LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

LA-UR-89-2618

LA-UR--89-2618

DE89 016605

TITLE CORE EVOLUTION: EMERGENCE OF COOPERATIVE STRUCTURES IN A COMPUTATIONAL CHEMISTRY

AUTHOR(S) Steen Rasmussen, Rasmus Feldberg, Morten Hinsholm, and Carsten Knudsen

SUBMITTED TO Proceedings of the Los Alamos (CNLS) sponsored Annual 9th Conference, "Emergent Computation ...," held in Los Alamos, New Mexico, May 22-26, 1989

DISCLAIMER

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

CORE-EVOLUTION: EMERGENCE OF COOPERATIVE STRUCTURES IN A COMPUTATIONAL CHEMISTRY

Steen Rasmussen[1,2,*], Rasmus Feldberg[1], Morten Hindsholm[1], and Carsten Knudsen[1]

[1]Physics Laboratory III

The Technical University of Denmark

DK-2800 Lyngby

DENMARK

[2]Center for Nonlinear Studies and Theoretical Division, MS-B258

Los Alamos National Laboratory

Los Alamos, NM 87545

USA

*Communication to Steen Rasmussen

Los Alamos National Laboratory

e-mail: steen@cardinal.lanl.gov.arpa

Key words:

adaptability, artificial life, biological evolution, cellular automata, complexity measure, computational ecology, computer virus, emergent computation, open-ended evolution, origin of life, self-organization, Von Neumann assembler.

ABSTRACT

We have developed an artificial ecology in the computer core, where one is able to evolve assembler-automaton code without any predefined evolutionary path. The system, in the present version has one dimension, is updated in parallel, the instructions are only able to communicate locally, and the system is continuously subjected to noise. The system also has a notion of computational resources. Depending on the specified parameters and the level of complexity, or distance from a randomized core, this electronic garden is started at, we see different evolutionary paths. For several initial conditions the system is able to develop extremely viable cooperative programs (organisms ?) which totally dominates the core. This demonstrates the emergence of complex functional properties in a computational environment.

1. INTRODUCTION

Up till now most quantitative work on evolution, and origin of life has been done by *a priori* outlining a specific evolutionary route. Assuming that the chosen path is relevant, modeling and calculation has been performed in order to analyze the different evolutionary consequences (Eigen (1971), Eigen and Schuster (1979), Kauffman (1986), Farmer et. al. (1986), Rasmussen (1989), Rasmussen et. al. (1989)). By such an approach one unfortunately neglects an infinity of alternative possibilities.

However, in origin of life studies the selection of a single possible path seems necessary both due to the number and due to the complexity of all the possible chemical reaction here on Earth. One can't take all possibilities into account. The same is true for the study of the evolution of organisms. One is forced to focus on a few factors. Further, looking at the history of science, much progress has been achieved using the approach: An observed phenomenon is explained by selecting a possible mechanism, which without violating the laws of Nature is able to model the phenomenon. Experiments then help us eliminate irrelevant models. Unfortunately experimental work in the area of evolution is very cumbersome, both because we are standing with the products of evolution and therefore can only guess under which conditions they were developed, and also because evolutionary experiments by definition demand large populations observed over many generations.

Another quantitative approach without any predefined evolutionary route *is* possible using a much simpler, but "artificial" chemistry. By defining a universal set of low level rules, an open evolutionary process can be started in the computer. Depending on the internal structure and parameters in such a set up, the system then chooses its own evolutionary direction and sometimes develops very interesting structures. In some cases such a system will first find a stable subset in its possible interaction phase space and from there start to "develop" structures from suitable building blocks. On Earth the interesting subset seems to be carbon-chain chemistry.

The different approaches are further discussed in Fig. 1.

Figure 1

Inspired by the computer game "core wars" (Dewdney, 1984), we have build a core simulator system in which we are only operating few basic instructions. The object of the core wars game is to cause the opposing player to terminate abnormally. The object for our system is to be able to create new computational properties. In this first version of our simulator, we have followed the specifications for a core simulator given in Int. Core Wars Soc., 1986 quite closely, although there are a number of significant differences, which allows us to evolve programs in the core.

The basic instructions in our core together with their interaction rules define our artificial chemistry. With these low level rules we want to be able to evolve functional properties in our system, similar to the way in which Dawkins (1989) evolves his "biomorph" shapes, but without any external human selection mechanism. We named our simulator VENUS hoping that it would be able to create interesting properties.

Related work in progress in this area includes Collins, Jefferson, and Tayler (1989), Langton (1986), McCaskill (1988), and Packard (1989). For a discussion of open-ended evolution, see for instance Farmer and Packard (1986) and Rasmussen (1985). For a general introduction to the field of "artificial life" see Langton (1989).

2. THE CORE SIMULATOR VENUS

The core in our simulator is one-dimensional and has periodic boundary conditions. Each address in the core is occupied by one of the basic instructions. An instruction is executed if it's address has a pointer, and if it's neighborhood has sufficient computational resources. The pointer location is opdated after the instruction, at which it is located, is executed. Unless the executed instruction tells the pointer to move to a specified location, the pointer moves to the next address in the memory, as the core is updated. The core is further described in figure 2, and some simple examples of programs are described in figure 3. The basic instruction set, called red-code, is the same as is used in the "core-wars" game. In table 1, the ten instructions and their addressing modes are discussed.

Figure 2

Figure 3

In many respects the core operates as an ordinary multitasking Von Neumann assembler level machine.

Each address in the core is associated with a certain amount of computational resources, $r(x,t)$, which is measured in fractions of one execution, one *exec*. An *exec* is equivalent to what is used in the execution for a single instruction. Thus, an instruction with a pointer can only be executed if it's neighborhood has computational resources equivalent to at least one *exec*. The resource neighborhood is defined by a radius R_{res}. The number of addresses each instruction can obtain resources from is therefore $2R_{res} + 1$. The simulator executes instructions and hereby removes resources in a sequence determined by the order

in which the pointers occur in the execution queue. Resources are renewed by an amount Δr at each core update. However the resources are never allowed to exceed a maximum resource level, r_{max}. The resource level at any address at any time is therefore determined by

$$r(x, t+1) = min[r_{max}\ ,\ \ r_L(x,t)\ +\ \Delta r] \tag{1}$$

where r_{max} is a global parameter always smaller than one *exec*. Δr, also $\leq\ 1$, is a global parameter and is the amount by which the resources associated with each address is increased with at each update of the core. $r_L(x,t)$ is recursively defined as an iteration over the entiere pointer execution queue of length L. In general

$$r_j(x,t) = \begin{cases} \alpha(x_j)\ r_{j-1}(x,t), & if\ x \in N(x_j),\ and\ if \sum_{k=x_j-R_{res}}^{x_j+R_{res}} r_{j-1}(k,t) \geq 1 \\ r_{j-1}(x,t), & else \end{cases} \tag{2}$$

where $r_0(x,t)$ is defined as $r(x,t)$. $N(x_j)$ defines the resource neighborhood for the j-th pointer located at position x_j, and the sum counts the aviable resources in the neighborhood. $\alpha(x_j)$ is defined as

$$\alpha(x_j) = \frac{\sum_{k=x_j-R_{res}}^{x_j+R_{res}} r_{j-1}(k,t)\ \ -\ 1}{\sum_{k=x_j-R_{res}}^{x_j+R_{res}} r_{j-1}(k,t)} \tag{3}$$

Large Δr (and r_{max}) defines "jungle" conditions opposite to small Δr (and r_{max}) which defines the "desert". A small r_{max} can of course be compensated by a large R_{res} but is not computational effective for our simulations.

Besides the limited computational resources associated with each core address, the system also has a limited number of active pointers. The current system has 220 places in the execution queue ($L = 220$), which means that each core update only executes at most 220 instructions.

The VENUS core also has another important locality parameter, the operation radius R_{opr}. This radius defines how far away each instruction is allowed to access and alter data

relative to its own address. The operation radius R_{opr} and the resource radius R_{res} allow us to achieve locality in our core.

The core is updated in parallel. When the system is running it has many pointers active at the same time. All the instructions associated with these pointers are updated before changes in the core are made. In this way each instruction "sees" the same core, when we are simulating on a sequential machine. In case of conflict between two instructions the instruction with the highest number in the execution queue will have its changes effected. However, the details on how conflicts are resolved does not affect the system's ability to evolve.

Table 1

As described so far our system still misses a fundamental property: the creative aspect of evolution or the notion of *noise*. We have introduced random fluctuations in two different ways. Whenever a MOV-instruction is executed there is a certain probability P_{mut}, that the affected word, the word the MOV-instruction copys (the source operand of the MOV-instruction), mutates. Each word has in this situation an equal probability to change into any of the ten instructions. The operands for the new instruction are also chosen at random. A traveling pointer dies whenever it meets a DAT-instruction. Although a pointer is duplicated whenever it meets a SPL-instruction, there is a finite probability that every pointer in the core dies, because every instruction initially is equally distributed. In order to assure that the system is always active, we disturb the core by introducing new pointers at random with a low pointer appearance frequency P_{point}, each time the core is updated (One core update is of course equivalent to one generation in the simulator). This is an additional way of driving the system besides the influx of computational resources, Δr.

Given these properties, the low level rules for our system are defined somewhere between a classic cellular automaton (CA) and a classic Von Neumann assembler language. We shall refer to it as an assembler-automaton. It is a compact and powerfull form for writing a CA with a very high number of possible states. This form has the effect that any mutation in an instruction causes the instruction to change into one of the ten legal instructions. Changes are thereby restricted.

Looking at the functional properties of a particular instruction and the instructions it communicate with, almost any mutation in an instruction will cause a significant change in the functional properties of the involved communicating instructions. Any change will normally cause the instruction and its neighborhood to "jump" to another location in the space of functional properties (not in the instruction space). The system misses a fundamental notion of continuity of functional properties with respect to perturbations.

In the next section we shall se how this assembler-automaton combination works as a simple artificial chemistry.

The present version of our system is implemented on an IBM PS 80. On this system we are able to take the "artificial garden" (the core) out and analyze it after we have grown it for some time. In this way we are able to save interesting gardens, and eventually grow them at some later point. When we start a simulation we are also able to "seed" an "engineered" program sequence at a random location in the core. We have three different ways to investigate the dynamics of the evolution interactively: (1) The whole core can be shown at the same time, with each instruction having a different color, and with the active pointers shown as highlighted underscores, (2) A selected part of the core can be followed in more detail, with successive generations of the selected core area can be seen simultaneosly (like for a one dimensional cellular atomaton), (3) The sequence of currunt executing instructions can be written to the screen. On top of these features, we have a number of tools useful for inspecting the core. A typical simulation follows the execution of $\sim$ 22,000,000 instructions over 100,000 core updates, and takes about twelve hours.

3. EVOLUTION IN VENUS

The initial work with this new world was both exciting and very frustrating. It was obvious that something was happening as the simulation proceeded. The core was changing. Because the basic physics of VENUS is so different from our "real" physics, we did'nt have a clue about, what would develop in the system. We did'nt know what we should be looking for in the core. We therefore had to dump many, many cores (each hardcopy of a core occupies several paper meters) and simply see if the human eye was able to catch anything of interest. In this period, we learned to distinguish between a core developed under desert conditions and a core grown as a jungle. We also became pretty good at "dating" a core, i.e. to guess for how many generations it had been active.

From these "field studies" a pattern of different interesting structures slowly emerged. We named some of the structures and were later able to recognize them when they appeared in other cores. In the desert with no initial structure we normally meet a number of "simple loops" after some thousand core updates. In the jungle we also found loops, some of them more complicated than the ones we found in the desert. The jungle seems to be characterized by the development of "SPL-alls": pointer-dense structures kept alive by one or more SPL-instructions. These were never found in the desert, due to their high density of pointers. Looking very carefully, we also became aware of the existense of "fossils" in cores which had been simulated for some time. These fossils could for instance be traces of the simple self-propagating instruction (see figure 3, (a)), and varius loops, where for instance a DJN-instruction earlier closing the loop, was eventualy counted down to zero, and thereby allowed the pointer to pass. We could locate these loops when some instruction inside the loop had a significant effect on the surroundings. Another thing we learned was that any "human engineered" organism - both programs we designed and the programs designed to participate in the core-wars - were too brittle to survive in the noisy VENUS universe. They die after a number of generations, depending on the noise level. However, later we learned that the system was able to develop its own stable organisms.

In many respects we were in a situation similar to Eigens group in Göttingen, performing *in vitro* evolutionary experiments with RNA. To decribe the nature of their work both on the development and on the operation of their evolution machine Günter Bauer said that, "...it took us one and a half years to build our evolution machine, it took us three hours to run the first experiment on the machine, and it has now taken us more than three months to analyze the huge amount of data this experiment has produced..." .

It took us a very long time before we were ready to test evolution of different environments in a more systematic way.

From the systematic simulations we have performend with VENUS up til now, we can draw some conclusions we believe will hold:

(i) A different parameter set in general causes a different evolutionary path, although some continuity is seen for many parameter combinations. In these parameter ranges a slight change of the parameters does not seem to effect the path.

(ii) Jungles develop more interesting structures than deserts. We have not been able to develop any really viable structures in the desert environment.

(iii) For some parameter sets, the system always seems to develop the same features independently of the detailed structure of the initial randomized core. This is for instance the case for deserts with very low inflow of resources. Also a jungle with no initial structure and a high operational radius seems to follow the same evolutionary path every time.

(iv) For other parameter sets, the evolutionary path is extremely sensitive to details of the initial random core and the noise. For instance this is the case when a jungle is evolved with some initial structure and a large operational radius. In such a situation, we have a high sensitivity to initial conditions, and the system has many coexisting attractors.

In the following, we shall discuss some of the evolutionary processes in details.

By default we start the simulations with one of two different active initial seeds in the core. One is an evolution from a total randomized environment and the other is an evolution with a simple kind of replication present from the very beginning. We can start

the simulation by placing an active JMP($0) somewhere in the randomized core. This instruction does not effect any of the other addresses in the core (recall table 1). Alternatively we can start the simulation by introducing a more sophisticated self-replicating program. The details of this program are not important. (The full self-replicating program is shown in the appendix) It consists of 8 instructions and has a cycle of 18 core updates. It has, however, an instruction copying loop similar to the loop shown in example (b), figure 3. The presence of this loop is important for what we are able to develop within reasonable spatio-temporal limits with the totally randomized core we use. The major effect of this program is therefor to replicate a copying loop, which after some mutations are able to multiply different instructions. The self-replicating program has a number of copying loops as closest "Hamming" neighbors in the space of funtional properties. It is important to note that this program only is used to create a certain kind of inhomogenuity in the randomized core. We belive that the same effect could be achieved by using a biased core instead of this replication program.

Some of the sensitivity to initial conditions we meet in simulations where we use the self-replicating instruction program as initial seed, is caused by the fluctuating number of functional loops the program is able to produce before some mutation causes it to colapse, and which instructions the copying loops turns out to be able to multiply.

A typical result for the evolution of a desert with small operation radius and no initial structure is that it will develop into a relatively stable core with (a) many fixed points for the pointers, (b) some simple loops, (c) and few more complicated loops. The precise parameters are shown in Table 2. The fixed points are JMP(# X), JMZ(# X, 0), JMN(# 0, Y), and DJN(# X, Y) where X can be anything and Y is different from zero. These instructions are all characterized by pointing to themselves. The JMZ(# X, 0) is rare because it is unlikely that the second argument will be exactly zero. They can only interact with other instructions, by changing their B-fields. A typical simple loop structure consists of some kind of a JMP-instruction, which sends the pointer back in the core, where

it travels forwards in the core until it again meets the JMP-instruction. After some time a loop may dissappear, due to some instructions inside altering an address somewhere. More complicated loops with overlapping loops, nested loops, and loops in series are also found.

The evolution in the desert initiated with some instruction copying loops, and parameters as shown in Table 2, does not seem to be very different from the above situation. Slightly more complicated loops are seen here, which probably is caused by the remains of the mutated self-replicating programs. The evolution has only changed the global chemical composition a little bit. The global distribution of the different instructions is almost identical with the initial instruction distribution.

Table 2.

The jungle simulated with a small operation-radius R_{opr}, is able to develop more complicated structures. The significant new feature appering in the jungle is clusters of dense programs, driven by pointers from SPL-instructions. Earlier we called these structures "SPL-falls". For a large R_{opr} (larger than 500) and no initial structure in the core, a typical evolutionary path in the jungle is to develop a number of loops with some JMP-instruction in the end to return the pointers. These loops will dominate the core for some thousand core updates. However, more and more pointers will be traped at simple fixed points. In one simulation (R_{opr} = 2,000, Δr = 0.25, and else a jungle defined like in table 2) we found 7 JMP(#), 3 JMN(#), and 202 DJN(#); 212 out of 220 possible pointers after 455,000 generations ($\sim$ 96 million instructions). This state of the core is very stable.

The system seems to develop more interesting structures for smaler R_{opr}, when no initial structure is present. However, the story is different if some initial structure is used.

An evolutionary history of a jungle initiated with one of the self-replicating program in the core is seen in Fig. 4.

Fig. 4.

Setting $R_{opr} = 100$ and starting with one self-replicating program in the core, we are able to develop larger areas in the core where the instructions interact in an interesting way. After some ten thousand core updates we often find stationary "organisms", which primarily consists of SPL-instructions. These programs are typically 20 to 100 instructions long. They are quite robust, because of the many pointers continously being produced in the area. A graphical representation of such a core is shown in figure 5.

It turns out that we need to have an R_{opr} at least of the order hundred, to allow the system to explore evolutionary paths leading to really interesting structures. The small SPL- organisms mentioned above give a glimpse of what will come.

In another simulation it took the same jungle about 100,000 generations, ($\sim$ 18,000,000 instructions) to develop two huge organisms mainly consisting of a mixture of SPL- and MOV-instructions. The core had in this case more than 800 copies of both the SPL- and the MOV-instruction. This means that these two instructions occupied more than one third of the universe. These organisms were able to move in the core, as they copied SPL- and MOV-instructions outside themselvs. On the path which created these SPL-MOV organisms, the change of chemical composition is very obvious. The system here went through something we may characterize as a phase transition in the distribution of different instructions. From a fairly randomized core the system now mainly consists of the SPL- and the MOV-instructions indicating a form of self-organization.

Fig. 5.

This demonstrates an evolutionary path where the system indeed finds a stable area in the rule space. The mature SPL-MOV combination is extremely stable, and any of the

pertubations caused by the copying part of MOV and by the introduction of new pointers in this system will be damped. The MOV- instruction usually copies either a MOV-instruction or a SPL-instruction, guarenteeing the reproduction. The SPL-instruction hands out pointers either to another SPL-instruction or to a MOV-instruction, guarenteeing hereby that the organism is kept alive.

Expanding R_{opr} to the size of the core and using one self-replicating program as initiator, allows the system to evolve away from the initial distribution of instructions in a even more radical way. A simulation with identical initial conditions gave a core almost solely consisting of a mixture of SPL- and MOV-instructions. In this situation we saw a belt of active pointers sweeping through the core, always altering the details of the core, but keeping the macroscopic mixture of the two instructions. Actually this may have been the mature state for the two large SPL-MOV organsims described above. However, we never gave them a chance to develop further.

The pathway leading to the SPL-MOV organism is only one among many possible directions the evolution can take. Another interesting evolutionary product was found after 110,000 generations ($\sim$ 24,000,000 instructions), where 2859 CMP-instructions (out of 3584 possible) were present. In this core, a funny discontinuous motion of small moving and "jumping" programs was seen everywhere (see the core statistics in figure 6(a)). Yet another evolutionary path (182,000 generations $\sim$ 40,000,000 instructions) led the system to develop a core with 1276 SPL-instructions and only 42 MOV-instructions. In this core large areas were boosted with pointers for some time, whereafter the activity died out locally, only to re-emerge at another location. These organisms indeed had an irregular metabolism! Core statistics in figure 6(b).

A jungle which is silent to watch after 100,000 core updates, but still has a potential to develop interesting behaviour, is shown in figure 6(c). With the high number of MOV-instructions we must expect a change at some point.

It turns out that another common phenomenon found in cores with a large operation radius R_{opr}, is the development of an exciting and very complicated transient, which even-

tually dies out and stabilizes after some large number of instructions. The relaxed system has relatively few active pointers, either stationary or moving in a rather trivial pattern. This is probably a phenomenon similar to what is found on the lattice with Conways CA-rule "Life" (Gardner, 1970). In Life the complicated behavior always seems to be transient. In one of the runs where we experienced this behaviour in our system, the transient core at one point even had a relatively high number of active MOV- and SPL-instructions. The core activity eventually collapsed to a silent state where only 78 of the 220 possible pointers were active. The core had at that time, after 145,000 generations ($\sim$ 32,000,000 instructions), only 15 SPL-instructions left in the whole core, which was totaly floted with 1323 ADD-instructions (core statistics in figure 6(d)). A similar transient behaviour resulted in 2654 JMZ-instructions after 30,000,000 instructions. In this core only 60 pointers were active. In these silent cores almost all of the remaining pointers are traped at fixed points, i.e. at some kind of JMP-instruction. Recall the discussion of fixed points for pointers.

The mechanism for this amplification of single instructions is of course the presence of a copying loop. Depending on which instruction the copying is directed at, we see a build up of this particular instruction. The situations where we see a collapse are normally associated with a virtual extinction of the MOV-instruction. This instruction is responsible both for a major part of the fluctuations (instruction-mutations) and the ability to re-arrange programs. Therefore the system is only perturbed by the spontaneous pointers, when the MOV- instructions are virtually extinct.

Fig. 6.

4. EVALUATION OF THE DEVELOPED FUNCTIONAL PROPERTIES

To compare the different structures developed in the VENUS-cores, we have to measure their functional properties in some way. It would be nice to be able to order them on an adaptability scale, or maybe more appropriately, on a scale of distance from non-interesting functional properties. We can adopt the basic intuitive properties a complexity measure must have, and from there try to construct a crude measure. Peter Grassberger (1986, 1988), argues that a complexity measure must favor: (1) a large number of interacting entities, and (2) diversity among the interacting entities. We can add (3) a notion of stability to perturbations among the interacting entities. For a first approximation we can simply, for a given interacting part of the core, take the product of the number of interacting instructions, the number of different instructions minus one, and a brittleness index, to obtain some evaluation number. The brittleness index could be the number of non-fatal perturbations over the number of possible perturbations caused by our noise. By non-fatal perturbations we mean changes that conserve the overall functional properties among the interacting entities. In this way the brittleness index is defined between zero and one. We define our complexity measure, Γ, as

$$\Gamma = \Im_{all}(\Im_{diff} - 1)\,\beta, \tag{4}$$

where $\Im_{all}$ is the number of interacting instructions, $\Im_{diff}$ is the number of different instructions among the interacting instructions and β is the brittleness index. β is defined as

$$\beta = \frac{\mathbf{P}_{\mathbf{non-fatal}}}{\mathbf{P}_{\mathbf{possible}}}\ , \tag{5}$$

where $\mathbf{P}_{\mathbf{non-fatal}}$ is the number of non-fatal perturbations and $\mathbf{P}_{\mathbf{possible}}$ is the total number of possible perturbations within the interacting entities.

Without going into details, let us evaluate some different structures:

(i) The self-propagating instruction MOV($\$$ 0, $\$$ 1) or even a large number of these instructions following one another, as in example (a) in figure 3, will have a measure identical to

zero, due to the fact that it only consists of a single kind of instruction. This holds for any single instruction being able to survive in the core (JMP(# X), JMZ(# X, 0), etc., where X can be any number.

(ii) Also the simple self-replicating program will obtain score very close to zero, due to it's failure to maintain the basic functional properties if subjected to noise. If any of its instructions are changed, or if another pointer is introduced in the program area, the structure is no longer able to replicate. This will be true for almost any human "engineered" organism, with nice, well defined properties.

(iii) The developed MOV-SPL organism is, on the other hand, able to obtain a high score, because none of the factors in the measure become zero. As argued earlier, this structure is very robost to the noise in our system, it can involve a high number of interacting instructions, and it consists of more than one kind of instruction.

Although this simple complexity measure (or measure of structures of interest), surely does not apply directly to every environment, it has the properties such a measure at least must have.

John McCaskill (1989) has suggested another way of evaluating the functional properties of structures developed in such a system. Related to our system one could place the SPL-MOV organism in a new environment controlled by us. An evaluation of how the structure performs in a sequence of such predefined environments could then tell us something about its adaptability or evolutionary sophistication.

5. DISCUSSION

As the evolutionary process proceeds, the local environment changes. The very chemical activity changes the universe, exactly like the chemical activities do in our "real world". However, the cooperative structures we are able to evolve in this system are, in several aspects, different from the living things we know in the "real world". An interesting difference is the way they reproduce and develop. For instance the SPL-MOV organism does not make a true copy of itself. It expands as it moves through the core, and it interacts with whatever it meets on its way. It does not have a well defined geno-type / pheno-type distingtion, like modern life forms have. We may interpretate the specific SPL-instruction(s) and the specific MOV-instruction(s) to be a kind of a gene, because they do not change as the organism grows, whereas the actual mixture of these instructions, i.e. the sequence in which they appear in the organism, may be interpretated as the phenotype. If we want to relate this organism to a "corresponding" organism build from our "real world" chemistry, it would be a cooperative, probably autocatalytic, chemical network which reproduces it's elements through the cooperative chemical reactions. Such a system would also interact with everything it meets in it's environment and it does not either have a clear geno/pheno-type distingtion.

It is interesting to note that the core in the significant parameter regime - the jungle with large R_{opr} - evolves through four successive macroscopic states, which are characterized by very different functional properties. The first state is the randomized core with the initial seed of the self-replicating program. This seed causes the core to be populated with copying loops. Eventually this process is slowed down, because more and more copies of the program loses their ability to replicate in a proper manner. Some of them turns into instruction copying loops similar to the one shown in example (b), Fig 3. As a result of that process more and more of the core is overwritten by the single instructions these loops copy. Eventually this process also saturates as more and more of the loops disappear. The new instruction mixture resulting from these processes may then start actively to develop

and hereby eventually take over the core, as we have seen in several examples. We cannot claim that this is truely open-ended evolution. First of all the evolution process gets trapped after three successions. It apperently reaches some kind of an attractor. Secondly we have to some extent "designed" the first succession. However, one has to conclude that the system is autonomously able to evolve through more than two macroscopic states, characterized by very different functional proprties.

The evolutionary succession process is summerized in Fig. 7.

Fig. 7.

By introducing this simplified artificial chemistry it does not at all seems as if the system has lost the fascinating property of being able to self-organize interesting structures. VENUS is able to develop very interesting cooperative programs. Let us now try to sharpen some of our tentative conclusions from the evolutionary experiments given in section 3, and relate them to what we know about biological evolution. We may even from these simple experiments be able to give some hints towards what was important in the origin of "real" life.

(i) Local fluctuations in the chemical composition seems to be of major importance to the evolution of interesting structures. Not any chemical environment is able to develop life, although the fundamental chemical laws are the same everywhere.

(ii) A certain flow of energy and resources are important.

(iii) Life probably did not emerge in the deserts. The deserts were presumeably populated by organisms originated elsewhere.

(iv) Optimal environmental conditions are not enough to ensure the evolution of cooperative structures. Chance plays a central role in the outcome of a particular process.

(v) Cooperative structures seems to be necessary for the evolution of anything of viable

nature. In real world this may corrospond to autocatalytic structures.

(vi) It seems easier to evolve a "metabolic networks" than a clean "genetic" system.

(vii) Even with a very brittle computational chemistry it is possible to evolve higly stable and viable organisms.

(viii) It may take a chemistry a long time to create an environment in which the right subset of chemical rules are active. To find the "right" area in rule space may be the crusial problem for any evolutionary process.

Obviously our simulations also relate to the concept: computer virus. Computer virus as they are known today, are human engineered "smart" programs, which either via the network or via human transported disks are able to invade or infect other computers and hereby via a replication process occupy all the available memory and all the available computational resources. Up till now, we have not seen any computer virus capable of adaptation to totally new computer environments by mutations or some other kind of real learning. All the properties such a virus has are given to it by it's creator. The environment the computer virus program "lives" in, the computer memory, is strictly controlled to be noiseless, in order to facilitate a precise execution of any assembler program the programmer wants executed. In this respect a "real" computer core is different from our VENUS core. However, the competition for computational resources between any illegal computer virus program in the core and any of the legal user's program is very similar to the the competition we see between different programs in our core.

From any computer owners point of view it would be very desirable to be able to kill any non-authorized pointer executing in his computer. However, it is very difficult *a priori* to distinguish between legal and illegal processes. Put in an other way: it is very difficult to build an immunesystem for a computer, because such a system should be able to discriminate itself from what is comming in from outside, and then after the discrimation be able kill the intruders. How this actually is going on in modern biological systems is not yet known in detail (Perelson, 1988). Besides the ideas of an internally encrypting

of all the legal processes in a computer, and in this way be able to discriminate illegal processes by the leak of the right encrypting, it may actually be possible to distinguish between wanted and non-wanted processes in the computer memory by looking at the qualitatively different trajectories a virus program and a "normal" program has in the core. By graphically representing the core on the screen like we do in VENUS, it should be possible via field studies to learn the patterns of "good" and "bad" programs. However, we don't belive that there are any short cuts in this game.

From now on, as in biology, it will presumeably allways be an arms race between the computers immunesystem and parasites wanting to get access to the computional resources "inside".

Returning to our original ideas of creating a new simple chemistry and showing that this chemistry is sufficient to create cooperative computation, it seems so far as if the evolutionary process in VENUS saturated at some point. One of the major problems with this version of our simulator is, as far as we see it, the brittleness of the more sophisticated structures. The assembler-automaton simulator inherited some of the unfortunate properties of its ancesters: Both the assembler language and the simple cellular automaton suffers from computational brittleness. Due to this brittleness and the very small instruction set, compared to the "real" chemistry, the system is locked when it has found a stable area in the rule space. In this situation the system has too few basic instructions which, if introduced, will maintain the stability of the system. Therefore more diverse structures cannot develop. However, the brittleness of more sophisticated structures is not a problem peculiar to our assembler-automaton system. If we imbed any modern biochemical pathway in a random chemical environment it will surely collaps. In the creation of life, it presumeably took evolution a major part of its effort to "find" or create a stable chemical sub-space within which sophisticated chemical reaction networks could emerge.

The system also suffers from the fact that it only has one dimension. Two different organisms are not able to "pass" each other in this world. When they meet, they immediately interact, changing both of them. Everything that meets has sex! In such a

universe it is difficult to distinguish between an organism and its environment or between two different organisms. However, these problems are a consequense of one of the very attractive simplistic properties of this universe. The very sequence of instructions - i.e. the interaction rules - also constitute the single geometrical dimension of this system.

From the very beginning of this project, it was clear that we wanted to call our simulator VENUS, in response to the "core wars " system, which was called MARS (*M*odular *A*rray *R*edcode *S*imulator). This was due to the fact that we wanted to play a very different game from the "core wars" game, where the main purpose is to kill one another. Our system should be able to create new properties. However, to find a proper name VENUS could be an abbreviation of, turned out to be one of the major difficulties associated with the project. We ended on *V*irtual *E*volution in a *N*on-deterministic *U*niverse *S*imulator. The next generation of simulator, which we are working on now, has more than one dimension and its basic chemistry need not be of the assembler-automaton type. In this system one can define one's own favorite artificial chemistry. It is obvious that we have to name this system EARTH, as an early point suggested by Doyne Farmer. Fortunately, the suggestment also included an interpretation of the abbreviation EARTH: *E*volutionary *A*dvantageous *R*egion of *T*hermodynamical *H*eterogenity. The near future will tell us if it is able to live up to its name!

6. CONCLUSION

We shall not try to judge whether the cooperative structures we have evolved in VENUS are alive or not. However, it seems clear that such a simple universe is a good vehicle for studying fundamental properties of emergent computation, evolution and artificial life. Despite the brittleness of the individual instructions, our system is indeed able to evolve stable cooperative programs. The dynamics of this simple system has many properties similar to real evolution. Through four successive macroscopic core epochs, the system is able to develop very life-like behaviors, although they are different from modern biological life forms. The interplay between chance and necessity changes significantly for different parameters in our chemistry. In some parameter regimes the evolutionary path seems quite deterministic, whereas other regimes support multiple coexisting attractors with what we may characterize as fractal borders between basins of attraction. These, together with other properties of the evolutionary processes in VENUS are used to discuss the properties of biological evolution. We have also developed a very crude complexity measure with which we are able to evaluate the different functional properties evolved in VENUS. Finally the approach we have used to understand the emergent computational properties in our core simulator is discussed in relation to computer virus. This approch may also be appropriate in the construction of an immunesystem for a computer.

ACKNOWLEDGMENTS

We would like to thank the Core Wars people present at the first Artificial Life Conference, September 1987, at the Center for Nonlinear Studies, Los Alamos National Laboratory, from whom we got the original idea of using a simulated computer core as our universe. We are greatful to Doyne Farmer and Chris Langton with whom we have discussed some of the later parts of the development of VENUS and the interpretation of some of our results, and who also have critizised earlier versions of this paper. Peter Grassberger is acknowledged for a number of discussions on how to measure complexity, and John McCaskill is acknowledged for his discussion on the evaluation of functional properties in artificial chemistries. Finally Y.C. Lee is acknowledged for discussions on stable subsets in Turing Machines and computer virus.

REFERENCES

1 R. Collins, D. Jefferson, and C. Taylor, are at present developing code (on the Connection Machine) for evolving finite state machines and neural networks on a grid, 1989.

2 R. Dawkins, 'The Evolution of Evolvability", in Artificial Life, SFI Studies in the Sciences of Complexity, Vol. III, ed. C. Langton, Addison-Wesley, 201-220, 1989.

3 A. Dewdney, "In the game called Core War hostile programs engage in the battle of bits", Sci. Amc., May 1984.

4 A. Dewdney, "A program called MICE nibbles its way to victory at the first Core War tournament", Sci. Amc., January 1987.

5 M. Eigen, "Self-Organization of Matter and Evolution of Biological Macromolecules," Naturwissenschaften **58**, 465, 1971.

6 M.Eigen and P.Schuster, "The Hypercycle - A Principle of Natural Self-Organization", Springer-Verlag, Heidelberg, 1979.

7 J. D. Farmer and N. H. Packard, "Evolution, Gamcs, and Learning: Models for Adaptation in Machines and Nature", Physica **22** D, vii-xii, 1986.

8 J. D. Farmer, S. A. Kauffman, and N. H. Packard, "Autocatalytic Replication of Polymers," Physica **22** D, 50, 1986.

9 M. Gardner, "The Fantastic Combinations of John Conways New Solitaire Game "Life"", Scientific American 223, 120-123, 1970.

10 P. Grassberger, "Toward a Quantitative Theory of Self-Generated Complexity", Int. J. Theoret. Phys. **25**, 907, 1986.

11 P. Grassberger, "Complexity and Forcasting in Dynamical Systems", preprint, 1988.

12 Int. Core Wars Soc., 1986, Note on the Core Wars simulator, 8619 Wi-sall, Wichita, Kansas 67210-1934, USA.

13 C. Langton, "Studying Artificial Life with Cellular Automata", Physica **22** D, 120-149, 1986.

14 C. Langton, "Artificial Life", in Artificial Life, SFI Studies in the Sciences of Complexity, Vol. VI, ed. C. Langton, Addison-Wesley, 1-47, 1989,

15 S. Kauffman, "Autocatalytic Sets of Proteins," J. Theor. Bio. **119**, 1-24 (1986).

16 J. McCaskill, "A Minimal Integreted Recognition-Processing Model for Macromolecular Evolution", preprint 1988.

17 N. Packard, "Intrinsic Adaptation in a Simple Model for Evolution" in Artificial Life, SFI Studies in the Sciences of Complexity, Vol. VI, ed. C. Langton, Addison-Wesley, 141-155, 1989, and some later developments of the code on "Evolving Bugs in an Artificial Ecology".

18 A. Perelson, "Theoretical Immunology, I and II", SFI Studies in the Sciences of Complexity, Vol. II and III, ed. A. Perelson, Addison-Wesley, 1988.

19 S. Rasmussen, "Aspects of Instabilities and Self-Organizing Processes," (in Danish), Ph.D. thesis, Physics Laboratory III, The Technical University of Denmark, 1985.

20 S. Rasmussen, "Towards a Quantitative Theory of the Origin of Life", in Artificial Life, SFI Studies in the Sciences of Complexity, Vol. VI, ed. C. Langton, Addison-Wesley, 79-104, 1989.

21 S. Rasmussen, B. Bollobás, and E. Mosekilde, "Elements of a Quantitative Theory of Prebiotic Evolution", submitted to J. Theor. Bio.

FIGURE CAPTIONS

Fig. 1. The predefined model can only tell somthing about one out of a possibly infinite number of alternative evolutionary pathways. In a more open-ended approach the model is able to find its own evolutionary path based on a set of low level "chemical rules" and the initial conditions. Note that such a model has many evolutionary paths and more than one attracting area. In general, one can define an artificial and very simple possibility space W_i. This is what we have done, hoping that such a model can tell us something about the emergence and evolution of complex functional properties.

Fig. 2. The central part of VENUS is the core. The core has 3584 addresses and each address contains one word. The core is cyclic modulus the core size. Each word consists of an instruction and up to two operands, A and B. At the magnified core area also the resource radius R_{res} and the operation radius R_{opr} is shown. The queue of execution pointers is symbolized by L. Note that one of the execution pointers is pointing at the address where the JMN(@ 5, $ 10)-instruction is located.

Fig. 3. Examples showing (a) a simple self-propagating instruction, and (b) a simple loop structure. All the addressing is relative to the executing instruction. Note that a loop structure like this is very powerful in multiplying any instruction. Similar loops are responsible for the phase transitions we sometimes meet in the distribution of instructions in the core.

Fig. 4. The history of a specific core, a jungle with $R_{opr} = 20$, told by its distribution of instructions at two different times, (a) at generation 0 and (b) at generation 72,700. The histogram shows the frequency of each instruction as it is found alone, as it is found in pairs, tripples, etc., up to 14-tupples. Higher order tupples may occur but are not shown. The total number of instructions in the core is 3584. The scale of the box is 325 instructions. The macroscopic composition of such a system only changes

very little over many generations. However, several large programs kept alive by SPL-instructions together with a number of complicated loops are found in this garden. This shows that significant changes has occured at the micro level.

Fig. 5. (a) Screen dump of a core evolved under jungle conditions after 5,000 generations. Each different instruction is shown with a different color, following the color code given in the bottom of the picture. The word at address 0 is shown in the upper left corner, words are shown with addresses increasing horizontally towards rigth. The last word in row 1 has the address 127, the first word in row 2 has the address 128 etc. The word in the lower right corner is associated with the address 3583. The colored squares indicate that a change has occured at that address since the last time the screen was cleared here approximately 100 generations. Some of the squares has a white underscore, which indicates the prescence of a pointer at that address, awaiting execution. An important characteristica of the core at this evolutionary stage is the many sequences of identical instructions. This indicates that the system is somewhere on the transition between the second and third macroscopic level as discussed in connection with figure7.
(b) shows a screen dump of a time trace (CA-view) of a little part of the same core at a later stage in the evolution (130,000 generations). Here we can see the time trace of the specific changes. Times progresses downward from the top and is cyclic. Note the group of pointers moving to the right towards increasing addresses. Also note the large number of identical SPL-instructions as can be seen in the more details in the lower right corner. Here the instructions associated with the thick white underscore in the CA view is shown in disassembled form. In the right corner we see part of a SPL organism, see also text. The core is now in the fourth evolutionary stage (see also figure 7.).

Fig. 6. Histograms telling how the frequency of the different instructions in the core has changed after one hundred thousand generations. All cores shown originated from

jungles with R_{opr} = coresize. They are all on the same scale; box corrosponding to 325 instructions. Compare with the randomized core in figure 4. In (a) 2859 CMP-instructions are developed after 110,000 generations. In this core, a funny discontinuous motion of small moving and "jumping" programs was seen everywhere. In (b) the jungle developed 1276 SPL-instructions after 182,000 core updates. In this core large areas were boosted with pointers for some time, whereafter the activity died out locally, only to re-emerge at another location. (c) shows a core with 1471 MOV-instructions. With that many MOV-instructions one must expect that the core still is developing after 100,000 generations. Not all pointers were active in this core (204 out of 220 possible). In (d) the core is floated with 1323 ADD-instructions after 145,000 generations. The current activity in this core is very low, only 78 of the 220 possible pointers were present. However, earlier this core had a very active period with all 220 pointers active. This kind of transient behaviour is further discussed in the text.

Fig. 7. A step towards open-ended evolution. Our core is able to evolve through four functionally very different macroscopic states, or epochs.

Table 1. The redcode instructions and their addressing modes.

Table 2. Parameters defining a desert and a jungle.

APPENDIX

Listing of the self-replicating program MICE, originally written by Chip Wendell, (Dewdney, 1987).

```
    DAT #7
    MOV #7 , $-1
    MOV @-2 , <5
    DJN $-1 , $-3
    SPL @3
    ADD #417 , 32
    JMZ $-5 , $-6
    DAT #714
```

Fig. 1

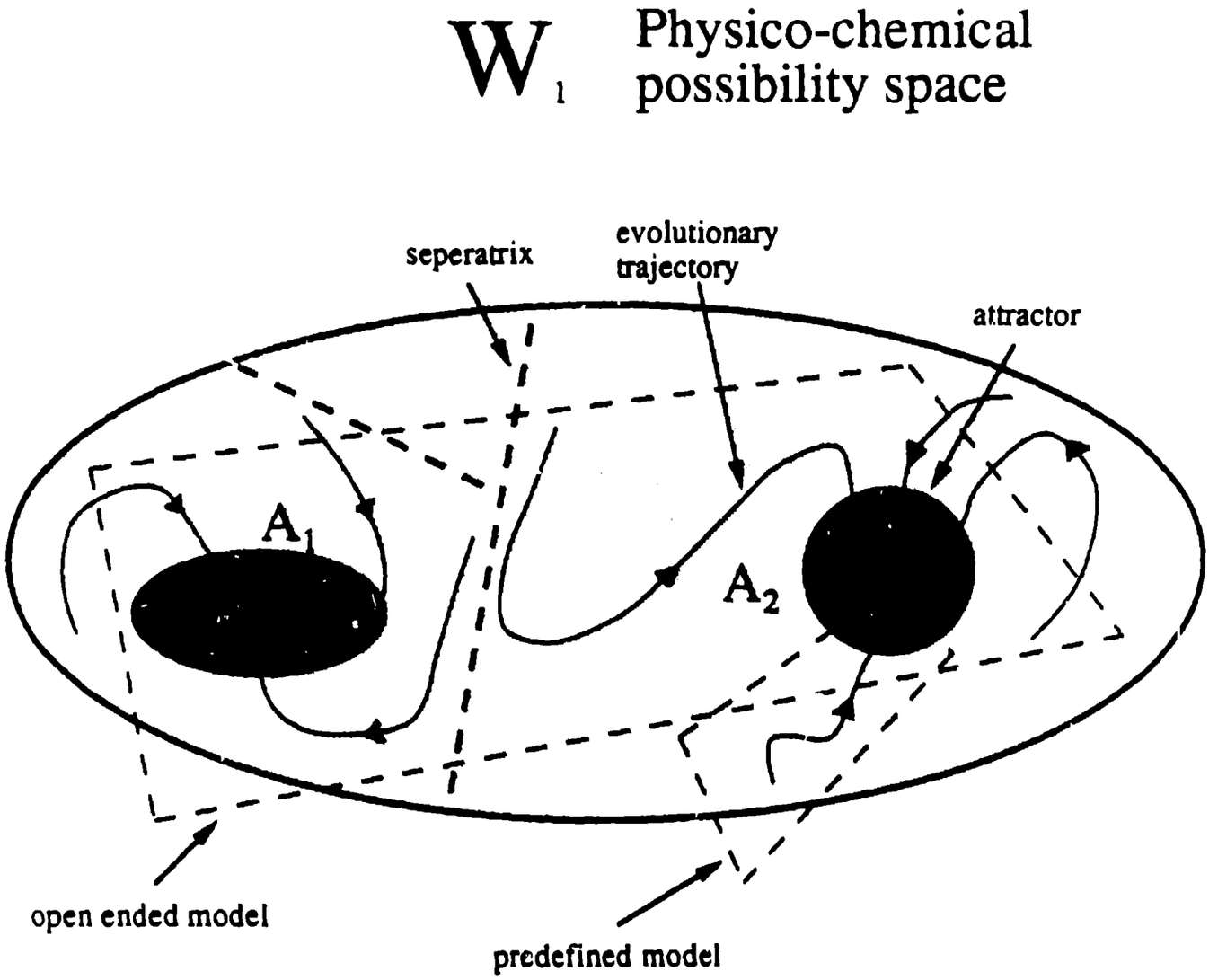
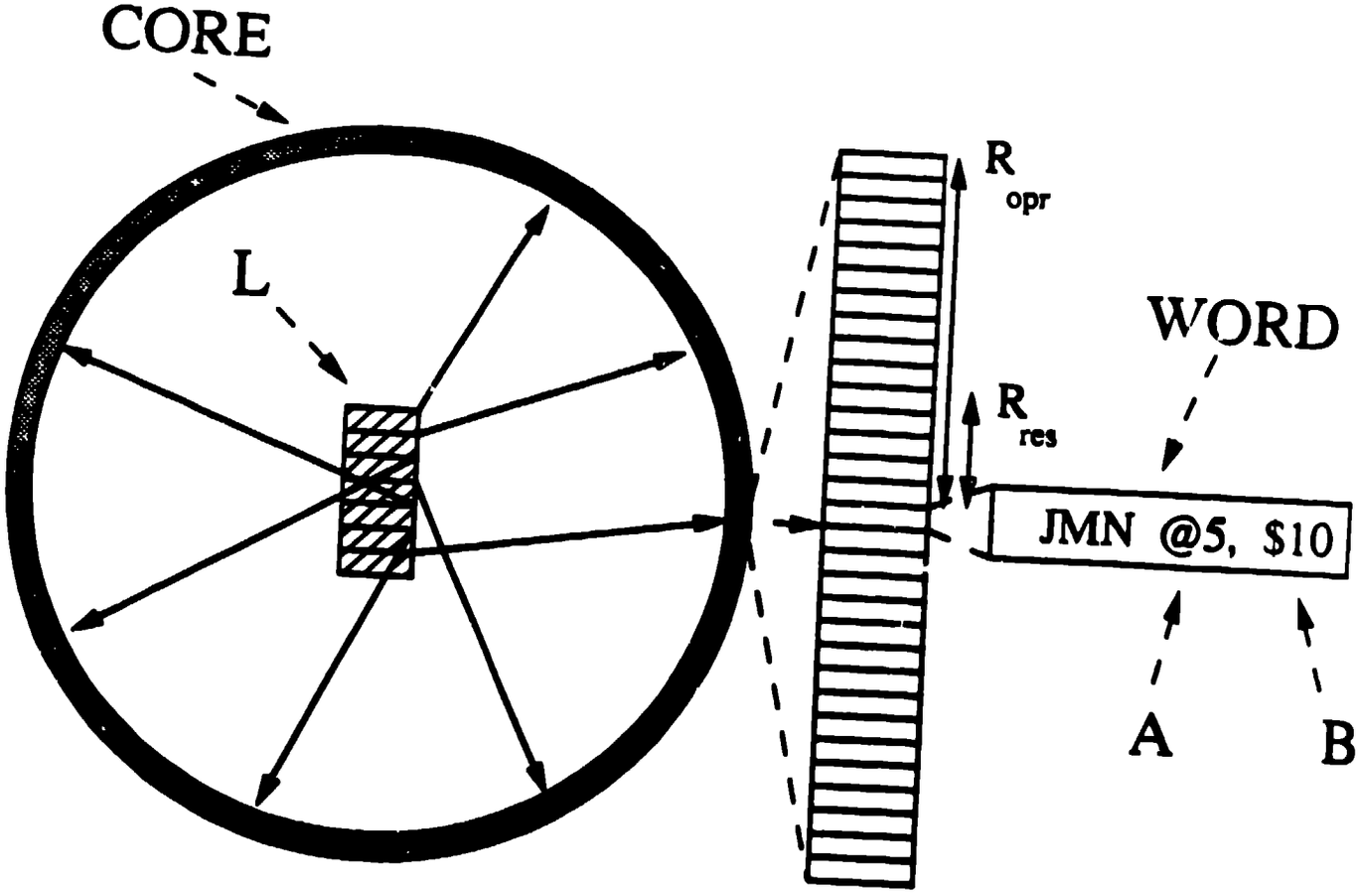
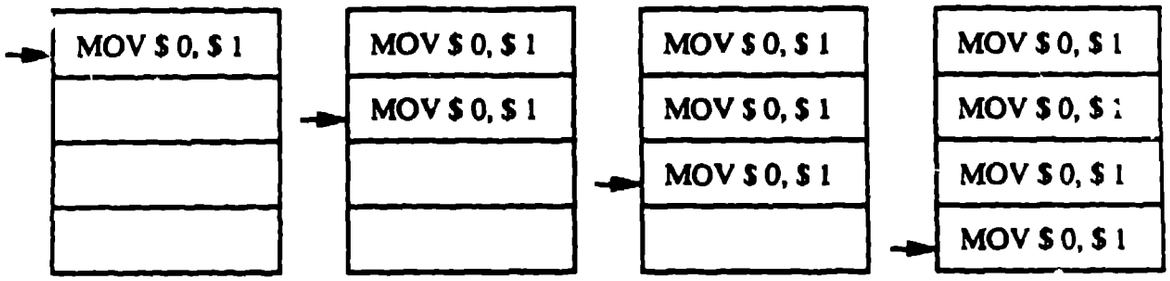

W1
Physico-chemical
possibility space
seperatrix
evolutionary
trajectory
attractor
A1
A2
open ended model
predefined model

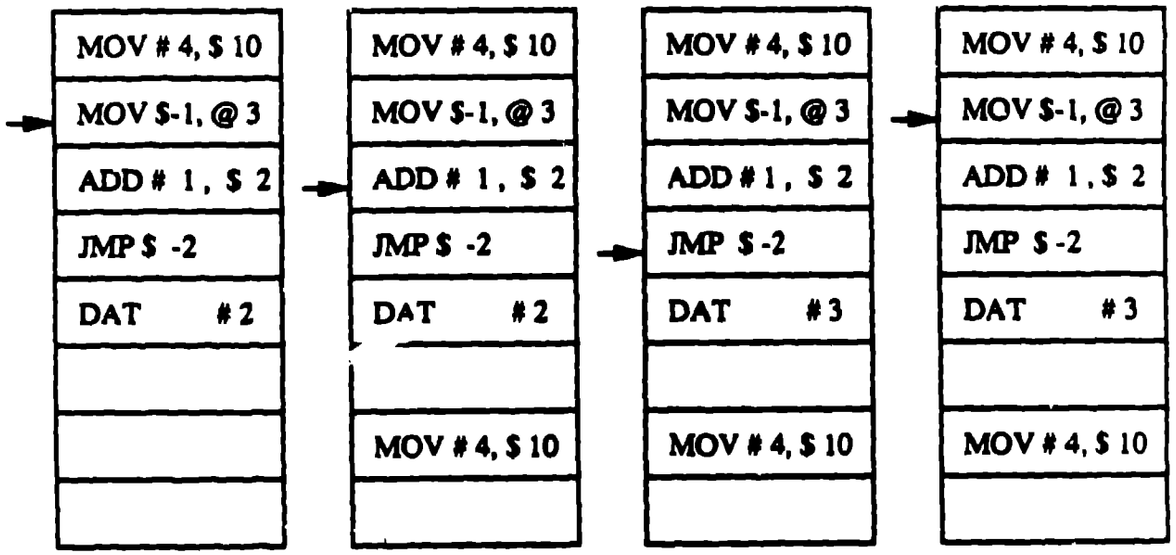
Fig. 2



(a)



(b)



time = 1 time = 2 time = 3 time = 4

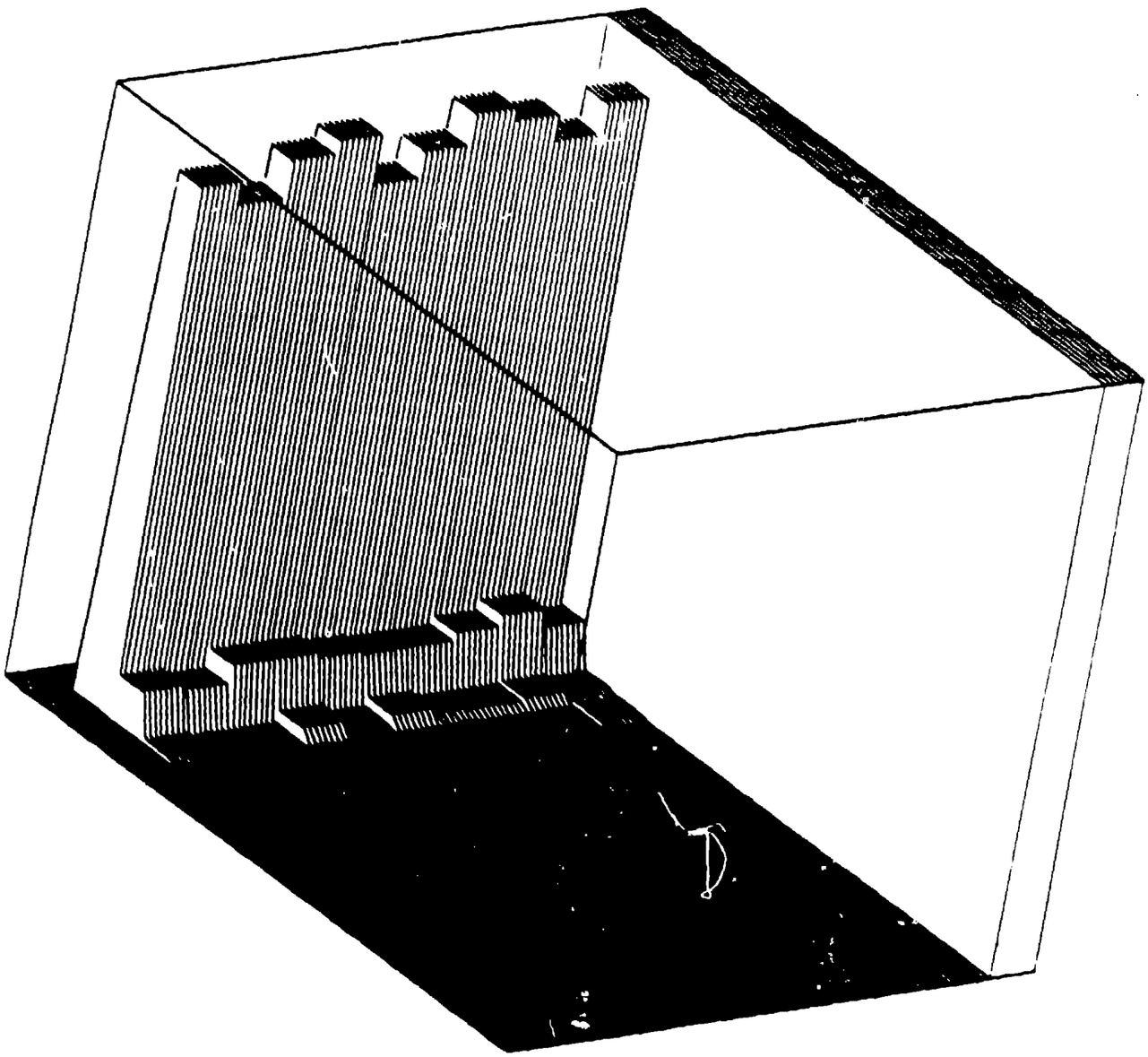


Fig 4(6)

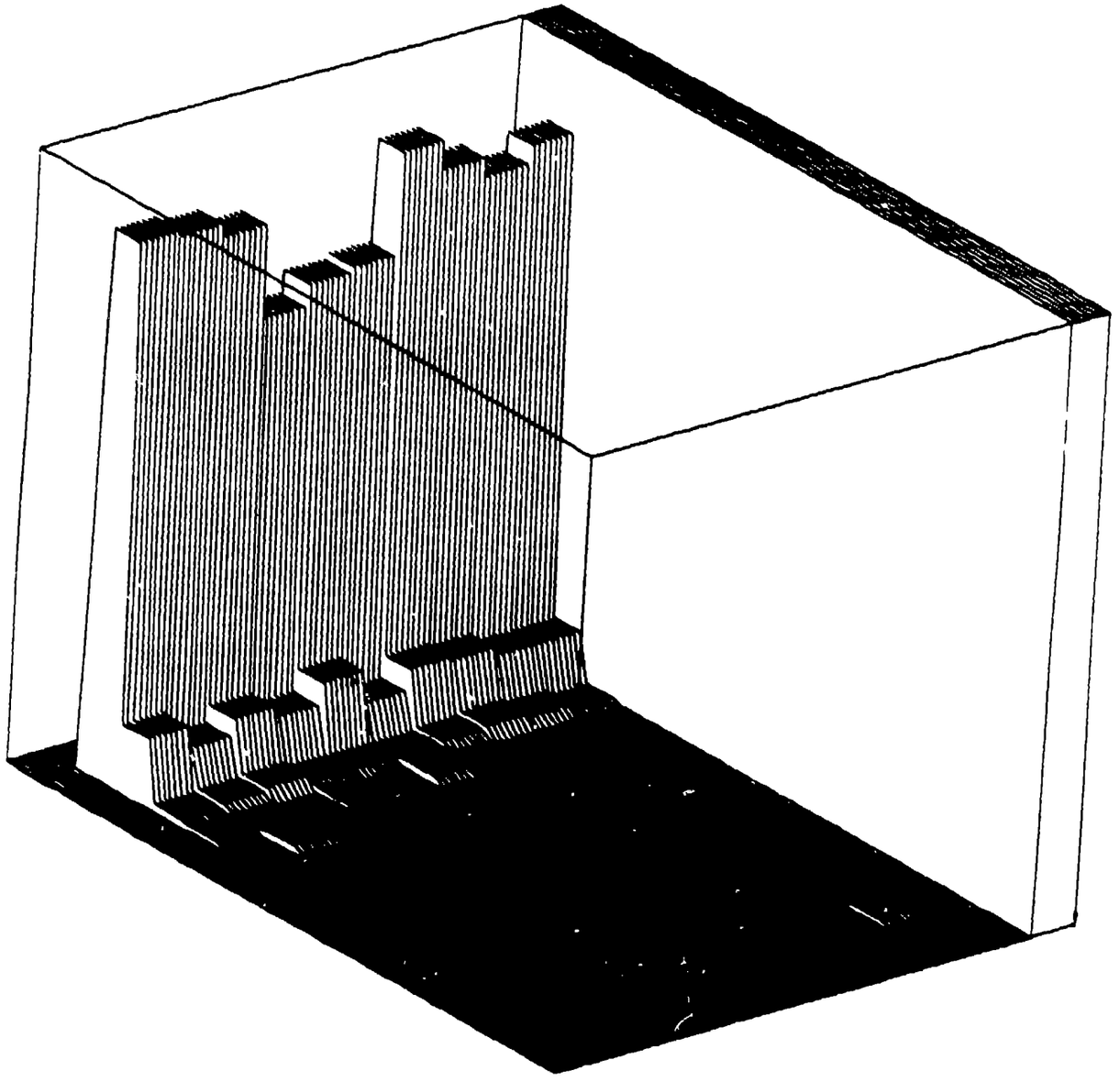
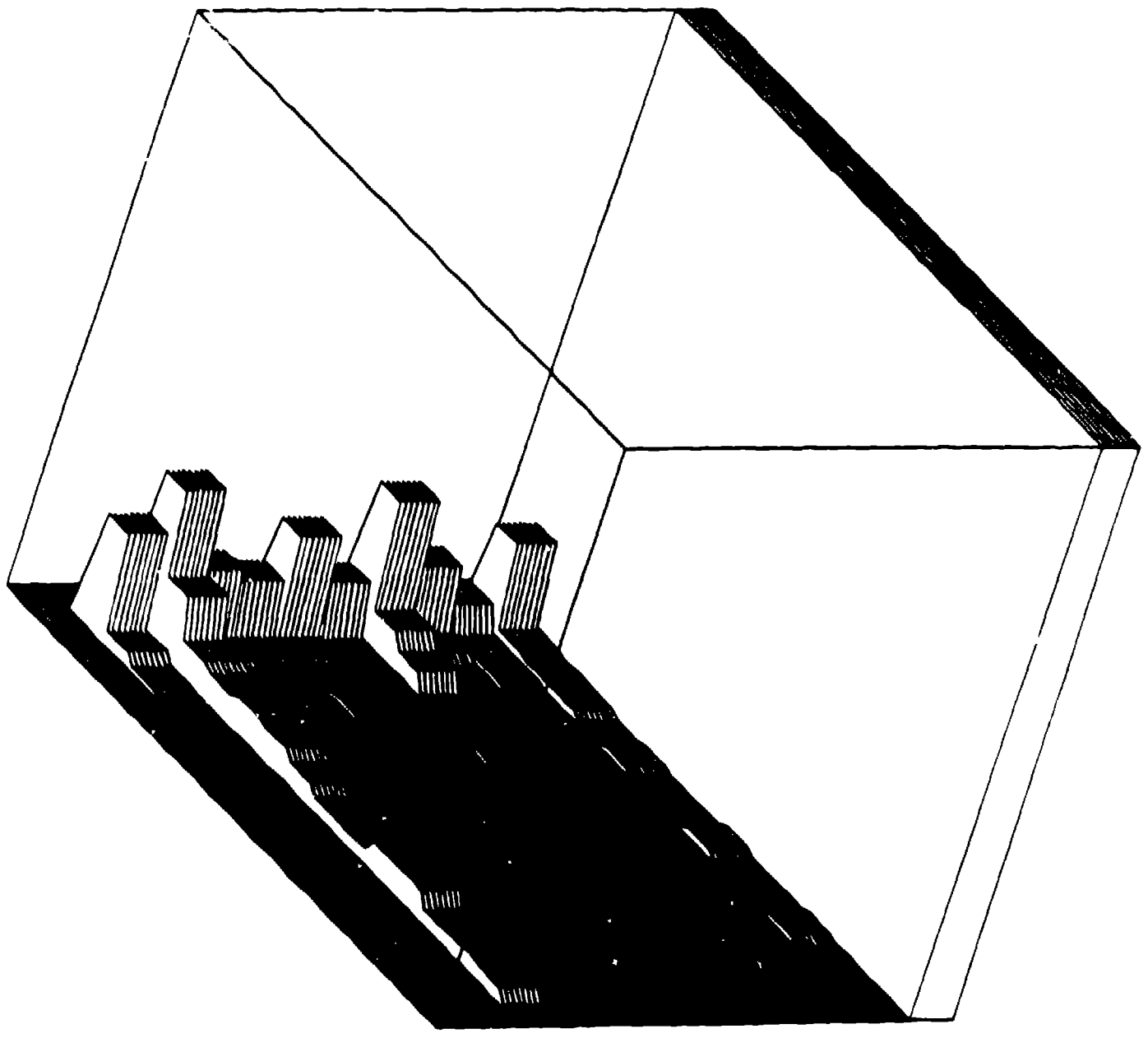
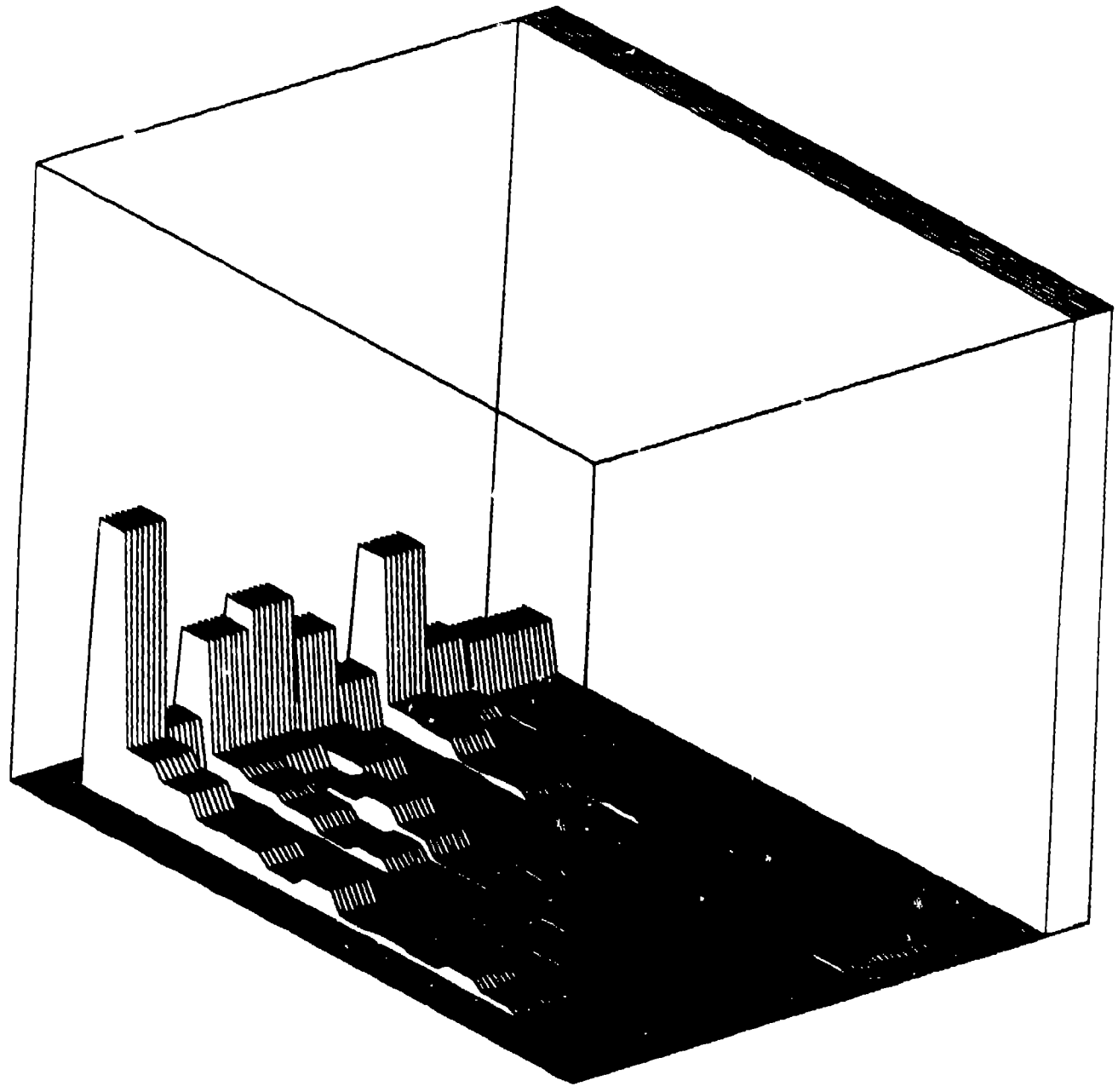


Figure 1

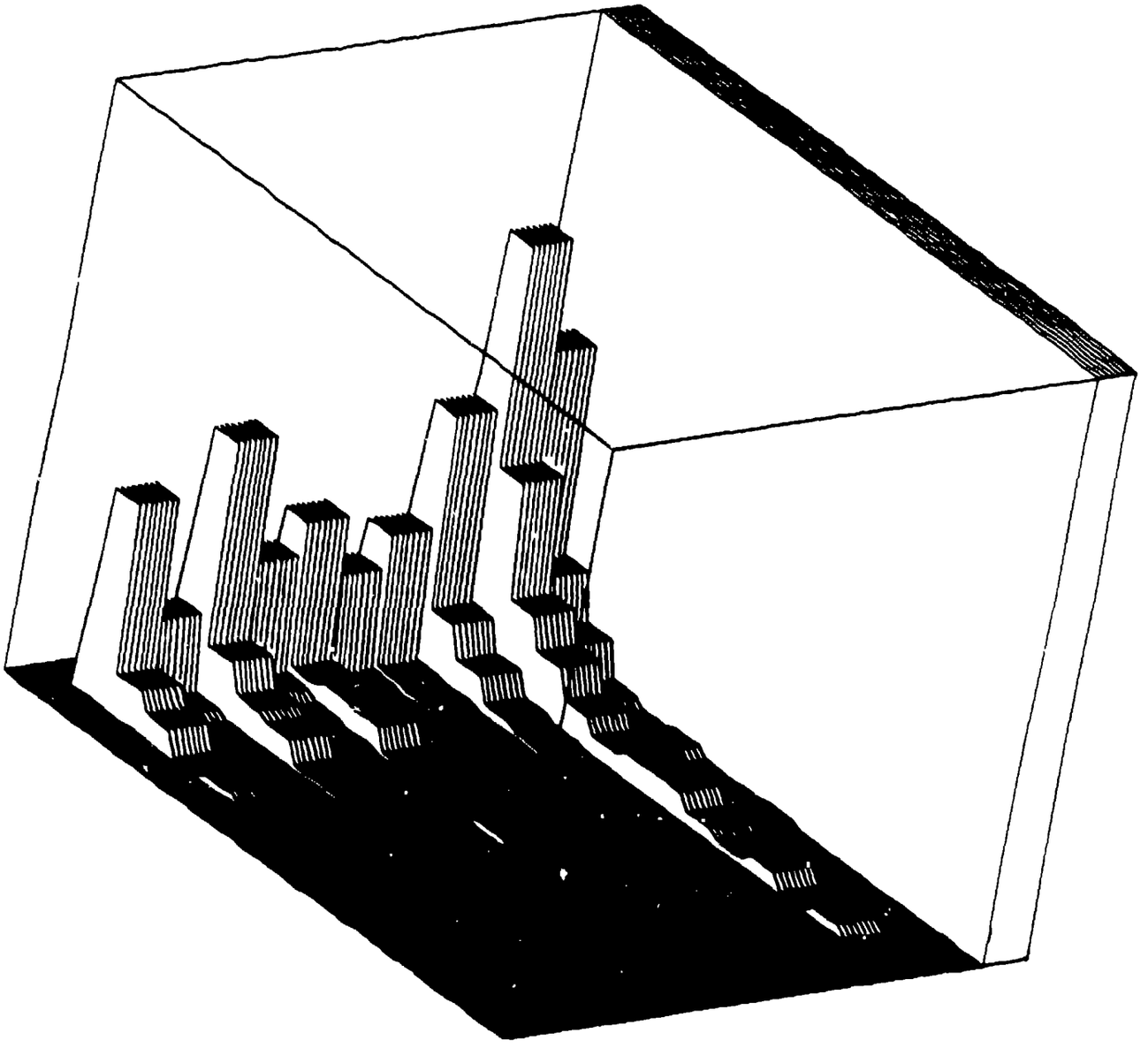


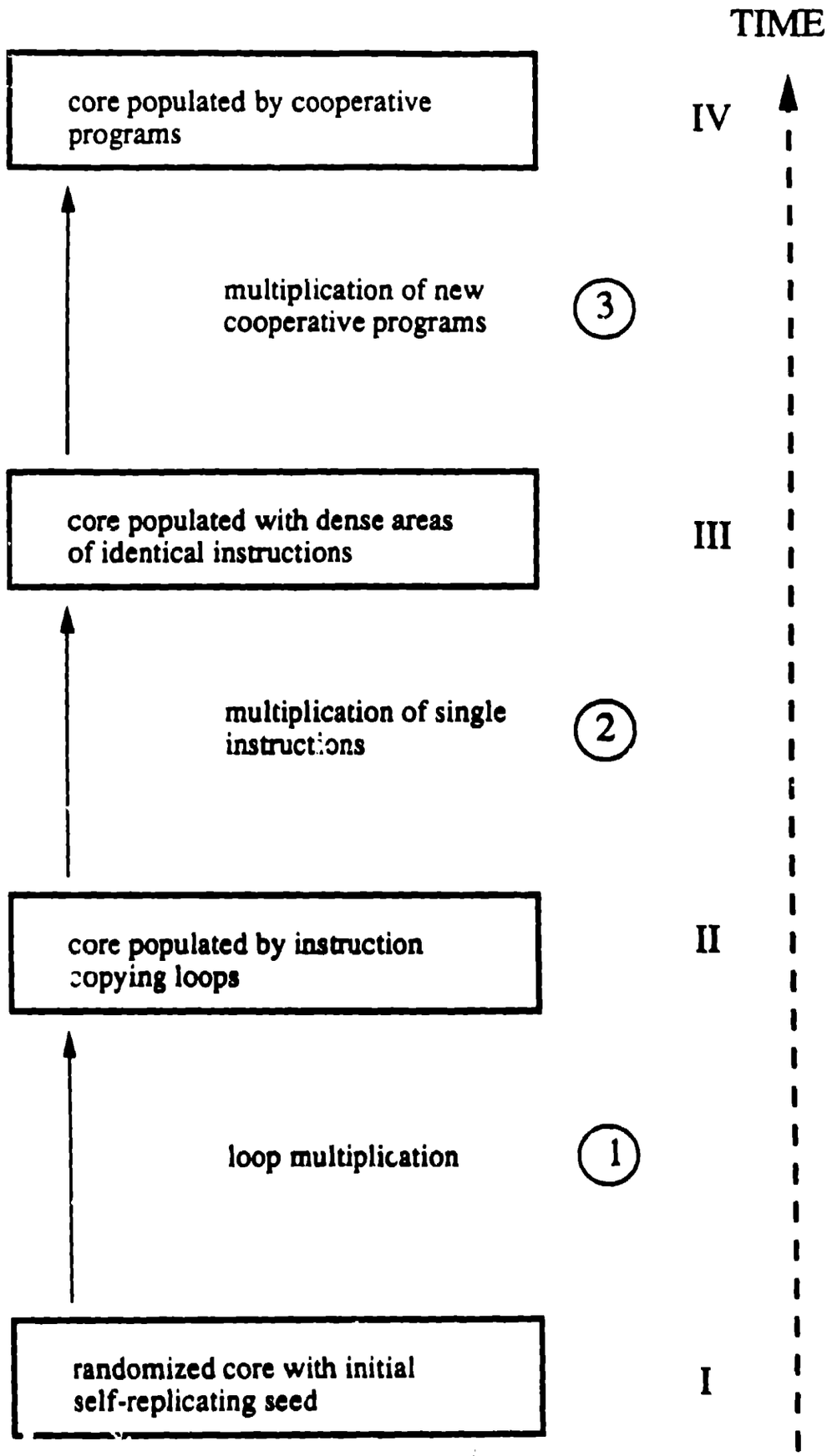
TJ 0161

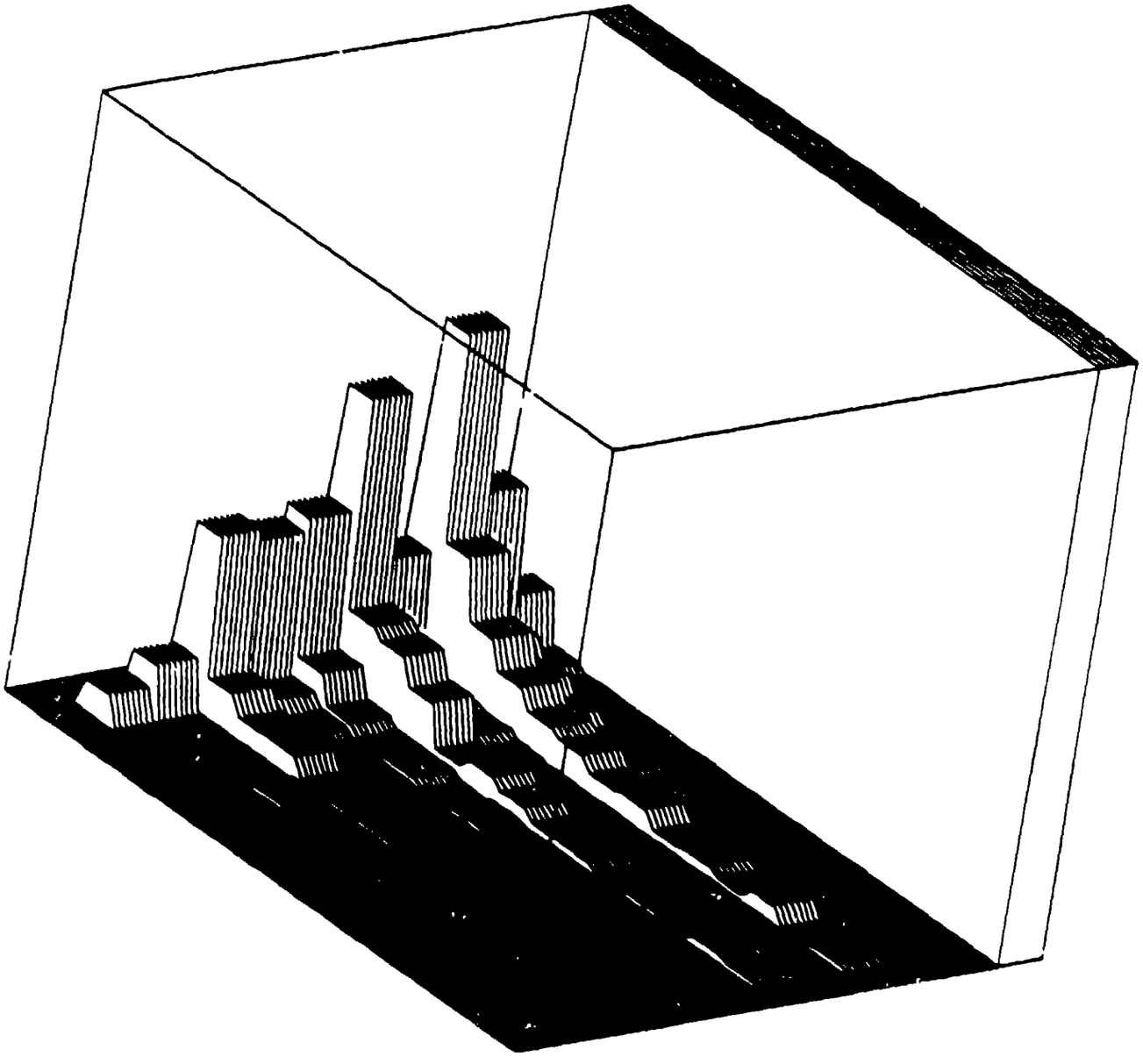


F

Fig 6(c)




TIME
core populated by cooperative programs
IV
multiplication of new cooperative programs
3
core populated with dense areas of identical instructions
III
multiplication of single instructions
2
core populated by instruction copying loops
II
loop multiplication
1
randomized core with initial self-replicating seed
I



Parameters defining a desert and a jungle:

desert	jungle
$R_{res} = 5{,}00;$	$R_{res} = 3{,}00;$
$\Delta r = 0{,}10;$	$\Delta r = 0{,}50;$
$r_{max} = 0{,}50;$	$r_{max} = 0{,}50;$
$P_{mut} = 0{,}05;$	$P_{mut} = 0{,}05;$
$P_{point} = 0{,}05;$	$P_{point} = 0{,}05;$

Tabelle 1

Red-code words

DAT B	non-executable statment reserves space for data
JMP A	transfer program pointer to A
JMZ A, B	transfer program pointer to A if B equals 0
JMN A, B	transfer program pointer to A if B differs from 0
DJN A, B	decrement B and execute JMN A, B
ADD A, B	add the content of A to B and put result in B
SUB A, B	subtract the content of A from B and put result in B
MOV A, B	move the content of A to B
CMP A, B	compare A and B and skip next statement if unequal
SPL B	split execution between B and next statement

Statement Operands:

Each statement operand consist of an addressing mode and a value. The actual contents of A and B depends upon the addressing mode used.

Addressing Modes:

immediate

the actual operand is the value

$ direct

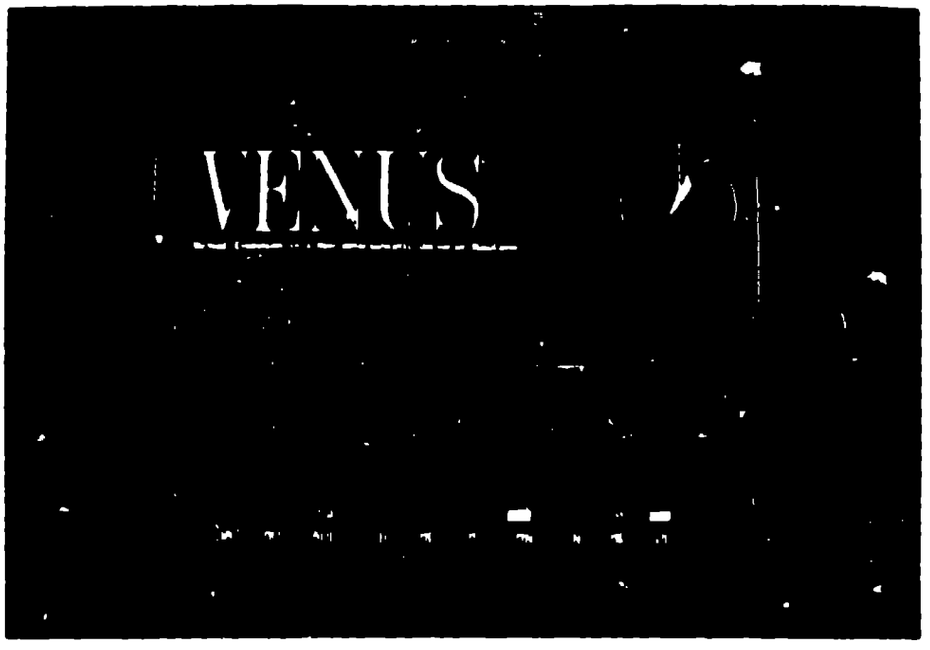
the actual operand is the statment pointed to by the value

@ indirect

the actual operand is the statement pointed to by the value pointed to by the value

< auto-decrement indirect

the actual operand is the statment pointed to by the decremented value pointed to by the value



(a)

(b)