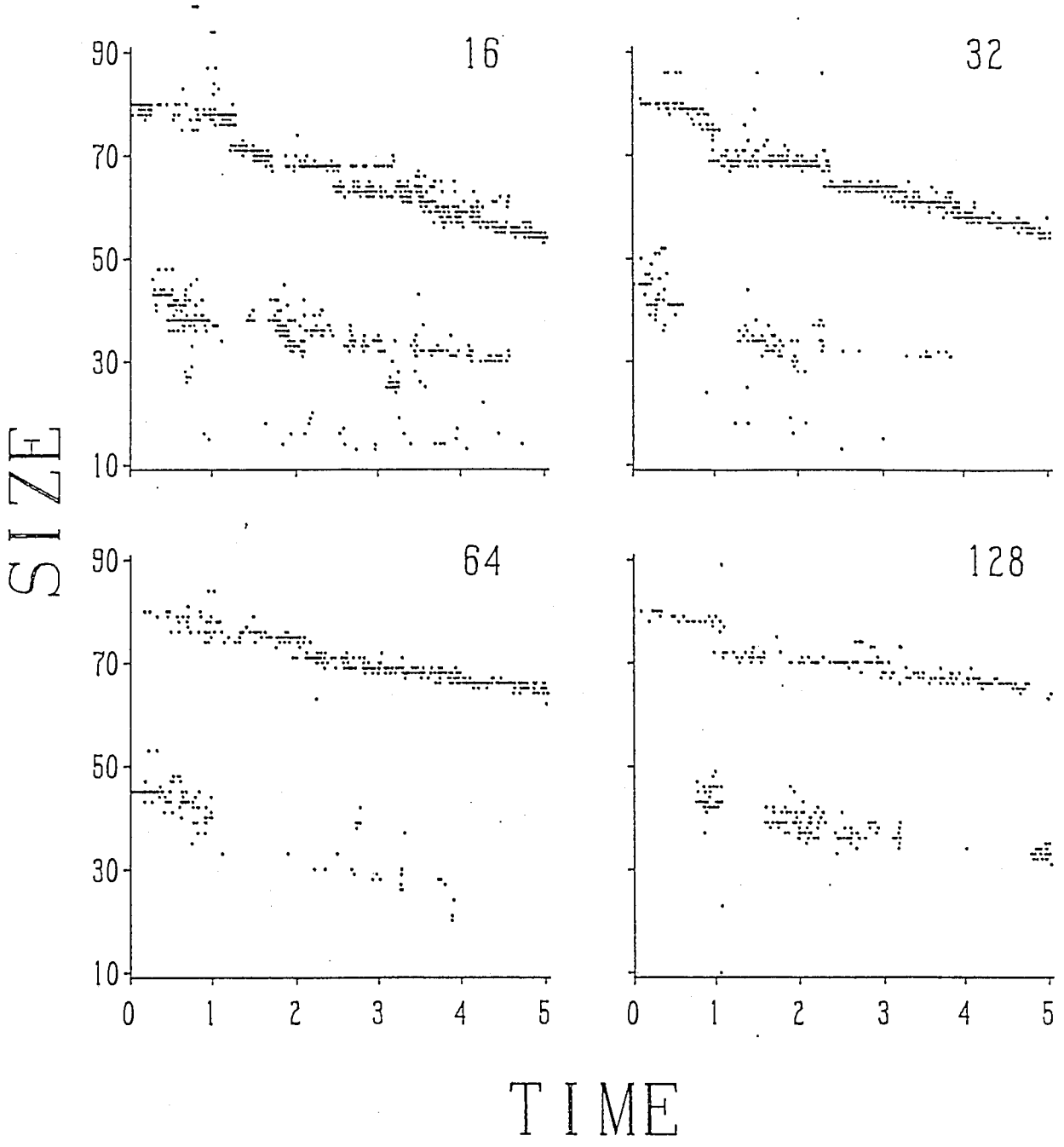


Fig. 5

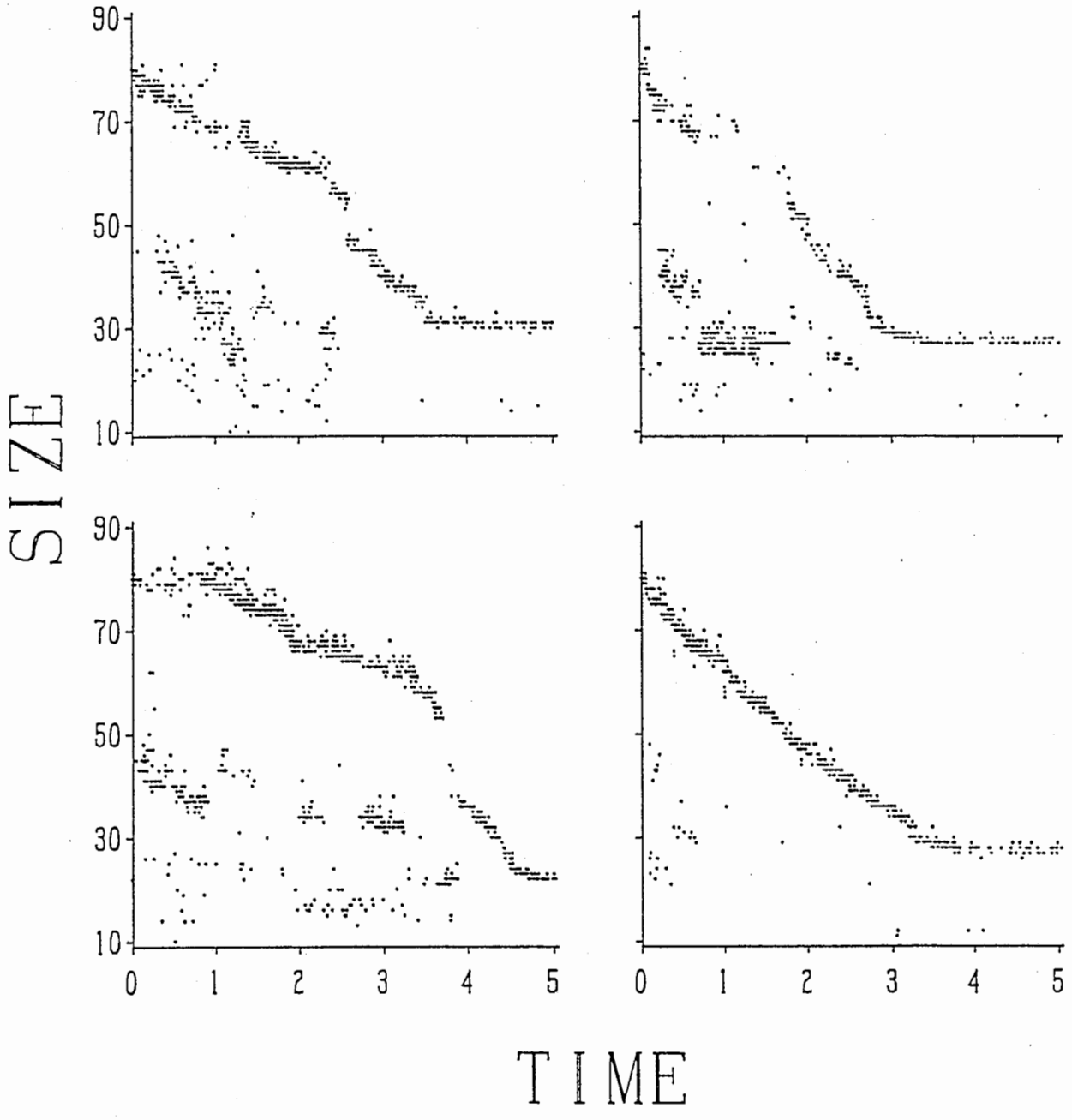


Fig. 6

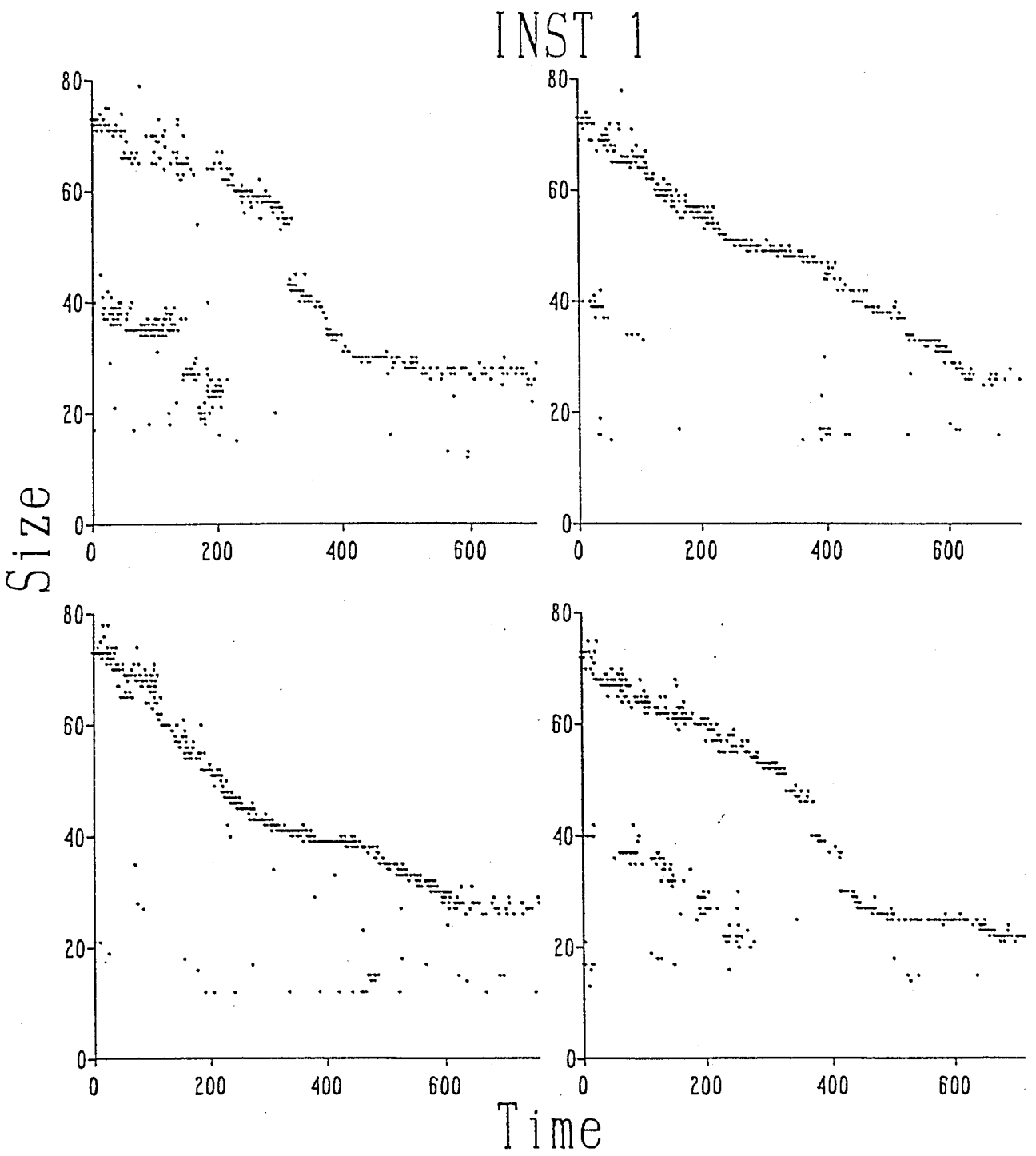


Fig. 6

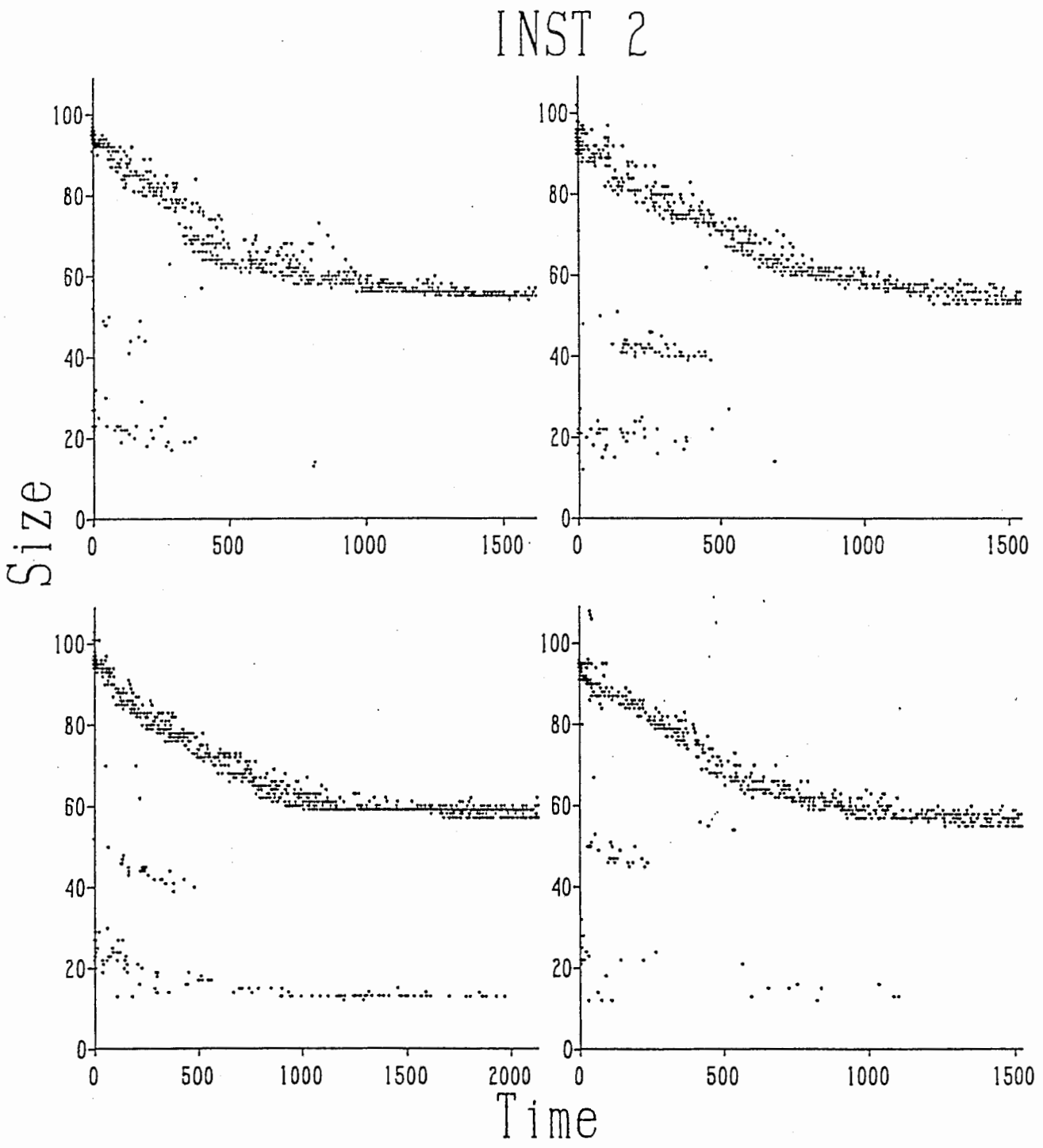


Fig. 6

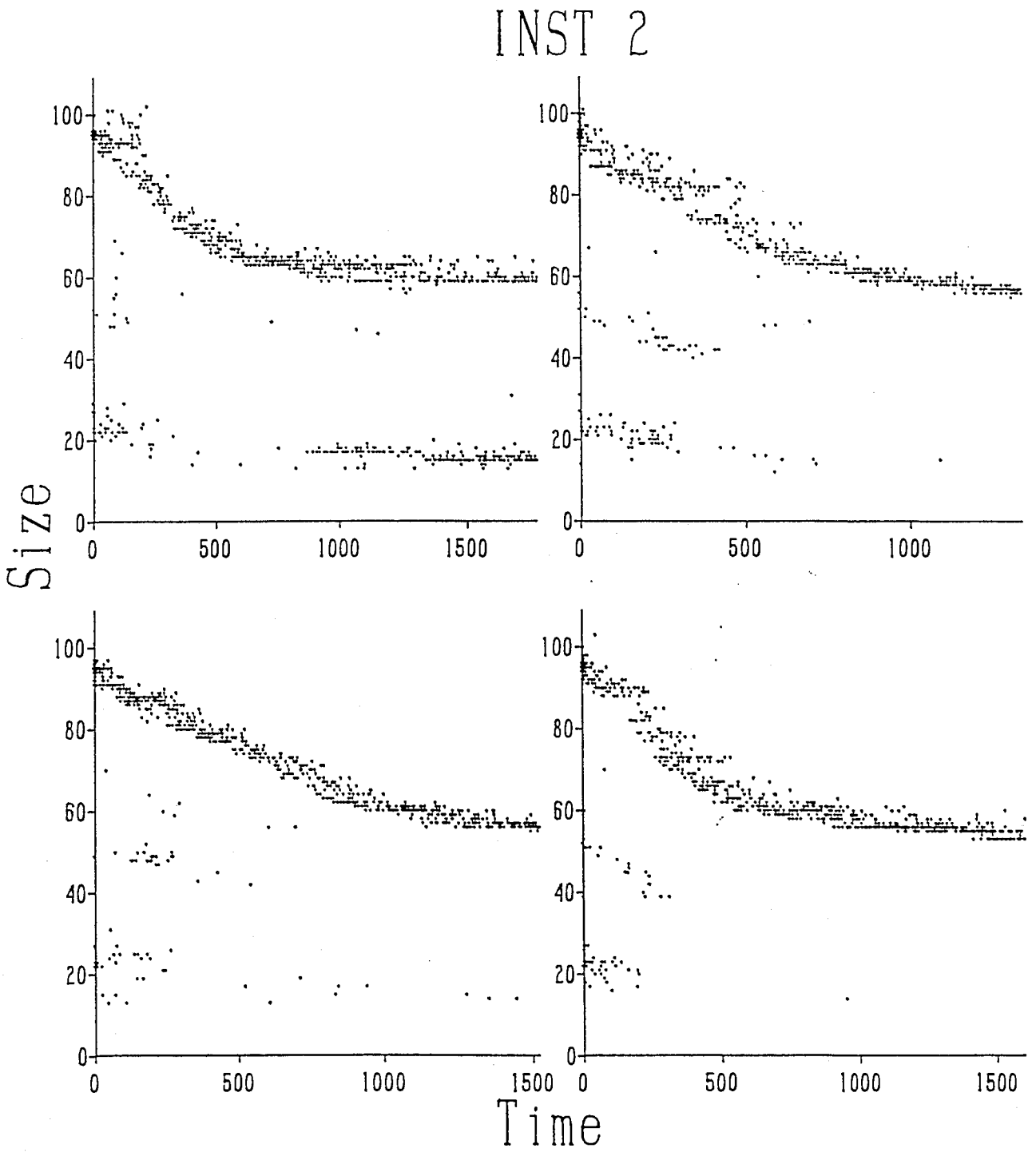


Fig. 6

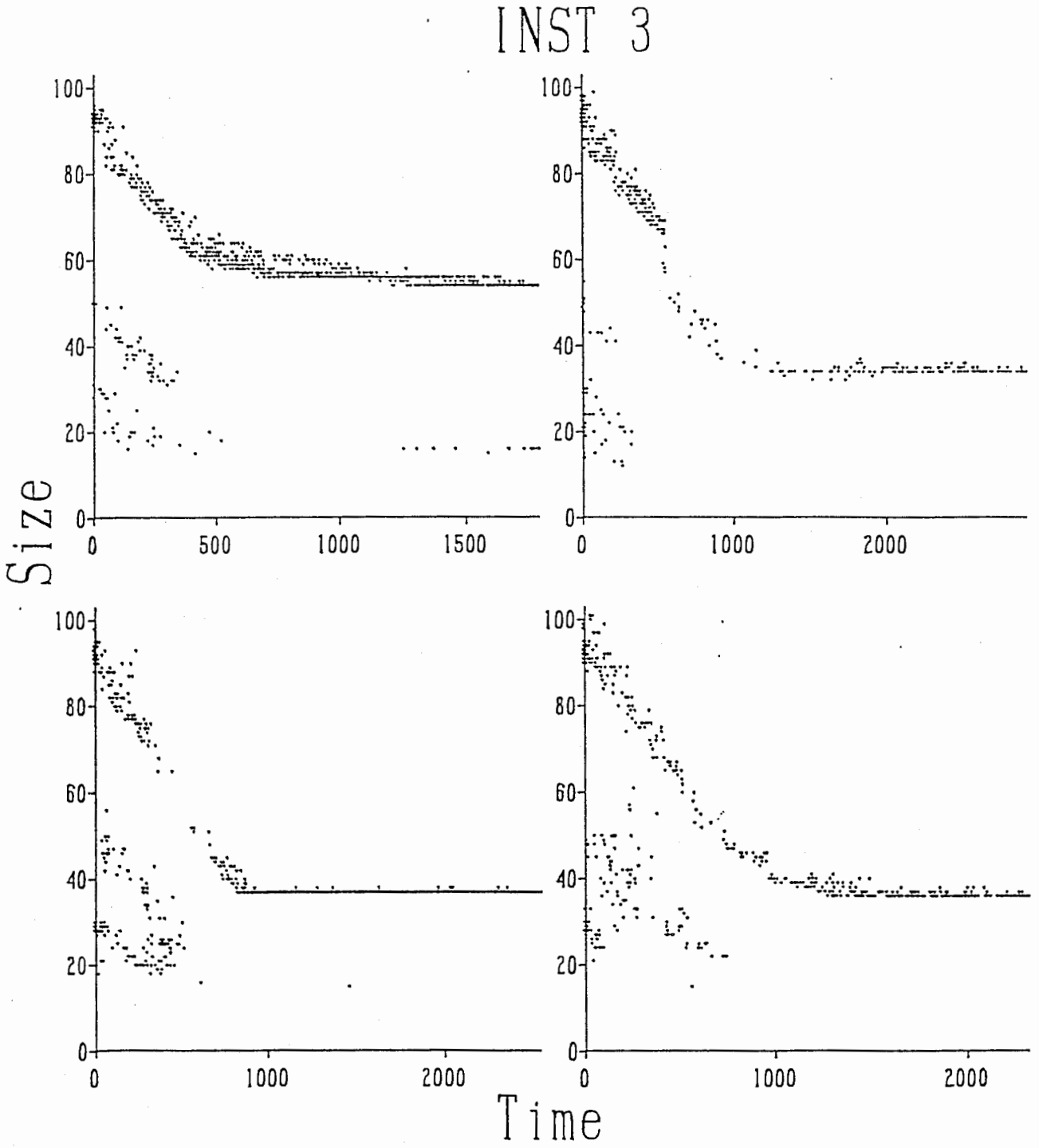


Fig. 6

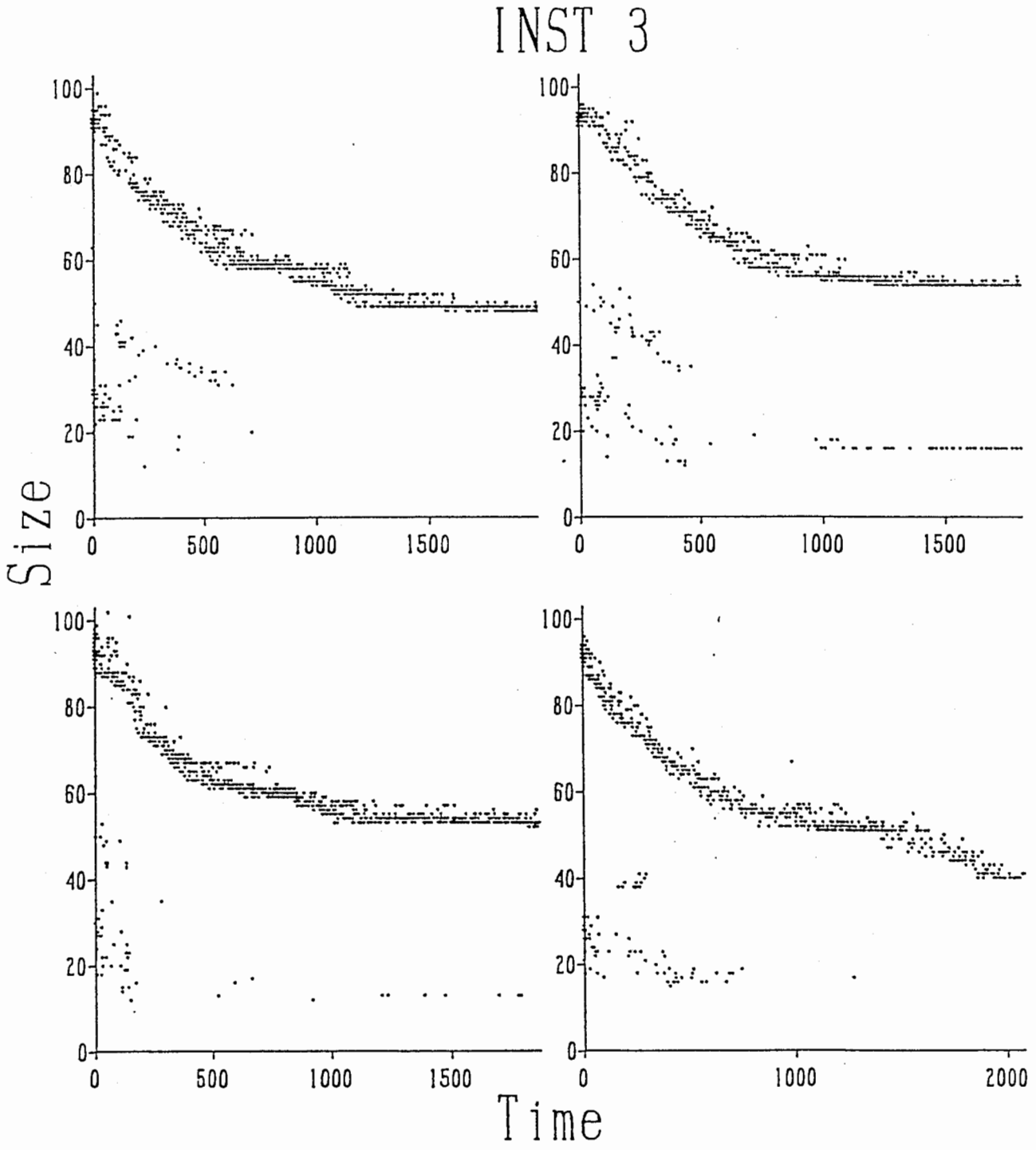


Fig. 6

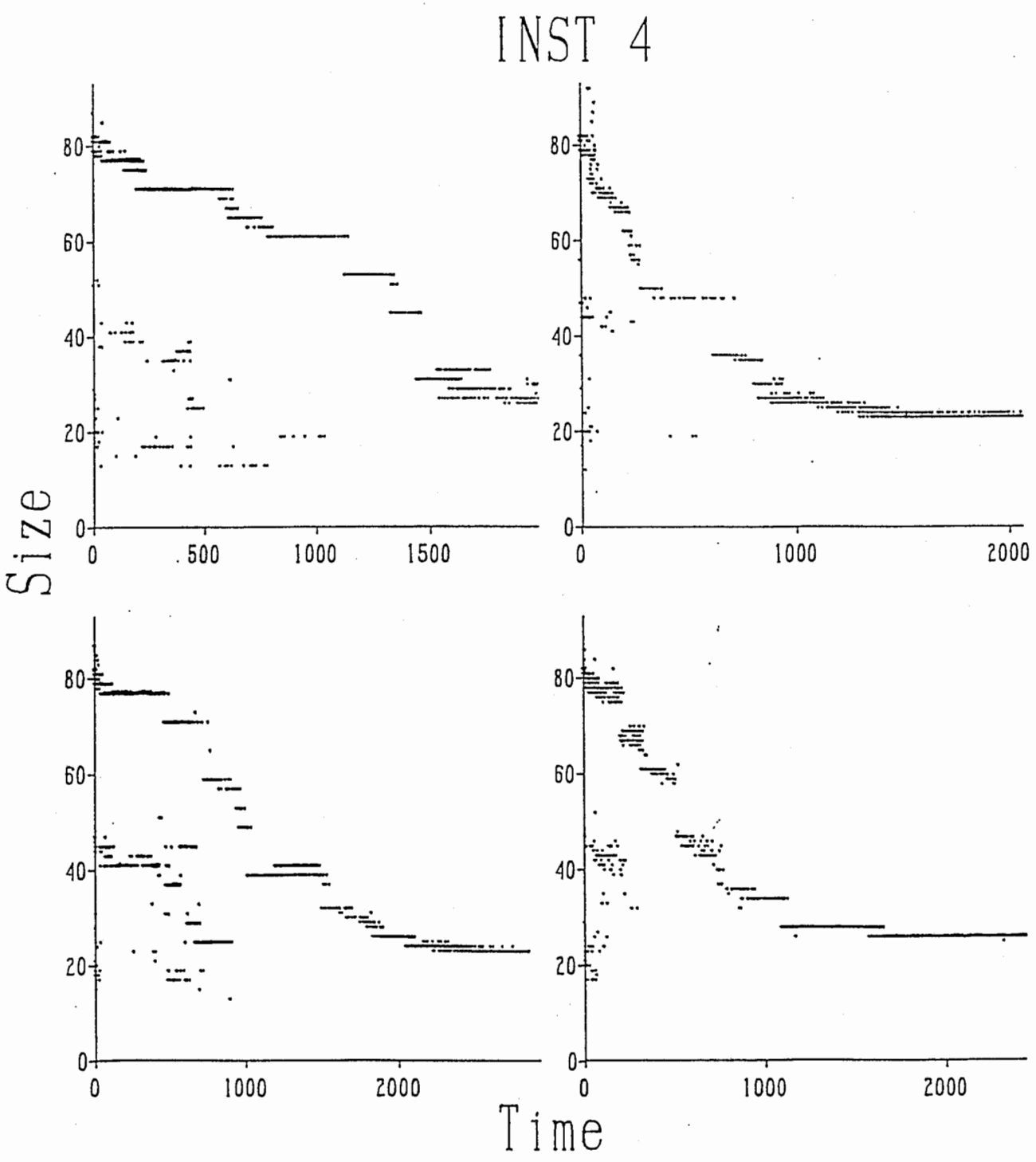


Fig. 6

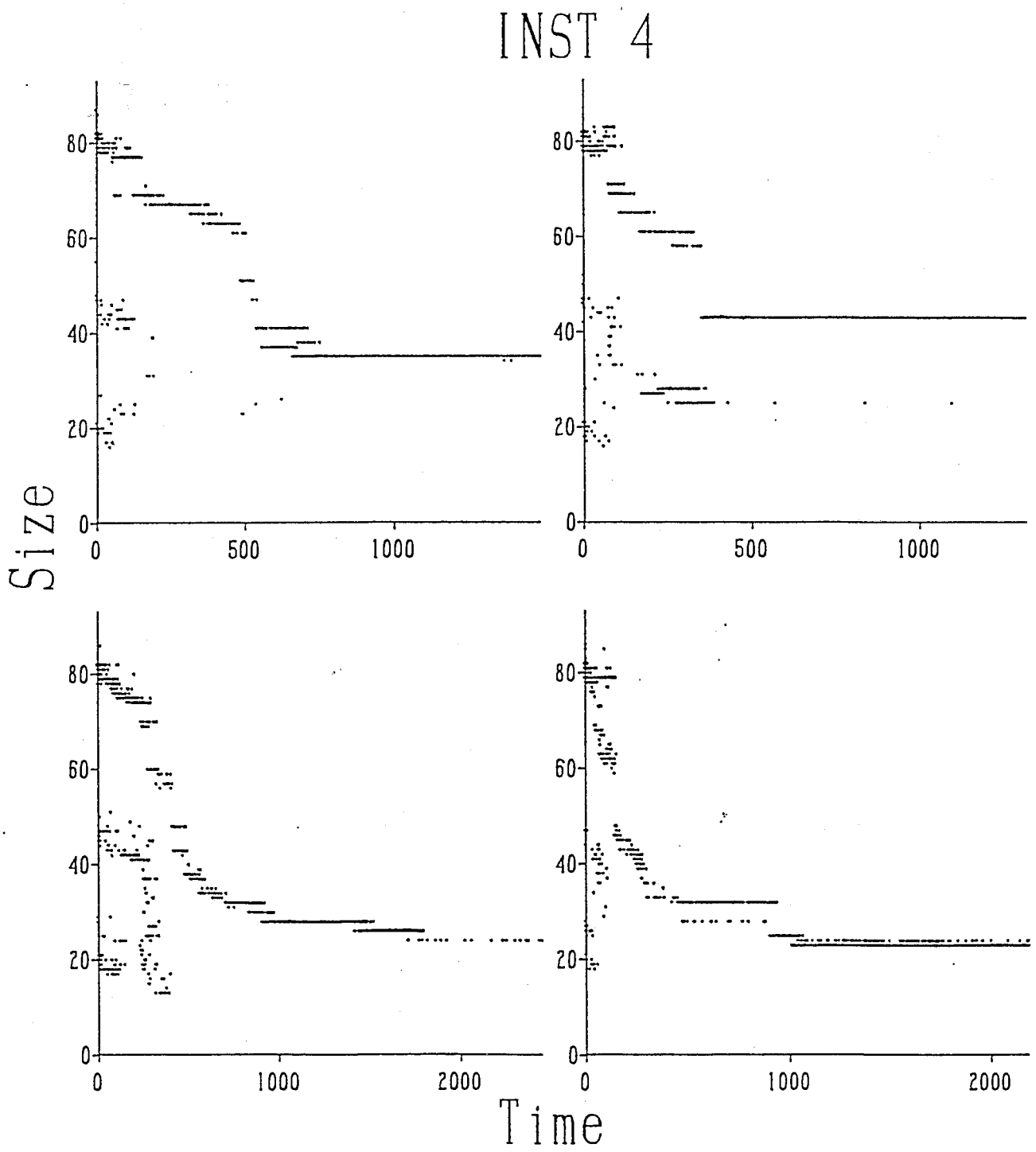
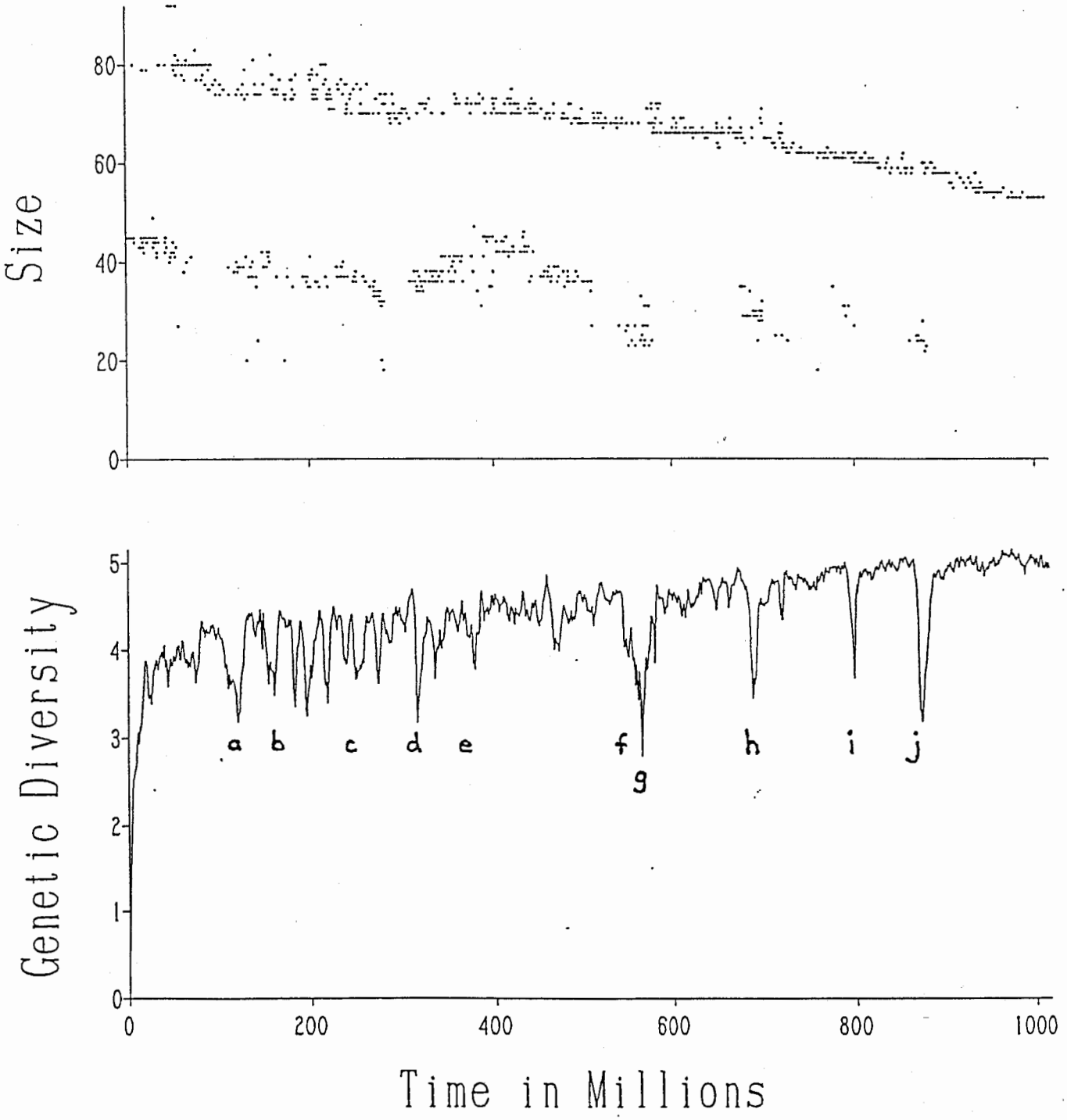


Fig. 7


Size
80
60
40
20
0
Genetic Diversity
5
4
3
2
1
0
a
b
c
d
e
f
g
h
i
j
0
200
400
600
800
1000
Time in Millions