1

ARTIFICIAL LIFE

CHRISTOPHER G. LANGTON

1. THE BIOLOGY OF POSSIBLE LIFE

Biology is the scientific study of life—in principle anyway. In practice, biology is the scientific study of life on Earth based on carbon-chain chemistry. There is nothing in its charter that restricts biology to the study of carbon-based life; it is simply that this is the only kind of life that has been available to study. Thus, theoretical biology has long faced the fundamental obstacle that it is impossible to derive general principles from single examples.

Without other examples it is extremely difficult to distinguish essential properties of life—properties that must be shared by any living system *in principle*—from properties that may be incidental to life, but which happen to be universal to life on Earth due *solely* to a combination of local historical accident and common genetic descent. Since it is quite unlikely that organisms based on different physical chemistries will present themselves to us for study in the foreseeable future, our only alternative is to try to synthesize alternative life-forms ourselves—*Artificial Life*: life made by man rather than by nature.

1.1. Artificial Life

Biology has traditionally started at the top, viewing a living organism as a complex biochemical machine, and has worked *analytically* down from there through the hierarchy of biological organization—decomposing a living organism into organs, tissues, cells, organelles, and finally molecules—in its pursuit of the mechanisms of life. Analysis means 'the separation of an intellectual or substantial whole into constituents for individual study'. By composing our individual understandings of the dissected component parts of living organisms,

 Reprinted from L. Nadel and D. Stein (eds.), *1991 Lectures in Complex Systems* (Santa Fe Institute Studies in the Sciences of Complexity, Lectures, 4; Reading, Mass: Addison-Wesley, 1992), 189–241. Reprinted by permission. This is an updated version of a paper that originally appeared in Langton (1989*b*).

traditional biology has provided us with a broad picture of the mechanics of life on Earth.

But there is more to life than mechanics—there is also dynamics. Life depends critically on principles of dynamical self-organization that have remained largely untouched by traditional analytic methods. There is a simple explanation for this—these self-organizing dynamics are fundamentally non-linear phenomena, and non-linear phenomena in general depend critically on the interactions *between* parts: they necessarily disappear when parts are treated in isolation from one another, which is the basis for the analytic method.

Rather, non-linear phenomena are most appropriately treated by a *synthetic* approach. Synthesis means 'the combining of separate elements or substances to form a coherent whole'. In non-linear systems, the parts must be treated in each other's presence, rather than independently from one another, because they behave very differently in each other's presence than we would expect from a study of the parts in isolation.

Artificial Life is simply the synthetic approach to biology: rather than take living things apart, Artificial Life attempts to put living things together.

But Artificial Life is more than this. To understand the overall aims of the Artificial Life enterprise, one needs to do the following: (1) Broaden the scope of the attempts, beyond simply recreating 'the living state', to the synthesis of any and all biological phenomena, from viral self-assembly to the evolution of the entire biosphere. (2) Couple this with the observation that there is no reason, in principle, why the parts we use in our attempts to synthesize these biological phenomena need be restricted to carbon-chain chemistry. (3) Note that we expect the synthetic approach to lead us not only to, but quite often *beyond*, known biological phenomena: beyond *life-as-we-know-it* into the realm of *life-as-it-could-be*.

Thus, for example, Artificial Life involves attempts to (1) synthesize the process of evolution (2) in computers, and (3) will be interested in whatever emerges from the process, even if the results have no analogues in the 'natural' world. It is certainly of scientific interest to know what kinds of things *can* evolve in principle, whether or not they happened to do so here on Earth.

1.2. AI and the Behaviour Generation Problem

Artificial Life is concerned with generating *lifelike* behaviour. Thus it focuses on the problem of creating *behaviour generators*. A good place to start is to identify the mechanisms by which behaviour is generated and controlled in natural systems, and to recreate these mechanisms in artificial systems. This is the course we shall take later in this paper.

The related field of Artificial Intelligence is concerned with generating *intelligent* behaviour. It, too, focuses on the problem of creating behaviour generators. However, although it initially looked to natural intelligence to identify its underlying mechanisms, these mechanisms were not known, nor are they today. Therefore, following an initial flirt with neural nets, AI became wedded to the only other known vehicle for the generation of complex behaviour: the technology of serial computer programming. As a consequence, from the very beginning artificial intelligence embraced an underlying methodology for the generation of intelligent behaviour that bore no demonstrable relationship to the method by which intelligence is generated in natural systems. In fact, AI has focused primarily on the production of intelligent *solutions* rather than on the production of intelligent *behaviour*. There is a world of difference between these two possible foci.

By contrast, Artificial Life has the great good fortune that many of the mechanisms by which behaviour arises in natural living systems are known. There are still many holes in our knowledge, but the general picture is in place. Therefore, Artificial Life can start by recapturing natural life and has no need to resort to the sort of initial infidelity that is now coming back to haunt AI.

The key insight into the natural method of behaviour generation is gained by noting that *nature is fundamentally parallel*. This is reflected in the 'architecture' of natural living organisms, which consist of many millions of parts, each one of which has its own behavioural repertoire. Living systems are highly distributed and quite massively parallel. If our models are to be true to life, they must also be highly distributed and quite massively parallel. Indeed, it is unlikely that any other approach will prove viable.

2. HISTORICAL ROOTS OF ARTIFICIAL LIFE

Mankind has a long history of attempting to map the mechanics of his contemporary technology on to the workings of nature, trying to understand the latter in terms of the former.

It is not surprising, therefore, that early models of life reflected the principal technology of their era. The earliest models were simple statuettes and paintings—works of art which captured the static form of living things. These statues were provided with articulated arms and legs in the attempt to capture the dynamic form of living things. These simple statues incorporated no internal dynamics, requiring human operators to make them behave.

The earliest mechanical devices that were capable of generating their own behaviour were based on the technology of water transport. These were the early Egyptian water-clocks called *Clepsydra*. These devices made use of a

rate-limited process—in this case the dripping of water through a fixed orifice—to indicate the progression of another process—the position of the sun. Ctesibius of Alexandria developed a water-powered mechanical clock around 135 BC which employed a great deal of the available hydraulic technology—including floats, a siphon, and a water-wheel-driven train of gears.

In the first century AD, Hero of Alexandria produced a treatise on *Pneumatics*, which described, among other things, various simple gadgets in the shape of animals and humans that utilized pneumatic principles to generate simple movements.

However, it was really not until the age of mechanical clocks that artefacts exhibiting complicated internal dynamics became possible. Around AD 850 the *mechanical escapement* was invented, which could be used to regulate the power provided by falling weights. This invention ushered in the great age of clockwork technology. Throughout the Middle Ages and the Renaissance, the history of technology is largely bound up with the technology of clocks. Clocks often constituted the most complicated and advanced application of the technology of an era.

Perhaps the earliest clockwork simulations of life were the so-called 'Jacks', mechanical 'men' incorporated in early clocks who would swing a hammer to strike the hour on a bell. The word 'jack' is derived from 'jaccomarchiadus', which means 'the man in the suit of armour'. These accessory figures retained their popularity even after the spread of clock dials and hands—to the extent that clocks were eventually developed in which the function of timekeeping was secondary to the control of large numbers of figures engaged in various activities, to the point of acting out entire plays.

Finally, clockwork mechanisms appeared which had done away altogether with any pretence at timekeeping. These 'automata' were entirely devoted to imparting lifelike motion to a mechanical figure or animal. These mechanical automaton simulations of life included such things as elephants, peacocks, singing birds, musicians, and even fortune-tellers.

This line of development reached its peak in the famous duck of Vaucanson, described as 'an artificial duck made of gilded copper who drinks, eats, quacks, splashes about on the water, and digests his food like a living duck'.[1] Vaucanson's goal is captured neatly in the following description:

> In 1735 Jacques de Vaucanson arrived in Paris at the age of 26. Under the influence of contemporary philosophic ideas, he had tried, it seems, to reproduce life artificially.

Unfortunately, neither the duck itself nor any technical descriptions or diagrams remain that would give the details of construction of this duck. The complexity

[1] All quotes concerning these mechanical ducks are from Chapuis and Droz (1958).

of the mechanism is attested to by the fact that one single wing contained over 400 articulated pieces.

One of those called upon to repair Vaucanson's duck in later years was a 'mechanician' named Reichsteiner, who was so impressed with it that he went on to build a duck of his own—also now lost—which was exhibited in 1847. Here is an account of this duck's operation from the newspaper *Das Freie Wort*:

> After a light touch on a point on the base, the duck in the most natural way in the world begins to look around him, eyeing the audience with an intelligent air. His lord and master, however, apparently interprets this differently, for soon he goes off to look for something for the bird to eat. No sooner has he filled a dish with oatmeal porridge than our famished friend plunges his beak deep into it, showing his satisfaction by some characteristic movements of his tail. The way in which he takes the porridge and swallows it greedily is extraordinarily true to life. In next to no time the basin has been half emptied, although on several occasions the bird, as if alarmed by some unfamiliar noises, has raised his head and glanced curiously around him. After this, satisfied with his frugal meal, he stands up and begins to flap his wings and to stretch himself while expressing his gratitude by several contented quacks. But most astonishing of all are the contractions of the bird's body clearly showing that his stomach is a little upset by this rapid meal and the effects of a painful digestion become obvious. However, the brave little bird holds out, and after a few moments we are convinced in the most concrete manner that he has overcome his internal difficulties. The truth is that the smell which now spreads through the room becomes almost unbearable. We wish to express to the artist inventor the pleasure which his demonstration gave to us.

Fig. 1.1 shows two views of one of the ducks—there is some controversy as to whether it is Vaucanson's or Reichsteiner's. The mechanism inside the duck would have been completely covered with feathers and the controlling mechanism in the box below would have been covered up as well.

2.1. The Development of Control Mechanisms

Out of the technology of the clockwork regulation of automata came the more general—and perhaps ultimately more important—technology of *process control*. As attested to in the descriptions of the mechanical ducks, some of the clockwork mechanisms had to control remarkably complicated actions on the part of the automata, not only *powering* them but *sequencing* them as well.

Control mechanisms evolved from early, simple devices—such as a lever attached to a wheel which converted circular motion into linear motion—to later, more complicated devices—such as whole sets of cams upon which would ride many interlinked mechanical arms, giving rise to extremely complicated automaton behaviours.

Eventually *programmable controllers* appeared, which incorporated such devices as interchangeable cams, or drums with movable pegs, with which one could program arbitrary sequences of actions on the part of the automaton. The

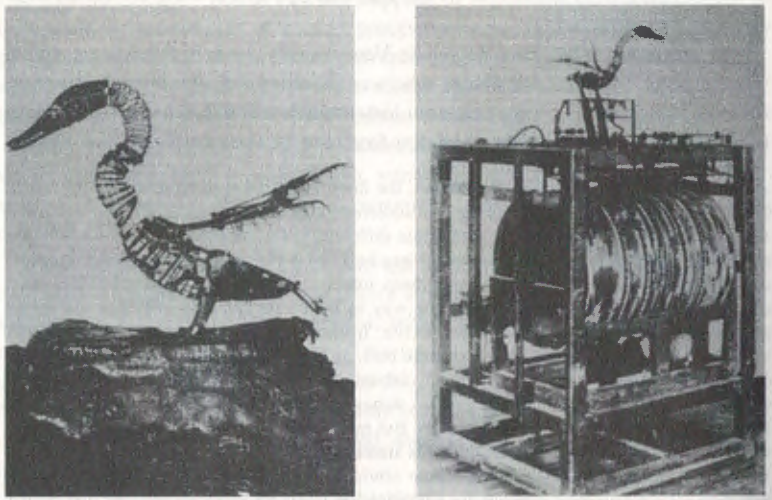


FIG. 1.1. Two views of the mechanical duck attributed to Vaucanson. From Chapuis and Droz (1958). Reprinted by permission

writing and picture-drawing automata of Fig. 1.2, built by the Jaquet-Droz family, are examples of programmable automata. The introduction of such programmable controllers was one of the primary developments on the road to general-purpose computers.

2.2. *Abstraction of the Logical 'Form' of Machines*

During the early part of the twentieth century, the formal application of logic to the mechanical process of arithmetic led to the abstract formulation of a 'procedure'. The work of Church, Kleene, Gödel, Turing, and Post formalized the notion of a logical sequence of steps, leading to the realization that the essence of a mechanical process—the 'thing' responsible for its dynamic behaviour—is not a thing at all, but an abstract control structure, or 'program'—a sequence of simple actions selected from a finite repertoire. Furthermore, it was recognized that the essential features of this control structure could be captured within an abstract set of rules—a formal specification—without regard to the material out of which the machine was constructed. The 'logical form' of a machine was separated from its material basis of construction, and it was found that 'machineness' was a property of the former, not of the latter. Today, the

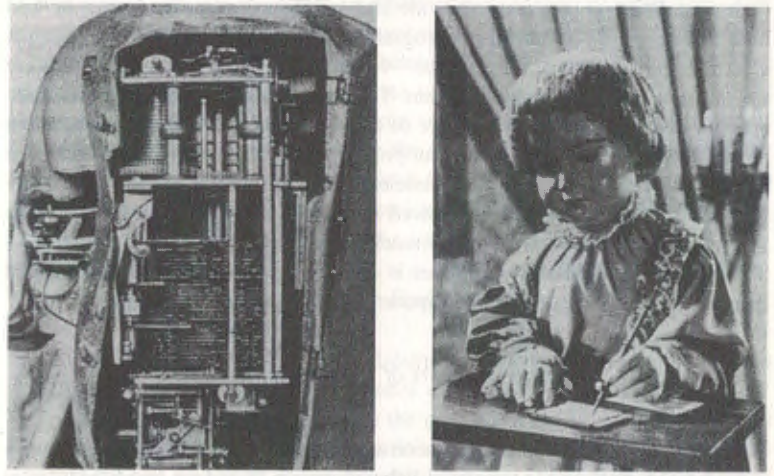


FIG. 1.2. Two views of a drawing automaton built by the Jaquet-Droz family. From Chapuis and Droz (1958). Reprinted by permission

formal equivalent of a 'machine' is an *algorithm*: the logic underlying the dynamics of an automaton, regardless of the details of its material construction. We now have many formal methods for the specification and operation of abstract machines: such as programming languages, formal language theory, automata theory, recursive function theory, etc. All these have been shown to be logically equivalent.

Once we have learned to think of machines in terms of their abstract, formal specifications, we can turn around and view abstract, formal specifications as potential machines. In mapping the machines of our common experience to formal specifications, we have by no means exhausted the space of *possible* specifications. Indeed, most of our individual machines map to a very small subset of the space of specifications—a subset largely characterized by methodical, boring, uninteresting dynamics.

2.3. *General-Purpose Computers*

Various threads of technological development—programmable controllers, calculating engines, and the formal theory of machines—have come together in the general-purpose, stored-program computer. Programmable computers are extremely general behaviour generators. They have no intrinsic behaviour of

their own. Without programs, they are like formless matter. They must be told how to behave. By submitting a program to a computer—that is: by giving it a formal specification for a machine—we are telling it to behave as if it were the machine specified by the program. The computer then 'emulates' that more specific machine in the performance of the desired task. Its great power lies in its plasticity of behaviour. If we can provide a step-by-step specification for a specific kind of behaviour, the chameleon-like computer will exhibit that behaviour. Computers should be viewed as *second-order* machines—given the formal specification of a first-order machine, they will 'become' that machine. Thus the space of possible machines is directly available for study, at the cost of a mere formal description: computers 'realize' abstract machines.

2.4. Formal Limits of Machine Behaviours

Although computers, and by extension other machines, are capable of exhibiting a bewilderingly wide variety of behaviours, we must face two fundamental limitations on the kinds of behaviours that we can expect of computers.

The first limitation is one of *computability in principle*. There are certain behaviours that are 'uncomputable'—behaviours for which *no* formal specification can be given for a machine that will exhibit that behaviour. The classic example of this sort of limitation is Turing's famous *Halting Problem*: can we give a formal specification for a machine which, when provided with the description of *any* other machine together with its initial state, will—by inspection alone—determine whether or not that machine will reach its halt state? Turing proved that no such machine can be specified. In particular, Turing showed that the best that such a proposed machine could do would be to emulate the given machine to see whether or not it halted. If the emulated machine halted, fine. However, the emulated machine might run forever without halting, and therefore the emulating machine could not answer whether or not it would halt. Rice and others (in Hopcroft and Ullman 1979) have extended this undecidability result to the determination—by inspection alone—of *any* non-trivial property of the future behaviour of an arbitrary machine.

The second limitation is one of *computability in practice*. There are many behaviours for which we do not know how to specify a sequence of steps that will cause a computer to exhibit that behaviour. We can automate what we can explain how to do, but there is much that we cannot explain how to do. Thus, although a formal specification for a machine that will exhibit a certain behaviour may be possible *in principle*, we have no formal procedure for producing that formal specification in practice, short of a trial-and-error search through the space of possible descriptions.

We need to separate the notion of a formal specification of a machine—that is, a specification of the *logical structure* of the machine—from the notion of a formal specification of a machine's behaviour—that is, a specification of the sequence of transitions that the machine will undergo. In general, we cannot derive behaviours from structure, nor can we derive structure from behaviours.

The moral is: in order to determine the behaviour of some machines, there is no recourse but to run them and see how they behave! This has consequences for the methods by which we (or nature) go about *generating* behaviour generators themselves, which we shall take up in the section on evolution.

2.5. John von Neumann: From Mechanics to Logic

With the development of the general-purpose computer, various researchers turned their attention from the *mechanics* of life to the *logic* of life.

The first computational approach to the generation of lifelike behaviour was due to the brilliant Hungarian mathematician John von Neumann. In the words of his colleague Arthur W. Burks, von Neumann was interested in the general question:

> What kind of logical organization is sufficient for an automaton to reproduce itself? This question is not precise and admits to trivial versions as well as interesting ones. Von Neumann had the familiar natural phenomenon of self-reproduction in mind when he posed it, but he was not trying to simulate the self-reproduction of a natural system at the level of genetics and biochemistry. *He wished to abstract from the natural self-reproduction problem its logical form.*[2]

This approach is the first to capture the essence of Artificial Life. To understand the field of Artificial Life, one need only replace references to 'self-reproduction' in the above with references to any other biological phenomenon.

In von Neumann's initial thought-experiment (his 'kinematic model'), a machine floats around on the surface of a pond, together with lots of machine parts. The machine is a *universal constructor*: given the description of any machine, it will locate the proper parts and construct that machine. If given a description of itself, it will construct itself. This is not quite self-reproduction, however, because the offspring machine will not have a description of itself and hence could not go on to construct another copy. So, von Neumann's machine also contains a *description copier*: once the offspring machine has been constructed, the 'parent' machine constructs a copy of the description that it worked from and attaches it to the offspring machine. This constitutes genuine self-reproduction.

Von Neumann decided that this model did not properly distinguish the logical

[2] From Burks (1970), emphasis added.

form of the process from the material of the process, and looked about for a completely formal system within which to model self-reproduction. Stan Ulam—one of von Neumann's colleagues at Los Alamos[3]—suggested an appropriate formalism, which has come to be known as a *cellular automaton* (CA).

In brief, a CA consists of a regular lattice of *finite automata*, which are the simplest formal models of machines. A finite automaton can be in only one of a finite number of states at any given time, and its transitions between states from one time-step to the next are governed by a *state-transition table*: given a certain input and a certain internal state, the state-transition table specifies the state to be adopted by the finite automaton at the next time-step. In a CA, the necessary input is derived from the states of the automata at neighbouring lattice-points. Thus the state of an automaton at time $t + 1$ is a function of the states of the automaton itself and its immediate neighbours at time t. All the automata in the lattice obey the same transition table and every automaton changes state at the same instant, time-step after time-step. CAs are good examples of the kind of computational paradigm sought after by Artificial Life: bottom-up, parallel, local determination of behaviour.

Von Neumann was able to embed the equivalent of his kinematic model as an initial pattern of state assignments within a large CA lattice using twenty-nine states per cell. Although von Neumann's work on self-reproducing automata was left incomplete at the time of his death, Arthur Burks organized what had been done, filled in the remaining details, and published it.[4] Fig. 1.3 shows a schematic diagram of von Neumann's self-reproducing machine.

Von Neumann's CA model was a constructive proof that an essential characteristic behaviour of living things—self-reproduction—*was* achievable by machines. Furthermore, he determined that any such method must make use of the information contained in the description of the machine in two fundamentally different ways:

1. *Interpreted*, as instructions to be executed in the construction of the offspring.
2. *Uninterpreted*, as passive data to be duplicated to form the description given to the offspring.

Of course, when Watson and Crick unveiled the structure of DNA, they discovered that the information contained therein was used in precisely these two ways in the processes of transcription/translation and replication.

In describing his model, von Neumann pointed out that:

[3] Ulam (1962) also investigated dynamic models of pattern production and competition.

[4] Together with a transcription of von Neumann's 1949 lectures at the University of Illinois entitled 'Theory and Organization of Complicated Automata', in which he gives his views on various problems related to the study of complex systems in general (von Neumann 1966).

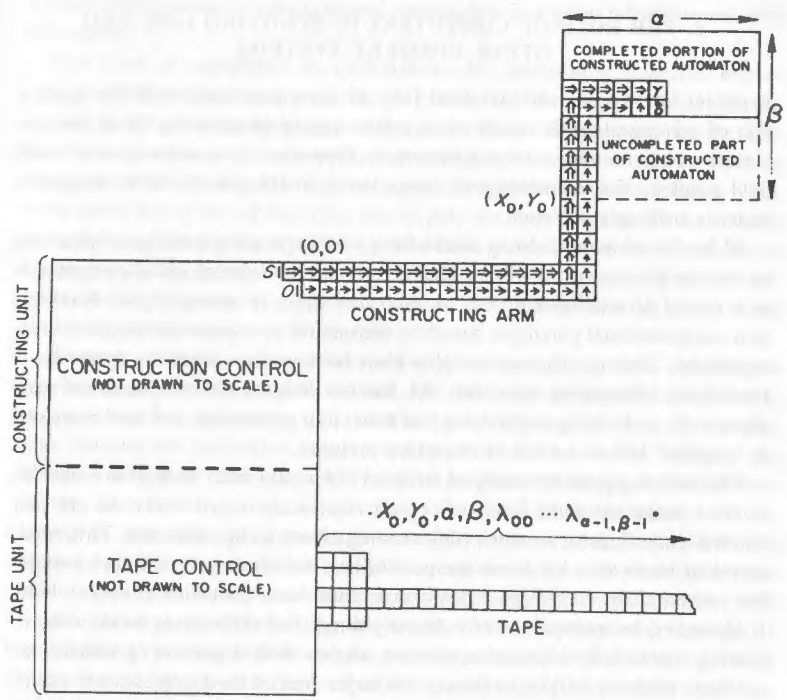


FIG. 1.3. Schematic diagram of von Neumann's CA self-reproducing configuration. From Burks (1970). Reprinted by permission

By axiomatizing automata in this manner, one has thrown half of the problem out the window, and it may be the more important half. One has resigned oneself not to explain how these parts are made up of real things, specifically, how these parts are made up of actual elementary particles, or even of higher chemical molecules.[5]

Whether or not the more important half of the question has been disposed of depends on the questions we are asking. If we are concerned with explaining how the life that we know emerges from the known laws of physics and organic chemistry, then indeed the interesting part has been tossed out. But if we are concerned with the more general problem of explaining how lifelike behaviours emerge out of low-level interactions within a population of logical primitives, we have retained the more interesting portion of the question.

[5] From Burks (1970).

3. THE ROLE OF COMPUTERS IN STUDYING LIFE AND OTHER COMPLEX SYSTEMS

Artificial Intelligence and Artificial Life are each concerned with the application of computers to the study of complex, natural phenomena. Both are concerned with generating complex behaviour. However, the manner in which each field employs the technology of computation in the pursuit of its respective goals is strikingly different.

AI has based its underlying methodology for generating intelligent behaviour on the computational paradigm. That is, AI uses the technology of computation as a model of intelligence. AL, on the other hand, is attempting to develop a new computational paradigm based on the natural processes that support living organisms. That is, AL uses insights from biology to explore the dynamics of interacting information structures. AL has not adopted the computational paradigm as its underlying methodology of behaviour generation, nor does it attempt to 'explain' life as a kind of computer program.

One way to pursue the study of artificial life would be to attempt to create life *in vitro*, using the same kinds of organic chemicals out of which we are constituted. Indeed, there are numerous exciting efforts in this direction. This would certainly teach us a lot about the possibilities for alternative life-forms *within* the carbon-chain chemistry domain that could have (but didn't) evolve here.

However, biomolecules are extremely small and difficult to work with, requiring rooms full of special equipment, replete with dozens of 'postdocs' and graduate students willing to devote the larger part of their professional careers to the perfection of electrophoretic gel techniques. Besides, although the creation of life *in vitro* would certainly be a scientific feat worthy of note—and probably even a Nobel prize—it would not, in the long run, tell us much more about the space of *possible* life than we already know.

Computers provide an alternative medium within which to attempt to synthesize life. Modern computer technology has resulted in machinery with tremendous potential for the creation of life *in silico*.

Computers should be thought of as an important laboratory tool for the study of life, substituting for the array of incubators, culture dishes, microscopes, electrophoretic gels, pipettes, centrifuges, and other assorted wet-lab paraphernalia, one simple-to-master piece of experimental equipment devoted exclusively to the incubation of information structures.

The advantage of working with information structures is that information has no intrinsic size. The computer is *the* tool for the manipulation of information, whether that manipulation is a consequence of our actions or a consequence of the actions of the information structures themselves. Computers themselves will not be alive, rather they will support informational universes within which

dynamic populations of informational 'molecules' engage in informational 'biochemistry'.

This view of computers as workstations for performing scientific experiments within artificial universes is fairly new, but it is rapidly becoming accepted as a legitimate, even necessary, way of pursuing science. In the days before computers, scientists worked primarily with systems whose defining equations could be solved analytically, and ignored those whose defining equations could *not* be so solved. This was largely the case because, in the absence of analytic solutions, the equations would have to be integrated over and over again, essentially simulating the time behaviour of the system. Without computers to handle the mundane details of these calculations, such an undertaking was unthinkable except in the simplest cases.

However, with the advent of computers, the necessary mundane calculations can be relegated to these idiot-savants, and the realm of numerical simulation is opened up for exploration. 'Exploration' is an appropriate term for the process, because the numerical simulation of systems allows one to 'explore' the system's behaviour under a wide range of parameter settings and initial conditions. The heuristic value of this kind of experimentation cannot be overestimated. One often gains tremendous insight for the essential dynamics of a system by observing its behaviour under a wide range of initial conditions. Most importantly, however, computers are beginning to provide scientists with a new paradigm for modelling the world. When dealing with essentially unsolvable governing equations, the primary reason for producing a formal mathematical model—the hope of reaching an analytic solution by symbolic manipulation—is lost. Systems of ordinary and partial differential equations are not very well suited for implementation as computer algorithms. One might expect that other modelling technologies would be more appropriate when the goal is the *synthesis*, rather than the *analysis*, of behaviour.[6]

This expectation is easily borne out. With the precipitous drop in the cost of raw computing power, computers are now available that are capable of simulating physical systems from first principles. This means that it has become possible, for example, to model turbulent flow in a fluid by simulating the motions of its constituent particles—not just approximating *changes* in concentrations of particles at particular points, but actually computing their motions exactly (Frisch *et al.* 1986; Wolfram 1986; Toffoli and Margolus 1987).

What does all this have to do with the study of life? The most surprising lesson we have learned from simulating complex physical systems on computers is that *complex behaviour need not have complex roots*. Indeed, tremendously interesting and beguilingly complex behaviour can emerge from collections of

[6] See Toffoli (1984) for a good exposition.

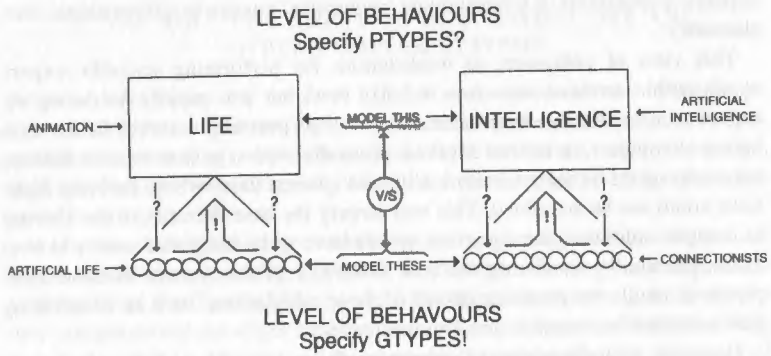


FIG. 1.4. The bottom-up versus the top-down approach to modelling complex systems. From Langton (1989*a*)

extremely simple components. This leads directly to the exciting possibility that much of the complex behaviour exhibited by nature—especially the complex behaviour that we call life—*also* has simple generators. Since it is very hard to work backwards from a complex behaviour to its generator, but very simple to create generators and synthesize complex behaviour, a promising approach to the study of complex natural systems is to undertake the general study of the kinds of behaviour that can emerge from distributed systems consisting of simple components (Fig. 1.4).

4. NON-LINEARITY AND LOCAL DETERMINATION OF BEHAVIOUR

4.1. Linear vs. Non-linear Systems

As mentioned briefly above, the distinction between linear and non-linear systems is fundamental, and provides excellent insight into why the principles underlying the dynamics of life should be so hard to find. The simplest way to state the distinction is to say that *linear systems* are those for which the behaviour of the whole is just the sum of the behaviour of its parts, while for *non-linear systems*, the behaviour of the whole is *more* than the sum of its parts.

Linear systems are those which obey the *principle of superposition*. We can break up complicated linear systems into simpler constituent parts, and analyse these parts *independently*. Once we have reached an understanding of the parts in isolation, we can achieve a full understanding of the whole system by *composing* our understandings of the isolated parts. This is the key feature of linear systems: by studying the parts in isolation, we can learn everything we need to know about the complete system.

This is not possible for non-linear systems, which do *not* obey the principle of superposition. Even if we could break such systems up into simpler constituent parts, and even if we could reach a complete understanding of the parts in isolation, we would not be able to compose our understandings of the individual parts into an understanding of the whole system. The key feature of non-linear systems is that their primary behaviours of interest are properties of the *interactions between parts*, rather than being properties of the parts themselves, and these interaction-based properties necessarily disappear when the parts are studied independently.

Thus analysis is most fruitfully applied to linear systems. Analysis has *not* proved anywhere near as effective when applied to non-linear systems: the non-linear system must be treated as a whole.

A different approach to the study of non-linear systems involves the inverse of analysis: *synthesis*. Rather than start with the behaviour of interest and attempting to analyse it into its constituent parts, we start with constituent parts and put them together in the attempt to *synthesize* the behaviour of interest.

Life is a property of *form*, not *matter*, a result of the organization of matter rather than something that inheres in the matter itself. Neither nucleotides nor amino acids nor any other carbon-chain molecule is alive—yet put them together in the right way, and the dynamic behaviour that emerges out of their interactions is what we call life. It is effects, not things, upon which life is based—life is a kind of behaviour, not a kind of stuff—and as such, it is constituted of simpler behaviours, not simpler stuff. *Behaviours themselves* can constitute the fundamental parts of non-linear systems—*virtual parts*, which depend on non-linear interactions between physical parts for their very existence. Isolate the physical parts and the virtual parts cease to exist. It is the *virtual parts* of living systems that Artificial Life is after, and synthesis is its primary methodological tool.

4.2. The Parsimony of Local Determination of Behaviour

It is easier to generate complex behaviour from the application of simple, *local* rules than it is to generate complex behaviour from the application of complex, *global* rules. This is because complex global behaviour is usually due to non-linear interactions occurring at the local level. With bottom-up specifications, the system computes the local, non-linear interactions explicitly and the global behaviour, which was implicit in the local rules, emerges spontaneously without being treated explicitly.

With top-down specifications, however, local behaviour must be implicit in global rules! This is really putting the cart before the horse! The global rules must 'predict' the effects on global structure of many local, non-linear interactions—something which we have seen is intractable, even impossible, in the general case. Thus top-down systems must take computational short cuts and explicitly deal with special cases, which results in inflexible, brittle, and unnatural behaviour.

Furthermore, in a system of any complexity, the number of possible global states is astronomically enormous, and grows exponentially with the size of the system. Systems that attempt to supply *global* rules for *global* behaviour simply *cannot* provide a different rule for every global state. Thus the global states must be classified in some manner, and categorized using a coarse-grained scheme according to which the global states within a category are indistinguishable. The rules of the system can only be applied at the level of resolution of these categories. There are many possible ways to implement a classification scheme, most of which will yield different partitionings of the global-state space. Any rule-based system must necessarily *assume* that finer-grained differences do not matter, or must include a finite set of tests for 'special cases', and then must assume that no *other* special cases are relevant.

For most complex systems, however, fine differences in the global state can result in enormous differences in global behaviour, and there may be no way in principle to partition the space of global states in such a way that specific fine differences have the appropriate global impact.

On the other hand, systems that supply *local* rules for *local* behaviours *can* provide a different rule for each and every possible local state. Furthermore, the size of the local-state space can be completely independent of the size of the system. In local rule-governed systems, each local state, and consequently the global state, can be determined exactly and precisely. Fine differences in the global state will result in very specific differences in the local state and, consequently, will affect the invocation of local rules. As fine differences affect local behaviour, the difference will be felt in an expanding patch of local states, and in this manner—propagating from local neighbourhood to local neighbourhood—fine differences in global state can result in large differences in global behaviour. The only 'special cases' explicitly dealt with in locally determined systems are exactly the set of all possible local states, and the rules for these are just exactly the set of all local rules governing the system.

5. BIOLOGICAL AUTOMATA

Organisms have been compared to extremely complicated and finely tuned biochemical machines. Since we know that it is possible to abstract the logical form of a machine from its physical hardware, it is natural to ask whether it is possible to abstract the logical form of an organism from its biochemical wetware. The field of Artificial Life is devoted to the investigation of this question.

In the following sections we shall look at the manner in which behaviour is generated in a bottom-up fashion in living systems. We then generalize the mechanisms by which this behaviour generation is accomplished, so that we may apply them to the task of generating behaviour in artificial systems.

We shall find that the essential machinery of living organisms is quite a bit different from the machinery of our own invention, and we would be quite mistaken to attempt to force our preconceived notions of abstract machines on to the machinery of life. The difference, once again, lies in the exceedingly parallel and distributed nature of the operation of the machinery of life, as contrasted with the singularly serial and centralized control structures associated with the machines of our invention.

5.1. Genotypes and Phenotypes

The most salient characteristic of living systems, from the behaviour generation point of view, is the *genotype/phenotype* distinction. The distinction is essentially one between a specification of machinery—the genotype—and the behaviour of that machinery—the phenotype.

The *genotype* is the complete set of genetic instructions encoded in the linear sequence of nucleotide bases that makes up an organism's DNA. The *phenotype* is the physical organism itself—the structures that emerge in space and time as the result of the interpretation of the genotype in the context of a particular environment. The process by which the phenotype develops through time under the direction of the genotype is called *morphogenesis*. The individual genetic instructions are called *genes* and consist of short stretches of DNA. These instructions are 'executed'—or *expressed*—when their DNA sequence is used as a template for transcription. In the case of protein synthesis, transcription results in a duplicate nucleotide strand known as a *messenger RNA*—or *mRNA*—constructed by the process of base-pairing. This mRNA strand may then be modified in various ways before it makes its way out to the cytoplasm where, at bodies known as *ribosomes*, it serves as a template for the construction of a linear chain of *amino acids*. The resulting *polypeptide* chain will fold up on itself in a complex manner, forming a tightly packed molecule known as a *protein*. The finished protein detaches from the ribosome and may go on to serve as a passive structural element in the cell, or may have a more active role as an *enzyme*. Enzymes are *the* functional molecular 'operators' in the logic of life.

One may consider the genotype as a largely unordered 'bag' of instructions, each one of which is essentially the specification for a 'machine' of some sort—

passive or active. When instantiated, each such 'machine' will enter into the ongoing logical 'fray' in the cytoplasm, consisting largely of local interactions between other such machines. Each such instruction will be 'executed' when its own triggering conditions are met and will have specific, local effects on structures in the cell. Furthermore, each such instruction will operate within the context of all the other instructions that have been—or are being—executed.

The phenotype, then, consists of the structures and dynamics that emerge through time in the course of the execution of the parallel, distributed 'computation' controlled by this genetic 'bag' of instructions. Since genes' interactions with one another are highly non-linear, the phenotype is a non-linear function of the genotype.

5.2. Generalized Genotypes and Phenotypes

In the context of Artificial Life, we need to generalize the notions of *genotype* and *phenotype*, so that we may apply them in non-biological situations. We shall use the term *generalized genotype*—or GTYPE—to refer to any largely unordered set of low-level rules, and we shall use the term *generalized phenotype*—or PTYPE—to refer to the behaviours and/or structures that emerge out of the interactions among these low-level rules when they are activated within the context of a specific environment. The GTYPE, essentially, is the specification for a set of machines, while the PTYPE is the behaviour that results as the machines are run and interact with one another.

This is the bottom-up approach to the generation of behaviour. A set of entities is defined, and each entity is endowed with a specification for a simple behavioural repertoire—a GTYPE—that contains instructions which detail its reactions to a wide range of *local* encounters with other such entities or with specific features of the environment. Nowhere is the behaviour of the set of entities as a whole specified. The global behaviour of the aggregate—the PTYPE—emerges out of the collective interactions among individual entities.

It should be noted that the PTYPE is a multi-level phenomenon. First, there is the PTYPE associated with each particular instruction—the effect which that instruction has on an entity's behaviour when it is expressed. Second, there is the PTYPE associated with each individual entity—its individual behaviour within the aggregate. Third, there is the PTYPE associated with the behaviour of the aggregate as a whole.

This is true for natural systems as well. We can talk about the phenotypic trait associated with a particular gene, we can identify the phenotype of an individual cell, and we can identify the phenotype of an entire multi-cellular organism—its body, in effect. PTYPES *should* be complex and multi-level. If we want

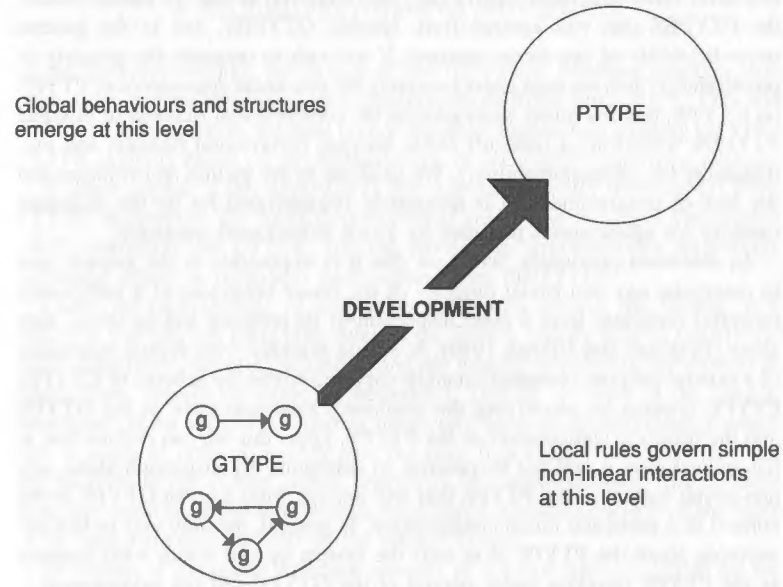


FIG. 1.5. The relationship between GTYPE and PTYPE. From Langton (1989*a*)

to simulate life, we should expect to see hierarchical structures emerge in our simulations. In general, phenotypic traits at the level of the whole organism will be the result of many non-linear interactions between genes, and there will be no single gene to which one can assign responsibility for the vast majority of phenotypic traits.

In summary, GTYPES are low-level rules for behavors—i.e. abstract specifications for 'machines'—which will engage in local interactions within a large aggregate of other such behavors. PTYPES are the behaviours—the structures in time and space—that *develop* out of these non-linear, local interactions (Fig. 1.5).

5.3. Unpredictability of PTYPE from GTYPE

Non-linear interactions between the objects specified by the GTYPE provide the basis for an extremely rich variety of possible PTYPES. PTYPES draw on the full combinatorial potential implicit in the set of possible interactions between

low-level rules. The other side of the coin, however, is that we cannot predict the PTYPES that will emerge from specific GTYPES, due to the general unpredictability of non-linear systems. If we wish to maintain the property of predictability, then we must restrict severely the non-linear dependence of PTYPE on GTYPE, but this forces us to give up the combinatorial richness of possible PTYPES. Therefore, a trade-off exists between behavioural richness and predictability (or 'programmability'). We shall see in the section on evolution that the lack of programmability is adequately compensated for by the increased capacity for adaptiveness provided by a rich behavioural repertoire.

As discussed previously, we know that it is impossible in the general case to determine *any* non-trivial property of the future behaviour of a sufficiently powerful computer from a mere inspection of its program and its initial state alone (Hopcroft and Ullman 1979). A Turing machine—the formal equivalent of a general-purpose computer—can be captured within the scheme of GTYPE/PTYPE systems by identifying the machine's transition table as the GTYPE and the resulting computation as the PTYPE. From this we can deduce that in the general case it will not be possible to determine, by inspection alone, any non-trivial feature of the PTYPE that will emerge from a given GTYPE in the context of a particular initial configuration. In general, the only way to find out anything about the PTYPE is to start the system up and watch what happens as the PTYPE develops under control of the GTYPE and the environment.

Similarly, it is not possible in the general case to determine what specific alterations must be made to a GTYPE to effect a desired change in the PTYPE. The problem is that any specific PTYPE trait is, in general, an effect of many, many non-linear interactions between the behavioural primitives of the system (an 'epistatic trait' in biological terms). Consequently, given an arbitrary proposed change to the PTYPE, it may be impossible to determine by any formal procedure exactly what changes would have to be made to the GTYPE to effect that—and *only* that—change in the PTYPE. It is not a practically computable problem. There is no way to calculate the answer—short of exhaustive search—*even though there may be an answer!*[7]

The only way to proceed in the face of such an unpredictability result is by a process of trial and error. However, some processes of trial and error are more efficient than others. In natural systems, trial and error are interlinked in such a way that error guides the choice of trials under the process of evolution by natural selection. It is quite likely that this is the *only* efficient, *general* procedure that could find GTYPES with specific PTYPE traits when non-linear functions are involved.

[7] An example in biology would be: What changes would have to be made to the genome in order to produce six fingers on each hand rather than five?

6. RECURSIVELY GENERATED OBJECTS

In the previous section, we described the distinction between genotype and phenotype, and we introduced their generalizations in the form of GTYPES and PTYPES. In this section, we shall review a general approach to building GTYPE/PTYPE systems based on the methodology of *recursively generated objects*.

A major appeal of this approach is that it arises naturally from the GTYPE/PTYPE distinction: the local developmental rules—the recursive description itself—constitute the *GTYPE*, and the developing structure—the recursively generated object or behaviour itself—constitutes the *PTYPE*.

Under the methodology of recursively generated objects, the 'object' is a structure that has sub-parts. The rules of the system specify how to modify the most elementary, 'atomic' sub-parts, and are usually sensitive to the *context* in which these atomic sub-parts are embedded. That is, the state of the 'neighbourhood' of an atomic sub-part is taken into account in determining which rule to apply in order to modify that sub-part. It is usually the case that there are no rules in the system whose context is the entire structure; that is, there is no use made of *global* information. Each piece is modified solely on the basis of its own state and the state of the pieces 'nearby'.

Of course, if the initial structure consists of a single part—as might be the case with the initial seed—then the context for applying a rule is necessarily global. The usual situation is that a structure consists of *many* parts, only a local subset of which determine the rule that will be used to modify any one sub-part of the structure.

A recursively generated object, then, is a kind of PTYPE, and the recursive description that generates it is a kind of GTYPE. The PTYPE will emerge under the action of the GTYPE, developing through time via a process akin to morphogenesis.

We shall illustrate the notion of recursively generated objects with examples taken from the literature on L-systems, cellular automata, and computer animation.

6.1. Example 1: Lindenmayer Systems

Lindenmayer systems (L-systems) consist of sets of rules for rewriting strings of symbols, and bear strong relationships to the formal grammars treated by Chomsky. We shall give several examples of L-systems illustrating the methodology of recursively generated objects.[8]

In the following '$X \rightarrow Y$' means that one replaces every occurrence of

[8] For a more detailed review, see Prusinkiewicz (1991).

symbol *X* in the structure with string *Y*. Since the symbol *X* may appear on the right as well as the left sides of some rules, the set of rules can be applied 'recursively' to the newly rewritten structures. The process can be continued *ad infinitum* although some sets of rules will result in a 'final' configuration when no more changes occur.

Simple Linear Growth. Here is an example of the simplest kind of L-system. The rules are *context free*, meaning that the context in which a particular part is situated is *not* considered when altering it. There must be only one rule per part if the system is to be deterministic.

The rules (the 'recursive description' or GTYPE):

1) A → CB
2) B → A
3) C → DA
4) D → C

When applied to the initial seed structure 'A', the following structural history develops (each successive line is a successive time-step):

time	structure	rules applied (L to R)
0	A	(initial 'seed')
1	C B	(rule 1 replaces A with CB)
2	D A A	(rule 3 replaces C with DA and rule 2 replaces B with A)
3	C C B C B	(rule 4 replaces D with C and rule 1 replaces the two As with CBs)
4	. . . (etc.) . . .	

And so forth.

The 'PTYPE' that emerges from this kind of recursive application of a simple, local rewriting rule can get extremely complex. These kinds of grammars (whose rules replace single symbols) have been shown to be equivalent to the operation of finite-state machines. With appropriate restrictions, they are also equivalent to the 'regular languages' defined by Chomsky.

Branching Growth. L-systems incorporate meta-symbols to represent branching points, allowing a new line of symbols to branch off from the main 'stem' (see Fig. 1.6).

The following grammar produces branching structures. The '( )' and '[ ]' notations indicate left and right branches, respectively, and the strings within them indicate the structure of the branches themselves.

n=5, δ=18°

ω : plant
p_1 : plant → internode + [plant + flower] – – / /
[– – leaf] internode [+ + leaf] –
[plant flower] + + plant flower
p_2 : internode → F seg [/ / & & leaf] [/ / ∧ ∧ leaf] F seg
p_3 : seg → seg F seg
p_4 : leaf → [' { +f–ff–f+ | +f–ff–f }]
p_5 : flower → [& & & pedicel ' / wedge / / / / wedge / / / /
wedge / / / / wedge / / / / wedge]
p_6 : pedicel → FF
p_7 : wedge → [' ∧ F] [{ & & & & –f+f | –f+f }]

FIG. 1.6. An L-system plant grown from rules incorporating graphical rendering information. From Prusinkiewicz (1991)

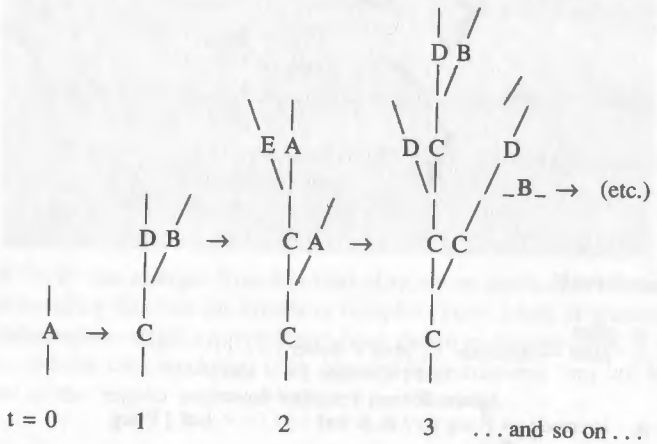
The rules—or GTYPE:

1) A → C[B]D
2) B → A
3) C → C
4) D → C(E)A
5) E → D

When applied to the starting structure 'A', the following sequence develops (using linear notation):

time	structure	rules applied (L to R)
0	A	initial 'seed'
1	C[B]D	rule 1
2	C[A]C(E)A	rules 3, 2, 4
3	C[C[B]D]C(D)C[B]D	rules 3, 1, 3, 5, 1
4	C[C[A]C(E)A]C(C(E)A)C[A]C(E)A	rules 3, 3, 2, 4, 3, 4, 3, 2, 4

In two dimensions, the structure develops as follows:



Note that at each step, *every symbol is replaced*, even if just by another copy of itself. This figure shows the result of growing a structure using the rules shown, which contain graphical rendering information in addition to the usual 'structural' information.

Signal Propagation. In order to propagate signals along a structure, one must have something more than just a single symbol on the left-hand side of a rule. When there is more than one symbol on the left-hand side of a rule, the rules are *context sensitive*—i.e. the 'context' within which a symbol occurs (the symbols next to it) are important in determining what the replacement string will be. The next example illustrates why this is critical for signal propagation.

In the following example, the symbol in '{ }' is the symbol (or string of symbols) to be replaced, the rest of the left-hand side is the context, and the symbols '[' and ']' indicate the left and right ends of the string, respectively. Suppose the rule set contains the following rules:

1) [{C} → C a 'C' at the left-end of the string remains a 'C'
2) C{C} → C a 'C' with a 'C' to its left remains a 'C'
3) *{C} → * a 'C' with an '*' to its left becomes an '*'
4) {*}C → C an '*' with a 'C' to its right becomes a 'C'
5) {*}] → * an '*' at the right end of the string remains an '*'

Under these rules, the initial structure '*CCCCCCC' will result in the '*' being propagated to the right, as follows:

time	structure
0	*CCCCCCC
1	C*CCCCCC
2	CC*CCCCC
3	CCC*CCCC
4	CCCC*CCC
5	CCCCC*CC
6	CCCCCC*C
7	CCCCCCC*

This would not be possible without taking the 'context' of a symbol into account. In general, these kinds of grammars are equivalent to Chomsky's 'context-sensitive' or 'Turing' languages, depending on whether or not there are any restrictions on the kinds of strings on the left- and right-hand sides.

The capacity for signal propagation is extremely important, for it allows arbitrary computational processes to be embedded within the structure, which may directly affect the structure's development. The next example demonstrates how embedded computation can affect development.

6.2. Example 2: Cellular Automata

Cellular automata (CA) provide another example of the recursive application of a simple set of rules to a structure. In CA, the structure that is being updated

is the entire universe: a lattice of finite automata. The local rule set—the GTYPE—in this case is the transition function obeyed homogeneously by every automaton in the lattice. The local context taken into account in updating the state of each automaton is the state of the automata in its immediate neighbourhood. The transition function for the automata constitutes a *local* physics for a simple, discrete space/time universe. The universe is updated by applying the local physics to each local 'cell' of its structure over and over again. Thus, although the physical structure itself does not develop over time, its *state* does.

Within such universes, one can embed all manner of processes, relying on the context sensitivity of the rules to local neighbourhood conditions to propagate information around within the universe 'meaningfully'. In particular, one can embed general-purpose computers. Since these computers are simply particular configurations of states within the lattice of automata, *they can compute over the very set of symbols out of which they are constructed.* Thus, structures in this universe can compute and construct other structures, which also may compute and construct.

For example, here is the simplest known structure that can reproduce itself:

```
  2 2 2 2 2 2 2 2
2 1 7 0 1 4 0 1 4 2
2 0 2 2 2 2 2 2 0 2
2 7 2         2 1 2
2 1 2         2 1 2
2 0 2         2 1 2
2 7 2         2 1 2
2 1 2 2 2 2 2 2 1 2 2 2 2 2
2 0 7 1 0 7 1 0 7 1 1 1 1 1 2
  2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Each number is the state of one automaton in the lattice. Blank space is presumed to be in state '0'. The '2'-states form a sheath around the '1'-state data path. The '7 0' and '4 0' state pairs constitute signals embedded within the data path. They will propagate counter-clockwise around the loop, cloning off copies which propagate down the extended tail as they pass the T-junction between loop and tail. When the signals reach the end of the tail, they have the following effects: each '7 0' signal extends the tail by one unit, and the two '4 0' signals construct a left-hand corner at the end of the tail. Thus for each full cycle of the instructions around the loop, another side and corner of an 'offspring-loop' will be constructed. When the tail finally runs into itself after four cycles, the collision of signals results in the disconnection of the two loops as well as the construction of a tail on each of the loops.

After 151 time-steps, this system will evolve to the following configuration:


```
                2
              2 1 2
              2 7 2
              2 0 2
              2 1 2
    2 2 2 2 2 2 7 2     2 2 2 2 2 2 2 2
  2 1 1 1 7 0 1 7 0 2   2 1 7 0 1 4 0 1 4 2
  2 1 2 2 2 2 2 2 1 2   2 0 2 2 2 2 2 2 0 2
  2 1 2       2 7 2     2 7 2       2 1 2
  2 1 2       2 0 2     2 1 2       2 1 2
  2 4 2       2 1 2     2 0 2       2 1 2
  2 1 2       2 7 2     2 7 2       2 1 2
  2 0 2 2 2 2 2 2 0 2   2 1 2 2 2 2 2 2 1 2 2 2 2 2
  2 4 1 0 7 1 0 7 1 2   2 0 7 1 0 7 1 0 7 1 1 1 1 1 2
    2 2 2 2 2 2 2 2       2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Thus, the initial configuration has succeeded in reproducing itself.

Each of these loops will go on to reproduce itself in a similar manner, giving rise to an expanding *colony* of loops, growing out into the array.

These embedded self-reproducing loops are the result of the recursive application of a rule to a seed structure. In this case, the primary rule that is being recursively applied constitutes the 'physics' of the universe. The initial state of the loop itself constitutes a little 'computer' under the recursively applied physics of the universe: a computer whose program causes it to construct a copy of itself. The 'program' within the loop computer is also applied recursively to the growing structure. Thus, this system really involves a double level of recursively applied rules. The mechanics of applying one recursive rule within a universe whose physics is governed by another recursive rule had to be worked out by trial and error. This system makes use of the signal propagation capacity to embed a structure that itself *computes* the resulting structure, rather than having the 'physics' directly responsible for developing the final structure from a passive seed.

This captures the flavour of what goes on in natural biological development: the genotype codes for the constituents of a dynamic process in the cell, and it is this dynamic process that is primarily responsible for mediating—or 'computing'—the expression of the genotype in the course of development.

6.3. Example 3: Flocking 'Boids'

The previous examples were largely concerned with the growth and development of *structural* PTYPES. Here, we give an example of the development of a *behavioural* PTYPE.

Craig Reynolds (1987) has implemented a simulation of flocking behaviour. In this model—which is meant to be a general platform for studying the qualitatively similar phenomena of flocking, herding, and schooling—one has a large collection of autonomous but interacting objects (which Reynolds refers to as 'Boids'), inhabiting a common simulated environment.

The modeller can specify the manner in which the individual Boids will respond to *local* events or conditions. The global behaviour of the aggregate of Boids is strictly an emergent phenomenon, none of the rules for the individual Boids depends on global information, and the only updating of the global state is done on the basis of individual Boids responding to local conditions.

Each Boid in the aggregate shares the same behavioural 'tendencies':

- to maintain a minimum distance from other objects in the environment, including other Boids,
- to match velocities with Boids in its neighbourhood, and
- to move towards the perceived centre of mass of the Boids in its neighbourhood.

These are the only rules governing the behaviour of the aggregate.

These rules, then, constitute the generalized genotype (GTYPE) of the Boids system. They say nothing about structure, or growth and development, but they determine the behaviour of a set of interacting objects, out of which very natural motion emerges.

With the right settings for the parameters of the system, a collection of Boids released at random positions within a volume will collect into a dynamic flock, which flies around environmental obstacles in a very fluid and natural manner, occasionally breaking up into sub-flocks as the flock flows around both sides of an obstacle. Once broken up into sub-flocks, the sub-flocks reorganize around their own, now distinct and isolated centres of mass, only to re-merge into a single flock again when both sub-flocks emerge at the far side of the obstacle and each sub-flock feels anew the 'mass' of the other sub-flock (Fig. 1.7).

The flocking behaviour itself constitutes the generalized phenotype (PTYPE) of the Boids system. It bears the same relation to the GTYPE as an organism's morphological phenotype bears to its molecular genotype. The same distinction between the *specification* of machinery and the *behaviour* of machinery is evident.

6.4. Discussion of Examples

In all the above examples, the recursive rules apply to *local structures* only, and the PTYPE—structural or behavioural—that results at the global level emerges out of all local activity taken collectively. Nowhere in the system are there rules

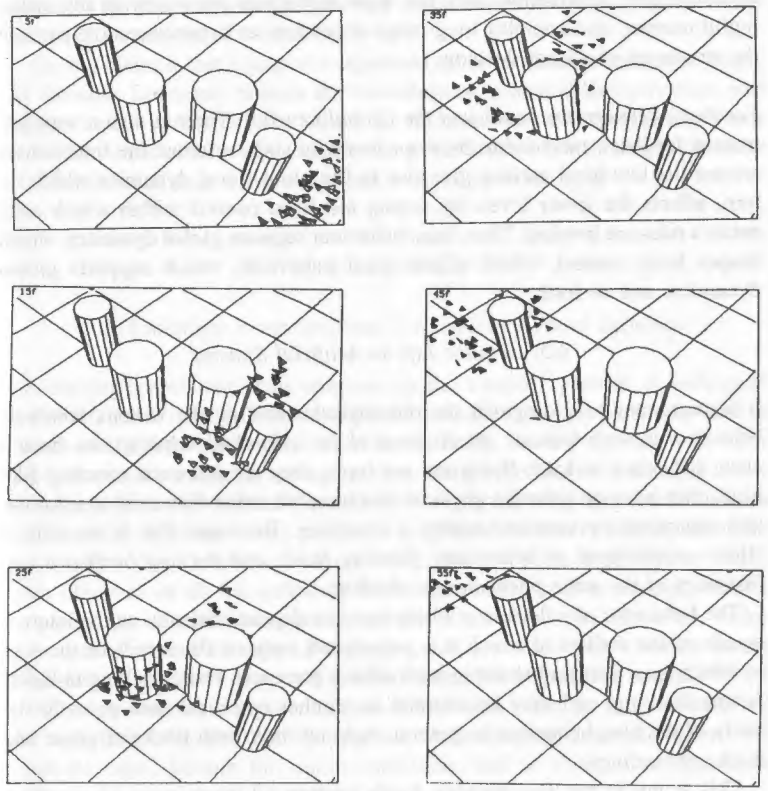


FIG. 1.7. A flock of 'Boids' negotiating a field of columns. Sequence generated by Craig Reynolds. From Langton (1989*a*)

for the behaviour of the system at the global level. This is a much more powerful and simple approach to the generation of complex behaviour than that typically taken in AI, for instance, where 'expert systems' attempt to provide global rules for global behaviour. Recursive, 'bottom up' specifications yield much more natural, fluid, and flexible behaviour at the global level than typical 'top down' specifications, and they do so *much* more parsimoniously.

Importance of Context Sensitivity. It is worth while to note that *context-sensitive* rules in GTYPE/PTYPE systems provide the possibility for non-linear interactions among the parts. Without context sensitivity, the systems would be linearly

decomposable, information could not 'flow' throughout the system in any meaningful manner, and complex long-range dependencies between remote parts of the structures could not develop.

Feedback between the Local and the Global Levels. There is also a very important feedback mechanism *between* levels in such systems: the interactions among the low-level entities give rise to the global-level dynamics which, in turn, affects the lower levels by *setting the local context* within which each entity's rules are invoked. Thus, local behaviour supports global dynamics, which shapes local context, which affects local behaviour, which supports global dynamics, and so forth.

6.5. Genuine Life in Artificial Systems

It is important to distinguish the ontological status of the various levels of behaviour in such systems. At the level of the individual behavors, we have a clear difference in kind: Boids are *not* birds, they are not even remotely like birds, they have no cohesive physical structure, but rather they exist as information structures—processes—within a computer. But—and this is *the* critical 'But'—at the level of behaviours, *flocking Boids and flocking birds are two instances of the same phenomenon: flocking.*

The behaviour of a flock as a whole does not depend critically on the internal details of the entities of which it is constituted, only on the details of the way in which these entities behave in each other's presence. Thus, flocking in Boids is true flocking, and may be counted as another empirical data point in the study of flocking behaviour in general, right up there with flocks of geese and flocks of starlings.

This is *not* to say that flocking Boids capture *all* the nuances upon which flocking behaviour depends, or that the Boids' behavioural repertoire is sufficient to exhibit all the different modes of flocking that have been observed—such as the classic 'V' formation of flocking geese. The crucial point is that we have captured, within an aggregate of artificial entities, a *bona fide* lifelike behaviour, and that the behaviour emerges within the artificial system in the same way that it emerges in the natural system.

The same is true for L-systems and the self-reproducing loops. The constituent parts of the artificial systems are different kinds of things from their natural counterparts, but the emergent behaviours that they support are the same kinds of thing as their natural counterparts: genuine morphogenesis and differentiation for L-systems, and genuine self-reproduction in the case of the loops.

The claim is the following. The 'artificial' in Artificial Life refers to the component parts, not the emergent processes. If the component parts are implemented correctly, the processes they support are *genuine*—every bit as genuine as the natural processes they imitate.

The *big* claim is that a properly organized set of artificial primitives carrying out the same functional roles as the biomolecules in natural living systems will support a process that will be 'alive' in the same way that natural organisms are alive. Artificial Life will therefore be *genuine* life—it will simply be made of different stuff than the life that has evolved here on Earth.

7. EVOLUTION

7.1. Evolution: From Artificial Selection to Natural Selection

Modern organisms owe their structure to the complex process of biological evolution, and it is very difficult to discern which of their properties are due to chance and which to necessity. If biologists could 'rewind the tape' of evolution and start it over, again and again, from different initial conditions, or under different regimes of external perturbations along the way, they would have a full *ensemble* of evolutionary pathways to generalize over. Such an ensemble would allow them to distinguish universal, necessary properties (those which were observed in all the pathways in the ensemble) from accidental, chance properties (those which were unique to individual pathways). However, biologists cannot rewind the tape of evolution, and are stuck with a single, actual evolutionary trace out of a vast, intuited ensemble of possible traces.

Although studying computer models of evolution is not the same as studying the 'real thing', the ability to freely manipulate computer experiments—to 'rewind the tape', perturb the initial conditions, and so forth—can more than make up for their 'lack' of reality.

It has been known for some time that one can evolve computer programs by the process of natural selection among a population of variant programs. Each individual program in a population of programs is evaluated for its performance on some task. The programs that perform best are allowed to 'breed' with one another via *Genetic Algorithms* (Holland 1975; Goldberg 1989). The offspring of these better-performing parent programs replace the worst-performing programs in the population, and the cycle is iterated. Such evolutionary approaches to program improvement have been applied primarily to the tasks of function optimization and machine learning.

However, such evolutionary models have rarely been used to study evolution itself (Wilson 1989). Researchers have primarily concentrated on the *results*, rather than on the *process*, of evolution. In the spirit of von Neumann's research on self-reproduction via the study of self-reproducing *automata*, the following

sections review studies of the process of evolution by studying evolving populations of 'automata'.

7.2. Engineering PTYPES from GTYPES

In the preceding sections, we have mentioned several times the formal impossibility of predicting the behaviour of an arbitrary machine by mere inspection of its specification and initial state. In the general case, we must run a machine in order to determine its behaviour.

The consequence of this unpredictability for GTYPE/PTYPE systems is that we cannot determine the PTYPE that will be produced by an arbitrary GTYPE by inspection alone. We must 'run' the GTYPE in the context of a specific environment, and let the PTYPE develop in order to determine the resulting structure and its behaviour.

This is even further complicated when the environment consists of a population of PTYPES engaged in non-linear interactions, in which case the determination of a PTYPE depends on the behaviour of the specific PTYPES it is interacting with, and on the emergent details of the global dynamics.

Since, for any interesting system, there will exist an enormous number of potential GTYPES, and since there is no formal method for deducing the PTYPES from the GTYPES, how do we go about finding GTYPES that will generate lifelike PTYPES? Or PTYPES that exhibit any other particular sought-after behaviour?

Until now, the process has largely been one of guessing at appropriate GTYPES, and modifying them by trial and error until they generate the appropriate PTYPES. However, this process is limited by our preconceptions of what the appropriate PTYPES would be, and by our restricted notions of how to generate GTYPES. We should like to be able to automate the process so that our preconceptions and limited abilities to conceive of machinery do not overly constrain the search for GTYPES that will yield the appropriate behaviours.

7.3. Natural Selection among Populations of Variants

Nature, of course, has had to face the same problem, and has hit upon an elegant solution: *evolution by the process of natural selection among populations of variants*. The scheme is a very simple one. However, in the face of the formal impossibility of predicting behaviour from machine description alone, it may well be the only efficient, general scheme for searching the space of possible GTYPES.

The mechanism of evolution is as follows. A set of GTYPES is interpreted within a specific environment, forming a population of PTYPES which interact

with one another and with features of the environment in various complex ways. On the basis of the relative performance of their associated PTYPES, *some* GTYPES are duplicated in larger numbers than others, and they are duplicated in such a way that the copies are similar to—but not exactly the same as—the originals. These variant GTYPES develop into variant PTYPES, which enter into the complex interactions within the environment, and the process is continued *ad infinitum* (Fig. 1.8). As expected from the formal limitations on predictability, GTYPES must be 'run' (i.e. turned into PTYPES) in an environment and their behaviours must be evaluated explicitly, their implicit behaviour cannot be determined.

7.4. Genetic Algorithms

In the spirit of von Neumann, John Holland (1975, 1986) has attempted to abstract 'the logical form' of the natural process of biological evolution in what he calls the 'Genetic Algorithm' (GA). In the GA, a GTYPE is represented as a character string that encodes a potential solution to a problem. For instance, the character string might encode the weight matrix of a neural network, or the transition table of a finite-state machine. These character strings are rendered as PTYPES via a problem-specific interpreter, which constructs, for example, the neural net or finite-state machine specified by each GTYPE, evaluates its performance in the problem domain, and provides it with a specific fitness value, or 'strength'.

The GA implements natural selection by making more copies of the character strings representing the better performing PTYPES. The GA generates variant GTYPES by applying *genetic operators* to these character strings. The genetic operators typically consist of *reproduction*, *cross-over*, and *mutation*, with occasional usage of *inversion* and *duplication*.

Recently, John Koza (1991) has developed a version of the GA, which he calls the *Genetic Programming Paradigm* (GPP), that extends the genetic operators to work on GTYPES that are simple expressions in a standard programming language. The GPP differs from the traditional GA in that these program expressions are not represented as simple character strings but rather as the parse trees of the expressions. This makes it easier for the genetic operators to obey the syntax of the programming language when producing variant GTYPES.

Fig. 1.9 shows some examples of GTYPES in the GA and GPP paradigms.

The Genetic Operators. The genetic operators work as follows.

Reproduction is the most basic operator. It is often implemented in the form of *fitness proportionate reproduction*, which means that strings are duplicated in direct proportion to their relative fitness values. Once all strings have been

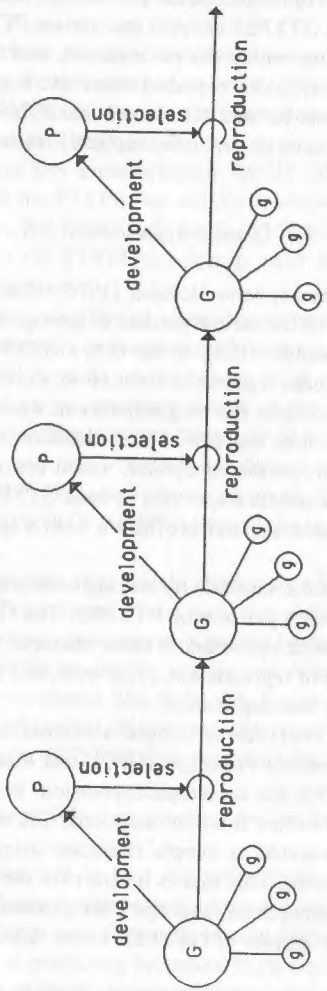


FIG. 1.8. The process of evolution by natural selection. From Langton (1989*a*)

(a) 1100101011101011010100010101011101011

(b) (OR (NOT a) (AND b c))

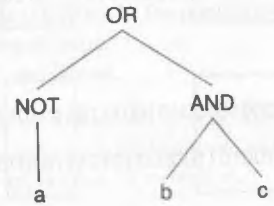


FIG. 1.9. GTYPES in the GA and GPP paradigms. From Langton (1989*a*)

evaluated, the average fitness of the population is computed, and those strings whose fitness is higher than the population average have a higher probability of being duplicated, while those strings whose fitness is lower than the population average have a lower probability of being duplicated. There are many variations on this scheme, but most implementations of the GA or the GPP use some form of fitness proportionate reproduction as the means to implement 'selection'. Another form of this is to simply keep the top 10 per cent or so of the population and throw away the rest, using the survivors as breeding stock for the next generation.

Mutation in the GA is simply the replacement of one or more characters in a character string GTYPE with another character picked at random. In binary strings, this simply amounts to random bit flips. In the GPP, mutation is implemented by picking a sub-tree of the parse tree at random, and replacing it with a randomly generated sub-tree whose root node is of the same syntactic type as the root node of the replaced sub-tree.

Cross-over is an analogue of sexual recombination. In the GA, this is accomplished by picking two 'parent' character strings, lining them up side by side, and interchanging equivalent sub-strings between them, producing two new sub-strings that each contain a mix of their parent's genetic information. Cross-over is an extremely important genetic operator. Whereas mutation is equivalent to random search, cross-over allows the more 'intelligent' search strategy of putting things that have proved useful in new combinations.

In the GPP, cross-over is implemented by picking two 'parent' parse trees, locating syntactically similar sub-trees within each, and swapping them.

Fig. 1.10 illustrates the cross-over operation in the GA and GPP.

Inversion is used rarely in order to rearrange the relative locations of specific pieces of genetic information in the character strings of the GA.

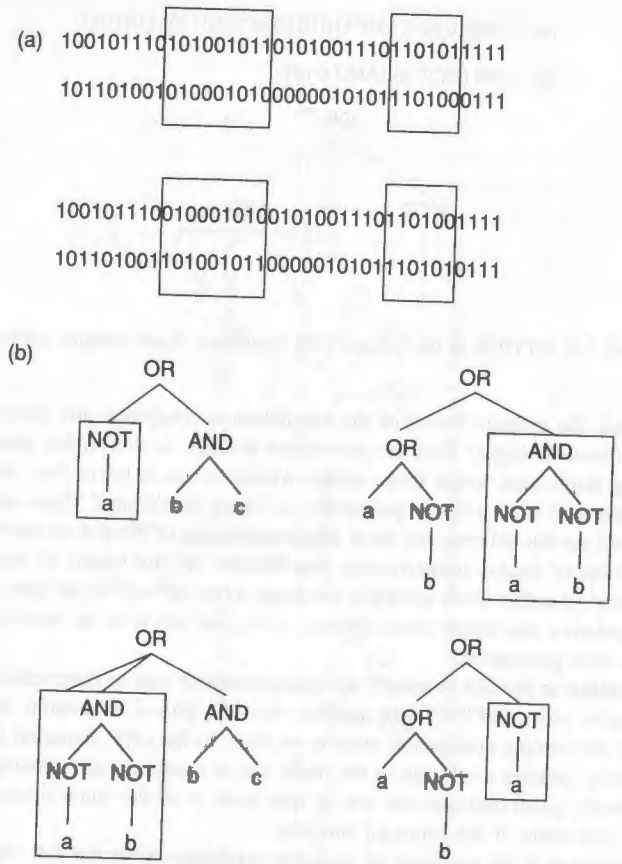


FIG. 1.10. Cross-over operation in the GA and GPP

Duplication is sometimes used in situations where it makes sense for the genome to grow in length, representing, for instance, larger neural nets, or bigger finite-state machine transition tables.

The Operation of the Genetic Algorithm. The basic outline of the genetic algorithm is as follows:

1. Generate a random initial population of GTYPES.
2. Render the GTYPES in the population as PTYPES and evaluate them in the problem domain, providing each GTYPE with a fitness value.

3. Duplicate GTYPES according to their relative fitness using a scheme like fitness-proportionate reproduction.
4. Apply *genetic operators* to the GTYPES in the population, typically picking cross-over partners as a function of their relative fitness.
5. Replace the least-fit GTYPES in the population with the offspring generated in the last several steps.
6. Go back to step 2 and iterate.

Although quite simple in outline, the genetic algorithm has proved remarkably powerful in a wide variety of applications, and provides a useful tool for both the study and the application of evolution.

The Context of Adaptation. GAs have traditionally been employed in the contexts of machine learning and function optimization. In such contexts, one is often looking for an explicit, optimal solution to a particular, well-specified problem. This is reflected in the implementation of the evaluation of PTYPES in traditional GAs: each GTYPE is expressed as a PTYPE independently of the others, tested on the problem, and assigned a value representing its individual fitness using an explicit fitness function. Thus, one is often seeking to evolve an *individual* that *explicitly encodes* an optimal solution to a precisely specified problem. The fitness of a GTYPE in such cases is simply a function of the problem domain, and is independent of the fitnesses of the other GTYPES in the population.

This is quite different from the context in which natural biological evolution has taken place, in which the behaviour of a PTYPE and its associated fitness are highly dependent on which other PTYPES exist in the environment, and on the dynamics of their interactions. Furthermore, in the natural context, it is generally the case that there is no single, explicitly specified problem confronting the population. Rather, there is often quite a large set of problems facing the population at any one time, and these problems are only implicitly determined as a function of the dynamics of the population and the environment themselves, which may change significantly over time. *In such a context, nature has often discovered that the collective behaviour emerging from the interactions among a set of PTYPES will address a subset of the implicitly defined problems.*

Thus, the proper picture for the natural evolutionary context is that of a large cloud of implicit collective solutions addressing a large cloud of implicit collective problems. Both these clouds are implicit in the spatio-temporal dynamics of the population.

The dynamics of such systems are very complex and impossible to predict. One can think of them as the dynamical equivalent of many-body orbital mechanics problems: two-body problems can be treated analytically, whereas three- or more body problems are non-analytic.

The important point here is that non-linearities and emergent collective phenomena are properties that are to be exploited, rather than avoided as has been the traditional engineering viewpoint. Emergent non-linear solutions may be harder to understand or to engineer, but there are far more of them than there are non-emergent, analysable linear solutions. The true power of evolution lies in its ability to exploit emergent collective phenomena; it lies, in fact, in evolution's inability to *avoid* such phenomena.

7.5. From Artificial Selection to Natural Selection

In *The Origin of Species*, Darwin used a very clever device to argue for the agency of natural selection. In the first chapter of *Origin*, Darwin lays the groundwork of the case for *natural* selection by carefully documenting the process of *artificial* selection. Most people of his time were familiar with the manner in which breeders of domestic animals and plants could enhance traits arbitrarily by selective breeding of their stock. Darwin carefully made the case that the wide variety of domestic animals and plants extant at his time were descended from a much smaller variety of wildstock, due to the selective breedings imposed by farmers and herders throughout history.

Now, Darwin continues, simply note that environmental circumstances can fill the role played by the human breeder in artificial selection, and *voilà!* one has natural selection. The rest of the book consists in a very careful documentation of the manner in which different environmental conditions would favour animals bearing different traits, making it more likely that individuals bearing those traits would survive to mate with each other and produce offspring, leading to the gradual enhancement of those traits through time. A beautifully simple yet elegant mechanism to explain the origin and maintenance of the diversity of species on Earth—too simple for many of his time, particularly those of strong religious persuasion.

The abstraction of this simple elegant mechanism for the production and filtration of diversity in the form of the Genetic Algorithm is straightforward and obvious. However, as it is usually implemented, it is artificial, rather than natural, selection that is the agency determining the direction of computer evolution. Either we ourselves, or our algorithmic agents in the form of explicit fitness functions, typically stand in the role of the breeder in computer implementations of evolution. Yet it is plain that the role of 'breeder' can as easily be filled by 'nature' in the world inside the computer as it is in the world outside the computer—it is just a different 'nature'.

In the following sections, we shall explore a number of examples of computational implementations of the evolutionary process, starting with examples that clearly involve artificial selection, and working our way through to an example

that clearly involves natural selection. The key thing to keep track of throughout these examples is the manner in which we incrementally give over our role as breeder to the 'natural' pressures imposed by the dynamics of the computational world itself.

A Breeder's Paradise: Biomorphs. The first model, a clear-cut example of computational artificial selection, is due to the Oxford evolutionary biologist, Richard Dawkins, author of such highly regarded books as *The Selfish Gene*, *The Extended Phenotype*, and *The Blind Watchmaker*.

In order to illustrate the power of a process in which the random production of variation is coupled with a selection mechanism, Dawkins wrote a program for the Apple Macintosh computer that allows users to 'breed' recursively generated objects.

The program is set up to generate tree structures recursively by starting with a single stem, adding branches to it in a certain way, adding branches to those branches in the same way, and so on. The number of branches, their angles, their size relative to the stem they are being added to, the number of branching iterations, and other parameters affecting the growth of these trees are what constitute the GTYPES of the tree organisms—or 'biomorphs' as Dawkins calls them. Thus the program consists of a general-purpose recursive tree-generator, which takes an organism's GTYPE (parameter settings) as data and generates its associated PTYPE (the resulting tree).

The program starts by producing a simple default—or 'Adam'—tree and then produces a number of mutated copies of the parameter string for the Adam tree. The program renders the PTYPE trees for all these different mutants on the screen for the user to view. The user then selects the PTYPE (i.e. tree shape) he or she likes the best, and the program produces mutated copies of that tree's GTYPE, and renders the associated PTYPES. The user selects another tree, and the process continues. The original Adam tree together with a number of its distant descendants are shown in Fig 1.11.

It is clear that this is a process of artificial selection. The computer generates the variants, but the human user fills the role of the 'breeder', the active selective agent, determining which structures are to go on to produce variant offspring. However, the mechanics of the production of variants are particularly clear: produce slight variations on the presently selected GTYPE. The specific action taken by the human breeder is also very clear: choose the PTYPE whose GTYPE will have variations of it produced in the next round. There is both a producer and a selector of variation.

Algorithmic Breeders. In this section, we investigate a model which will take us two steps closer to natural selection. First, the human breeder is taken out

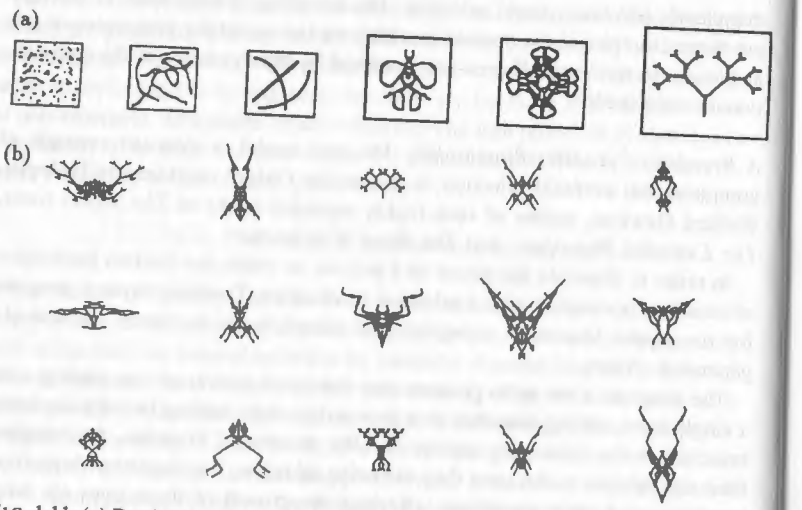


FIG. 1.11. (*a*) Dawkins's original Adam tree, and (*b*) a number of its distant descendants. From Dawkins (1989)

of the loop, replaced by a program he writes, which formalizes his selection criteria, so that the act of selection can be performed by his computational agent. Second, we see that our computational representative can itself be allowed to evolve—an important first step towards eliminating our externally imposed, *a priori* criteria from the process completely.

The system we discuss here is due to Danny Hillis, inventor of the Connection Machine and chief scientist of Thinking Machines Corporation. In the course of the work at TMC, they have a need to design fast and efficient chips for the hardware implementation of a wide variety of common computational tasks, such as sorting numbers. For many of these, there is no body of theory that tells engineers how to construct the optimal circuit to perform the task in question. Therefore, progress in the design of such circuits is often a matter of blind trial and error until a better circuit is discovered. Hillis decided to apply the trial-and-error procedure of evolution to the problem of designing sorting circuits.

In his system, the GTYPES are strings of numbers encoding circuit connections that implement comparisons and swaps between input lines. GTYPES are rendered into the specific circuits they encode—their PTYPES—and they are rated according to the number of circuit elements and connections they require, and by their performance on a number of test strings which they have to sort. This rating is accomplished by an explicit fitness function—Hillis's computational

representative—which implements the selection criteria and takes care of the breeding task. Thus, this is still a case of artificial selection, even though there is no human being actively doing the selection.

Hillis implemented the evolution problem on his Connection Machine CM2—a 64K processor SIMD parallel supercomputer. With populations of 64K sorting networks over thousands of generations, the system managed to produce a 65-element sorter, better than some cleverly engineered sorting networks, but not as good as the best-known such network, which has 60 components. After reaching 65-element sorters, the system consistently became stuck on local optima.

Hillis then borrowed a trick from the biological literature on the co-evolution of hosts and parasites (specifically Hamilton 1980, 1982) and in the process took a step closer to natural selection by allowing the evaluation function to evolve in time. In the previous runs, the sorting networks were evaluated on a fixed set of sorting problems—random sequences of numbers that the networks had to sort into correct order. In the new set of runs, Hillis made another evolving population out of the sorting problems. The task for the sorting networks was to do a good job on the sorting problems, while the task for the sorting problems was to make the sorting networks perform poorly.

In this situation, whenever a good sorting network emerged and took over the population, it became a target for the population of sorting problems. This led to the rapid evolution of sorting sequences that would make the network perform poorly and hence reduce its fitness. Hillis found that this co-evolution between the sorting networks and the sorting problems led much more rapidly to better solutions than had been achieved by the evolution of sorting networks alone, resulting in a sorting network consisting of 61 elements (see Fig. 1.12).

It is the co-evolution in this latter set of runs that both brings us one step closer to natural selection and is responsible for the enhanced efficiency of the search for an optimal sorting network. First of all, rather than having an absolute, fixed value, the fitness of a sorting network depends on the specific set of sorting problems it is facing. Likewise, the fitness of a set of sorting problems depends on the specific set of sorting networks it is facing. Thus the 'fitness' of an individual is now a relative quantity, not an absolute one. The fitness function depends a little more on the 'nature' of the system; it is an evolving entity as well.

Co-evolution increases the efficiency of the search as follows. In the earlier runs consisting solely of an evolving population of sorting networks, the population of networks was effectively hill-climbing on a multi-peaked fitness landscape. Therefore, the populations would encounter the classic problem of getting stuck on local maxima. That is, a population could reach certain structures which lie on relatively low fitness peaks, but from which any deviations result

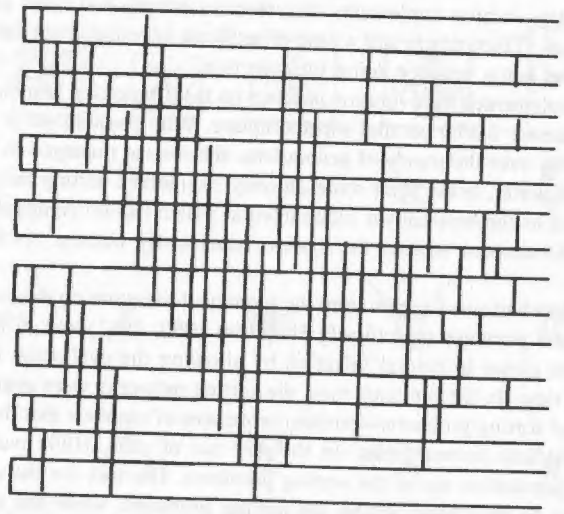


FIG. 1.12. An evolved sorting network showing sequencing of comparisons and swaps. From Hillis (1991)

in lower fitness, which is selected against. In order to find another, higher peak, the population would have to cross a fitness valley, which it is difficult to do under simple Darwinian selection (Fig. 1.13(*a*)).

In the co-evolutionary case, here's what happens (Fig. 1.13(*b*)). When a population of sorting networks gets stuck on a local fitness peak, it becomes a target for the population of sorting problems. That is, it defines a new peak for the sorting problems to climb. As the sorting problems climb their peak, they *drive down the peak on which the sorting networks are sitting*, by finding sequences that make the sorting networks perform poorly, therefore lowering their fitness. After a while, the fitness peak that the sorting networks were sitting on has been turned into a fitness valley, from which the population can escape by climbing up the neighbouring peaks. As the sorting networks climb other peaks, they drive down the peak that they had provided for the sorting problems, which will then chase the sorting networks to the new peaks they have achieved and drive those down in turn.

In short, each population dynamically deforms the fitness landscape being traversed by the other population in such a way that both populations can continue to climb uphill without getting stuck on local maxima. When they do get stuck, the maxima get turned into minima which can be climbed out of by

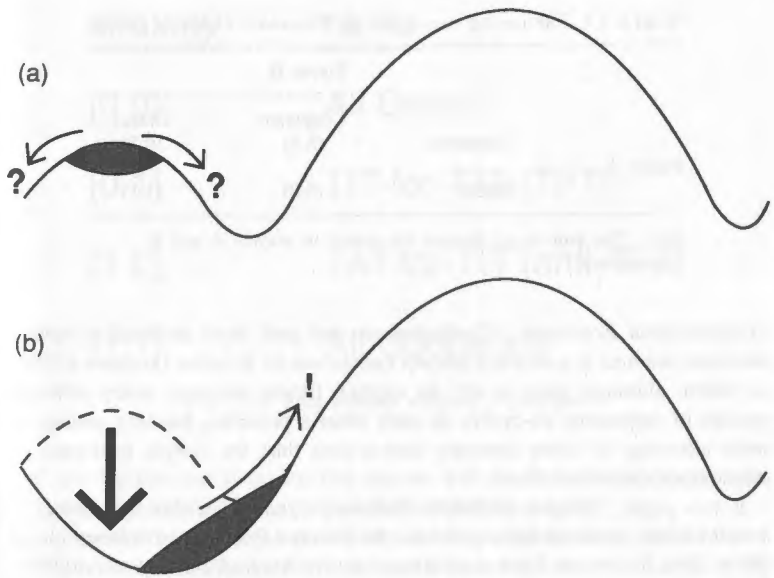


FIG. 1.13. (*a*) Population of working networks stuck on a local fitness peak. In order to attain the higher peak, the population must cross a fitness 'valley', which is difficult to achieve under normal Darwinian mechanisms. (*b*) The co-evolving parasites deform the fitness landscape of the sorting networks, turning the fitness peak into a fitness valley, from which it is easy for the population to escape

simple Darwinian means. Thus coupled populations evolving by Darwinian means can bootstrap each other up the evolutionary ladder far more efficiently than they can climb it alone. By competing with one another, coupled populations improve one another at increased rates.

Thus when coupled in this way, a population may get hung up on local optima for a while, but eventually it will be able to climb again. This suggests immediately that the structure of the evolutionary record for such systems should show periods of stasis followed by periods of evolutionary change. The stasis comes about as populations sit at the top of local fitness peaks, waiting around for something to come along and do them the favour of lowering the peaks they are stuck on. The periods of change come about when populations are released from local optima and are freed to resume climbing up hills, and are therefore changing in time. Hillis has, in fact, carefully documented this kind of *Punctuated Equilibria* in his system.

TABLE 1.1. *The pay-off matrix for the Prisoner's Dilemma Game*

		Player B	
		Cooperate	Defect
Player A	Cooperate	(3,3)	(0,5)
	Defect	(5,0)	(1,1)

Note: The pair (s_1,s_2) denotes the scores to players A and B, respectively.

Computational Ecologies. Continuing on our path from artificial to natural selection, we turn to a research project carried out by Kristian Lindgren (1991), in which, although there is still an explicit fitness measure, many different species of organisms co-evolve in each other's presence, forming ecological webs allowing for more complex interactions than the simple host–parasite interactions described above.

In this paper, Lindgren studies evolutionary dynamics within the context of a well-known game-theoretic problem: the *Iterated Prisoner's Dilemma* model (IPD). This model has been used effectively by Axelrod and Hamilton (1981) in their studies of the evolution of cooperation.

In the prisoner's dilemma model, the pay-off matrix (the fitness function) is constructed in such a way that individuals will garner the most pay-off collectively in the long run if they 'cooperate' with one another by *avoiding* the behaviours that would garner them the most pay-off individually in the short run. If individuals only play the game once, they will do best by not cooperating ('defecting'). However, if they play the game repeatedly with one another (the 'iterated' version of the game), they will do best by cooperating with one another.

The pay-off matrix for the prisoner's dilemma game is shown in Table 1.1. This pay-off matrix has the following interesting property. Assume, as is often assumed in game theory, that each player wants to maximize his immediate pay-off, and let's analyse what player **A** should do. If **B** cooperates, then **A** should defect, because then **A** will get a score of 5 whereas he only gets a score of 3 if he cooperates. On the other hand, if **B** defects, then again, **A** should defect, as he will get a score of 1 if he defects while he only gets a score of 0 if he cooperates. So, no matter what **B** does, **A** maximizes his immediate pay-off by defecting. Since the pay-off matrix is symmetric, the same reasoning applies to player **B**, so **B** should defect no matter what **A** does. Under this reasoning, each player will defect at each time-step, giving them 1 point each per play. However, if they could somehow decide to cooperate, they would

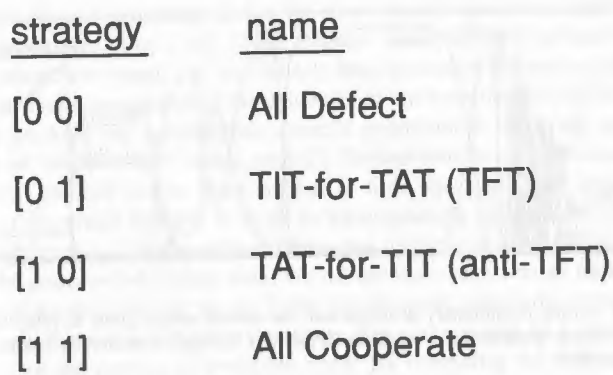


FIG. 1.14. Four possible memory 1 strategies

each get 3 points per play: the two players will do better in the long run by foregoing the action that maximizes their immediate pay-off.

The question is, of course, can ordinary Darwinian mechanisms, which assume that individuals selfishly want to maximize their immediate pay-off, lead to co-operation? Surprisingly, as demonstrated by Axelrod and Hamilton, the answer is yes.

In Lindgren's version of this game, strategies can evolve in an open-ended fashion by learning to base their decisions on whether to cooperate or defect upon longer and longer histories of previous interactions.

The scheme used by Lindgren to represent strategies to play the Iterated Prisoner's Dilemma game is as follows. In the simplest version of the game, players make their choice of whether to cooperate or defect based solely on what their opponent did to them in the last time-step. This is called the *memory 1* game. Since the opponent could have done only one of two things, cooperate or defect, a strategy needs to specify what it would do in either of those two cases. As it has two moves it can make in either of those two cases, cooperate or defect, there are four possible memory 1 strategies. These can be encoded in bit strings of length 2, as illustrated in Fig. 1.14.

If the players should base their decisions by looking another move into the past, to see what they did to their opponent before their opponent made his move, then we would have the *memory 2* game. In this case, there are two moves with two possible outcomes each, meaning that a memory 2 strategy must specify whether to cooperate or defect for each of four possible cases. Such a strategy can be encoded using four bits, twice the length of the memory 1 strategies, so there will be 16 possible memory 2 strategies. Memory 3 strategies require

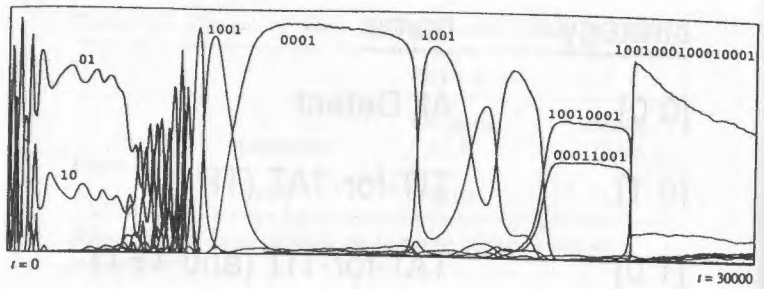


FIG. 1.15. During evolutionary development, the system settles down to relatively long periods of stasis 'punctuated' irregularly by periods of rapid evolutionary change. From Lindgren (1991)

another doubling of the encoding bit string, i.e. 8 bits, yielding 256 possible strategies. In general, memory n strategies require 2^n bits for their encoding, and there will be 2^{2^n} such strategies.

In order to allow for the evolution of higher memory strategies, Lindgren introduces a new genetic operator: gene duplication. As a memory n strategy is just twice as long as a memory $n-1$ strategy, a memory n strategy can be produced from a memory $n-1$ strategy by simply duplicating the memory n strategy and concatenating the duplicate to itself. In Lindgren's encoding strategy, gene duplication has the interesting property that it is a *neutral mutation*. Simple duplication alone does not change the PTYPE, even though it has doubled the length of the GTYPE. However, once doubled, mutations in the longer GTYPE will alter the behaviour of the PTYPE.

Once again, evolution proceeds by allowing populations of different organisms to bootstrap each other up coupled fitness landscapes, dynamically deforming each other's landscapes by turning local maxima into local minima. Again, the fitness of strategies is not an absolute fixed number that is independently computable. Rather, the fitness of each strategy depends on what other strategies exist in the 'natural' population.

Many complicated and interesting strategies evolve during the evolutionary development of this system. More important, however, are the various phenomenological features exhibited by the dynamics of the evolutionary process. First of all, as we might expect, the system exhibits a behaviour that is remarkably suggestive of Punctuated Equilibria. After an initial irregular transient, the system settles down to relatively long periods of stasis 'punctuated' irregularly by periods of rapid evolutionary change (Fig. 1.15).

Second, the diversity of strategies builds up during the long periods of stasis, but often collapses drastically during the short, chaotic episodes of rapid evolutionary succession (Fig. 1.16). These 'crashes' in the diversity of species constitute 'extinction events'. In this model, these extinction events are observed to be a natural consequence of the dynamics of the evolutionary process alone, without invoking any catastrophic, external perturbations (there are no comet impacts or 'nemesis' stars in this model!). Furthermore, these extinction events happen on multiple scales: there are lots of little ones and fewer large ones.

This is important because in order to understand the dynamics of a system that is subjected to constant perturbations, one needs to understand the dynamics of the *un*perturbed system first. We do not have access to an unperturbed version of the evolution of life on Earth; consequently, we could not have said definitively that extinction events on many size scales would be a natural consequence of the process of evolution itself. By comparing the perturbed and unperturbed versions of model systems like Lindgren's, we may very well be able to derive a universal scaling relationship for 'natural' extinction events, and therefore be able to explain deviations from this relationship in the fossil record as due to external perturbations such as the impact of large asteroids.

Third, the emergence of ecologies is nicely demonstrated by Lindgren's model. It is usually the case that a mix of several different strategies dominates the system during the long periods of stasis. In order for a strategy to do well, it must do well by cooperating with other strategies. These mixes may involve three or more strategies whose collective activity produces a stable interaction pattern that benefits all of the strategies in the mix. Together, they constitute a more complex, 'higher order' strategy, which can behave as a group in ways that are impossible for any individual strategy.

It is important to note that, in many cases, the 'environment' that acts on an organism, and in the context of which an organism acts, is primarily constituted of the other organisms in the population and their interactions with each other and the physical environment. There is tremendous opportunity here for evolution to discover that certain sets of individuals exhibit emergent, collective behaviours that reap benefits to all of the individuals in the set. Thus, evolution can produce major leaps in biological complexity, without having to produce more complex individuals by simply discovering, perhaps even 'tripping over', the many ways in which collections of individuals at one level can work together to form aggregate individuals at the next higher level of organization (Buss 1987).

This is thought to be the case for the origin of eukaryotic cells, which are viewed as descended from early cooperative collections of simpler, prokaryotic cells (Margolus 1970). It is also the process involved in the origin of multicellular organisms, which led to the Cambrian explosion of diversity some 700 million years ago. It was probably a significant factor in the origin of the prokaryotes

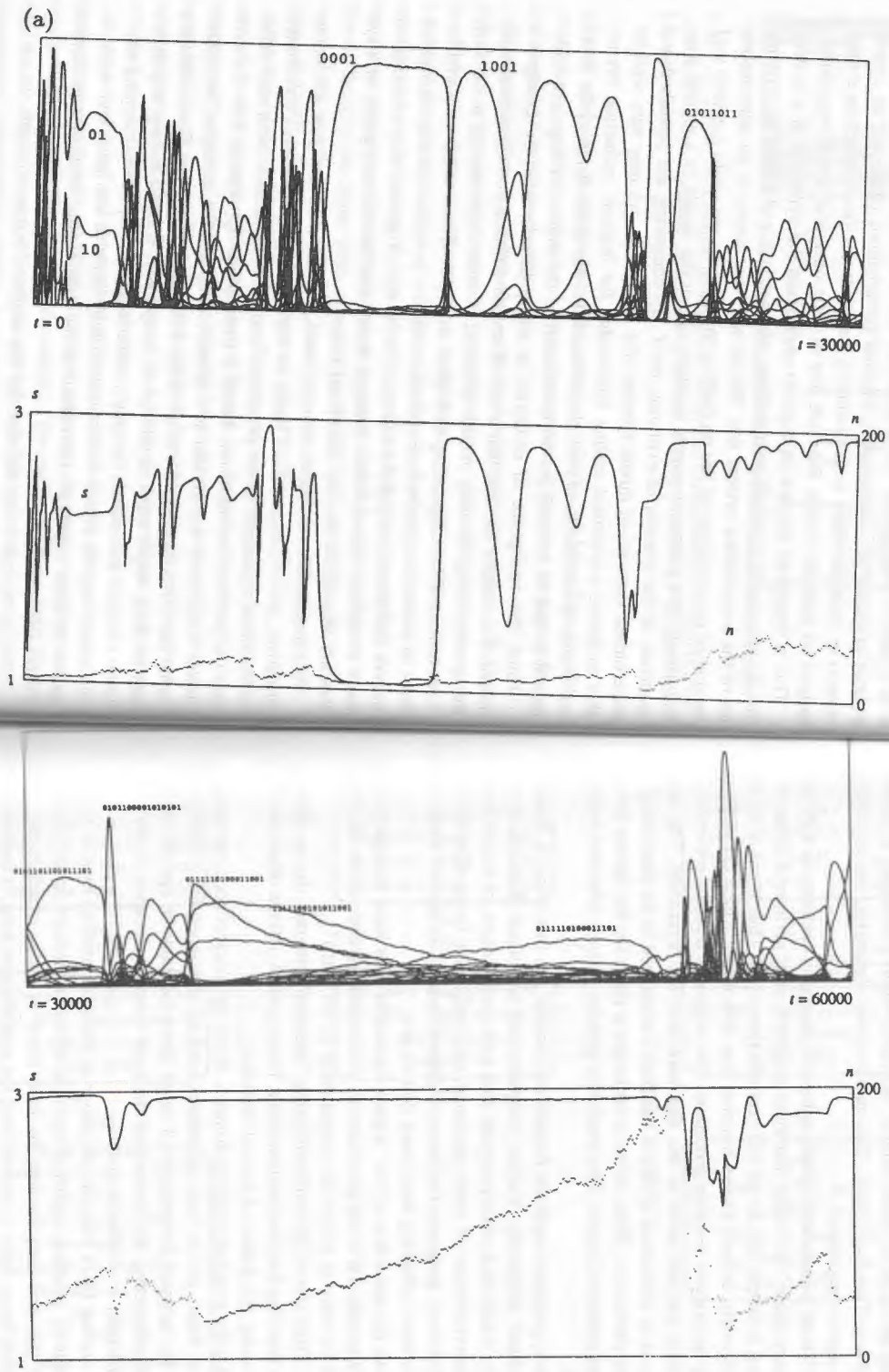


FIG. 1.16. The evolutionary dynamics of strategies in the iterated prisoner's dilemma system of Lindgren. In both cases, the top trace plots the changing concentration of strategies in the population while the bottom trace shows two things: the solid line plots the average fitness of the population, while the dotted line plots the diversity of species (the number of different strategies in the population at any time). In all cases, time is traced on the horizontal axis. The top traces illustrate the interplay between metastable and chaotic episodes, while the bottom traces illustrate the 'extinction events' that are often associated with the end of metastable periods. These extinction events can be quite large, as is seen in the bottom trace

themselves, and it has been discovered independently at least seven times by the various social insects (including species of wasps, bees, ants, and termites).

The final step in eliminating our hand from the selection/breeding process and setting the stage for true 'natural' selection within a computer is taken in a model due to Tom Ray (1991). This step involves eliminating our algorithmic breeding agent completely.

In his 'Tierra' simulation system, computer programs compete for CPU time and memory space. The 'task' that these programs must perform in order to be reproduced is simply the act of self-reproduction itself! Thus there is no need for an externally defined fitness function that determines which GTYPES get copied by an external copying procedure. The programs reproduce themselves, and the ones that are better at this task take over the population. The whole external task of evaluation of fitness has been internalized in the function of the organisms themselves. Thus, there is no longer a place for the human breeder or his computational agent. This results in genuine natural selection within a computer.

In Tierra, programs replicate themselves 'noisily', so that some of their offspring behave differently. Variant programs that reproduce themselves more efficiently, which trick other programs into reproducing them, or which capture the execution pointers of other programs, etc., will leave more offspring than others. Similarly, programs that learn to defend themselves against such tricks will leave more offspring than those that do not.

We shall discuss a few of the 'digital organisms' that have emerged within the Tierra system. (It is not necessary to understand the code in the illustrated programs in order to follow the explanation in the text.)[9]

Fig. 1.17(*a*) shows the self-replicating 'ancestor' program that is the only program Tom Ray has ever written in the Tierra system. All the other programs evolved under the action of natural selection.

The ancestor program works as follows. In the top block of code, the program locates its 'head' and its 'tail', templates marking the upper and lower boundaries of the program in memory. It saves these locations in special registers and, after subtracting the location of the head from the location of the tail, it stores its length in another register.

In the second block of code, the program enters an endless loop in which it will repeatedly produce copies of itself. It allocates memory space of the appropriate size and then invokes the final block of code, which is the actual reproduction loop. After it returns from the reproduction loop, it creates a new execution pointer to its newly produced offspring, and cycles back to create another offspring.

[9] The details are to be found in Ray (1991); see below, Ch. 3.

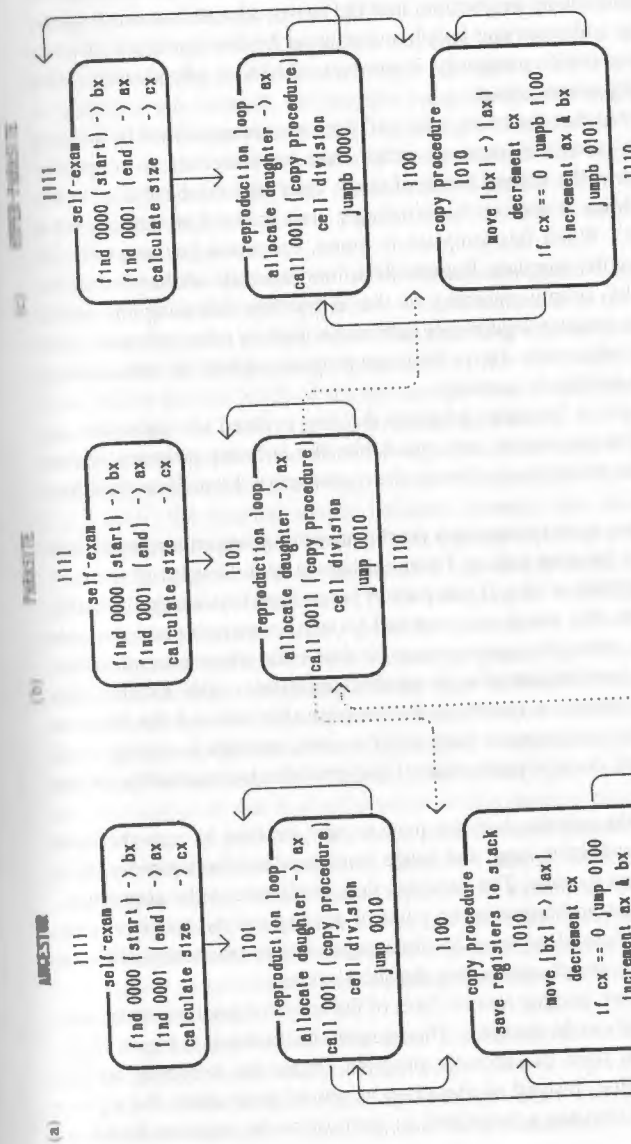


FIG. 1.17. Digital organisms from Ray's (1991) Tierra simulation system: (*a*) self-reproducing ancestor; (*b*) an early parasite of the ancestor; (*c*) a descendant of the ancestor that is immune to the parasite

In the third and final block of code, the reproduction loop, the program copies itself, instruction by instruction, into the newly allocated memory space, making use of the addresses and length stored away by the first block of code. When it has copied itself completely, it returns to the block of code that called it, in this case, the second block.

It should be noted that 'function calls' in Tierra are accomplished by seeking for a specific bit pattern in memory rather than by branching to a specific address. Thus, when the second block of code 'calls' the third block of code, the reproduction loop, it does so by initiating a seek forward in memory for a specific 'template'. When this template is found, execution begins at the instruction following the template. Returns from function calls are handled in the normal manner, by simply returning to the instruction following the initial function call. This template-addressing scheme is used in other reference contexts as well, and helps make Tierra language programs robust to mutations, as well as easily relocatable in memory.

Fig. 1.17(*b*) shows a 'parasite' program that has evolved to exploit the ancestor program. The parasite is very much like the ancestor program, except that it is missing the third block of code, the reproduction loop. How then does it copy itself?

The answer is that it makes use of a nearby ancestor program's reproduction loop! Recall that a function call in Tierra initiates a seek forward in memory for a particular template of bits. If this pattern is not found within the initiating program's own code, the search may proceed forward in memory into the code of other organisms, where the template may be found and where execution then begins. When the invoked function in another organism's code executes the 'return' statement, execution reverts to the program that initiated the function call. Thus, organisms can execute each other's code, and this is exactly what the parasite program does: it makes use of the reproductive machinery of the ancestor host.

This means that the parasite does not have to take the time to copy the code constituting the reproductive loop, and hence can reproduce more rapidly, as it has fewer instructions to copy. The parasites thus proliferate in the population. However, they cannot proliferate to the point of driving out the ancestor hosts altogether, for they depend on them for their reproductive machinery. Thus, a balance is eventually struck optimizing the joint system.

Eventually, however, another mutant form of the ancestor emerges which has developed an immunity to the parasites. This program is illustrated in Fig. 1.17(*c*). Two key differences from the ancestor program confer the immunity to the parasite programs. First, instead of executing a 'return' instruction, the reproduction loop instead initiates a jump back in memory to the template found in

the instruction that calls the reproduction loop. This has the same effect as a return statement when executed by the immune program, but has a very different effect on the parasite. The second important difference is that following the cell division in the second block (which allocates a new execution pointer to the offspring just created), the program jumps back to the beginning of the first block of code, rather than to the beginning of the second block. Thus, the immune program constantly resets its head, tail, and size registers. This seems useless when considering only the immune organism's own reproduction, but let's see what happens when a parasite tries to execute the reproduction loop in an immune organism.

When a parasite attempts to use the immune program's reproduction code, the new jump transfers the parasite's execution pointer to the second block of the immune program's code, rather than returning it to the second block of the parasite code, as the parasite expects. Then, this execution pointer is further redirected to the first block of the immune program, where the registers originally containing the head, tail, and length of the parasite are reset to contain the head, tail, and length of the immune organism. The immune program has thus completely captured the execution pointer of the parasite. Having lost its execution pointer, the parasite simply becomes dormant data occupying memory, while the immune program now has two execution pointers running through it: its own original pointer, plus the pointer it captured from the parasite. Thus, the immune program now reproduces twice as rapidly as before. Once they emerge, such immune programs rapidly drive the parasites to extinction.

Complex interactions between variant programs like those described above continue to develop within evolutionary runs in Tierra. From a uniform population of self-reproducing ancestor programs, Ray, a tropical biologist by training, notes the emergence of whole 'ecologies' of interacting species of computer programs. Furthermore, he is able to identify many phenomena familiar to him from his studies of real ecological communities, such as competitive exclusion, the emergence of parasites, key-stone predators and parasites, hyper-parasites, symbiotic relationships, sociality, 'cheaters', and so forth.

Again, the actual 'fitness' of an organism is a complex function of its interactions with other organisms in the 'soup'. Collections of programs can cooperate to enhance each other's reproductive success, or they can drive each other's reproductive success down, thus lowering fitness and kicking the population off of local fitness peaks.

Not surprisingly, Ray, too, has noted periods of relative stasis punctuated by periods of rapid evolutionary change, as complex ecological webs collapse and new ones stabilize in their place. Systems like Ray's Tierra capture the proper context for evolutionary dynamics, and natural selection is truly at play here.

8. CONCLUSION

This article is intended to provide a broad overview of the field of Artificial Life, its motivations, history, theory, and practice. In such a short space, it cannot hope to go into depth in any one of these areas. Rather, it attempts to convey the 'spirit' of the Artificial Life enterprise via several illustrative examples coupled with a good deal of motivating explanation and discussion.

The field of Artificial Life is in its infancy, and is currently engaged in a period of extremely rapid growth, which is producing many new converts to the principles detailed here. However, it is also raising a significant amount of controversy, and is not without its critics. The notion of studying biology via the study of patently non-biological things is an idea that is hard for the traditional biological community to accept. The acceptance of Artificial Life techniques within the biological community will be directly proportional to the contributions it makes to our understanding of biological phenomena.

That these contributions are forthcoming, I have no doubt. However, high-quality research in Artificial Life is difficult, because it requires that its practitioners be experts in both the computational sciences and the biological sciences. Either of these alone is a full-time career, and so the danger lurks of doing either masterful biology but trivial computing, or doing masterful computing but trivial biology.

Therefore, I strongly suggest incorporating a trick from nature: cooperate! As is amply illustrated in many of the examples discussed in this article, nature often discovers that collections of individuals easily solve problems that would be extremely difficult or even impossible for individuals to solve on their own. Collaborations between biologists and computer scientists are quite likely to be the most appropriate vehicles for making significant contributions to our understanding of biology via the pursuit of Artificial Life.

So, if you are a computer expert dying to hack together an evolution program, go find yourself a top-notch evolutionary biologist to collaborate with, one who will bring to the enterprise an in-depth understanding of the subtleties of the evolutionary process plus a proper set of open questions about evolution towards which your evolution program might be addressed.

On the other hand, if you are a field biologist interested in doing some numerical simulations in order to understand the ecological dynamics you are observing in the field, hook up with a top-notch parallel-computing expert, who will bring to the enterprise a thorough knowledge of the subtleties involved in multi-agent interactions, and will be in possession of an equally open set of questions, which you very well might find to be strikingly related to your own.

Above all, when in doubt, turn to Mother Nature. After all, she is smarter than you![10]

REFERENCES

Axelrod, R. (1984), *The Evolution of Cooperation* (New York: Basic Books).
—— and Hamilton, W. D. (1981), 'The Evolution of Cooperation', *Science*, 211: 1390–6.
Burks, A. W. (1970) (ed.), *Essays on Cellular Automata* (Urbana, Ill.: University of Illinois Press).
Buss, L. (1987), *The Evolution of Individuality* (Princeton: Princeton University Press).
Chapuis, A., and Droz, E. (1958), *Automata: A Historical and Technological Study*, trans. A. Reid (London: Batsford Ltd.).
Dawkins, R. (1989), 'The Evolution of Evolvability', in Langton (1989*b*), 201–20.
Frisch, U., Hasslacher, B., and Pomeau, Y. (1986), 'Lattice Gas Automata for the Navier-Stokes Equation', *Physical Reviews and Letters*, 56: 1505–8.
Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning* (Reading, Mass.: Addison-Wesley).
Hamilton, W. D. (1980), 'Sex Versus Non-Sex Versus Parasite', *OIKOS*, 35: 282–90.
—— (1982), 'Pathogens as Causes of Genetic Diversity in their Host Populations', in R. M. Anderson and R. M. May (eds.), *Population Biology of Infectious Diseases* (Berlin: Springer-Verlag, 1982), 269–96.
Hillis, W. D. (1991), 'Co-evolving Parasites Improve Simulated Evolution as an Optimization Procedure', in Langton *et al.* (1991), 313–24.
Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence* (Ann Arbor: University of Michigan Press).
—— (1986), 'Escaping Brittleness: The Possibilities of General Purpose Learning Algorithms Applied to Parallel Rule-Based Systems', in R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (eds.), *Machine Learning II* (New York: Kaufman), 593–623.
Hopcroft, J. E., and Ullman, J. D. (1979), *Introduction to Automata Theory, Languages, and Computation* (Menlo Park, Calif.: Addison-Wesley).
Koza, J. R. (1991), 'Genetic Evolution and Co-evolution of Computer Programs', in Langton *et al.* (1991), 603–30.
Langton, C. G. (1984), 'Self-Reproduction in Cellular Automata', *Physica D*, 10/1–2: 135–44.
—— (1986), 'Studying Artificial Life with Cellular Automata', *Physica D*, 22: 120–49.

[10] A large number of people have assisted in writing this paper, which is based on my lecture notes for the Complex Systems Summer School and on the overview of Artificial Life that served as an introduction to the proceedings of the first Artificial Life workshop. Besides the people credited in the latter paper, I should like to thank the following people for their help with this version: Tom Ray, Kristian Lindgren, Danny Hillis, and John Koza. I should also like to thank Ronda Butler-Villa and Della Ulibarri for their patience with me and for their skill in preparing the figures and text for publication.

Langton, C. G. (1989*a*), 'Artificial Life', in Langton (1989*b*).
—— (1989*b*), *Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems* (Santa Fe Institute Studies in the Sciences of Complexity, Proceedings, 6; Redwood City, Calif.: Addison-Wesley).
—— Taylor, C., Farmer, J. D., and Rasmussen, S. (1991) (eds.), *Artificial Life II* (Santa Fe Institute Studies in the Sciences of Complexity, Proceedings, 10; Redwood City, Calif.: Addison-Wesley).
Lindgren, K. (1991), 'Evolutionary Phenomena in Simple Dynamics', in Langton *et al.* (1991), 295–312.
Margolus, L. (1970), *Origin of Eucaryotic Cells* (New Haven: Yale University Press).
Prusinkiewicz, P. (1991), *The Algorithmic Beauty of Plants* (Berlin: Springer-Verlag).
Ray, T. S. (1991), 'An Approach to the Synthesis of Life', in Langton *et al.* (1991), 371–408, and Chapter 3 below.
Reynolds, C. W. (1987), 'Flocks, Herds, and Schools: A Distributed Behavioral Model', Proceedings of SIGGRAPH '87, *Computer Graphics* V 21/4: 25–34.
Toffoli, T. (1984), 'Cellular Automata as an Alternative to (Rather than an Approximation of) Differential Equations in Modeling Physics', in J. D. Farmer, T. Toffoli, and S. Wolfram (eds.), *Cellular Automata: Proceedings of an Interdisciplinary Workshop* (Los Alamos, New Mexico, March 7–11, 1983) = *Physica D* (special issue), 10/1–2.
—— and Margolus, N. (1987), *Cellular Automata Machines* (Cambridge, Mass.: MIT Press).
Ulam, S. (1962), 'On Some Mathematical Problems Connected with Patterns of Growth of Figures', *Proceedings of Symposia in Applied Mathematics*, 14: 215–24. Repr. In Burks (1970).
Von Neumann, J. (1966), *Theory of Self-Reproducing Automata*, ed. and completed by A. W. Burks (Urbana, Ill.: University of Illinois Press).
Wilson, S. W. (1989), 'The Genetic Algorithm and Simulated Evolution', in Langton (1989), 157–65.
Wolfram, S. (1986), 'Cellular Automaton Fluids 1: Basic Theory', *Journal of Statistical Physics*, 45: 471–526.

2

AUTONOMY AND ARTIFICIALITY

MARGARET A. BODEN

1. THE PROBLEM—AND WHY IT MATTERS

When Herbert Simon wrote his seminal book on 'the sciences of the artificial' (Simon 1969), he had in mind artificial intelligence (AI) and cybernetics. Now, the sciences of the artificial include artificial life (A-Life) also. A-Life uses informational concepts and computer-modelling to study the functional principles of life in general (Langton 1989). Simon's use of the word 'sciences' was well chosen. The interests of A-Life, as of AI, are largely scientific, not technological. That is, many researchers in these two fields hope to contribute to theoretical biology and/or psychology.

The relations between A-Life and AI are complex. One might define A-Life as the abstract study of life, and AI as the abstract study of mind. But if one assumes that life prefigures mind, that cognition is—and must be—grounded in self-organizing adaptive systems, then AI may be seen as a sub-class of A-Life. Certainly, A-Life is theoretically close to some recent work in AI (described below). However, A-Life workers often go out of their way to distance their research from AI. In doing so, they usually stress the concept of autonomy—which, they say, applies to A-Life models but not to AI.

Since autonomy of some kind is generally thought to be an important characteristic of both life and mind, A-Life and AI should have implications for our understanding of human autonomy, or freedom. These implications are not purely abstract, to be forgotten when one leaves one's study to play a game of backgammon. What science tells us about human autonomy is practically important, because it affects the way in which ordinary people see themselves—which includes the way in which they believe it is possible to behave.

If science tells us that we lack freedom, we may be less likely to try to exercise it. An observable decline of personal autonomy was reported many

 An emended version of a paper called 'Artificial Intelligence and Human Dignity', in J. Cornwell (ed.), *Nature's Imagination: The Frontiers of Scientific Vision* (Oxford: Oxford University Press, 1995), 148–60. Reprinted by permission of Oxford University Press.