Design of Evolvable Computer Languages

Charles Ofria, Christoph Adami, and Travis C. Collier

***Abstract*—We investigate common design decisions for constructing a computational genetic language in an autoadaptive system. Such languages must support self-replication and are typically Turing-complete so as not to limit the types of computations they can perform. We examine the importance of using templates to denote locations in the genome, the methods by which those templates are located (direct-matching versus complement-matching), methods used in the calculation of genome length and the size and complexity of the language. For each test, we examine the effects on the rate of evolution of the populations and isolate those factors that contribute to it, most notably the organisms' ability to withstand mutations.**

***Index Terms*—Auto-adaptive systems, computer languages, digital life, evolvability, robustness.**

I. Introduction

As current trends in the growth of computer hardware and software continue, we face difficulties with the development of complex operating systems and the programs designed to run under them. Computer programs can comprise many millions of lines of code, yet are intended to interact smoothly with other software, written independently. These systems are becoming effectively untestable and their behavior is unpredictable. New paradigms of code generation, testing, and assembly are borrowing principles from nature, by interpreting living organisms as complex machines that are constructed from—and operating on—"software" (the genome) several billions of lines long, assembled from multiple sources, and operating in a robust and fault-tolerant manner.

Early experiments in genetic programming [8] and evolutionary programming [5] focused on the evolution of tree-like structures in which each "atom" already had a functionality related to the problem to be solved. Also, genetic algorithms [6] can be viewed as a tool to evolve specialized problem-solving code. In both instances, the brittleness of the coding—the tendency of evolved code to easily break under mutations—seems to go hand-in-hand with the specialization of the atomic instructions used and, therefore, as the price to pay to ensure fast adaptation.

Unlike traditional genetic programming and genetic algorithms, in systems of self-replicating computer codes (autoadaptive systems) robustness to mutations is intrinsically selected for, as high-fidelity information transmission equates to having more living offspring in the next generation. All of the systems discussed in this paper consist of populations of computer programs coded in Turing-complete programming languages and made to exist in a noisy environment. The individual programs must produce copies of themselves (self-replicate) to survive over long periods of time. Errors in replication are akin to mutations in biological systems and drive the evolutionary process. The underlying genetic language used in any of these systems is of critical importance, as it shapes the range of mutational effects that are possible in the system and determines the ways in which genetic space can be exploited [10].

Evolution of fault tolerance as well as mutation sensitivity has been considered previously in the context of the evolution of robot controllers, where fault tolerance was explicitly included in the fitness function [15]. Mutational robustness has been explored independently by van Nimwegen *et al.* [16] and the concept of "neutrality" discussed there is similar to the one used here.

In this paper, we are taking initial steps in examining what characteristics are required to develop a genetic language that is flexible in the computations it can perform (computationally universal in both a theoretical and practical sense) and is highly evolvable. Rather than settling for a single instance of such an "artificial chemistry," we explicitly test elements that influence the adaptability of programs using the avida platform (see below). In particular, we study the role that evolution plays in generating populations of programs that react to noisy environments in a robust and predictable manner, while maintaining evolvability.

II. Exploring Artificial Chemistries

Isolating those aspects of an instruction set's design that are directly responsible for evolvability is of fundamental importance if dedicated evolvable instruction sets are to be designed. The original experiments with populations of self-replicating programs performed by Rasmussen [12] in the Coreworld system seemed to rule out evolution because these programs turned out to be extremely fragile: self-replicating digital organisms written in the redcode language could not survive even miniscule amounts of stochasticity in the replication process, leading to "dying" populations. Thus, redcode represents a computationally effective chemistry that, however, does not survive mutations (i.e., it is extremely brittle).

A critical step was taken by Ray [13], who recognized that the brittleness of redcode is due primarily to the *argumented* instruction set: independent mutations in the instruction and its arguments are unlikely to lead to a meaningful combination. In experiments with a version of redcode designed to run on the avida system we determined that, in fact, over 99.7% of all nontrivial[1] mutations are deleterious in this architecture, making evolvability in this language very limited. Those few mutations that were not deleterious were almost entirely neutral. After the first 70 generations, not a single mutation in any of the 100 trials was beneficial and all trials progressed for at least 2000 generations without significant adaptation.

These experiments have, however, shown that information can be preserved in large populations if programs have protected memory. When mutations are applied to an arbitrary population without any such protection, a single broken organism may write to portions of hundreds of others before it dies, effectively killing each of them and, in time, bringing the entire population to extinction.

Memory protection and an argument-free instruction set led to the first successful evolutionary experiments with an assembly-like language in Ray's tierra world. Rather than using arguments for direct addressing, Ray's instruction set relied on patterns of instructions whose execution has no effect [no-operation (nop) instructions] for relative addressing. Such instructions play a role analogous to untranslated binding sites in biochemical code (e.g., promoter sequences). Self-replicating programs survive well in the tierra world and can adapt to user-specified fitness landscapes [1] and grow in complexity [3].

Manuscript received November 9, 2000; revised December 26, 2001. This work was supported in part by Microsoft Research and in part by the National Science Foundation under Award PHY-9723972 and Award DEB-9981397.

C. Ofria is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: charles@alife.org).

C. Adami is with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109 USA.

T. Collier is with the Division of Organizmic Biology, Ecology, and Evolution, University of California at Los Angeles, Los Angeles, CA 90095 USA.

Publisher Item Identifier 10.1109/TEVC.2002.802442.

[1]A trivial mutation is defined as one that affects only nonexecuted portions of code.

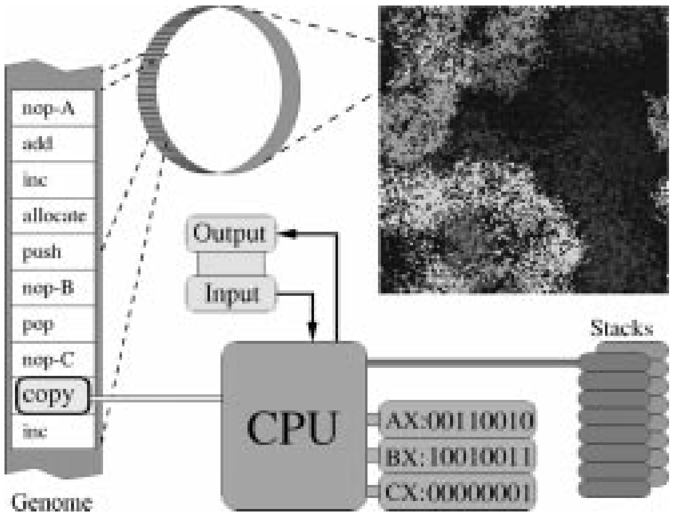


Fig. 1. Genome of a digital organism and its virtual CPU, with registers, buffers, and stack architecture. Each genome and CPU occupies a position on the lattice. Genomes are shaded according to genotype.

The artificial chemistry of the avida world[2] (see, e.g., [2]) differs significantly from the coreworld and tierra systems. In avida, each program has a natural memory protection as it occupies a unique location on a two-dimensional lattice that other programs cannot directly access.[3] (see Fig. 1). Consequently, interactions between programs are local, constrained to neighboring lattice sites, and organisms cannot corrupt the genome of other organisms by overwriting, as happens in coreworld. This feature contributes significantly to the stability of populations in this world, as it ensures that a program's fidelity does not depend on its neighbors. Avida also features a much more complex environment than tierra, and the organisms evolve to reflect this complexity. From a simple ancestor, adapted programs readily emerge, with increased code lengths that accommodate the new "genes" (sequences of instructions) used to take advantage of the environment [3]. Because length changes are important in the evolution of complexity, we also tested the size-change mechanisms allowed for in the various instruction set.

We have systematically tested (within the avida system) the influence of design choices on evolvability by constructing several chemistries:

Set I is the standard instruction set, with 28 instructions and minimal redundancy. It has three nop instructions (`nop-A`, `nop-B`, and `nop-C`) that are used to modify (i.e., act in conjunction with) other instructions. As there are no explicit arguments in the default instruction set, this is the only way in which instructions can explicitly influence the behavior of other instructions. Such instructions naturally lead to epistasis in the code. The nop instructions can also be used to construct patterns of instructions called templates that can be searched for (relative addressing). Each pattern of nop instructions has a complementary pattern, obtained by replacing `nop-A` by `nop-B`, `nop-B` by `nop-C`, and `nop-C` by `nop-A` (cyclic complementarity).

This set contains four flow-control instructions. The `jump-f` and `jump-b` instructions move the instruction pointer to a complement template, i.e., the nops that follow the jump instruction are read and converted to a complement pattern, which is located in the forward or backward direction (depending on which of the instructions is used). The `call` instruction uses the templates in a similar fashion, but will push the address of the instruction that directly follows the initial template onto the stack. The `return` instruction will pop the first number off of the stack and jump back to the address associated with it.

The set also contains three conditionals (`if-n-equ`, `if-less`, and `if-bit-1`) that will skip the following instruction if the condition they are testing for is not met. These can be combined with the flow control statements to give the digital organisms a significant ability to regulate their execution.

For basic mathematics, the default instruction set supplies the organisms with seven instructions: `shift-r`, `shift-l`, `inc`, `dec`, `add`, `sub`, and `nand`, all of which manipulate numbers in the registers.

Four instructions are for management of internal state, allowing the organisms to control how numbers are moved around within their CPU. The `push`, `pop`, `swap-stk` instructions all work with the stacks and `swap` will exchange the contents of two registers.

Four sensory instructions are included so that the organisms can obtain information about themselves and their environment. The `search-f` and `search-b` instructions will find templates within the genome (in the forward or backward direction) and return their distance to the search command into a register. The `get` and `put` instructions will read numbers out of the environment or write them back in, respectively.

The final three instructions are all specific to the replicative process for the organisms. First, they must `allocate` memory space to write their offspring into. Then they must use the `copy` command once on each instruction to construct the offspring (typically by going through a "copy loop") and, finally, they must `divide` to release the offspring.

Set II tests the importance of cyclic complementarity of nop patterns (defined in Set I). This direct-matching set is identical to standard except that complementarity is direct rather than cyclic (i.e., `nop-A` matches `nop-A`, not `nop-B`, etc.) The size of this set (the number of different available instructions) is identical to standard.

Set III tests the rationality for including nop instructions at all. The no-nop set lacks all three nop instructions entirely. The instructions `jump-f`, `jump-b`, and `call` all require a value in the BX register (as opposed to a template) that set the distance to be jumped. Additionally, the `search-f` and `search-b` instructions are removed. Finally, instead of `push` and `pop`, the register-specific `push-AX`, `push-BX`, `push-CX`, `pop-AX`, `pop-BX`, and `pop-CX` are used, for a total of 27 instructions in this set.

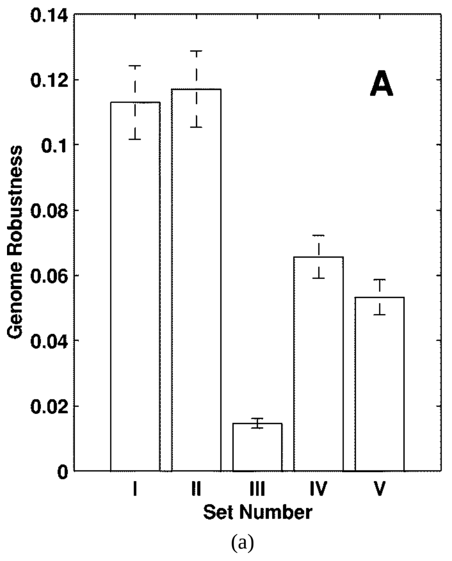
Set IV is designed to study the mechanism of length modification on evolvability. It contains all of the instructions from standard, with the single addition of the `memsize` instruction. In all of the other sets, an organism must calculate its own genome length (memory size) before they can allocate new memory to copy their offspring into. Often, this size-calculation mechanism is fragile, forcing the organism's lineage to become stuck at a fixed genome length, severely limiting further evolution. The `memsize` instruction provides a single instruction that will return the genome length without complex self-inspection, making robust length changes possible. The size of this set is 29.

Set V tests the influence of instruction set size on the dynamics of evolution. This long set comprises 84 unique instructions, with no additional functionality beyond the standard set. The new instructions are all variants of the normal instructions. For example, from the conditionals provided in the standard set, any numerical comparison can be constructed. However, the long set contains the additional conditionals `if-equ`, `if-grt`, `if->=`, `if-<=`, `if-equ-0`, `if-not-0`, `if-grt-0`, `if-less-0`, `if->=-0`, `if-<=-0`, `if-A!=B`, `if-B!=C`, and `if-A!=C`.

For each experiment in each chemistry, we test the robustness to mutations of any program that is ever most abundant (dominant) in the population. This is done by iterating through every possible

[2]The avida software (including source code) is freely available at http://dllab.caltech.edu.

[3]The instruction pointer cannot execute or overwrite code of neighboring organisms except in special chemistries that explicitly allow for this. Thus, the phenomenon of parasitism [13] is excluded in the present experiments.



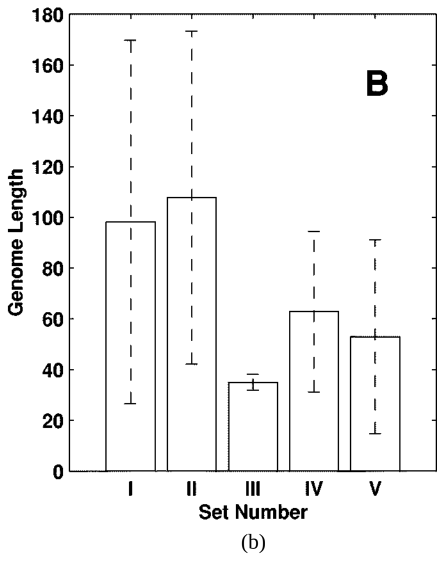


Fig. 2. (a) Genome robustness averaged over the dominant genotypes at the end of 100 trials for each of the five chemistries, with error bars indicating a confidence interval around this value, using standard error. (b) Genome length for the same genotypes, with error bars depicting the standard deviation across trials.

single-step mutation and examining each resulting organism in an isolated test-CPU to determine its relative fitness as compared to the wild type. We record the fraction of mutations that fall into each of the categories of fatal, deleterious, neutral, or beneficial. In practice, all deleterious mutations prove fatal for the mutant as it is out-competed. Most beneficial mutations turn out to be fatal for the rest of the population, as the edge in replicative ability spells doom for inferior genotypes in this single-niche environment, yet the percentage of advantageous mutations is so small that no statistical significance can be attributed to it. Consequently, we can classify almost all mutations as either neutral or effectively fatal. In this case, an obvious measure of robustness is the fraction f_ν of single mutations of a genome that are neutral or beneficial, i.e.,

$$f_\nu = \frac{N_\nu}{(\mathcal{D} - 1)\ell} \tag{1}$$

where N_ν is the number of all neutral or beneficial single mutants of the wild type, $\mathcal{D}$ is the size of the instruction set, and ℓ is the genome length.

III. Results

Due to the contingent nature of evolution, trials with identical conditions but different random number seeds can—and do—lead to wildly different outcomes. Here, we have the opportunity to repeat such trials many times (i.e., multiple replicas) to gain statistical significance and extract global characteristics that set one chemistry apart from another. We focus on two such measures: the fraction of neutral or beneficial mutations to measure robustness [see Fig. 2(a)] and the average fitness [relative replication rate with respect to the ancestor (see Fig. 3)] as a function of time to measure evolvability. As in previous experiments [3], [9], the programs adapt to a world in which all binary logical operations on up to three random numbers (provided by the environment when the digital organisms issues a `get` instruction and evaluated on a `put` instruction) are rewarded with bonus resource (CPU time), which effectively increases the replicatory speed of those organisms that developed the code necessary to trigger it.

For each of 100 replicas in each of the five chemistries,[4] the robustness measurement is obtained by extracting the dominant genotype at the end of each trial and examining all possible one-point mutations in order to calculate the fraction of them that are nondetrimental. These values are then averaged across the replicas of a particular set to produce Fig. 2(a).

The robustness of the initial ancestor used in these experiments is very low at 0.005 (no doubt due to the clumsy human design), but evolution has moved it to the levels shown. We can see that there are significant differences in robustness between chemistries, and that this measure seems to be strongly correlated to sequence length [see Fig. 2(b)].

The robustness of a genome can be thought of as the fraction of its length that is impervious to mutations and thus carries no information. The amount of information about the environment the sequence evolves in can then be approximated by the sequence length minus the number of neutral instructions [3]. Thus, learning events (evolutionary transitions in which information is being acquired) decrease robustness if the sequence length stays constant, while size increases without commensurate acquisition of information increase robustness [3]. The mechanism by which the sequence length changes, thus, is crucial for both robustness and in turn evolvability [4], [7].

In three of the chemistries (sets I, II, and V), length changes are possible only as long as the program calculates its length by finding its end (marked by a pattern of `nop` instructions), which is the algorithm used by the ancestral program. Any insertions or deletions that occur in such an organism will be measured properly, and thus accounted for in the offspring's memory allocation. If, however, length is calculated by some other means, such as a mathematical operation that produces a result equal to the length, then this calculation is not likely to change (and certainly not commensurately) when mutations have led to an altered length.

The chemistry in which length changes are easiest is set IV, as it contains a single instruction (`mem-size`) that directly returns the sequence length without self-inspection. If `mem-size` is used for length calculation (as was the case in 96 of the 100 trials in set IV) insert or delete mutations become more neutral and length changes occur as needed. The opposite extreme is set III in which no nop instructions are

[4]Each population of 60 × 60 programs was allowed to adapt for 50 000 updates subject to a mutation rate of 7.5 × 10^{-3} errors per instruction copied, as well as a 5% probability for a single insert or delete mutation per gestation cycle.

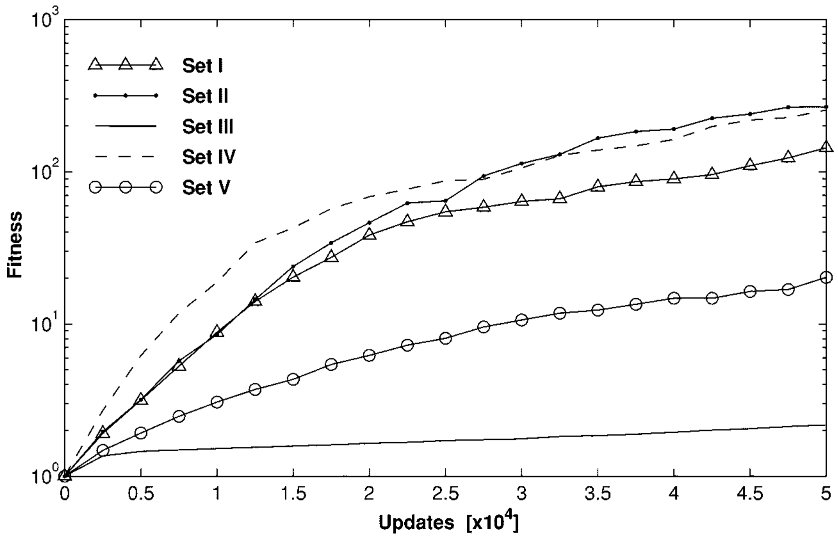


Fig. 3. Fitness averaged over 100 trials for each of the five chemistries, as a function of time (in updates).

available rendering template matching impossible. For the experiments with this no-nop set, the standard ancestor must be replaced by one in which the genome length is explicitly coded into the sequence. Consequently, changes in length are rare as they involve significant code rearrangements.

Instruction set II differs from the standard set only in that the search for a pattern of nop's seeks the pattern itself rather than its complement, as described earlier. This seemingly innocuous change leads to important differences in the evolutionary history. In the ancestral organism, two pairs of patterns are required for correct functioning: a pattern at the initial point of the search and its paired pattern (either a complement or direct match) to mark the end of the genome and a second pair framing the copy loop of the organism. Since, in set I, the patterns in each pair must be different, a total of four distinct patterns have to be used. As such, they are cumbersome to maintain and with complementary template matching, we have witnessed that only 18% of the replicas retain a flexible size-calculation structure. In most others, it is replaced by one in which the organism's search for its end returns the location of the copy loop instead, then finds its own length ("accidentally") by manipulating this number, typically by doubling it. This construct can require significantly fewer instructions and is therefore selected for as it will decrease the organism's replication time. While this is an effective way to calculate program length, it is also brittle: length changes can occur only if several instructions are changed in a commensurate manner. As a consequence, the standard set develops difficulties in adjusting the program's length the moment the new algorithm is locked in. On the contrary, in set II, the original algorithm is maintained more frequently (in 29% of the replicas) because the direct matching of templates tends to avoid such misdirected searches and is therefore more difficult to replace.

The differences in the way length changes occur is reflected in both evolvability and robustness. First, robustness is inherently higher for chemistries that lead to the development of junk code, i.e., loci that do not code for information [see Fig. 2(a)]. These are chemistries I and II where length changes occur frequently, but the direct-matching chemistry (set II) holds the edge after 20 000 updates when 72.5% of the trials in set I have locked in a nonrobust algorithm for length calculation (as compared to only 58% of set II) leading to problems in the acquisition of more information. In fact, 44.5% of the trials in set I lock in within the first 2000 updates (before there is much opportunity for any computational tasks to be acquired), as compared to only 3% of set II. If these early lock-ins are removed, the final average fitness of set I is 74.76, greater than (though statistically equivalent to) the average fitness of set II, 63.13.

The chemistry that is deprived of the possibility of relative addressing offered by templates of nop instructions (set III) is extremely inflexible: length changes are infrequent [indicated by the small lengths with a very low standard deviation, as shown in Fig. 2(b)], leading to poor adaptation. In that respect, it is more akin to the redcode chemistry mentioned earlier.

Populations in set IV change genome length most easily and use this ability to eliminate junk code. Consequently, its robustness is lower except for the early stages of evolution where, in fact, it increases to its final robustness level almost immediately giving it a significant head start in information acquisition (data not shown).

Set V, which consists of 84 instructions, lags in fitness. We attribute this to the smaller rate of advantageous substitutions (due to the larger set of instructions to choose from), while the versatility of the instructions seems to result in smaller sizes and less junk code.

IV. Conclusion

We have examined five artificial chemistries with respect to their evolvability and found that the differences among them are attributable mainly to their robustness to mutations and the manner in which genome-size changes occur. While fitness continues to increase during the evolutionary process, the robustness (on average) stays constant, suggesting that the adaptive process has led the population to a "comfortable" level that avoids evolutionary dead ends.

Even though the artificial chemistries examined here (and the artificial physics represented by the avida world) may appear ad hoc, we believe that some of the results we have seen (such as the tendency of evolvability to go hand in hand with mutational robustness or neutrality) to be universal. This conviction stems from work unrelated to the investigation of instruction set design, where avidian digital organism were used to study standard problems in evolutionary biology [3], [9], [11], [17] and appeared to give results at least as independent from the physics and chemistry it is framed in as experiments with bacteria as opposed to, say, flatworms, are thought to be.

It is clear that the present paper is not an exhaustive analysis of factors affecting the evolvability of computer programs, but rather the first systematic one (Ray [14] examined four distinct instruction sets, but with only a case study of each). We hope that further studies of this issue within the avida world will ultimately lead to design decisions for an evolvable instruction set in a custom setting that will permit the large-scale evolution of useful, robust, and fault-tolerant, computer code.

Acknowledgment

The authors would like to thank D. Adler for many fruitful discussions, as well as R. E. Lenski and T. Ray for comments on the manuscript. Access to a Beowulf system was provided by the Center for Advanced Computation Research at the California Institute of Technology. Part of this work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] C. Adami, "Learning and complexity in genetic auto-adaptive systems," *Physica D*, vol. 80, no. 1–2, pp. 154–170, Jan. 1995.
[2] ——, *Introduction to Artificial Life*. New York: Springer-Verlag, 1998.
[3] C. Adami, C. Ofria, and T. C. Collier, "Evolution of biological complexity," *Proc. Nat. Acad. Sci. USA*, vol. 97, no. 9, pp. 4463–4468, Apr. 2000.
[4] M. Eigen, "The physics of molecular evolution," in *Molecular Evolution of Life*, H. Baltscheffsky, H. Jörnvall, and R. Rigler, Eds. Cambridge, U.K.: Cambridge Univ. Press, 1986, pp. 13–26.
[5] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Piscataway, NJ: IEEE Press, 1995.
[6] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. Michigan Press, 1975.
[7] M. Huynen, P. F. Stadler, and W. Fontana, "Smoothness within ruggedness: The role of neutrality in adaptation," *Proc. Nat. Acad. Sci. USA*, vol. 93, no. 1, p. 397, Jan. 1996.
[8] J. R. Koza, *Genetic Programming*. Cambridge, MA: MIT Press, 1992.
[9] R. E. Lenski, C. Ofria, T. C. Collier, and C. Adami, "Genome complexity, robustness, and genetic interactions in digital organisms," *Nature*, vol. 400, no. 6745, pp. 661–664, Aug. 1999.
[10] J. Maynard Smith, "Natural selection and the concept of a protein space," *Nature*, vol. 225, no. 232, p. 563, Feb. 1970.
[11] C. Ofria, C. Adami, and T. C. Collier, "Selective pressures on genomes," J. Theor. Biol., submitted for publication.
[12] S. Rasmussen, C. Knudsen, R. Feldberg, and M. Hindsholm, "The coreworld—Emergence and evolution of cooperative structures in a computational chemistry," *Physica D*, vol. 42, no. 1-3, p. 111, June 1990.
[13] T. S. Ray, "An approach to the synthesis of life," in *Proceedings of Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, Eds. Reading, MA: Addison-Wesley, 1992, p. 371.
[14] ——, "Evolution, complexity, entropy, and artificial reality," *Physica D*, vol. 75, no. 1–3, p. 239, Aug. 1994.
[15] A. Thompson, "Evolutionary techniques for fault tolerance," in *Proc. UKACC Int. Conf. Control*, 1996, IEE Conf. Pub. 427, pp. 693–698.
[16] E. van Nimwegen, J. P. Crutchfield, and M. Huynen, "Neutral evolution of mutational robsutness," *Proc. Nat. Acad. Sci. USA*, vol. 96, no. 17, pp. 9716–9720, Aug. 1999.
[17] C. O. Wilke, J. L. Wang, C. Ofria, and R. E. Lenski, "Evolution of digital organisms at high mutation rates leads to survival of the flattest," *Nature*, vol. 412, no. 6844, pp. 331–333, July 2001.