Available online at www.sciencedirect.com
SCIENCE DIRECT®
BioSystems 76 (2004) 291–301
BioSystems
www.elsevier.com/locate/biosystems

Evolving beyond perfection: an investigation of the effects of long-term evolution on fractal gene regulatory networks

Peter J. Bentley

Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK

Received 28 February 2003; received in revised form 11 July 2003; accepted 1 August 2003

Abstract

This paper continues a theme of exploring algorithms based on principles of biological development for tasks such as pattern generation, machine learning and robot control. Previous work has investigated the use of genes expressed as fractal proteins to enable greater evolvability of gene regulatory networks (GRNs). Here, the evolution of such GRNs is investigated further to determine whether evolution exhibits natural tendencies towards efficiency and graceful degradation of developmental programs. Experiments where "perfect" GRNs are evolved for a further thousand generations without the addition of any further selection pressure, confirm this hypothesis. After further evolution, the perfect GRNs operate in a more efficient manner (using fewer proteins) and show an improved ability to function correctly with missing genes. When the algorithm is applied to applications (e.g. robot control) this equates to efficient and fault-tolerant controllers.

Keywords: Fractal proteins; Self-organizing and self-repairing systems; Evolutionary algorithms; Bio-developmental systems; Gene networks

1. Introduction

It is a common misconception held by many that a good design must be the result of conscious analysis, understanding, planning and knowledge. This is, after all, how many see the processes of human design. So when anyone used to human design encounters evolutionary design (in nature or computer), the most common objection is that there is no analysis, understanding, planning or knowledge. "How can random chance produce better designs than our carefully thought out designs?" the critics complain.

Yet it is clear that evolution does create designs that are better in many respects. Natural solutions often exploit the intrinsic properties of materials far more efficiently than human solutions. For example, what we might consider to be just hair—something for trapping air and insulating against cold—nature also uses for diverse purposes such as powerful horns or poison-filled defences. Natural solutions also, almost without exception, display graceful degradation. When damaged, life is designed to carry on working, whether the loss is a limb or a gene. Nature gives her designs an elegant efficiency and ability to survive damage that we can only admire. In this paper, we explore the idea that the reason why natural solutions show these capabilities is because of the process by which they were designed.

When we design something, our designs are clear, unambiguous plans that must followed to the letter. If they are not, then the result will not work. Our designs are *brittle*. Natural designs are not built as one-off plans, to be followed perfectly. Evolution must contend with random perturbations during processes of development, in environments and in reproduction (resulting in mutation). Although driven by this variability, evolution must also protect her solutions against the ravages of its randomness. Therefore, in addition to

E-mail address: p.bentley@cs.ucl.ac.uk (P.J. Bentley).

0303-2647/$ – see front matter © 2004 Elsevier Ireland Ltd. All rights reserved.
doi:10.1016/j.biosystems.2004.05.019

having selection pressure towards solutions that must survive in the short term, evolution endures longer, implicit pressures towards the creation of robust and efficient solutions that will not be lost through genetic drift. It is the hypothesis of this work that an evolutionary algorithm—properly designed and set up—will also exhibit the same natural tendencies towards the improvement of robustness and efficiency of developmental processes. This paper provides evidence that supports this hypothesis.

2. Background

Evolutionary algorithms have long been (incorrectly) regarded simply as optimisers, where the goals are normally to find the globally best solution with the least amount of computation. Should efficient solutions be desired, then additional selection pressures (fitness criteria) are added. For example, to reduce solution size in genetic programming (GP), functions that penalise larger solutions are employed. More recently, research in more advanced evolutionary systems that employ developmental stages from genotype to phenotype is increasing. Researchers such as Hornby (2003), Bongard (2002), and Kumar and Bentley (2003) have demonstrated that various types of development can enable smaller genotypes to represent more complex phenotypes through the ability of development to discover modularities and repetition. However, the idea that evolution may have a natural tendency to improve the efficiency of such developmental processes further does not seem common in the literature.

Other scientists in the field have been focussing on the ability of evolution, and more commonly developmental methods, to enable self-repairing behaviour and graceful degradation of solutions. For example, Sipper (2002) demonstrates a simple self-repair capability for electronic circuits by the use of cellular-automata-like “biodules”. Using ideas inspired from embryology, the circuits can functionally self-organise, while redundant biodules enable damage to be overcome. Similarly the work of Andy Tyrrell and his group create fault-tolerant hardware inspired by ideas of embryology and immune systems (Jackson and Tyrrell, 2002). Adrian Thompson has spent some years investigating how evolvable hardware can provide robust solutions, for example circuits that handle large variations in temperature and fabrication, by testing designs in different environments during evolution (Thompson and Layzell, 2002). More recently, Julian Miller has described experiments evolving developmental programs to create “French Flag” patterns (Miller and Banzhaf, 2003). He shows that development is able to regenerate these patterns should some of their cells be removed. Finally, current work by Mahdavi and Bentley (2003) demonstrates how adaptive evolutionary control can enable a “Smart Snake” to redevelop new movement strategies even after the loss of a crucial muscle (Nitinol wire).

However, it is the work of Thompson (1997) that most resembles the idea investigated here. In his research on fault-tolerant systems, Thompson describes how “graceful degradation for free” can be achieved in theory and in practice for robot controllers, “from the nature of the evolutionary process.” Thompson suggests that mutation-insensitive individuals will, in the long term, survive better, thus producing a pressure towards fault-tolerant solutions. Significantly, these findings have only ever been tested on systems where there is a direct mapping from gene to phenotypic feature and hence from mutation-insensitivity to fault-tolerance. In this work, the use of developmental processes means that there are no direct mappings: pleiotropy and polygeny are prevalent, and genes are reused over many developmental iterations. Nevertheless, it is the hypothesis of this work that through the Baldwin effect, even mutations that cause highly indirect and seemingly inconsequential changes to developmental programs will eventually result in solutions becoming more efficient and fault-tolerant.

The work described in this paper forms part of a project called MOBIUS (modelling biology using smart materials). The aims of this research are to free evolution from the traditional constraints imposed by evolutionary computation (EC). Smart materials will be used to provide a rich environment in which evolution can be embodied (Quick et al., 1999), enabling the intrinsic evolution of potentially unconventional and diverse robot morphologies. Evolution will employ new genetic representations based on natural developmental processes, designed to be evolvable, scalable and free of constraints. Previous papers have described initial research on this topic: how evolved genes can

be expressed into fractal proteins that form themselves into complex and desirable gene regulatory networks, (Bentley, 2003a,b). Current work has shown how such GRNs can be used to learn paths for a robot through a maze.

This paper uses the concept of a fractal gene regulatory network as a framework in which to test the hypothesis that an evolutionary algorithm can exhibit natural tendencies towards the improvement of robustness and efficiency of developmental systems. If shown to be correct, it demonstrates a way of automatically generating efficient and fault-tolerant developmental procedures (that can be applied to applications such as machine learning and robot control).

3. Fractal proteins

Development is the set of processes that lead from egg to embryo to adult. Instead of using a gene for a parameter value as we do in standard EC (i.e., a gene for long legs), natural development uses genes to define proteins. If expressed, every gene generates a specific protein. This protein might activate or suppress other genes, might be used for signalling amongst other cells, or might modify the function of the cell it lies within. The result is an emergent "computer program" made from dynamically forming gene regulatory networks (GRNs) that control all cell growth, position and behaviour in a developing creature (Lewis et al., 2001). In this work, a biologically plausible model of gene regulatory networks is constructed through the use of genes that are expressed into *fractal proteins*—subsets of the Mandelbrot set that can interact and react according to their own fractal chemistry. The motivations behind this work can be listed as follows: (further motivations and discussions on fractal proteins are provided in (Bentley, 2003a,b)).

1. Natural evolution extensively exploits the complexity, redundancy and richness of chemical systems in the design of DNA and the resulting developmental systems in organisms. Providing a computer system with genes that define fractal proteins gives the system complexity, redundancy and richness to exploit.
2. It is extremely difficult and computationally intensive to model natural chemical systems accurately in an artificial chemistry. Fractal proteins have many of the same properties as natural proteins, without any modelling overheads.
3. A fractal protein (with the infinite complexity of the Mandelbrot set) can be defined by just three genes.
4. The "fractal genetic space" is highly evolvable—a small change to a gene produces a small change to the fractal protein, while the self-similarity of fractals ensures that any specific shape can be found in an infinite number of places.
5. When fractal proteins are permitted to interact according to their morphologies, a hugely complex (and eminently exploitable) fractal chemistry emerges naturally.
6. Calculating subsets of Mandelbrot sets is *fast* so there is little overhead.

It should be noted that the system used for the experiments in this paper is an extended version of systems described elsewhere. While the detail provided below is in some respects a distraction from the theme of this paper, for reasons of reproducibility it is provided in full.

3.1. Representation

Currently in this representation, there exist:

fractal proteins, defined as subsets of the Mandelbrot set;
environment, which can contain one or more fractal proteins (expressed from the environment gene(s)), and one or more *cells*;
cell, which contains a *genome* and *cytoplasm*, and which has some *behaviours*;
cytoplasm, which can contain one or more fractal proteins;
genome, which comprises *structural genes* and *regulatory genes*. In this work, the structural genes are divided into different types: *cell receptor genes*, *environment genes* and *behavioural genes*;
regulatory gene, comprising operator (or promoter) region and coding (or output) region;
cell receptor gene, a structural gene with a coding region which acts like a mask, permitting variable portions of the environmental proteins to enter the corresponding cell cytoplasm;

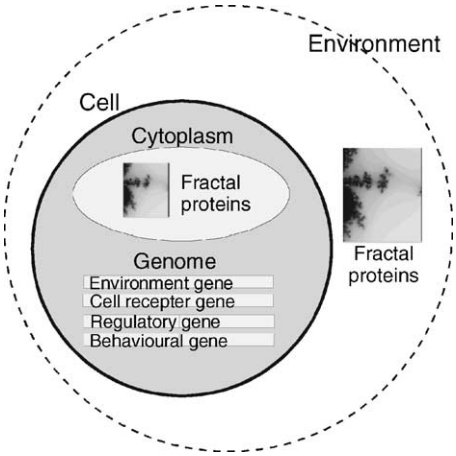

Fig. 1. Representation using fractal proteins.

environment gene, a structural gene which determines which proteins (maternal factors) will be present in the environment of the cell(s);
behavioural gene, a structural gene comprising operator region and a cellular behaviour region.

Fig. 1 illustrates the representation. Fig. 2 provides an overview of the algorithm used to develop a phenotype from a genotype. Note how most of the dynamics rely on the interaction of fractal proteins. Evolution is used to design genes that are expressed into fractal proteins with specific shapes, which result in developmental processes with specific dynamics.

FRACTAL DEVELOPMENT

For every developmental time step:
  For every cell in the embryo:
    Express all environment genes and
    calculate shape of merged environment fractal proteins
    Express cell receptor genes as receptor fractal proteins
    and use each one to mask the merged environment proteins
    into the cell cytoplasm.
    If the merged contents of the cytoplasm match a promoter
    of a regulatory gene, express the coding region of the gene,
    adding the resultant fractal protein to the cytoplasm.
    If the merged contents of the cytoplasm match a promoter of a
    behavioural gene, use coding region of the gene to specify a
    cellular function.
    Update the concentration levels of all proteins in the cytoplasm.
    If the concentration level of a protein falls to zero, that protein
    does not exist.

Fig. 2. The fractal development algorithm.

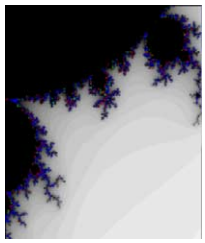

Fig. 3. Example of a fractal protein defined by ($x = 0.132541887$, $y = 0.698126164$, $z = 0.468306528$).

3.2. Defining a fractal protein

In more detail, a fractal protein is a finite square subset of the Mandelbrot set, defined by three codons (x, y, z) that form the coding region of a gene in the genome of a cell. Each (x, y, z) triplet is expressed as a protein by calculating the square fractal subset with centre coordinates (x, y) and sides of length z, see Fig. 3 for an example. In this way, it is possible to achieve as much complexity (or more) compared to natural protein folding in nature.

In addition to shape, each fractal protein represents a certain *concentration* of protein (from 0 meaning "does not exist" to 200 meaning "saturated"), determined by protein production and diffusion rates.

3.3. Fractal chemistry

Cell cytoplasms and the environment usually contain more than one fractal protein. In an attempt to harness the complexity available from these fractals, multiple proteins are merged. The result is a product of their own "fractal chemistry" which naturally emerges through the fractal interactions.

Fractal proteins are merged (for each point sampled) by iterating through the fractal equation of all proteins in "parallel", and stopping as soon as the length of any is unbounded (i.e. greater than 2). Intuitively, this results in black regions being treated as though they are transparent, and paler regions "winning" over darker regions. See Fig. 4 for an example.

3.4. Calculating concentration levels

The total concentration of two or more merged fractal proteins is the mean of the different concentrations

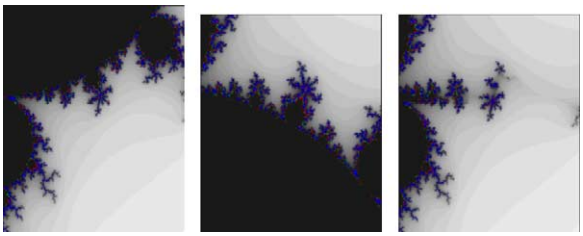

Fig. 4. Two fractal proteins (left and middle) and the resulting merged fractal protein combination (right).

seen in their merged product. For example, Fig. 4 shows how fractal proteins are merged to form a new fractal shape. Fig. 5 illustrates the resultant areas of different concentration in the product. When being compared to the (*xp*, *yp*, *zp*) promoter region of a gene (the "conditional" part of the gene to be matched, see later section on genes), the concentration seen on that promoter is described by all those regions that "fall under" the promoter, see Fig. 5. In other words, the merged product is masked by the promoter fractal, and the total concentration on the promoter is the mean of the resulting concentrations, see Fig. 6.

3.5. Updating protein concentration levels

Every developmental time step, the new concentration of each protein is calculated (synchronously). This is formed by summing two separate terms: the previous concentration level after diffusion (*Dc*) and the new concentration output by a gene (*Gc*). These two terms model the reduction in concentration of proteins over time, and the production of new proteins

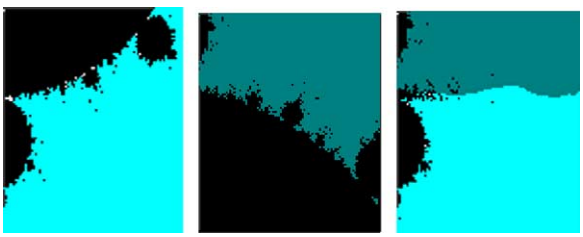

Fig. 5. The different concentrations of the two fractal proteins (left and middle) and the concentration levels in their merged product (right).

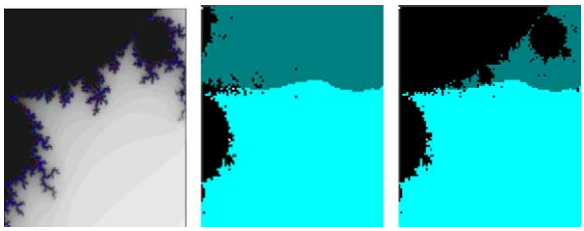

Fig. 6. The shape of the desired protein as defined by a promoter (left), the shape and concentration levels of merged proteins in the cytoplasm (middle) and the concentration levels seen on that promoter (right), where total concentration is taken as mean. Note that although a merged protein may decrease affinity (similarity) to the promoter, should the second protein have a higher concentration level to the first, it will boost overall concentration seen by the promoter, i.e., act like a catalyst to speed up (or slow down, if lower) the "reaction".

over time, respectively, where

$$Dc = Pc - Pc/\mathrm{C_p} + 0.2$$

Pc is protein concentration in previous time step, $\mathrm{C_p}$ is a constant normally set to 5, the final addition of 0.2 ensures a minimum level of diffusion and:

$$Gc = Tc \times Cm,$$

Tc is the mean concentration seen at the promoter, *Cm* is a concentration multiplier, where:

$$Cm = \tanh((Tc - ct)/\mathrm{C_w})/\mathrm{C_i}$$

where *ct* is the concentration threshold from the gene promoter, $\mathrm{C_w}$ is a constant (set to 30 for these experiments), $\mathrm{C_i}$ is a constant (set to 2 for these experiments).

3.6. Genes

The environment gene, cell receptor gene, regulatory genes, and behavioural genes all contain seven real-coded values:

xp	*yp*	*zp*	Affinity threshold	Concentration threshold	*x*	*y*	*z*	Type

where *xp*, *yp*, *zp*, *affinity threshold*, *concentration threshold* defines the promoter (operator or precondition) for the gene and (*x*, *y*, *z*) defines the coding region of the gene. The *type* value defines which type of gene is being represented, and can be one or all of the following: *environment*, *receptor*, *behavioural*,

or *regulatory*. This enables the type of genes to be set independently of their position in the genome, enabling variable-length genomes. It also enables genes to be multi-functional, i.e. a gene might be expressed both as an environmental protein and a behaviour.

When *affinity threshold* is a positive value, one or more proteins must match the promoter shape defined by (*xp*, *yp*, *zp*) with a difference equal to or lower than *affinity threshold* for the gene to be activated. When *affinity threshold* is a negative value, one or more proteins must match the promoter shape defined by (*xp*, *yp*, *zp*) with a difference equal to or lower than |*affinity threshold*| for the gene to be repressed (not activated).

To calculate whether a gene should be activated, all fractal proteins in the cell cytoplasm are merged (including the masked environmental proteins, see later) and the combined fractal mixture is compared to the promoter region of the gene.

The similarity between two fractal proteins (or a fractal protein and a merged fractal protein combination) is calculated by sampling a series of points in each and summing the difference between all the resulting values. (Black regions of fractals are ignored.) Given the similarity matching score between cell cytoplasm fractals and gene promoter, the activation probability of a gene is given by

$$Pa = (1 + \tanh((m - At - C_t)/C_s))/2$$

where m is the matching score, At is *Affinity threshold* (the matching threshold from the gene promoter), C_t is a threshold constant (normally set to 50), C_s is a sharpness constant (normally set to 50).

3.6.1. Regulatory gene

Should a regulatory gene be activated by other protein(s) in the cytoplasm (which have concentrations above 0) matching its promoter region, its corresponding coding region (*x*, *y*, *z*) is expressed (by calculating the subset of the Mandelbrot set) and new concentration level calculated. To do this, the concentration of the resulting protein is modified by incrementing with *geneoutputconc*, the result of a function of the concentration threshold (*ct*) and the mean total concentration seen at the gene promoter (*totalconc*), as given in Section 3.5. In this way, higher concentrations of protein on the promoter will cause an increased rate of output protein concentration growth, while lower concentrations (below the *ct* threshold) will increase the diffusion rate of the output protein (its concentration will decrease at a higher rate).

The cell cytoplasm, which holds all current proteins, is updated at the end of the developmental cycle.

3.6.2. Cell receptor gene

At present, the promoter region of the cell receptor gene is ignored, and this gene is always activated. As usual, the corresponding coding region (*x*, *y*, *z*) is expressed by calculating the subset of the Mandelbrot set. However, the resultant fractal protein is treated as a mask for the environmental proteins, where all black regions of the mask are treated as opaque, and all other regions treated as transparent. For an example, see Fig. 7. If there is more than one receptor gene, only the first in the genome is used.

3.6.3. Environment gene

Like the cell receptor gene, this gene is always activated. It produces environmental factors for all cells: fractal proteins of concentration 200. If there is more than one environmental gene, the expressed environmental proteins are merged before being masked by the receptor protein.

3.6.4. Behavioural gene

A behavioural gene is activated when other protein(s) in the cytoplasm match its promoter region and the overall concentration is above its *concentration threshold* value. Instead of the coding region (*x*, *y*, *z*) being expressed as a protein, these three real values are decoded to specify a range of different cellular functions, depending on the application.

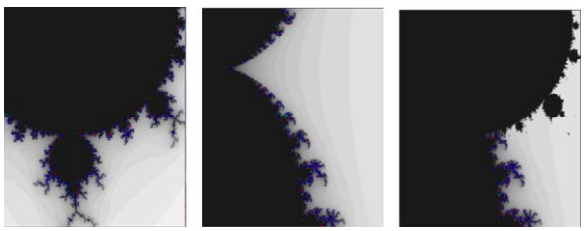

Fig. 7. Cell receptor protein (left), environment protein (middle), resulting masked protein to be combined with cytoplasm (right).

If there are more behavioural genes than are required, only the first encountered in the genome are used.

3.7. Fractal sampling

All fractal calculations (masking, merging, comparisons) are performed at the same time, by sampling the fractals at a resolution of 15 × 15 points. Note that the comparison is normally performed between the single fractal defined by (*xp*, *yp*, *zp*) of a gene and the merged combination of all other proteins currently in the cytoplasm. The fractal being compared is treated a little like the cell receptor mask—only those regions that are not black are actually compared with the contents of the cytoplasm.

3.8. Development

As was illustrated in Fig. 2, an individual begins life as a single cell in a given environment. To develop the individual from this zygote into the final phenotype, fractal proteins are iteratively calculated and matched against all genes of the genome. Should any genes be activated, the result of their activation (be it a new protein, receptor or cellular behaviour) is generated at the end of the current cycle. Development continues for *d* cycles, where *d* is dependent on the problem. Note that if one of the cellular behaviours includes the creation of new cells, then development will iterate through all genes of the genome in all cells.

3.9. Evolution

The genetic algorithm used in this work has been used extensively elsewhere for other applications (including GADES (Bentley, 1999)). A dual population structure is employed, where child solutions are maintained and evaluated, and then inserted into a larger adult population, replacing the least fit. The fittest *n* are randomly picked as parents from the adult population. The degree of negative selection pressure can be controlled by modifying the relative sizes of the two populations. Likewise the degree of positive selection pressure is set by varying *n*. When child and adult population sizes are equal, the algorithm resembles a canonical or generational GA. When the child population size is reduced, the algorithm resembles a steady-state GA. Typically the child population size is set to 80% of the adult size and $n = 40\%$. (For further details of this GA, refer to (Bentley, 1999).)

Unless specified, alleles are initialised randomly, with (*xp*, *yp*, *zp*) and (*x*, *y*, *z*) values between −1.0 and 1.0 and *thresh* between −10,000 and 10,000. The ranges and precision of the alleles are limited only by the storage capacity of *double* and *long* 'C' data types—no range constraints were set in the code.

3.9.1. Genetic operators

Genes are real-coded, but genomes may comprise variable numbers of genes. Given two parent genomes, the crossover operator examines each gene of parent1 in turn, finding the most similar gene of the same type in parent 2. Similarity is measured by calculating the differences between values of operator and coding regions of genes. One of the two genes is then randomly allocated to the child. If the genome of parent 2 is shorter, the child inherits the remaining genes from parent 1. If the genomes are the same length, this crossover acts as uniform crossover.

Mutation is also interesting, particularly since these genes actually code for proteins in this system. There are four main types of mutation used here

1. Creep mutation, where (*xp*, *yp*, *zp*) and (*x*, *y*, *z*) values are incremented or decremented by a random number between 0 and 0.5, *affinity threshold* is incremented or decremented by a random number between 0 and 16,384 and *concentration threshold* is incremented or decremented by a random number between 0 and 200.
2. Duplication mutation, where a (*xp*, *yp*, *zp*) or (*x*, *y*, *z*) region of one gene randomly replaces a (*xp*, *yp*, *zp*) or (*x*, *y*, *z*) of another gene. (This permits evolution to create matching promoter regions and coding regions quickly).
3. Gene mutation, where a random gene in the genome is either removed or a duplicate added.
4. Sign flip mutation, where the sign of *affinity threshold* is reversed.

Crossover is always applied; all mutations occur with probability 0.01 per gene.

4. Experiments

A single cell with 1 environment gene, 1 receptor gene, 1 behavioural gene and 6 regulatory genes was used. (Note that with variable length genomes, evolution was free to modify these numbers). The operator and coding regions of the genes were randomly initialised with the alleles that defined 10 previously evolved protein fractals (Bentley, 2003a). 16 developmental steps were employed, and the evolutionary algorithm ran for up to 1000 generations. If a perfect GRN evolved (i.e., one in which the state of the behavioural gene matches the desired output pattern at every developmental step), the system then continued to evolve for a further 1001 generations. All other parameters were as described above.

A simple fitness function was used: it measured the output from the behavioural gene at every developmental step, and calculated how much this output deviated from a predefined pattern, see Table 1. In addition, the number of incorrect positive and negative edges in the pattern of ons and offs was measured, with penalties given in proportion to the number wrong. Note that the fitness function only evaluated correctness of the GRN pattern, it did not measure efficiency or robustness in any way. The system was run 20 times; the *type* value in genes was not evolved.

Although the desired pattern may seem elementary, it involves switching on the behavioural gene on step 5 (i.e., counting to four), switching it off on step 9 (counting to four again), and then on again on step 13 (after counting to four yet again). Clearly if the GRN can learn to repeat the pattern from 1 to 8 on steps 9–16, then things are slightly simpler, but there is still significant complexity here. (Further work has demonstrated how evolution of other developmental patterns can be used for robot control, where the pattern determines the path taken by the robot through a maze. Here, we focus on a more regular and easier-to-analyse target.)

5. Results and analysis

The results were fascinating and in some respects surprising. Out of 20 runs, 12 evolved perfect GRN patterns (the others all were very nearly perfect). Table 2 provides details of the results. Because of the complexity of fractal protein interactions, a full analysis of how even a simple GRN operates is an extensive undertaking. Purely for illustrative purposes, Fig. 8 shows the changes in protein concentrations over time for two solutions to this problem; a full analysis of fractal protein interactions is given in (Bentley, 2003a).

For this work, to assess the efficiency of the GRNs, they were analysed to determine

1. how many proteins had concentrations above zero (any with concentrations of zero for all 16 developmental steps mean that the protein is never used, hence the corresponding regulatory gene has no effect);
2. how many genes could be individually removed in turn from the genome without detrimentally affecting the activation pattern by the behavioural gene.

The second-to-last run in Table 2 provides a nice example of the effects of further evolution. Here the perfect pattern was found after 36 generations, with the total number of genes, and number of genes of each type unchanged from the start. The analysis shows that 5 of the 6 regulatory genes are used in this solution, and out of the total 9 genes in the genome, only 2 could be removed without damaging the pattern. After 1037 generations, the genome still contains 9 genes, but now there are 2 environment, 2 behavioural and only 4 regulatory. The analysis shows that only 4 regulatory genes are now being used to generate the pattern, and because of the use of redundancy, 4 out of 9 genes could be individually removed without damaging the pattern.

Table 1
Desired output pattern over 16 developmental steps to be created by GRN in developing phenotype ('na' means behavioural gene not activated, 'a' means b-gene activated)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
na	na	na	na	a	a	a	a	na	na	na	na	a	a	a	a

Table 2
Number of generations, genes in genomes, proteins with positive concentrations in the GRN, and robustness of the 12 perfect solutions

Perfect solution after *n* generations:	Number of genes (env, rec, beh, reg)	No. of proteins used	Genes that can be individually removed without affecting solution
34	1, 1, 1, 6	5	1 (/9)
1035	1, 1, 2, 5	4	4 (/9)
68	1, 1, 1, 7	6	3 + 1 near-perfect score (/10)
1069	2, 1, 1, 6	4	3 + 1 near perfect score (/9)
78	1, 1, 1, 6	5	2 (/9)
1079	2, 1, 1, 7	4	5 (/10)
39	1, 1, 1, 6	2	4 (/9)
1040	1, 1, 3, 5	2	5 (/10)
16	1, 1, 1, 6	4	2 (/9)
1017	1, 1, 4, 4	2	5 (/10)
36	1, 1, 1, 6	6	0 + 2 near perfect scores (/9)
1037	1, 1, 2, 10	5	7 + 1 near perfect score (/14)
40	1, 1, 1, 6	3	3 (/9)
1041	1, 1, 1, 4	3	1 (/7)
47	1, 1, 1, 7	5	3 + 1 near perfect score (/10)
1048	2, 1, 1, 8	4	5 + 1 near perfect score (/12)
164	1, 1, 1, 6	4	4 (/9)
1165	1, 2, 1, 8	2	8 (/12)
292	1, 1, 1, 6	3	3 + 1 near perfect (/9)
1293	2, 1, 1, 6	3	4 + 1 near perfect (/10)
36	1, 1, 1, 6	5	2 (/9)
1037	2, 2, 1, 4	4	4 (/9)
200	1, 1, 1, 7	3	4 + 1 near perfect (/10)
1201	1, 1, 2, 6	2	5 (/10)

As can be seen in Table 2, in every case further evolution improves the "perfect" solutions by

- *increasing efficiency:* reducing the number of proteins (actual genes used to make the solution) and/or;
- *increasing robustness:* increasing the percentage of genes that can be removed without affecting the solution (either by reducing number of proteins or by introducing redundant duplicates).

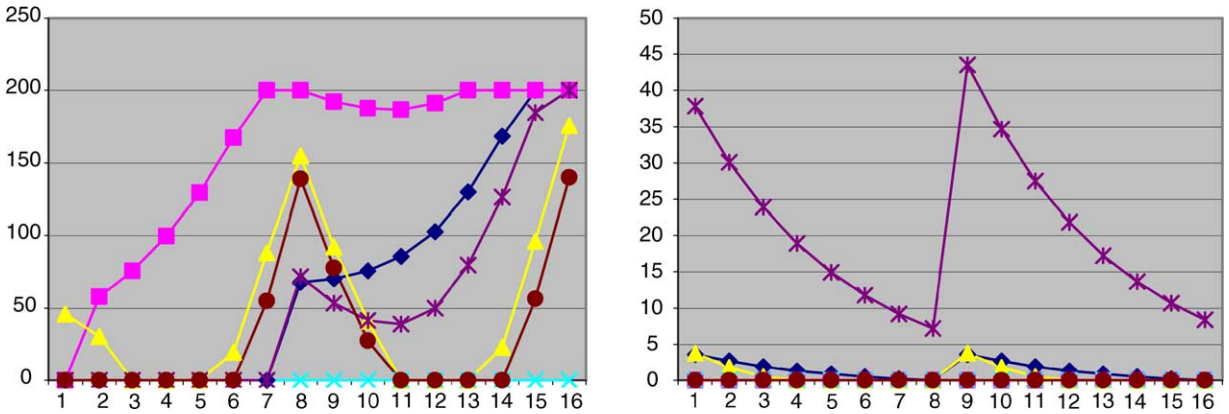

Fig. 8. Protein concentration levels during development of two perfect solutions to the problem. Note how radically different dynamics can produce the same behavioural gene pattern.

A typical evolutionary run seems to find the desired GRN pattern by merging the effects of many different genes and proteins. Once found, the perfect solution is fine-tuned by removing the genes that have smaller effects resulting in shorter genomes, followed sometimes by intron growth. Without any additional selection pressures, the natural tendency of this system is to reduce the number of genes used in solutions and increase the ability of the solutions to survive damage.

Finally, it is clear that this system, although allowing variable genomes, does not behave in the same manner as GP systems. By the end of the further 1001 generations, in 7/12 runs evolution had added one or more non-functioning genes (intron growth or bloat). However, perhaps because of the high reactivity of fractal proteins with each other, or perhaps because probability of mutation is fixed per gene and not per genome so larger genomes do not imply fewer average mutations, in the other five cases the total number of genes remained constant or even fell. Also note that although intron growth is occasionally apparent, it seems likely that the presence of introns would enable quicker evolution of new solutions should the selection pressure change towards other patterns. So even the apparently wasteful addition of introns may actually be a useful feature in this kind of system.

6. Conclusions

Researchers are now focussing their attention on the creation of technologies with some of the same features of natural systems. This paper has focussed on two of these capabilities: efficiency and graceful degradation, in the context of gene regulatory networks. It has been shown that given 1000 generations of evolution after a "perfect" solution is found, an evolutionary algorithm that generates fractal gene regulatory networks within a developmental process has a natural tendency towards the creation of efficient and compact solutions by reducing the number of genes and proteins employed within solutions. It has also been shown that the same system has a natural tendency towards more robust solutions, increasing the ability of GRNs to survive damage by the removal of genes.

These tendencies were not explicitly demanded by selective pressures, nor were they deliberately "designed into" the system. It is conceivable that improvements of performance may be possible if these measures are taken. However, the important finding of this work is the utility of evolution as a method of design. As shown by Thompson (1997) and Eigen (1987) in simpler systems, evolution can provide fault tolerance "for free." This research shows that the same findings are evident for evolutionary development. Over long time-scales, randomness causes evolution to preserve her developmental programs by making them more efficient and giving them abilities to work even when damaged. When evolutionary developmental algorithms such as this are applied to applications (e.g. robot control) this equates to efficient and fault-tolerant controllers.

Acknowledgements

Thanks to Sanjeev Kumar for his comments. This material is based upon work supported by the European Office of Aerospace Research and Development (EOARD), Airforce Office of Scientific Research, Airforce Research Laboratory, under contract no. F61775-02-WE014. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of EOARD. MOBIUS is an project.

References

Bentley, P.J., 2003a. Fractal proteins. To appear in Genet. Program. Evol. Machines J.

Bentley, P.J. 2003b. Evolving fractal proteins. In: Proceedings of ICES'03, the Fifth International Conference on Evolvable Systems: from Biology to Hardware.

Bentley, P.J., 1999. From coffee tables to hospitals: generic evolutionary design. In: Bentley, P.J. (Ed), Evolutionary Design by Computers, chapter 18. Morgan Kaufmann Publishers, San Francisco, pp. 405–423.

Bongard, J.C., 2002. Evolving Modular Genetic Regulatory Networks. In: Proceedings of the IEEE 2002 Congress on Evolutionary Computation (CEC2002). IEEE Press, pp. 1872–1877.

Eigen, M., 1987. New concepts for dealing with the evolution of nucleis acids. In: Cold Spring Harbor Symposium on Quantitative Biology, vol. LII.

Hornby, G.S., 2003. Ph.D. dissertation. Generative representations for evolutionary design automation. Department of Computer Science, Brandeis University.

Jackson, A.H., Tyrrell, A.M., 2002. Implementing asynchronous embryonic circuits using AARDVArc. In: Proceedings of 2002

NASA/DoD Conference on Evolvable Hardware (EH-2002). IEEE Computing Society, Alexandria, Virginia, 15–18 July 2002, pp. 231–240.

Kumar, S., Bentley, P.J., 2003. Computational embryology: past, present and future. In: Ghosh, Tsutsui (Eds.), Theory and Application of Evolutionary Computation: Recent Trends. Springer Verlag, UK.

Mahdavi, S., Bentley, P.J., 2003. Adaptive evolutionary motion of smart robots. To appear in EvoROB2003, second European Workshop on Evolutionary Robotics.

Miller, J., Banzhaf, W., 2003. Evolving the program for a cell: from French flags to Boolean circuits. In: Kumar, S., Bentley, P.J. (Eds.), On Growth, Form and Computers. Academic Press (to appear as an invited chapter).

Quick, T., Dautenhahn, K., Nehaniv, C., Roberts, G., 1999. The essence of embodiment: a framework for understanding and exploiting structural coupling between system and environment. In: Proceedings of the Third International Conference on Computing Anticipatory Systems (CASYS'99), Symposium 4 on Anticipatory, Control and Robotic Systems. Liege, Belgium, pp. 16–17.

Sipper, M., 2002. Machine Nature: the Coming of Age of Bio-Inspired Computing. McGraw-Hill, New York.

Thompson, A., 1997. Evolving inherently fault-tolerant systems. In: Proceedings of the Institutional Mechanical Engineers.

Lewis, W., Beddington, R., Jessell, T., Lawrence, P., Meyerowitz, E., Smith, J. Principles of Development, second ed. Oxford University Press.