A Three-Dimensional Environment for Self-Reproducing Programs

Marc Ebner

Universität Würzburg, Lehrstuhl für Informatik II,
Am Hubland, 97074 Würzburg, Germany
ebner@informatik.uni-wuerzburg.de
http://www2.informatik.uni-wuerzburg.de/staff/ebner/welcome.html

Abstract Experimental results with a three-dimensional environment for self-reproducing programs are presented. The environment consists of a cube of virtual CPUs each capable of running a single process. Each process has access to the memory of 7 CPUs, to its own as well as to the memory of 6 neighboring CPUs. Each CPU has a particular orientation which may be changed using special opcodes of the machine language. An additional opcode may be used to move the CPU. We have used a standard machine language with two operands. Constants are coded in a separate section of each command and a special mutation operator is used to ensure strong causality. This type of environment sets itself apart from other types of environments in the use of redundant mappings. Individuals have read as well as write access to neighboring CPUs and reproduce by copying their genetic material. They need to move through space in order to spawn new individuals and avoid overwriting their own offspring. After a short time all CPUs are filled by self-reproducing individuals and competition between individuals sets in which results in an increased rate of speciation.

1 Introduction

Evolution of computer programs, Genetic Programming [17,18,2], is usually a very difficult task because small changes to the program's code are often lethal. Changing a single byte in any large application program is very likely to cause such a severe error that the mutated program no longer performs its intended function. This violates the principle of strong causality [31] which states that small changes to the genotype should have a small effect on the phenotype. In essence, we do not want an entirely random fitness landscape.

Nature's search space is highly redundant [15]. This redundancy is caused in part by a redundancy in the genetic code. In addition, different phenotypes may perform the same function (e.g. different sequences code for the same shape and are able to perform the same enzymatic function because local shape space is only finite). Redundant mappings induce so called neutral networks [14,37]. Neutral networks are genotypes mapping to the same phenotype which are connected via point mutations. It has been argued (see Ebner [9] and Shipman [33]) that mappings with similar characteristics like nature's search space should also be beneficial for an artificial search technique, i.e. a genetic algorithm [13,11,20]. Redundant mappings for artificial evolution were investigated by Shipman et al. [34] and Shackleton et al. [32]. We have shown previously that redundant mappings which possess highly intertwined neutral networks increases the evolvability of a population of bit strings [10]. The extent of the neutral networks affects the interconnectivity of the search space and thereby affects evolvability.

In this paper we propose the use of highly redundant mappings for self-reproducing programs. The redundancy in the genotype-phenotype mapping creates the same robustness as it is present in the genetic code. A single exchange of a base-pair in a DNA sequence does not necessarily result in a different sequence of amino acids because the mapping from codon to amino acid is redundant. Therefore, many mutations in our environment are neutral. According to the neutral theory of evolution [16] a large fraction of all mutations in nature are neutral and only a small fraction of the mutations that actually have an effect on the phenotype are beneficial. If the same mechanism is introduced into artificial evolution robustness of programs is increased. A single mutation is no longer lethal for the individual.

We have devised a virtual environment for self-reproducing programs which uses redundant mappings to decode its machine language. Instead of using a highly simplified machine language (such as the Tierran language by Ray [27] which does not use operands), we are using a rather complex machine language were the opcode and two operands are stored in successive memory locations. Constants are also used in our machine language. We integrated them into the instructions and changed the mutation operator instead such that strong causality is preserved. It has been argued that the commands in the Tierra language are like the the amino acids which are used by nature to construct a protein out of the DNA sequence [27]. During a two stage process called transcription and translation, the sequence of base pairs on the DNA are converted into a string of amino acids which fold into a protein. Different protein can fulfill different functions. It is the three-dimensional structure of a protein which is responsible for producing a particular function. In nature, different types of protein may perform the same function if they have the same local structure. Therefore, instead of equating instructions of an assembly language with amino acids we equate instructions with proteins and our processes with cells. Analogies between natural and artificial life are discussed by Davidge [3].

2 The virtual environment

Several types of virtual environments for self-replicating programs have already been developed (e.g. Core War [4,5,6,7], Coreworld [24], Luna [23], the Computer Zoo [36], Tierra [26,25,27,28], Network Tierra [29,30], Avida [1] or CoreSys [8]). Spontaneous emergence of self-replication has been investigated by Koza [18] and Pargellis [22,21]. An overview about artificial self-replicating structures is given by Sipper [35]. Our environment consists of a cube of processors. At any

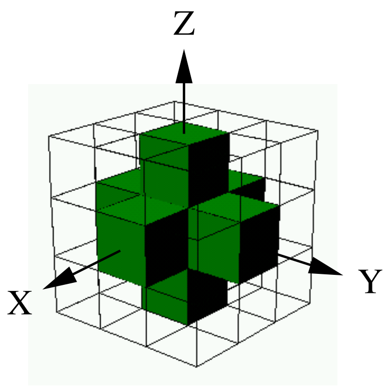

Figure 1. Neighborhood of CPU.

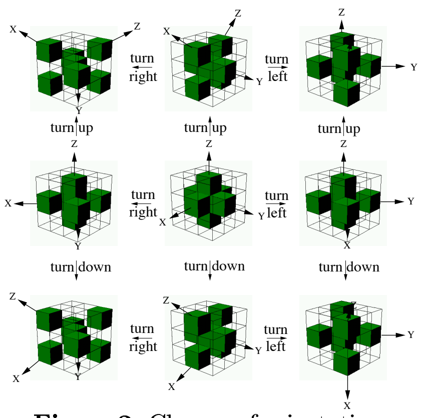

Figure 2. Change of orientation.

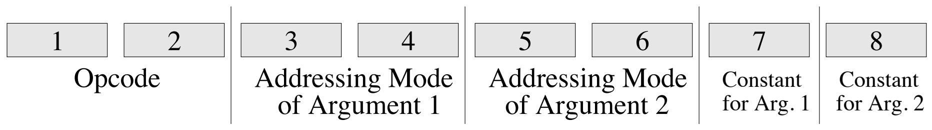

Figure 3. 8 byte structure of a single instruction.

point in time at most one process may be running on a CPU. Thus, each CPU can carry at most one individual. The instructions of each individual are stored inside the CPU's memory. Each CPU has 5 registers, a stack with 10 elements, a stack pointer marking the top of the stack, a program counter, a zero flag and a memory of 256 bytes. The registers may be used to hold data as well as to access memory. Each register consist of two parts. The first part specifies a neighboring CPU. The second part specifies an address within the CPU. If an individual tries to access a non-existing address we perform a modulo operation on this register. The neighborhood relation is shown in Figure 1. Each CPU has access to the address space of the CPUs on the left, right, in front of, behind as well as above and below its own CPU. Which CPUs are accessed by a command is determined by the orientation of the CPU. The orientation of each CPU is initially set randomly and may be changed by executing special commands during the run as shown in Figure 2.

We have used a two-address instruction format [19,12]. The list of opcodes is shown in Table 1. The result replaces the value of the second operand. The structure of the machine language is shown in Figure 3. Each command consists of 8 bytes. The first two bytes code for the opcode of the command. Bytes three and four code for the addressing mode of the first operand. Bytes five and six code for the addressing mode of the second operand. Constants are stored in bytes seven and eight. A redundant mapping is used to decode the 2 opcode bytes. Thus, different byte combinations may code for the same opcode. Opcodes are assigned randomly to the different byte combinations. Additional redundant mappings are used to decode the addressing modes of the first and

Table 1. List of opcodes. A "-" in the column for operand 1 or 2 indicates that the opcode does not use this operand. "A" denotes the addressing modes: register indirect, register indirect with auto-increment, register indirect with auto-decrement, and bind. "R" denotes the register addressing mode. "C" denotes the direct addressing mode, and "N" denotes the neighbor addressing mode.

opcode	operand 1	operand 2	name of operation
nop	-	-	no operation
lea	A, N	R	load effective address
move	C, R, A	R, A	move operand 1 to operand 2
clr	-	R, A	set operand to zero
inc	-	R, A	increment operand by one
dec	-	R, A	decrement operand by one
add	C, R, A	R, A	add operand 1 to operand 2
sub	C, R, A	R, A	subtract operand 1 from operand 2
neg	-	R, A	negate operand
not	-	R, A	negate bits
and	C, R, A	R, A	and operand 1 with operand 2
or	C, R, A	R, A	or operand 1 with operand 2
xor	C, R, A	R, A	xor operand 1 with operand 2
shl	-	R, A	shift operand left
shr	-	R, A	shift operand right
cmp	C, R, A	R, A	compare operand 1 with operand 2
beq	-	A	branch on equal (zero flag set)
bne	-	A	branch on not equal (zero flag not set)
jmp	-	A	jump
jsr	-	A	jump to subroutine
rts	-	A	return from subroutine
pop	-	R, A	pop from stack
push	C, R, A	-	push operand to stack
spawn	-	A	spawn new process
kill	-	R, N	kill CPU
turnl	-	-	turn CPU left
turnr	-	-	turn CPU right
turnu	-	-	turn CPU up
turnd	-	-	turn CPU down
fwd	-	-	move CPU forward
label	-	-	marker used for relative addressing

second operand. Because the number of allowed addressing modes can differ for each operator we have used separate mappings to decode the operands for each operator. A standard binary mapping is used to decode the constants.

The following addressing modes are available: direct (e.g. `move #1,r0`), register (e.g. `move r1,r0`), register indirect (e.g. `move (r1),r0`), register indirect with auto-increment (e.g. `move (r1)+,r0`), register indirect with auto-decrement (e.g. `move -(r1),r0`), bind (e.g. `lea <start>,r0`), and neighbor (e.g. `lea [lft],r0`). Depending on the type of command only a subset of all addressing modes can be used. The allowed addressing modes for each command are shown in Table 1. The addressing modes have the usual meaning except for the bind and neighbor addressing modes. Neighbor is used to refer to a neighboring CPU. The bind addressing mode may be used to access memory. This is an addressing mode which is similar to the complementary labels which are used in the Tierran assembly language. In our assembly language, labels may be compiled using the `label` opcode. Each label is assigned a unique constant. This constant is stored in byte 8 of the machine code. If the instruction pointer executes the `label` opcode nothing happens. The bind addressing mode may be used to search for a particular label inside its own or neighboring CPUs. The label with the best match is returned as the resulting address. Special opcodes have been added to change the orientation of a CPU. A CPU may change its orientation by turning to the left (`turnl`), right (`turnr`), up (`turnu`) or down (`turnd`). A process is also able to move through the environment by executing the `fwd` command. This command swaps the specified CPU with the CPU which executed the command. Swapping as opposed to simply copying or copying and erasing the original has been used in order to preserve the contents of the destination CPU.

A command to allocate memory is not included in the instruction set. Other virtual environment for self-reproducing individuals (e.g. Tierra [27] and Avida [1]) include such an instruction. This instruction must be called by each individual which wants to reproduce itself. In our environment there is no need to allocate memory because individuals have read as well as write access to neighboring CPUs. Individuals reproduce themselves by copying their genotype to an adjacent CPU and setting the program counter to the start of the program. Each process is only allowed to execute a limited amount of steps. Thus, it has to reproduce its genetic material before its lifetime is over. In order to avoid periodic effects of the evaluation method we evaluated CPUs at random. On average all CPUs are evaluated for the same number of steps.

3 Reproduction, variation and selection

Individuals reproduce by copying their genetic material to a neighboring CPU and setting the instruction pointer appropriately. In order for evolution to occur, we also need a source of variation and selection. Selection of individuals occurs for three reasons. First, the lifetime of each individual is limited. Each individual may only execute a fixed number of steps before its CPU stops executing instructions. Second, an individual has the possibility of killing another individual and third, space is finite. Several methods to kill another individual exist. An individual may use the kill instruction, reset the instruction pointer using the spawn instruction or simply overwrite the memory of its opponent. Thus, individuals have to protect themselves against other individuals in order to survive. Selection occurs naturally.

In order for evolution to occur, we also need some source of variation. Each memory location is mutated with probability p_{cosmic}. One may equate this type of variation with cosmic rays randomly hitting bytes in memory. If an opcode is to be mutated we simply choose two new bytes for the opcode. The mutations hitting the addressing mode are handled in the same way. Because redundant mappings are used to decode the operands as well as the addressing modes the mutation does not necessarily create a new phenotype. In order to ensure strong causality random mutations of constants are handled differently. If a cosmic mutation hits a constant (bytes 7 or 8 of our machine language) we simply increment or decrement the constant. Thus our environment differs from existing environments in that we use a non-uniform mutation operator. It is highly important that strong causality is preserved. Otherwise we would essentially be performing a random search in the space of self-replicating individuals. Ray has carefully constructed an instruction set with high evolvability by removing all operands [27]. Because we are free to specify the physics of our virtual environment one may as well change the mutation operator which is the option we have chosen for our environment.

Another source of variation results from errors which occur when determining a memory address. The address is either decreased or increased by 1 with probability p_{addr}. Bytes read from memory are decreased or increased by 1 with probability p_{read}. Bytes written to memory are decreased or increased by 1 with probability p_{write}. Thus, an offspring may be different from its parent because of addressing errors, read errors or write errors. In addition, with probability p_{shift} we randomly shift a memory section 8 bytes up or down after a spawn occurred. This causes one command to be removed and one command to be doubled. The command which was doubled is either left untouched, replaced with a NOP or replaced with a random command. The shift operation allows the self-replication programs to grow or shrink in size if necessary.

As the memory fills with self-reproducing individuals, selection sets in. Only those individuals which successfully defend themselves against other individuals will be able to replicate. Short individuals will have an advantage because they are able to reproduce quickly. Another strategy is to develop an elaborate defense mechanism and thereby ensure that the replication will succeed. This might even entail creating signals or tags to make sure that an older individual doesn't destroy its newly created offspring. In principle, it may be possible for the individuals to distinguish their own species from different species by looking at their genotype or inventing special tag bytes. In addition, crossover can develop naturally because at any point in time several individuals may be copying their genetic material into the memory of the same CPU.

4 Experimental results

Results are described for an environment consisting of a cube of $10 \times 10 \times 10$ CPUs. The probabilities p_{cosmic}, p_{read}, p_{write}, p_{addr}, and p_{shift} have been set to 10^{-6}, 10^{-4}, 10^{-4}, 10^{-3}, and 0.5 respectively. These probabilities were chosen such that the different mutations occur approximately equally often per reproduction. Each CPU has 256 bytes of memory. We filled the memory of all CPUs with NOPs and loaded a manually coded self-reproducing individual into a single CPU. The source code of this individual is shown in Figure 4. The individual first moves one step forward, turns to the left and then up. Next, it determines the beginning and the end of the program and copies its genotype back to front to the CPU on its left side. We performed a number of runs. However, results are only shown for a single run. Thus, the reader may see what actually happens during the run.

Initially, most CPUs are empty and can be used by the individual and its offspring to replicate themselves. Soon, almost all available CPUs are filled. Now individuals have to compete for CPUs. We have analyzed how the original

```
Addr   Bytes                Source Code          Comment
------------------------------------------------------------------------
0000 EB6F CB28 EAE2 0000   label  <00>
0008 FB3C 152C 1096 9E2C   fwd                   ; go forward
0010 B609 10CF 621A D479   turnl                 ; turn left
0018 C3C6 CED4 1EB5 5048   turnu                 ; turn up
0020 E5D3 3DC5 EA0D 0083   lea    <00>,R0        ; get beginning
0028 E25E F50F 5332 AA15   lea    <AA>,R1        ; get end
0030 0701 5D91 EF8C 08C9   add    #08,R1         ; add 8 bytes for label
0038 663B E709 F66B 459E   move   R1,R2          ; copy contents of reg.
0040 1B57 A934 F25B B35D   lea    [lft],R2       ; set cpu pointer
0048 87F9 188E 899E CC55   label  <55>
0050 FCC8 4552 E39D 7DC2   move   -(R1),-(R2)    ; copy one byte
0058 B8D6 D016 102A EB77   cmp    R2,R0          ; check if done
0060 51B3 14F3 1302 1455   bne    <55>           ; copy next byte
0068 4322 2624 D15A 3A59   spawn  (R2)           ; spawn new process
0070 DA32 5909 EB54 AB00   jmp    <00>           ; repeat
0078 8090 2B1D EBA6 61AA   label  <AA>
```

Figure 4. Self-reproducing program. The program copies itself back to front to the CPU on its left side. After all bytes are copied the program spawns a new process. The source code of the program is shown on the right, the assembled bytes on the left. The first column shows the memory addresses.

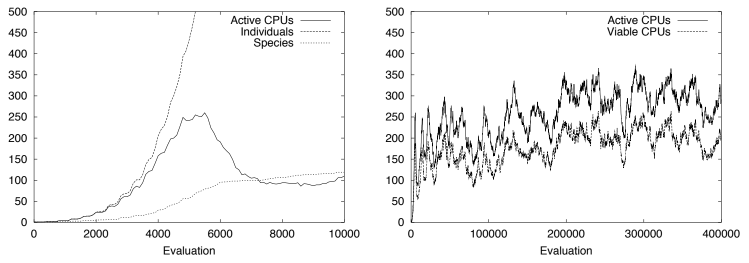

Figure 5. Relation between speciation and number of active CPUs.

individual speciates. An offspring and its parent are considered to belong to the same species if the offspring contains all of the bytes which were executed by its parent. A comparison between speciation and number of active CPUs is shown in Figure 5. The graph on the left shows how the total number of individuals and the total number of species increases over time. The number of active CPUs is also shown. One can clearly see that speciation happens at a faster rate if the number of active CPUs is high. The more active CPUs the higher the speciation rate. Thus, the co-evolutionary environment aids speciation in that due to the presence of other individuals the mutations accumulate in the offspring. All individuals try to replicate themselves, but only if there are a number of active CPUs in the neighborhood, mutations start to accumulate and new species are created. How the number of active CPUs changes during the run can be seen on the graph on the right. The number of individuals which have successfully created an offspring (viable individuals) are also shown in the same graph.

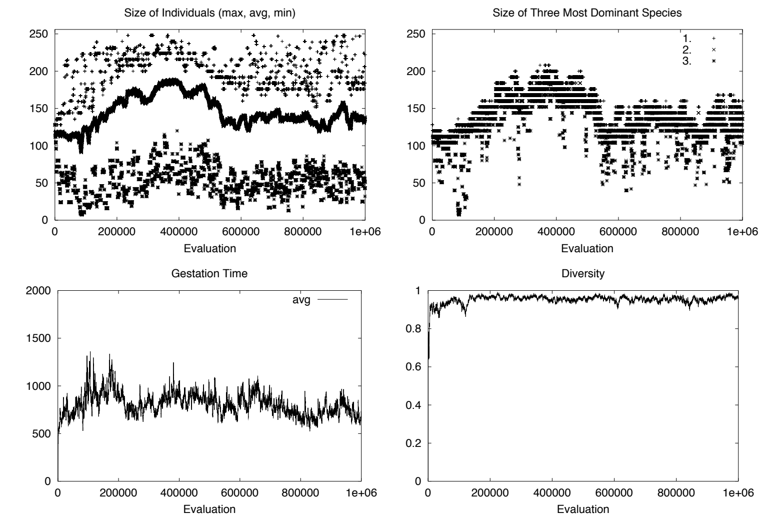

Figure 6. Size, gestation time and diversity of self-reproducing programs.

The average size of viable individuals is shown in Figure 6. Also shown in Figure 6 is the size of the three most dominant species, the average gestation time and the diversity of the population. We define gestation time as the number of steps executed over the total number of offspring. Simpson's index is used to calculate diversity. We group individuals into species depending on their size. The graph shows Simpson's index scaled with the factor 256/255 because individuals may reach a maximum size of 256 bytes.

5 Conclusion

Experiments with a three-dimensional environment of self-replicating computer programs are reported. The environment consists of a cube of CPUs. Each CPU may be running at most one individual. Individuals have read as well as write access to the memory of neighboring CPUs. They replicate by copying their instruction to a neighboring CPU and setting the instruction pointer appropriately. Individuals need to move in order to find new CPUs which may be used for replication. Speciation sets in after all CPUs have been filled and individuals need to compete in order to use CPUs for replication. Thus, selection comes about naturally in our environment.

Instead of using a highly simplified instruction set, we have chosen to experiment with a high level machine language. Redundant mappings have been used in order to increase evolvability. We have shown that artificial evolution can also be carried out in a complex assembly language. In addition we have used a mutation operator which takes the structure of the machine language into account to ensure strong causality. This type of environment aids speciation and leads to a highly diverse population of individuals. In future experiments we plan to investigate the impact of different redundant mappings on the evolvability of the population. A more detailed investigation of the evolutionary dynamics would also be interesting, i.e. do we get a similar diverse ecosystem with parasites and the like as in Tierra?

References

1. C. Adami. *Introduction to Artificial Life.* Springer-Verlag, New York, 1998.
2. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming - An Introduction: On The Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann Publishers, San Francisco, CA, 1998.
3. R. Davidge. Looking at life. In F. J. Varela and P. Bourgine, eds., *Toward a practice of autonomous systems: Proc. of the 1st Europ. Conf. on Artificial Life*, pp. 448–454, Cambridge, MA, 1992. The MIT Press.
4. A. K. Dewdney. Computer recreations: In the game called core war hostile programs engage in a battle of bits. *Scientific American*, 250(5):15–19, 1984.
5. A. K. Dewdney. Computer recreations: A core war bestiary of viruses, worms and other threats to computer memories. *Scientific American*, 252(3):14–19, 1985.
6. A. K. Dewdney. Computer recreations: A program called mice nibbles its way to victory at the first core war tournament. *Scientific American*, 256(1):8–11, 1987.
7. A. K. Dewdney. Computer recreations: Of worms, viruses and core war. *Scientific American*, 260(3):90–93, 1989.
8. P. Dittrich, M. Wulf, and W. Banzhaf. A vital two-dimensional assembler automaton. In C. C. Maley and E. Boudreau, eds., *Artificial Life VII Workshop Proceedings*, 2000.
9. M. Ebner. On the search space of genetic programming and its relation to nature's search space. In *Proc. of the 1999 Congress on Evolutionary Computation, Washington, D.C.*, pp. 1357–1361. IEEE Press, 1999.
10. M. Ebner, P. Langguth, J. Albert, M. Shackleton, and R. Shipman. On neutral networks and evolvability. In *Proc. of the 2001 Congress on Evolutionary Computation, Seoul, Korea.* IEEE Press, 2001.
11. D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Reading, MA, 1989.
12. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.
13. J. H. Holland. *Adaptation in natural and artifical systems: an introductory analysis with applications to biology, control, and artificial intelligence.* The MIT Press, Cambridge, MA, 1992.
14. M. A. Huynen. Exploring phenotype space through neutral evolution. *Journal of Molecular Evolution*, 43:165–169, 1996.
15. S. A. Kauffman. *The Origins of Order.* Oxford University Press, Oxford, 1993.
16. M. Kimura. *Population Genetics, Molecular Evolution, and the Neutral Theory: Selected Papers.* The University of Chicago Press, Chicago, 1994.
17. J. R. Koza. *Genetic Programming.* The MIT Press, Cambridge, MA, 1992.
18. J. R. Koza. *Genetic Programming II.* The MIT Press, Cambridge, MA, 1994.
19. M. M. Mano. *Computer System Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 3rd ed., 1993.
20. M. Mitchell. *An Introduction to Genetic Algorithms.* The MIT Press, Cambridge, MA, 1996.

21. A. N. Pargellis. The evolution of self-replicating computer organisms. *Physica D*, 98:111–127, 1996.
22. A. N. Pargellis. The spontaneous generation of digital "life". *Physica D*, 91:86–96, 1996.
23. S. Rasmussen, C. Knudsen, and R. Feldberg. Dynamics of programmable matter. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, eds., *Artificial Life II: SFI Studies in the Sciences of Complexity, Vol. X* , pp. 211–254, Redwood City, CA, 1991. Addison-Wesley.
24. S. Rasmussen, C. Knudsen, R. Feldberg, and M. Hindsholm. The coreworld: emergence and evolution of cooperative structures in a computational chemistry. *Physica D*, 42:111–134, 1990.
25. T. S. Ray. An approach to the synthesis of life. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, eds., *Artificial Life II: SFI Studies in the Sciences of Complexity, Vol. X* , pp. 371–408, Redwood City, CA, 1991. Addison-Wesley.
26. T. S. Ray. Is it alive or is it ga? In R. K. Belew and L. B. Booker, eds., *Proc. of the 4th Int. Conf. on Genetic Algorithms, University of California, SD*, pp. 527–534, San Mateo, CA, 1991. Morgan Kaufmann Publishers.
27. T. S. Ray. Synthetic life: Evolution and optimization of digital organisms. In K. R. Billingsley, H. U. Brown III, and E. Derohanes, eds., *Scientific Excellence in Supercomputing*, pp. 489–531, Athens, GA, 1992. The Baldwin Press.
28. T. S. Ray. Evolution and complexity. In G. Cowan, D. Pines, and D. Meltzer, eds., *Complexity: Metaphors, Models, and Reality. SFI Studies in the Sciences of Complexity, Proc. Vol. XIX* , pp. 161–176. Addison-Wesley, 1994.
29. T. S. Ray. Selecting naturally for differentiation. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, eds., *Genetic Programming 1997, Proc. of the 2nd Annual Conf., July 13-16, 1997, Stanford University*, pp. 414–419, San Francisco, CA, 1997. Morgan Kaufmann Publishers.
30. T. S. Ray and J. F. Hart. Evolution of differentiation in multithreaded digital organisms. In M. A. Bedau, J. S. McCaskill, N. H. Packard, and S. Rasmussen, eds., *Artificial Life VII, Proc. of the 7th Int. Conf. on Artificial Life*, pp. 132–140, Cambridge, MA, 2000. The MIT Press.
31. I. Rechenberg. *Evolutionsstrategie '94*. frommann-holzboog, Stuttgart, 1994.
32. M. Shackleton, R. Shipman, and M. Ebner. An investigation of redundant genotype-phenotype mappings and their role in evolutionary search. In *Proc. of the 2000 Congress on Evolutionary Computation, La Jolla, CA*, pp. 493–500. IEEE Press, 2000.
33. R. Shipman. Genetic redundancy: Desirable or problematic for evolutionary adaptation? In A. Dobnikar, N. C. Steele, D. W. Pearson, and R. F. Albrecht, eds., *4th Int. Conf. on Artificial Neural Networks and Genetic Algorithms*, pp. 337–344, New York, 1999. Springer-Verlag.
34. R. Shipman, M. Shackleton, M. Ebner, and R. Watson. Neutral search spaces for artificial evolution: A lesson from life. In M. A. Bedau, S. Rasmussen, J. S. McCaskill, and N. H. Packard, eds., *Artificial Life: Proc. of the 7th Int. Conf. on Artificial Life*. MIT Press, 2000.
35. M. Sipper. Fifty years of research on self-replication: An overview. *Artificial Life*, 4:237–257, 1998.
36. J. Skipper. The computer zoo - evolution in a box. In F. J. Varela and P. Bourgine, eds., *Toward a practice of autonomous systems: Proc. of the 1st Europ. Conf. on Artificial Life*, pp. 355–364, Cambridge, MA, 1992. The MIT Press.
37. E. van Nimwegen, J. P. Crutchfield, and M. Huynen. Neutral evolution of mutational robustness. *Proc. Natl. Acad. Sci. USA*, 96:9716–9720, 1999.