A Biologist's View of Software Evolution

DeLesley Hutchins

CISA, University of Edinburgh
d.s.hutchins@sms.ed.ac.uk

Abstract. The term "software evolution" is generally used as an analogy for biological evolution. This paper explores that analogy in more depth, analyzing software evolution from the biologist's point of view. I give a basic introduction to genetic algorithms, and describe two major issues that effect evolvability: local optima in the search space, and genotype to phenotype mappings. Aspects and meta-programs address both of these issues. Encapsulation can be seen as a technique for reducing epistasis, while meta-programming can be seen as an analog to morphogenesis.

1 Introduction

The term "software evolution" is usually used by way of analogy with biological evolution. Biological species gradually change over the course of tens of millions of years, adapting in response to changing environmental conditions. A piece of software likewise changes over time, in response to changing requirements and needs.

What distinguishes "software evolution" from other forms of software development is that each version of an evolving program must be valid. It must at least compile and run, even if it is not entirely bug-free. This discipline is enforced by weekly or even daily builds. In an truly evolutionary process, such as that advocated by Extreme Programming [3], it is not possible to do major code rewrites. Updates must be kept small, and each update must map from a working program to another, similar, working program.

This, too, is analogous to biological evolution. Each organism in the real world must produce viable offspring which are capable of surviving on their own. Major change occurs only after thousands of generations.

Is this just an analogy, or is there something deeper? The purpose of this paper is to look at the problem from a biologist's point of view. As it turns out, there is strong evidence which supports the idea that aspects [4] [10] and meta-programs are, indeed, important tools for software evolution.

2 Genetic Algorithms

One problem with applying biological models to software is that our current understanding of the genome is extremely limited. Although the DNA sequences for several species are now available, there is no "road map" of gene function, or even which portions of the genome encode useful information. Moreover, we can only extract DNA from living creatures — the genomes of past organisms are unavailable.

Genetic algorithms (GAs) are computational models which abstract away from the biochemical details of DNA. [5] [6] [12] GAs treat evolution as a form of *search* through the space of all possible solutions.

Let $\mathbb{S}$ be the set of all possible solutions to a problem. These solutions can be numbers, strings, graphs or even programs — the definition of what constitutes a "solution" depends on the problem being solved.

The cannonical GA is defined by the following:

A **fitness function** $f : \mathbb{S} \to \mathbb{R}$ computes a numeric value $f(s)$ for every solution $s \in \mathbb{S}$. This value, called the *fitness* of the solution, is a measure of "how good" the solution is. If $f(s_1) > f(s_2)$, then s_1 is a better solution than s_2.

A **representation function** $g : \mathbb{B}^n \to \mathbb{S}$ defines a binary encoding for solutions. This function takes a binary string of length n, and interprets it as a solution s. Usually this function is one-to-one and onto, which means that every solution in the search space has a unique binary encoding.

Genetic operators, namely crossover and mutation, generate new binary strings. The mutation operator flips a bit at random. The crossover operator mixes two different strings together in a process that mimics sexual recombination.

A GA performs search by using a population $b_1..b_m$ of binary strings, which are initially distributed randomly throughout the search space. At each iteration, the GA does the following:

- Calculate the fitness $f(g(b_i))$ of each binary string b_i in the population.
- Select 2 of the better strings – call them b_x and b_y. These are chosen at random, but the choice is weighted by fitness, so better solutions are more likely to be chosen.
- Use crossover and/or mutation to generate a new string from b_x and b_y, and add that new string to the population.
- Select the worst string, and remove it from the population.
- Repeat.

This algorithm is intended to mimic the process of Darwinian natural selection. The binary strings are artificial genomes. Each genome is interpreted to generate a solution, and that solution is then tested with a fitness function. The bad ones die off, while the good ones survive and reproduce.

2.1 Programming as Search

To see how this model relates to programming, consider a GA that operates on a population of size 1, using only mutation. Each iteration, the algorithm will flip a bit at random, and test the resulting solution. If the new solution is better than the old one, then the old one is discarded.

This form of search is called *hillclimbing*. The mutation operator defines a distance metric on the search space, where the distance between two solutions is the minimum number of mutations needed to get from one to the other. The fitness function defines a gradient. The search proceeds by taking an initial solution s, and incrementally improving it by following the fitness gradient. A good visual metaphor is that of a hiker who climbs to the top of a mountain, step by step, by always walking uphill.

Hillclimbing is a reasonably good model of a single human programmer. The program requirements constitute a fitness function. The programmer can be regarded as an "intelligent mutator". Whereas a GA relies on natural selection, a human can intelligently choose which modifications to make. This eliminates a great deal of trial and error, but the overall process of incremental improvement remains the same.

A GA with a population of size N corresponds to a development team of N people, who are all working on the same application. In this case, the application may have

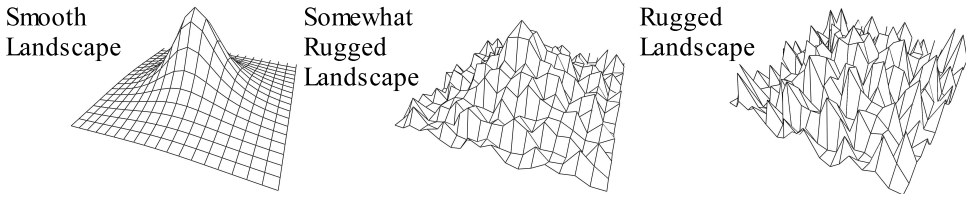
multiple branches or forks, which need to be merged together. The crossover operator is responsible for mixing code from one branch with code from another. Genetic crossover takes half the genes from one parent, half from the other, and then tests the result via natural selection. After many such trials, only the combinations which contain the "good genes" from both parents will survive.

3 Local Optima and Fitness Landscapes

The fundamental limitation of hillclimbing is that the search can get stuck in local optima. A local optimum is a solution s, which is not the best solution available, but for which all mutations result in a worse solution. The metaphor is that of a hiker who reaches the top of a small foothill. She can no longer proceed by going uphill; instead she must go down, off the hill, before she can climb the larger mountain.

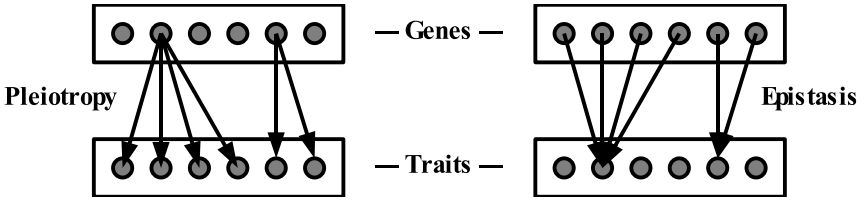
A software engineer encounters a local optimum whenever a small change to one part of the source code is not sufficient to create a working program. If the source code must be modified in several places simultaneously to keep it from breaking, then each such change corresponds to a "downhill step" necessary to get out of the local optimum. Human programmers are intelligent enough to make such simultaneous changes, even rewriting large amounts of code if need be, but true evolutionary systems do not have that luxury.

Genetic search spaces are often called *fitness landscapes*. [9] [8] A *rugged* landscape is one with many local optima, where the fitness gradient varies rapidly and is often misleading. Such landscapes are difficult to search. Landscapes with few optima and smooth fitness gradients are easier to search.



The principles of *pleiotropy* and *epistasis* are two of the main factors which influence how rugged the landscape is going to be. [2] [15] A gene is *pleiotropic* if it influences multiple unrelated traits in an organism. A group of genes are *epistatic* if they all interact in some way to control a single trait in an organism.

In the case of pleiotropy, the gene is hard to modify because a positive adaptation in one trait will likely cause a negative adaptation in another. In the case of epistasis, a change to one gene will influence the effects of all the others, with unpredictable results. The whole set of genes has to be changed at the same time, which cannot be done in gradual steps. These sorts of dependencies create local optima because a single change to one gene is much more likely to be disruptive, thus reducing overall fitness.



In a nutshell, this is a biologist's argument for modularity and encapsulation. A piece of code which is properly encapsulated behind a well-defined interface will have minimal dependencies on the rest of the code base. It is easier to make changes to such code without breaking anything else.

4 Genotype to Phenotype Mappings

The definition of a genetic algorithm given above involves two functions – a fitness function, and a representation function. Early work with GAs ignored the representation function; fitness was defined directly on genomes. It was assumed that the difficulty of searching for a solution was determined by the overall difficulty the problem being solved. One of the major surprises of this early work was that representation mattered a great deal. It is possible to transform an easy problem into a hard one, or vice versa, by altering the way in which solutions are encoded on the genome. [14]

For example, an integer can be encoded using either standard binary, or gray coding. A gray code is an encoding in which adjacent integers are a single bit flip apart, which seems to improve hill-climbing performance on some problems.

A more sophisticated example arises in genetic programming, where programs are represented as abstract syntax trees. Rather than encode such trees as a linear string of characters, they are represented directly as trees, and the mutation and crossover operators are modified so that that they operate on trees rather than bit-strings. [11] This dramatically improves performance.

Using an explicit representation function highlights the difference between genotype and phenotype. [1] [15] Genetic operators such as mutation and crossover operate on *genotypes*, which for biological organisms is DNA. The fitness function, however, evaluates *phenotypes* — the physical body of the organism. A phenotype can be characterized by a collection of *traits*, which are adaptations that directly contribute to fitness. For example, thick fur and blubber are survival traits for animals in the arctic. Branching angle and leaf size are traits of plants which compete for sunlight.

If there is a *natural* mapping between genotype and phenotype, then each gene (or small group of genes) in the genotype will correspond to a single trait in the phenotype. In this case, natural selection for traits translates directly to selection on the appropriate genes.

Note that a natural mapping is not the same as a *direct* mapping, where each gene maps to one specific part of the phenotype. In the case of thick fur, there is not a different gene for every hair on the body. Instead, a few genes control the length of all hairs. It is the overall thickness of the fur, not the length of an individual hair, that is the adaptive trait. The same is true of plants. A fern has many branches, but the angle of each branch is the same.

4.1 Morphogenesis = Meta-Programming

Multicellular organisms create a natural mapping between genotype and phenotype by means of *morphogenesis*: the process of growth and development, whereby a single cell repeatedly divides and differentiates to build an entire organism. This process is algorithmic. [13] While the exact mechanisms of morphogenesis are not well understood, we do know that a significant percentage of genes are active only during development. Developmental genes create chemical gradients, and switch on and off in response to signals

produced by other genes, thus producing the complex structures we see in living things. Because each cell in the body has a complete copy of the DNA, a single gene can express itself in many different physical locations.

Aspects and meta-programs serve the same role in software evolution that morphogenesis does in biological evolution. The phenotype of a program is the form that is actually executed by the CPU — a sequence of instructions in machine code. The genotype is the form that human programmers modify — the source code. A language like C has a fairly direct mapping between source code and machine code; every function or statement can be translated almost directly to (unoptimized) assembly. But as research with GAs has shown, direct mappings from genotype to phenotype are not necessarily evolvable.

Aspects and meta-programs introduce a more sophisticated genotype to phenotype mapping. A meta-program algorithmically generates an implementation that is quite different from the source code. This is ideal for situations such as parser generators and DSLs, where a great deal of repetitive code needs to be produced, but where the repetition cannot be encapsulated into simple loops or functions. Aspects are similar. An aspect can weave advice (such as logging code) throughout an application, thus algorithmically generating an implementation.

Work on evolving neural-networks suggests that generating solutions algorithmically does, in fact, lead to more modular and evolvable designs. [7]

5 Conclusion

The word "evolution" in software evolution is more than just an analogy. The set of program requirements constitute a fitness function, and the space of all possible programs constitute a fitness landscape. The act of developing and maintaining a piece of software plots a path across this landscape. Software maintenance can be seen as a form of human-guided search for a program that meets the specified requirements.

Encapsulation and modularity are basic techniques which improve software evolvability by "smoothing" the fitness landscape, and reducing the number of local optima. Parallel maintenance problems and major code rewrites are a symptoms of local optima. All forms of modularity, whether they be functions, classes, or aspects, are useful in this regard.

Aspects and meta-programs go beyond simple modularity, however, because they alter the genotype to phenotype mapping. By introducing a different mapping, it is possible to completely transform a rugged landscape into a smooth one.

The most sophisticated examples of such mappings are *domain-specific languages*. DSLs abstract away from the implementation language entirely; there is little similarity between the source code (the genotype), and the executable code (the phenotype). In a well-designed DSL, terms in the DSL map directly to concepts in the problem domain. Program requirements (the fitness function) are also domain-specific, which means that there is a natural mapping between terms in the DSL, and fitness-correlated traits.

The main weakness of current DSLs is that the translation from the DSL to the implementation language is fixed. Program requirements change over time, and this means that the DSL itself may have to change too. The clear lesson for software engineers is that we need meta-programming tools which allow DSLs to be easily constructed and modified. Good integration between the DSL and the implementation language remains a challenge.

References

1. Lee Altenberg. Genome growth and the evolution of the genotype-phenotype map. Evolution and Biocomputation: Computational Models of Evolution, 1995.
2. Lee Altenberg. Nk fitness landscapes. T. Back, D. Fogel, Z Michalewicz. editors. Handbook of Evolutionary Computation, Section B2.7.2, 1997.
3. Kent Beck and Cynthia Andres. *Extreme Programming Explained*. Addison-Wesley, 2004.
4. G. Kiczales et. al. Aspect-oriented programming. Proceedings of ECOOP, 1997.
5. David E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Reading: Addison-Wesley, 1989.
6. David E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Boston, MA, USA, 2002.
7. Frederic Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 318–325, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
8. Stuart A. Kauffman. Adaptation on rugged fitness landscapes. Lectures in the Sciences of Complexity. Addison-Wesley, 1989.
9. Stuart A. Kauffman and S. Levin. Towards a general theory of adaptive walks on rugged landscapes. Journal of Theoretical Biology 128: 11-45., 1987.
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. Proceedings of ECOOP, 2001.
11. John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.
12. Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996.
13. P. Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
14. Andrew Tuson. *No Optimization Without Representation: A Knowledge Based Systems View of Evolutionary/Neighborhood Search Optimization*. Ph.D. Thesis, University of Edinburgh, 1999.
15. Gunter P. Wagner and Lee Altenberg. Complex adaptations and the evolution of evolvability. Evolution 50 (3): 967-976, 1996.