Generative Representations for Evolving Families of Designs

Gregory S. Hornby

Mail Stop 269-3, NASA Ames Research Center
Moffett Field, CA 94035-1000
hornby@email.arc.nasa.gov

Abstract. Since typical evolutionary design systems encode only a single artifact with each individual, each time the objective changes a new set of individuals must be evolved. When this objective varies in a way that can be parameterized, a more general method is to use a representation in which a single individual encodes an entire class of artifacts. In addition to saving time by preventing the need for multiple evolutionary runs, the evolution of parameter-controlled designs can create families of artifacts with the same style and a reuse of parts between members of the family. In this paper an evolutionary design system is described which uses a generative representation to encode families of designs. Because a generative representation is an algorithmic encoding of a design, its input parameters are a way to control aspects of the design it generates. By evaluating individuals multiple times with different input parameters the evolutionary design system creates individuals in which the input parameter controls specific aspects of a design. This system is demonstrated on two design substrates: neural-networks which solve the 3/5/7-parity problem and three-dimensional tables of varying heights.

1 Introduction

Evolutionary algorithms have been used in a variety of different design domains, with each individual in the evolutionary design system typically encoding a single design. With this type of representation, each time the objective changes (such as the desired lift of an aircraft wing or the receptive properties of an antenna) a new evolutionary run must be performed. While one option is to use previous results to seed a new run – as Gruau did in evolving parity networks [1] – these additional evolutionary runs can be avoided by evolving individuals which take an input parameter that controls some feature of the resulting design.

One method for an individual to encode a family of designs is for each member of the family to be encoded separately in the genotype. Yet with such a representation the size of the genotype grows with the number and size of each family member and it does not easily generalize to produce a design not already encoded. The alternative is to encode a family of designs with an algorithm which reuses parts of the genotype for different family members.

* This version (v1.1) corrects a couple of typos in the conference proceedings.

In addition to efficiencies of space, the reuse of genotypic elements for multiple members in the design family has two other advantages. For consumer products it is often desirable to have different versions of a product that vary in some way, yet have the same style. Whereas the individuals produced by different evolutionary runs usually have a different structure to them, a single individual that generates a family of designs with a reuse of assemblies of components will produce designs with a similar style. This reuse of parts leads to a second advantage of evolving design families, which is improved manufacturability. Since the members of these design families have more components in common than designs produced by multiple runs, there are fewer different parts to test and having assemblies of parts in common across the entire family should result in lower manufacturing costs.

An algorithm for encoding families of designs can be described as a *program* for mapping a *seed* to a design. Using these definitions existing work in evolutionary design can be classified as evolving either a single seed, a program, or both together. For example, the evolution of a vector of parameters with Dawkins' Biomorphs [2] and Ventrella's stick creatures [3] is the evolution of a seed for a pre-defined creature-building program. More common is the evolution of programs for fixed seeds, such as the evolution of cellular automata rules for a fixed starting state [4, 5] and the evolution of Lindenmayer systems (L-systems) with a fixed axiom [6, 7]. Finally, both seeds and programs have been evolved together, such as Frazer's evolution of both starting condition and cellular-automata-style rules [8] and the evolution of the axiom and rules of L-systems by Jacob [9] and Hornby [10].

Previously we defined *generative representations* as the class of representations in which elements of the genotype are reused in the translation to the phenotype and demonstrated a generative representation in which the genotype contained both a program for creating designs and the input parameters for the starting rule [10]. Here we describe an extension of this work from evolving a single design, with reuse of modules within the design, to evolving design families, with a reuse of modules across different members of the family. To produce individuals which represent a family of designs we now encode in the genotype only the program for constructing a design, and then evaluate an individual multiple times by compiling the program with different starting parameters. For each of these evaluations a specific phenotypic property of the resulting design is then compared with the desired result and the individual's fitness is adjusted based on how closely they match. By testing an individual with different starting parameters in this way, individuals are evolved to be responsive to the parameter values. We demonstrate the generality of this approach by evolving families of designs on two different design substrates: neural-networks which correctly calculate 3/5/7-parity, and three-dimensional tables of varying height.

The rest of this paper is organized as follows. First the generative representation for encoding families of designs is described, followed by a description of the overall evolutionary design system. The next two sections describe the evolution of a family of networks for calculating the 3/5/7 odd-parity function and the evolution of three-dimensional tables of varying heights. Finally we close with a summary of this paper.

2 L-systems as a Generative Representation

The generative representation for our design families is based on parametric Lindenmayer systems (PL-systems) [11]. PL-systems are a grammar consisting of a set of production rules for string rewriting. Production rules are composed of a predecessor, which is the symbol to be replaced, followed by a number of condition-successor pairs. The condition is a boolean expression on the parameters to the production-rule, and the successor consists of a sequence of characters that replace the predecessor. For example in the production:

$$A(n_1, n_1) : n_2 > 5 \rightarrow B(n_2{+}1)cD(n_2{+}0.5, n_1{-}2)$$

the predecessor is A(n_1, n_2), the condition is $n_2 > 5$ and the successor is B(n_2+1) c D(n_2+0.5, n_1-2). Predecessor symbols are re-written by testing each of their conditions sequentially and then replacing the predecessor symbol with the successor of the first condition that succeeds.

To generate strings with the grammar a starting string is required. For example the following grammatical rules,

$$\begin{array}{l} A(n_1,n_2) : (n_1 > 0) \rightarrow a(n_1)\ B(n_2,n_1)\ A(n_1-1,n_2) \\ A(n_1,n_2) : (n_1 \leq 0) \rightarrow a(0) \\ B(n_1,n_2) : (n_1 > 1) \rightarrow b(n_2)\ B(n_1-1,n_2) \\ B(n_1,n_2) : (n_1 \leq 1) \rightarrow b(n_2) \end{array}$$

when compiled starting with the string, $A(3,2)$, produce the sequence,

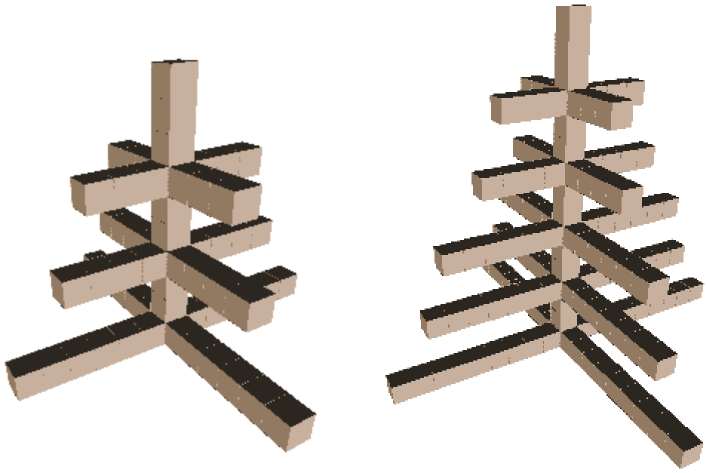
$$\begin{array}{c} A(3,2) \\ a(3)B(2,3)A(2,2) \\ a(3)b(3)B(1,3)a(2)B(2,2)A(1,2) \\ a(3)b(3)b(3)a(2)b(2)B(1,2)a(1)B(2,1)A(0,2) \\ a(3)b(3)b(3)a(2)b(2)b(2)a(1)b(1)B(1,1)a(0) \\ a(3)b(3)b(3)a(2)b(2)b(2)a(1)b(1)b(1)a(0) \end{array}$$

The combination of $A(3,2)$ and the grammatical rules is a generative representation for producing the final string in the sequence. In this case the seed consists of $A(3,2)$ and the program is the grammar. Alternatively, by using the starting string $A(n_1,n_2)$, the grammar is a program for creating a family of designs: the first parameter, n_1, controls the number of blocks of b's that are created and the second parameter, n_2, controls how many b's are in each block.

By assigning a meaning to each symbol, the strings produced by a PL-system can be used to construct artifacts. Consider the following PL-system:

$$\begin{array}{l} P0(n_1) :\ n_1 > 1.0 \rightarrow [\ P1(n_1 * 1.5)\ ]\ up(1)\ forward(3) \\ \qquad\qquad down(1)\ P0(n_1 - 1) \end{array}$$

$$P1(n_1) :\ n_1 > 1.0 \rightarrow \{\ [\ forward(n_1)\ ]\ left(1)\ \}(4)$$



(a) (b)

Fig. 1. Two example structures.

If this PL-system is started with the string P0(4), it produces the following sequence of strings,

1. *P0(4)*
2. *[P1(6)] up(1) forward(3) down(1) P0(3)*
3. *[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [P1(4.5)] up(1) forward(3) down(1) P0(2)*
4. *[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(4.5)] left(1) }(4)] up(1) forward(3) down(1) [P1(3)] up(1) forward(3) down(1) P0(1)*
5. *[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(4.5)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(3)] left(1) }(4)] up(1) forward(3) down(1)*
6. *[[forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1)] up(1) forward(3) down(1) [[forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1)] up(1) forward(3) down(1) [[forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1)] up(1) forward(3) down(1) forward(3)*

By interpreting the final string as a sequence of commands to a LOGO-style turtle, this PL-system creates the tree in figure 1.a.

To encode families of designs with this generative representation, the starting string (seed) is set to the predecessor of the first production rule with variables for starting parameters instead of numerical values. In this example the starting string is $P0(n_1)$ and different values of n_1 will produce different trees: the tree in figure 1.b is created from this system by starting it with n_1 equal to six.

3 Method

The system for evolving design families uses a canonical evolutionary algorithm (EA) with variation operators customized for the representation. A generational EA is used in which each individual encodes a design family using the generative representation described in section 2. Parents are selected with stochastic remainder selection [12] based on rank, using exponential scaling [13]. To create new individuals, the mutation and recombination operators of Hornby [10] are applied with equal probability. Mutation modifies an individual by changing one symbol with another, perturbing the parameter value of a symbol, adding/deleting some symbols, or recombining an individual with itself. With recombination, one parent is the main parent and it is modified by swapping some genetic material – either an entire rule, a single production body or substrings of a production body – with a second parent.

To produce individuals which encode for families of designs, individuals are evolved such that the value(s) of the input parameter(s) controls a certain feature of a design in a specific way. Each individual is tested with a range of different input values and each design's score is modified by how well the feature in the design matches the desired result. The following two sections will describe the application of this system for two design substrates.

4 Evolution of Parameter-Controlled n-Parity Networks

The first substrate for which families of designs are evolved is neural networks which calculate the odd-parity function. The odd-n-parity function returns `true` if the number of `true` inputs is odd and returns `false` otherwise. This function is difficult because the correct output changes for every change of an input value. In addition, the even/odd-n-parity functions have become a standard benchmark function in genetic programming (GP) and past experiments have shown that GP does not solve the five-parity (or higher) problem without automatically defined functions [14].

The method for using generative representations to encode neural networks is the same as our earlier work [15], which we now summarize. First the generative representation (section 2) is compiled into an assembly procedure and each neural network is constructed from an initial graph by executing this assembly procedure. The initial graph consists of a single neuron which has a single edge from itself to itself and the assembly procedure is a sequence of commands from the following command set, for which the current link connects from neuron A to neuron B:

- `add-input`(n), creates an input neuron with a link from it to neuron B with weight n.
- `add-output`(n), creates an output neuron with a link from B to it with weight n.
- `decrease-weight`(n), subtracts n from the weight of the current link. If the current link is a virtual link, it creates it with weight $-n$.

- `duplicate`(n), creates a new link from neuron A to neuron B with weight n.
- `increase-weight`(n), adds n to the weight of the current link. If the current link is a virtual link, it creates it with weight n.
- `loop`(n), creates a new link from neuron B to itself with weight n.
- `merge`(n), merges neuron A into neuron B by copying all inputs of A as inputs to B and replacing all occurrences of neuron A as an input with neuron B. The current link then becomes the nth input into neuron B.
- `next`(n), changes the from-neuron in the current link to its nth sibling.
- `output`(n), creates an output-neuron, with a linear transfer function, from the current from-neuron with weight n. The current-link continues to be from neuron A to neuron B.
- `parent`(n), changes the from-neuron in the current link to the nth input-neuron of the current from-neuron. Often there will not be an actual link between the new from-neuron and to-neuron, in which case a virtual link of weight 0 is used.
- `]`, pops the top state off the stack and makes it the current state.
- `[`, pushes the current state to the stack.
- `reverse`, deletes the current link and replaces it with a link from B to A with the same weight as the original.
- `set-function`(n), changes the transfer function of the to-neuron in the current link, B, with: 0, for sigmoid; 1, linear; and 2, for oscillator.
- `split`(n), creates a new neuron, C, with a sigmoid transfer function, and moves the current link from C to B and creates a new link connecting from neuron A to neuron C with weight n.

The design problem is to evolve an individual that specifies a family of networks with three/five/seven inputs which calculates the three/five/seven-odd-parity problem. To specify which network to construct, the first input parameter is set to 3.0, 5.0 and 7.0 to solve the three, five and seven parity problem. Input values are 1.0 for `true` and -1.0 for `false`. Networks are updated four times and then the value of the output neuron is examined to determine the parity value calculated for that set of input values. If the value of the output neuron is > 0.9 then the output of the network is taken as `true` and if the value of the output neuron is < 0.9 then the output of the network is taken as `false`. If the absolute value of the output neuron is < 0.9, the network is iteratively updated until its output value is either > 0.9 or < -0.9, for a maximum of six updates. The network receives a score of 2.0 for returning the correct parity value and a score of -1 for an incorrect answer. If the absolute value of the output neuron is less than 0.9 after six network updates, the network receives a score of 1.0 if the value of the output neuron is positive and the parity was `true` or if the value of the output neuron is negative and the parity is `false`. No penalty is given for having an incorrect value in this case. The fitness value of a network is the sum of its scores on all possible inputs and an individual's fitness score is the sum of its scores for the three networks.

Using the fitness function described in the previous paragraph, the graph in figure 2 contains a plot of the fitness of the best individual in the popu-

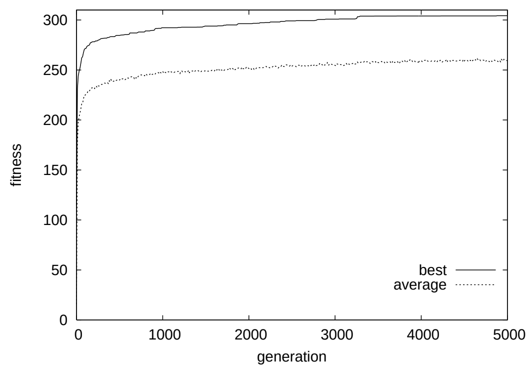


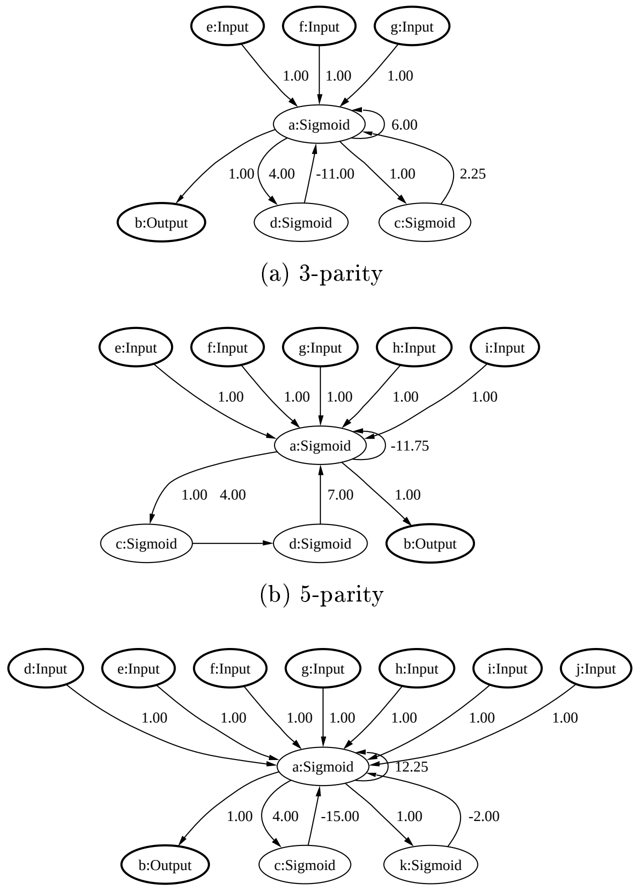
Fig. 2. Best fitness and average population fitness, averaged over fifty trials, for solving 3/5/7-parity. The maximum possible score is 336.

lation averaged over fifty trials. For these trials the generative representation was set to a maximum of fifteen productions, each with two parameters and three sets of condition-successor pairs. The maximum number of commands in each condition-successor pair is fifteen and the maximum length of an assembly procedure generated by the representation is ten thousand commands. The evolutionary algorithm used a population of five hundred individuals and was run for five thousand generations. Out of these fifty runs, the generative representation found a solution that produced correct 3/5/7-parity networks twelve times. For those runs that were successful it took 1800 generations, on average, to find a solution that produced correct networks. The smallest networks produced by a single individual that correctly calculates the 3/5/7-parity problems are shown in figure 3 (the genotype of this individual is listed in Appendix B of [10]).

5 Evolution of Parameter-Controlled Table Designs

The second design problem is that of evolving families of tables in which the input parameter controls the height of the table. With this substrate, the command set consists of commands for controlling a LOGO-style turtle in a three-dimensional grid [16]. As the turtle moves it fills in voxels, creating a three-dimensional object. The commands for this substrate are:

- `back`(n), move in the turtle's negative X direction n units.
- `clockwise`(n), rotate heading $n \times 90°$ about the turtle's X axis.
- `counter-clockwise`(n), rotate heading $n \times -90°$ about the turtle's X axis.
- `down`(n), rotate heading $n \times -90°$ about the turtle's Z axis.
- `forward`(n),move in the turtle's positive X direction n units.
- `left`(n), rotate heading $n \times 90°$ about the turtle's Y axis.
- `]`, pops the top state off the stack and makes it the current state.



(c) 7-parity

Fig. 3. Networks constructed to solve 3, 5, 7-parity from the same evolved network encoding.

- `[`, pushes the current state to the stack.
- `right`(n), rotate heading $n \times -90°$ about the turtle's Y axis.
- `up`(n), rotate heading $n \times 90°$ about the turtle's Z axis.

Section 2 contains an example of a design family encoded using this command set and the assembly procedure it compiles into.

Rather than have the input parameter exactly specify the height of the table, we evolve tables whose height is four times higher than the value of the input parameter. Using a volume of $40 \times 40 \times 40$ voxels, the maximum height of a table is forty units, so the valid range of input parameters is from one to ten. To allow us to later determine if an evolved generative representation will interpolate between tested input values and extrapolate beyond tested input values, we evaluate an individual using four input values that cover the range: `2.0`, `4.0`, `6.0`, and `8.0`. As with the previous set of experiments, individuals are encoded with the generative representation of section 2 using a maximum of fifteen production rules, each with two parameters and three condition-successor pairs. Since production rules take two input parameters, the second input value is an evolved value and is fixed for all trials.

To score the sensitivity of an individual to its input parameters, an individual's fitness is a combination of scores from the four designs created using four different input values. The fitness score for a single table design is based on that of our earlier work [16], with the objectives of maximizing stability and surface area while minimizing the amount of material used:

$$\begin{aligned} f_{surface} &= \text{the number of voxels at } Y_{max} \\ f_{stability} &= \sum_{y=0}^{Y_{max}} \text{area of the convex hull at height } y \\ f_{material} &= \text{number of voxels not on the surface} \end{aligned}$$

The overall score of a single table combines these objectives into a single function:

$$score(table) = f_{surface} \times f_{stability} / f_{material} \tag{1}$$

In addition, there is a height objective specified by the seed parameter:

$$f_{height} = \begin{cases} Y_{max}/height_{desired} & \text{if } Y_{max} < height_{desired} \\ 1.0 & \text{if } Y_{max} = height_{desired} \\ height_{desired}/Y_{max} & \text{if } Y_{max} > height_{desired} \end{cases}$$

This height objective is a value in the range of zero to one that penalizes a design for under-shooting or overshooting the desired height. A single fitness value for an individual is created by summing the scores for each of the four tables created by the four different seeds and multiplying them by the sum of the height penalties for all four tables:

$$fitness = \left(\sum_{i=1}^{4} f_{height}(table_i) \right) \times \left(\sum_{i=1}^{4} score(table_i) \right) \tag{2}$$

The reason for summing all the height penalties and applying them to the scores for all tables is to put pressure on the EA to evolve individuals which are sensitive to the seed parameter. With an early version of this test function in which the height penalty for a given table was applied only to that table,

$$fitness = \left(\sum_{i=1}^{4} f_{height}(table_i) \times score(table_i) \right) \tag{3}$$

evolved individuals tended to produce tables with high fitness for some of the seed parameters and had low fitnesses for others.

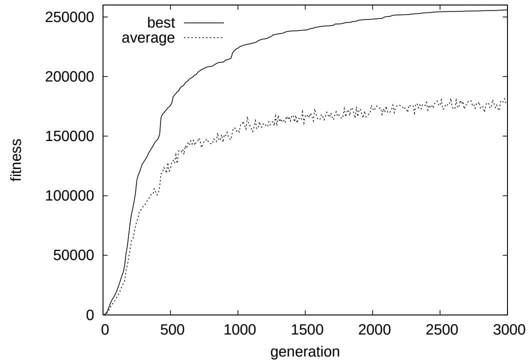


Fig. 4. Best fitness and average population fitness, averaged over fifty trials, for the table design problem.

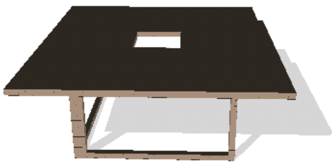
Using the fitness function of equation 2, figure 4 contains a graph plotting the fitness of the best individual in the population and average fitness of the entire population averaged over fifty trials. Evolved tables were responsive to the seed values and in all fifty trials the individuals in the final generation created tables within one voxel of the desired height for all four seed values. Figure 5 contains six tables in the design family for one of the evolved parameter-controlled tables. The tables in 5.a-d are the tables that are generated with tested input parameters 2.0, 4.0, 6.0 and 8.0 – the second parameter is evolvable and in this case is 2.0. The table in 5.e is an example with an input of 7.0, demonstrating that this individual can interpolate and generate a table with a seed value that is inside the tested range, and the table in 5.f is an example with an input of 10.0, which demonstrates that this individual can produce designs that extrapolate beyond the range tried during evolution.

6 Summary

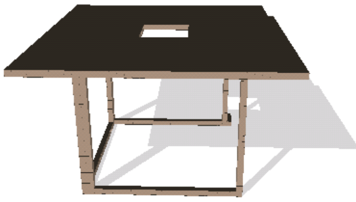
Typical evolutionary design systems must evolve a new set of individuals each time the design objective changes. Here we presented an evolutionary design system in which individuals use a generative representation to encode a family of designs. By encoding designs as a program and not directly, the generative representation uses an input parameter to control a design feature. Individuals are evolved to be sensitive to this input parameter by evaluating each one multiple times with different input values and combining the scores for the resulting designs into a single fitness function.

Using this system, families of designs were evolved on two different problem domains. On the first design problem, individuals were evolved that encoded three networks that calculated 3, 5 and 7 parity. On the second domain, individuals were evolved such that an input parameter controlled the height of a table.

(a) table(2.0, 2.0)



(b) table(4.0, 2.0)



(c) table(6.0, 2.0)

(d) table(8.0, 2.0)

(e) table(7.0, 2.0)

(f) table(10.0, 2.0)

Fig. 5. Parameter-controlled tables: (a)-(d) are the four trial parameters, (e) is an interpolation example, and (f) is an extrapolation example.

In addition, it was demonstrated that one evolved individual produces tables of the correct height for an input value in between those tested during evolution and for an input value greater than those tested during evolution: examples of interpolation and extrapolation. In general, evolving families of designs with a generative representation produced individuals with a reuse of modules among members of the design family. This reuse produced designs in a similar style and should lead to improved manufacturability.

References

1. Gruau, F.: Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm. PhD thesis, Ecole Normale Supérieure de Lyon (1994)
2. Dawkins, R.: The Blind Watchmaker. Harlow Longman (1986)
3. Ventrella, J.: Explorations in the emergence of morphology and locomotion behavior in animated characters. In Brooks, R., Maes, P., eds.: Proceedings of the Fourth Workshop on Artificial Life, Boston, MA, MIT Press (1994)
4. de Garis, H.: Artificial embryology : The genetic programming of an artificial embryo. In Soucek, B., the IRIS Group, eds.: Dynamic, Genetic and Chaotic Programming, Wiley (1992)
5. Bonabeau, E., Gurin, S., Snyers, D., Kuntz, P., Theraulaz, G.: Three-dimensional architectures grown by simple 'stigmergic' agents. BioSystems **56** (2000) 13–32
6. Ochoa, G.: On genetic algorithms and lindenmayer systems. In Eiben, A., Baeck, T., Schoenauer, M., Schwefel, H.P., eds.: Parallel Problem Solving from Nature V, Springer-Verlag (1998) 335–344
7. Coates, P., Broughton, T., Jackson, H.: Exploring three-dimensional design worlds using lindenmayer systems and genetic programming. In Bentley, P.J., ed.: Evolutionary Design by Computers. Morgan Kaufmann, San Francisco (1999)
8. Frazer, J.: An Evolutionary Architecture. Architectural Association Publications (1995)
9. Jacob, C.: Genetic L-system Programming. In Davidor, Y., Schwefel, P., eds.: Parallel Problem Solving from Nature III, Lecture Notes in Computer Science. Volume 866. (1994) 334–343
10. Hornby, G.S.: Generative Representations for Evolutionary Design Automation. PhD thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA (2003)
11. Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer-Verlag (1990)
12. Bäck, T.: Evolutionary Algorithms in Theory and Practice. Oxford University Press, New York (1996)
13. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, New York (1992)
14. Koza, J.R.: Genetic Programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, Mass. (1992)
15. Hornby, G.S., Pollack, J.B.: Creating high-level components with a generative representation for body-brain evolution. Artificial Life **8** (2002) 223–246
16. Hornby, G.S., Pollack, J.B.: The advantages of generative grammatical encodings for physical design. In: Congress on Evolutionary Computation. (2001) 600–607