A Developmental method for growing Graphs and Circuits

Julian F. Miller[1] and Peter Thomson[2]

[1] School of Computer Science, The University of Birmingham, UK j.miller@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~jfm
[2] School of Computing, Napier University, UK
p.thomson@napier.ac.uk
http://www.dcs.napier.ac.uk/~petert

Abstract. A review is given of approaches to growing neural networks and electronic circuits. A new method for growing graphs and circuits using a developmental process is discussed. The method is inspired by the view that the cell is the basic unit of biology. Programs that *construct* circuits are evolved to build a sequence of digital circuits at user specified iterations. The programs can be run for an arbitrary number of iterations so circuits of huge size could be created that could not be evolved. It is shown that the circuit building programs are capable of correctly predicting the next circuit in a sequence of larger even parity functions. The new method however finds building specific circuits more difficult than a non-developmental method.

1 Introduction

Natural evolution builds organisms using a process of biological development. In this a fertilized cell begins a process of replication and differentiation that culminates in an organism made of a huge number of specialist cells. Each cell carries the same genetic information yet somehow this decentralized, distributed community of cells builds a whole organism. Human beings design things in a top-down manner. Electronic circuits are built from high-level specification, to synthesis of symbolic components. These are then converted into transistor-level designs and finally wafers of silicon and wires are created. This method of design will become increasingly untenable as the size of electronic components continues to shrink.

In evolutionary computing a problem of scalability has become evident. The word scalability has a number of meanings: the evolution time increases markedly with problem size, the genotype length is proportional to problem size, the density of solutions in the search space associated with larger problems is a sharply decreasing function, and the time for fitness evaluation increases rapidly with problem size. The problem is particularly evident in the evolution of neural networks, where each link requires a floating-point weight that must be determined. In Genetic Programming researchers have generally concentrated on relatively small and simple problem such as parity functions and the Santa Fe Ant Trail. This has partly been motivated by scientific reasons in that these, though small are sufficiently difficult benchmarks for the evaluation of new techniques. Koza has been able to produce large and complex, human competitive analog circuit designs, but only by providing enormous computational power [19]. The problems of scalability have partly motivated a number of attempts to consider the use of a developmental approach to evolutionary design. In the field of evolvable hardware researchers have also tended to concentrate on relatively small circuit design problems. In digital circuit design the scalability problem is particularly acute [26].

2 Lindenmeyer systems, graph re-writing and developmental approaches

A number of researchers have studied the potential of Lindenmeyer systems [20] for developing artificial neural networks (ANNs) and generative design. Boers and Kuiper have adapted L-systems to develop the architecture of artificial neural networks (ANNs) (numbers of neurons and their connections) [3]. They used an evolutionary algorithm to evolve the rules of a L-system that generated feed-forward neural networks. Backpropagation was used, and the accuracy of the neural networks on test data was assigned to the fitness of the encoded rules. They found that this method produced more modular neural networks that performed better than networks with a predefined structure. Kitano had developed another method for evolving the architecture of an artificial neural network [17] using a matrix re-writing system that manipulated adjacency matrices. Although Kitano claimed that his method produced superior results to direct methods (i.e. a fixed architecture, directly encoded and evolved), it was later shown in a more careful study that the two approaches were of equal quality [23]. Gruau devised an elegant graph re-writing method called cellular encoding [10]. Cellular encoding is a language for local graph transformations that controls the division of cells that grow into artificial neural networks. This method was shown to be effective at optimizing both the architecture and weights at the same time, and they found that, to achieve as good performance with direct encoding, required the testing of many candidate architectures [11]. Others have successfully employed this approach in the evolution of recurrent neural networks that control the behaviour of simulated insects [18]. Koza has successfully employed a modified cellular encoding technique to allow the evolution of programs that produce human-competitive designs, especially in analog circuit design [19]. Recently Hornby and Pollack have also evolved context free L-systems to define three dimensional objects (table designs) [14]. They found that their generative system could produce designs with higher fitness and faster, than direct methods.

Jacobi created an impressive artificial genomic regulatory network, where genes code for proteins and proteins activate (or suppress) genes [16]. He used the proteins to define neurons with excitatory or inhibitory dendrites. This allowed him to define a recurrent ANN that was used to control a simulated Khepera robot for obstacle avoidance and corridor following. Nolfi and Parisi evolved encoded neuron position and branching properties of axonal trees that would spread out from the neurons and connect to other neurons [22] and in later work introduced cell division using a grammar [5]. Astor and Adami have created a developmental model of the evolution of ANN that utilizes an artificial chemistry [1].

Eggenberger suggests that the complex genotype-phenotype mappings typically employed in developmental models allow the reduction of genetic information without losing the complex behaviour. He stresses the importance of the fact that the genotype will not necessarily grow as the number of cells, thus he feels that developmental approaches will scale better on complex problems [8]. Sims evolved the morphology and behaviour of simulated agents [24]. Bongard and Pfeifer have evolved genotypes that encode a gene expression method to develop the morphology and neural control of multi-articulated simulated agents [4].

Bentley and Kumar examined a number of genotype-phenotype mappings on a problem of creating a tessellating tile pattern [2]. They found that the indirect developmental mapping (that they referred to as an implicit embryogeny) could evolve the tiling patterns much quicker, and further, that they could be subsequently grown (iterated) to much larger sized patterns. One drawback that they reported, was that the implicit embryogeny tended to produce the same types of patterns.

Hemmi, Mizoguchi and Shimohara evolved the rules of a rewriting system to define HDL programs [12]. Recently interest in developmental approaches in Evolvable Hardware has begun to increase. Haddow, Tufte and van Remortel have considered the use of Lindenmeyer systems for digital circuit design [13]. Their "cells" are like configurable logic blocks in FPGAs and cell interactions are defined by neighbouring cells in a two-dimensional grid system. They note that there are many epistatic effects that make evolution of the genotype difficult. Gordon and Bentley have compared a developmental evolutionary approach with a non-developmental encoding. They found that the could evolve these direct mappings more easily [9]. Edwards also examined developmental genotypes and possible physical implementations [7].

3 Cartesian Genetic Programming for circuits

Cartesian Genetic Programming was developed from methods developed for the automatic evolution of digital circuits [21]. Since electronic circuits can be represented by graphs (often referred to as netlists) it was natural to generalize it to the general problem of evolving computer programs. CGP represents a program or circuit as a list of integers that encode the connections and functions. The representation is readily understood from a small example. Consider a one bit binary adder circuit. This has three inputs that represent the two bits to be summed and the carry-in bit. It has two outputs: sum and carry-out. A possible implementation of this is shown below:

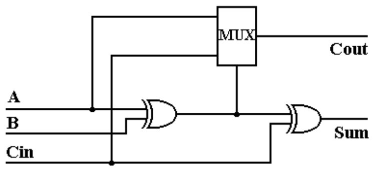

Fig. 3.1. One-bit adder circuit.

CGP employs an indexed list of functions that represent in this example various two input logic gates and also three input multiplexers. Suppose that XOR is function 10 and MUX is function 16. The three inputs A, B, Cin are labeled 0, 1, 2. The output of the left (right) XOR gate is labeled 3 (5). The output of the MUX gate is labeled 6. In CGP, this circuit could have the following genotype representation:

0 1 0 **10** 2 0 2 **6** 0 2 3 **16** 3 2 2 **10**

The integers in bold represent the functions, and the others represent the connections between gates, however, if it happens to be a two input gate then the third input is ignored. It is assumed that the circuit outputs are taken from the last two nodes. The second group of four integers (shown in grey) represent an AND gate that is not part of the circuit phenotype (hence doesn't appear in Fig 3.1). Since we are only interested in feedforward circuits here, it is important to note that the connections to any gate can only refer to gates that appear on its left. Typically in CGP we use point mutation (that is constrained to respect the feedforward nature of the circuit). Suppose that the first input of the MUX gate (0) was changed to 4. This would connect the AND gate into the circuit. Similarly, a point mutation might disconnect gates. Thus it is clear that in CGP we use a many to one genotype-phenotype mapping, as redundant nodes may be changed in any way and the genotypes would still be decoded to the same phenotype. We use an (1+4)-ES evolutionary algorithm that uses characteristics of this genotype-phenotype mapping to great advantage (i.e. genetic drift). This is given below:

1. Generate 5 chromosomes randomly to form the population
2. Evaluate the fitness of all the chromosomes in the population
3. Determine the best chromosome (called it *current_best*)
4. Generate 4 more chromosomes (offspring) by mutating the *current_best*
5. The *current_best* and the four offspring become the new population
6. Unless stopping criterion reached return to 2

Step 3 is a crucial step in this algorithm: if more than one chromosome is equally good then the algorithm always chooses the chromosome that is not the *current_best* (i.e. equally fit but genetically different). In a number of studies this step has been proved to allow a genetic drift process that turns out be of great benefit [25][29]. In later sections we refer to the mutation rate, this is the percentage of each chromosome that is mutated in step 4.

4 Developmental Cartesian Genetic Programming

The philosophical standpoint for the form of development described here is that *the cell is the basic unit of biology*. We see evolution as a process of evolving a cell. The cell is a very clever piece of machinery that can in co-operation with an environment self-replicate and differentiate to form a whole organism. Thus in developmental Cartesian GP (DCGP) we attempt to evolve a cell that can construct a larger program by iteration of the cell's program in its environment[1]. To elaborate this further, the idea is to define a program (encoded as a CGP graph) that is run inside the nodes (cells) of another graph (that is the phenotype or organism, or in this case the final digital circuit). It is important to note that in the context of the organism being a digital circuit, the cell is identified with a logic gate. All the gates of the final circuit are produced by running the same program inside all the nodes (gates) of the circuit graph for a given number of iterations. In the version of developmental CGP described here, we define the "environment" of a given cell to be the position of the cell and the integer labels of the cells that are connected to that cell. The analogues between the abstract graph re-writing rules and real biological entities are given below:

DNA bases: **Cartesian Cell Genotype**; *Translation*: **Genotype-Phenotype decoding**
Positional information: **Node position**; *Cell division*: **Node replication**
Cell type: **node function type (gate)**; *Zygote*: **seed node**
Neighbouring cell signals: **Node inputs (gate inputs)**

Of course, this is a highly idealized environment and further investigations are planned that will take into account the functions of the cells that are connected to the cell in question (this will allow semantic information to affect cell function). Imagine that we start with a 'seed' cell, this can only be connected to the program inputs. Here is one possible seed (and the one used for all the experiments described in this paper):

[1] The technique described here is much more biologically motivated than either L-systems or cellular encoding in addition the re-writing rules have the an enormously richer space of transformations that are possible with either, and with a much simpler conceptual apparatus.

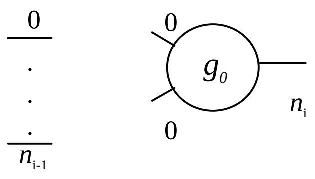

Fig. 4.1. Seed cell with n_i program inputs (0- n_{i-1}), g_0 denotes the first gate type in a list of possible gates.

Now we want to run a program inside the cell that uses information about the cell and its environment to construct a new one, indeed, we also want it to be able to replicate itself, so that we can grow a larger program. The cell needs four pieces of information, its two connections, its function and its position and the program inside the cell needs to take this information and create a new node with connections and a function and whether the cell should be duplicated. So the situation is now

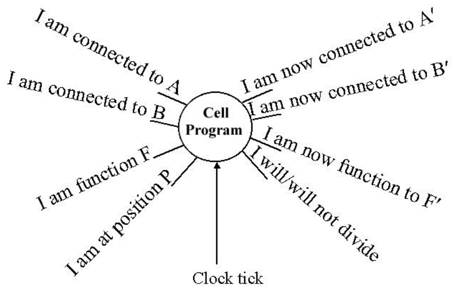

Fig. 4.2. Cell with program inputs and outputs.

The cell program is a mapping from the integers that define the cell's connections, function and position to a new set of integers defining its new connections, function and whether it will replicate itself. Since we are using this program to construct a circuit that is feed-forward , the integers that come out of the node program must take the correct values (i.e. the connections must be to nodes on the left of our current position, the functions must be ones on the list of valid functions, and finally the divide must be 0 or 1). Unfortunately it is very difficult to construct programs that will automatically respect the feed-forward requirement. After we have run the cell program we will need to carry out some sort of operation to bring the numbers into the correct ranges. A simple way to do this is to apply a modulo operation (note for the new function *F'* we use the $X \bmod N_F$ as the *address* in the function look up table). Finally, we are left with deciding how to allow the evolution of the program inside the cell. This can be accomplished using CGP with primitive functions that manipulate integers.

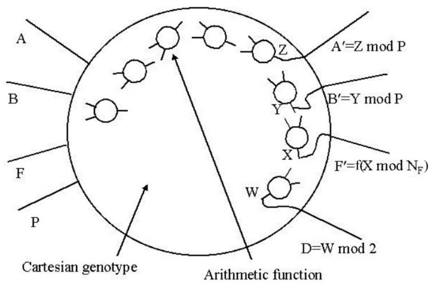

Fig. 4.3. Inside a cell is a Cartesian program defined using arithmetic functions. The phenotype nodes are valid feed-forward nodes due to the modulo operations.

To construct a circuit we run a single Cartesian program in each cell of a developing circuit (organism). This is called an iteration. Then at each iteration, we execute the same program. After a user defined number of iterations we test the developed circuit against the specification for the circuit (i.e. we count how many bits are correct). To clarify this still further let us consider an example of an evolved cell program that constructs the even-parity function with four inputs. Suppose that there are two primitive arithmetic functions f_0 and f_5 of arity two, defined as follows:

$$f_0 = f_0(x,y) = x + y \quad , \quad f_5 = f_5(x,y) = \begin{cases} 1 & x \geq y \\ 0 & x < y \end{cases} . \qquad (4.\mathbf{1})$$

Consider the following genotype

node labels	4	5	6	7	8	9	10	11
Cartesian genotype	1 2 **5**	4 4 **0**	3 2 5	2 0 **5**	2 1 **0**	8 7 **0**	4 3 **0**	1 5 **5**

There are four inputs: connection1 (A), connection2 (B), function (F), position (P) with labels 0, 1, 2, 3. The outputs of the program are taken from nodes 8, 9, 10, 11 and these outputs are denoted *Z*, *Y*, *X*, *W* (as seen in Fig. 4.3). Note that in this genotype node 6 is not referenced by nodes on its right, so is redundant. We can decode the genotype into a set of arithmetic rules dependent on the four inputs.

$$\begin{gathered} Z = F + B \ , \ Y = Z + f_5(F,A) \ , \\ X = P + f_5(B,F) \ , \ W = f_5(B, 2f_5(B,F)) \end{gathered} . \qquad (4.\mathbf{2})$$

To clarify the origin of these rules, consider the expression for W, this is the output of node 11, the first input is 1 (B), the second input refers to node 5. This is the addition of its two inputs, both of which refer to node 4. Node 4 is function 5 acting on it two inputs 1 and 2, that is, B and F (hence $2f_5(B,F)$ is the second argument of f_5 in *W*). Armed with the rules of equation 4.2 we can apply them to the seed cell and carry out the first iteration. Note that in this example we are using two functions in the phenotype, XOR (labelled as decimal 10) and XNOR (labelled as decimal 11). We are considering even-4 parity which has four inputs (labelled 0, 1, 2, 3). We found that the program worked much better when we allowed the right daughter cell to differentiate again (disallowing further replication)[2]. Thus, we define the word "iteration " to mean running the cell program in each cell and running the cell program (ignoring replication) in the right daughter cell. The reason why this helps markedly is still under investigation. The steps required for one iteration are shown in table 4.1. In the first "interpretation" line in the table a 10 appears (asterisk) because the cell program (after modulo operation) is 0, this refers to the zeroth function, which in this case is 10 denoting the XOR operation. After one iteration the circuit is represented by the encoded phenotype: 2 3 **10** 3 4 **11** which is the circuit shown in Fig. 4.4.

Table 4.1. Applying the cell program to the seed cell (first iteration)

Seed cell						Cell program outputs (see equation 4.2)			
A	*B*	*F*	*P*	f_5(F, A)	$f_5(B, F)$	*Z*	*Y*	*X*	*W*
0	0	10	4	1	0	10	11	4	1
						Zmod 4	Ymod4	Xmod2	Wmod2
New left cell						2	3	0	1
Interpretation						**2**	**3**	**10***	replicate
Right cell									
2	3	10	5	1	0	13	14	5	1
						Zmod 5	Ymod5	Xmod2	Wmod2
New right cell						3	4	1	1
Interpretation						**3**	**4**	**11**	ignore

[2] This step was found to allow more diversity in nodes in the phenotype, without it there tended to occur repeat node groups.

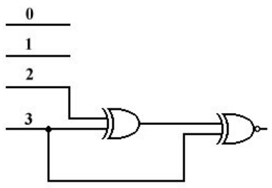

Fig. 4.4. The circuit, after one iteration.

In the next iteration the rules defined in equation 4.2 are then applied to *each* cell of the phenotype. Applying the rules to the cell defined by 2 3 **10** we obtain 1 2 **10** that is replicated (since D=1) giving 1 2 **10** 1 2 **10** 3 4 **11** and when the program is run with 1 2 **10** (node 5) we obtain 2 3 **11** (note D is ignored on the second differentiating step - see earlier). Finally applying the rules to the cell defined by 3 4 **11** we find that if replicates to 0 1 **11** 0 1 **11** and on differentiating the last cell we obtain the new encoded phenotype 1 2 **10** 2 3 **11** 0 1 **11** 5 6 **11**. This is even-four parity circuit below

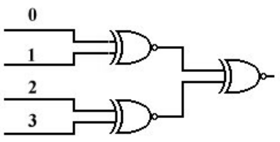

Fig. 4.5. The final grown circuit

The idea of growing graphs or circuits leads to some very interesting consequences. It is possible for us to provide rewards during the growth process. It also allows us to consider trying to evolve programs that construct classes of circuits, by requiring that at particular points in time the circuit have a particular behaviour. In the next section we describe experiments and results for the problem of growing the class of even-parity functions.

5 Experiments and results

In the experiments reported here we were able to choose from a set of seven primitive arithmetic functions to build the cell program. It is extremely difficult to know what would be a good choice for such functions[3]. The functions chosen are listed below:

[3] This is akin to trying to define a sequence of DNA base pairs that will give rise to a specific phenotypic trait. The function set in 5.2 was chosen so as to provide a "complete" set of arithmetic operations with the addition of comparisons (f_5) and controlled swapping (f_6).

$$f_0 = f_0(x,y) = x+y \;\;,\;\; f_1 = f_1(x,y) = x-y \qquad (5.2)$$
$$f_2 = f_2(x,y) = x \bmod (y+1) \;\;,\;\; f_3 = f_3(x,y) = xy \;\;,\;\; f_4 = f_4(x,y) = x/y \;.$$
$$f_5 = f_5(x,y) = \begin{cases} 1 & x \ge y \\ 0 & x < y \end{cases} \;\;,\;\; f_6 = f_6(x,y,z) = \begin{cases} x & z = 0 \\ y & z \ne 0 \end{cases}$$

Where x, y, z are the integer-value inputs to any node in the cell program. Note that in f_4 the operator is integer division. The circuit phenotype is built from the two binary input (a and b) functions $p(a, b)$ that are defined in table 5.2:

Table 5.2. Possible circuit functions

Integer code	***p* (*a*, *b*)**	**Integer code**	***p* (*a*, *b*)**
0	0	8	AND(NOT(*a*),*b*)
1	1	9	NOR(*a*, *b*)
2	*a*	10	XOR(*a*, *b*)
3	*b*	11	XNOR(*a*, *b*)
4	NOT(*a*)	12	OR(*a*, *b*)
5	NOT(*b*)	13	OR(*a*, NOT(*b*))
6	AND(*a*, *b*)	14	OR(NOT(*a*), *b*)
7	AND(*a*, NOT(*b*))	15	NAND(*a*, *b*)

In the first experiment we wanted to test the feasibility of the new approach, so we tried a simple problem of evolving the program that would construct a binary adder with the function defined in section 3. The experimental parameters are seen in table 5.3. We obtained three perfect adders designs. When observing the runs we observed that the algorithm reached fitness 14 (two bits incorrect) quite quickly but was often unable to improve. The average fitness obtained was 14.18. This does not compare well with a directly (non-developmental) evolved circuit (where the genotype directly encoded the circuit using the same representation as the phenotype) with the same parameters. We obtained 100 perfect adder circuits in that experiment. In the second experiment we tried to evolve a program that would build a series of even parity functions. The experimental parameters are shown in table 5.4.

Table 5.3. Parameters for experiment 1

Parameter	**value**
Population size	5
Mutation rate	8%
Number of generations	20,000
Number of runs	100
Number of circuits	1
Desired circuit	One-bit adder
Iteration at which circuit required	4
Maximum number of genotype nodes	20
Genotype node functions used	0 - 6
Phenotype node functions	6,7,10

Table 5.4 Parameters for experiment 2

Parameter	**value**
Population size	5
Mutation rate	3%
Number of generations	100,000
Number of runs	100
Number of circuits	3
Desired circuits	even-3, even-4, even5 parity
Iteration at which circuit required	1, 2, 3
Maximum number of genotype nodes	40
Genotype node functions used	0 - 6
Phenotype node functions	10,11

The circuit functions (phenotype) chosen for this experiment allow even parity functions to be built easily. The fitness of the cell program was defined as the sum of the correct truth table bits for all the target functions. We obtained 60 cell programs that could correctly build the even parity functions at the desired iterations. All these programs were subsequently tested at iteration four to see whether any of them could correctly build even-6 parity. One program was found that correctly built this. Unfortunately it proved not to be a general solution to the even-parity problem as it was found that it did not correctly build even-7 parity at the fifth iteration. Other researchers have successfully evolved completely general solutions to even-parity problems. However they all provide bit strings serially and work at a much higher level of abstraction.: machine code [15], recursive functions[27], and lambda abstractions [28].

6 Conclusions

We have presented a new way to evolve digital circuits that uses a developmental mapping that is based on the idea of evolving the program for a cell. It is capable of constructing a number of Boolean circuits at user specified points in time. The circuit construction program can in principle build circuits of arbitrary size. However, in practice the genotypes are not as evolvable as direct encodings. Designing an effective developmental mapping is a very difficult problem. The mapping we have presented is very general and flexible. However there are many aspects that require further investigation. In our representation the "neighbourhood" of a cell are the *positions* of the cells that are connected to it. Intuitively it seems more natural to take into account the *functions* of the neighbours. We have used a form of Cartesian GP that is linear in nature. In would be interesting to investigate the behaviour of the developmental algorithm if the circuit was defined in terms of a two-dimensional grid. One could evolve a circuit in which gates were arranged in columns and one could define the neighbours of a gate to be some gates that were *physically* close.

When the behaviour of the program that developed correct even parity functions was examined it was found that the circuits at successive iterations were generally quite different. One could potentially inherit more structure from previous circuits if the cell program was not run in every cell but only in the *last* cell. This of course, would depart from the biological inspired form of development. The form of development described here works at a quite abstract level in that the integers that form the input to the cell program are labels and addresses. It may be that a more direct representation would be more effective, for instance, one in which wires are particular types of cells. The problems we have studied here are all relatively simple and it may be that developmental approaches are more valuable when the circuits being grown are enormous. Many authors tend to believe that the shorter developmental genotypes will be more evolvable, however, this work suggests that the issue is much more complex.

References

1. Astor J. C., and Adami C. (2000), "A Development Model for the Evolution of Artificial Neural Networks", Artificial Life, Vol. 6, pp. 189-218.
2. Bentley P., and Kumar S. (1999), "Three ways to grow designs: A comparison of embryogenies for an Evolutionary Design Problem", in Proceedings of the Congress on Evolutionary Computation, IEEE Press, pp. 35-43.
3. Boers, E. J. W., and Kuiper, H. (1992), "Biological metaphors and the design of modular neural networks", Masters thesis, Department of Computer Science and Department of Experimental and Theoretical Psychology, Leiden University.
4. Bongard J. C. and Pfeifer R. (2001), "Repeated Structure and Dissociation of Genotypic and Phenotypic Complexity in Artificial Ontogeny", in Spector L. et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan-Kaufmann, pp. 829-836.

5. Cangelosi, A., Parisi, D., and Nolfi, S. (1993), "Cell Division and Migration in a 'Genotype' for Neural Networks", Technical report PCIA-93, Institute of Psychology, CNR, Rome.
6. Dellaert, F. (1995), "Toward a Biologically Defensible Model of Development", Masters thesis, Department of Computer Engineering and Science, Case Western Reserve University.
7. Edwards R. T (2003), "Circuit Morphologies and Ontogenies", in NASA/DOD Conference on Evolvable Hardware, IEEE Computer Society, pp. 251-260.
8. Eggenberger P. (1997), "Evolving morphologies of simulated 3D organisms based on differential gene expression", in Proceedings of 4th European Conference on Artificial Life, pp. 205-213.
9. Gordon T., and Bentley P. J. (2003) "Towards Development in Evolvable Hardware", in NASA/DOD Conference on Evolvable Hardware, , IEEE Computer Society, pp. 241-250.
10. Gruau, F. (1994), "Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm", PhD thesis, Ecole Normale Supérieure de Lyon.
11. Gruau, F., Whitley, D., and Pyeatt, L. (1996) "A Comparison between Cellular Encoding and Direct Encoding for Genetic Neural Networks", in Proceedings of the 1st Annual Conference on Genetic Programming, Stanford.
12. Hemmi H., Mizoguchi, J., and Shimohara K. (1994), "Development and Evolution of Hardware Behaviors", in Proceedings of Artificial Life IV, pp. 371-376.
13. Haddow, P. C., Tufte G., and van Remortel P. (2001) "Shrinking the genotype: L-systems for evolvable hardware", in 4th International Conference on Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, Vol. 2210, Springer-Verlag, pp. 128-139.
14. Hornby G. S., and Pollack J. B. (2001), "The Advantages of Generative Grammatical Encodings for Physical Design", in Proceedings of the Congress on Evolutionary Computation, IEEE Press, pp. 600-607.
15. Huelsbergen L. (1998), "Finding general Solutions to the Parity Problem bu Evolving Machine-Language Representations", in Proc. Conf. on Genetic Programming, pp. 158-166.
16. Jacobi, N. (1995), "Harnessing Morphogenesis", Cognitive Science Research Paper 423, COGS, University of Sussex.
17. Kitano, H. (1990), "Designing neural networks using genetic algorithms with graph generation system", Complex Systems, Vol. 4, pp. 461-476.
18. Kodjabachian, J. and Meyer, J-A. (1998), "Evolution and Development of Neural Controllers for Locomotion, Gradient-Following and Obstacle-Avoidance in Artificial Insects", IEEE Transactions on Neural Networks, Vol. 9, pp. 796-812.
19. Koza, J., Bennett III, F. H., Andre, D., and Keane, M. A. (1999). Genetic Programming III. Darwinian Invention and Problem Solving. Morgan Kaufmann.
20. Lindenmeyer, A. (1968), "Mathematical models for cellular interaction in development, parts I and II", Journal of Theoretical Biology, Vol. 18, pp. 280-315.
21. Miller, J. F. and Thomson, P. (2000), "Cartesian genetic programming", in Proceedings of the 3rd European Conference on Genetic Programming. LNCS, Vol. 1802, pp.121-132.
22. Nolfi, S., and Parisi, D. (1991) "Growing neural networks", Technical report PCIA-91-15, Institute of Psychology, CNR, Rome.
23. Siddiqi, A. A., and Lucas S. M. (1998), "A comparison of matrix rewriting versus direct encoding for evolving neural networks", in Proceedings of the 1998 IEEE International Conference on Evolutionary Computation, IEEE Press, pp. 392-397.
24. Sims K. (1994), "Evolving 3D morphology and behaviour by competition", in Proceedings of Artificial Life IV, pp. 28-39.

25. Vassilev V. K., and Miller J. F. (2000), "The Advantages of Landscape Neutrality in Digital Circuit Evolution", 3rd International Conference on Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, Vol. 1801, Springer-Verlag, pp. 252-263.
26. Vassilev V. K. and Miller J. F., (2000), "Scalability Problems of Digital Circuit Evolution", in Proceedings of the 2nd NASA/DOD Workshop on Evolvable Hardware, IEEE Computer Society, pp. 55-64
27. Wong M. L., and Leung K. S. (1996), " Evolving Recursive Functions for the Even-Parity Problem Using Genetic Programming", in Advances in Genetic Programming, MIT Press, Vol. 2, pp. 221-240.
28. Yu, T. (1999), "An analysis of the Impact of Functional Programming Techniques on Genetic Programming, Ph. D. Thesis, University College London, UK.
29. Yu, T. and Miller, J. (2001), "Neutrality and the evolvability of Boolean function landscape", in Proceedings of the 4th European Conference on Genetic Programming, Springer-Verlag, pp. 204-217.