Self Modifying Cartesian Genetic Programming: Parity

S. Harding, J. F. Miller and W. Banzhaf

***Abstract*— Self Modifying CGP (SMCGP) is a developmental form of Cartesian Genetic Programming(CGP). It differs from CGP by including primitive functions which modify the program. Beginning with the evolved genotype the self-modifying functions produce a new program (phenotype) at each iteration. In this paper we have applied it to a well known digital circuit building problem: even-parity. We show that it is easier to solve difficult parity problems with SMCGP than either with CGP or Modular CGP, and that the increase in efficiency grows with problem size. More importantly, we prove that SMCGP can evolve general solutions to arbitrary-sized even parity problems.**

I. Introduction

In biology, the process of transformation from genotype to phenotype is a complex interaction in which a genotype, together with the cellular machinery and the environment gives rise to a stage of the phenotype, which itself influences the decoding of the genotype for the next stage [1]. One can regard this process as one of self-modification which could take place both at the genotype or cellular level. Implicit in this notion is the concept of time or iteration. Accordingly, we define development to be the time-dependent process whereby genotype and phenotype, in interaction with each other and an external environment, produce a phenotype that can be selected for. This definition allows us to include many forms of development, such as models of genetic regulation, graph re-writing, and self-modification. It is our belief that self-modification is an interesting and unifying way to look at development. For instance, we can look at multi-cellular development as a process in which a phenotype modifies itself over time. We can also view development from the perspective of a single cell, where the genetic regulatory systems are a mechanism for development whereby a cell modifies its own phenotype through genetic self-modification over time. Kampis [2] has conducted an impressive philosophical analysis of the notion and importance of self-modification in biology and its relevance to 'emergent computation'.

In evolutionary computation, the idea of self-modification was discussed in the ontogenetic programming system of Spector and Stoffel [3], the graph re-writing system of Gruau [4] and the developmental method of evolving graphs and circuits of Miller [5]. Recently, however, much work in computational development has focused at a multi-cellular level and the aim has been to show that evolution could produce developmental cellular programs that could construct various cellular patterns (i.e. flags, or spheres, etc) [6]. Furthermore, another important aim has been to demonstrate that evolving developmental programs are a better way to evolve systems with an arbitrarily large number of parts than to directly evolve a genetic representation of such a system. While the former is an interesting goal, it is not *explicitly* computational in that often one must apply some other mapping process from the developed cellular structure into a computation.

In our previous work we showed that by utilizing self-modification operations within an existing computational method (a form of genetic programming, called Cartesian Genetic Programming, CGP) we could obtain a system that (a) could develop over time in interaction with environmental inputs and (b) would at every stage provide a computational function [7]. It could stop its own development, if required, without external input. Thus, if the computational task did not require development, evolution could decide for itself not to allow it. Another interesting feature of the approach is that, in principle, programs could be evolved which allow the replication of the original code. In this paper we have improved on our former work in SMCGP by concentrating on the scalability problem. Can evolution be used to produce arbitrarily large structures that represent *provably* general solutions to computational problems? We answer this question in the affirmative for the case of evolving a general solution to even parity (i.e. we obtain a program that can build a parity circuit for an arbitrary number of inputs). We have also compared the computational efficiency of this approach to non-developmental methods that use the same Cartesian genetic representation and we show that self-modifying CGP is more efficient.

II. Related Work

Parity is a well studied problem in Genetic Programming. Koza tackled up to 11-parity [8] using a GP system with automatically defined functions (ADFs), and found them difficult to evolve. Without ADFs, his approach failed to evolve circuits beyond 5 inputs [9].

In [10] very large parity circuits, with 22-inputs, are directly evolved. The authors describe three different approaches to solving the problem using a novel crossover operator, a submachine code level representation and a parallel population approach. This appears to be the largest, directly evolved parity circuit in the literature.

Other approaches have looked at finding general solutions. For example, in [11] machine language level programs were evolved that could iterate over the bits in a string and could be easily determine parity. The solutions would be suitable for any length bit string. Recursion has also been successfully used to solve the parity problem [12], [13]. These approaches produced programs rather than circuits to solve the problem.

S.Harding and W.Banzhaf are with the Department of Computer Science, Memorial University, Newfoundland, Canada; www.cs.mun.ca; email: (simonh,banzhaf)@cs.mun.ca. J. Miller is with the Department of Electronics, The University of York, UK; www.elec.york.ac.uk; email: jfm7@ohm.york.ac.uk.

978-1-4244-2959-2/09/$25.00 © 2009 IEEE

In contrast, the technique presented in this paper evolves programs that produce circuits.

An early developmental form of CGP successfully solved circuits up to 5 inputs [5]. However, none of the evolved solutions appeared to generalize. In [7], Self Modifying CGP was demonstrated for the first time, and parity circuits of up to 8-inputs were evolved. It was shown that SMCGP outperformed the best known results for a CGP based implementation.

Another developmental system uses L-systems to generate a grammar for modifying GP trees [14]. This approach is similar in concept to [7]. Circuits of up to 12-inputs were evolved with high success. The authors also found that the developmental approach was superior to previous, similar representations. [15] evolved artificial protein rules that configured FPGA blocks. The evolved rules could be iterated and used pattern matching and their behaviour depended on the matching of proteins at each time step. Again, circuits of up to 12-inputs were evolved. In contrast, the authors were unable to evolve circuits larger than 4 inputs using a direct encoding.

III. SELF MODIFYING CGP

A. Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming was originally developed by Miller and Thomson [16] for the purpose of evolving digital circuits and represents a program as a directed graph One of the benefits of this type of representation is the implicit re-use of nodes in the directed graph.

Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work on CGP always chose the number of rows to be one, thus giving a one-dimensional topology, as used in this paper. In CGP, the genotype is a fixed-length representation and consists of a list of integers which encode the function and connections of each node in the directed graph.

CGP uses a genotype-phenotype mapping that does not require all of the nodes to be connected to each other, resulting in a bounded variable length phenotype. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This type of neutrality has been investigated in detail [16], [17], [18] and found to be extremely beneficial to the evolutionary process on the problems studied.

B. SMCGP

In this paper, we use a slightly different genotype representation to previously published work using CGP.

Each node in the directed graph represents a particular function and is encoded by a number of genes. The first gene encodes the function the node is representing. This is followed by a number of connection genes (as in CGP) that indicate the location in the graph where the node takes its inputs from. Then three real-valued genes encode parameters required for the function. Finally there is a binary gene that indicates if the node should be used as an output. In this paper all nodes take two inputs, hence each node is specified by 7 genes. An example genotype is shown in Figure 1.

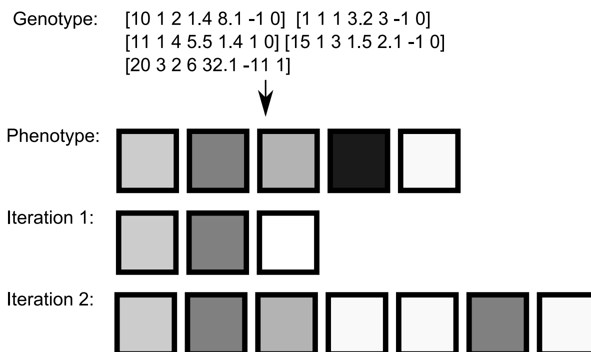

Fig. 1. The genotype maps directly to the initial graph of the phenotype. The genes control the number, type and connectivity of each of the nodes. The phenotype graph is then iterated to perform computation and produce subsequent graphs.

As in CGP, nodes take their inputs in a feed-forward manner from either the output of a previous node or from a program input (terminal). The actual number of inputs to a node is dictated by the arity of its function. However, unlike previous implementations of CGP, nodes are addressed *relatively* and specify how many nodes back in the graph they are connected to. Hence, if the connection gene is 1 it means that the node will connect to the previous node in the list, if the gene has value 2 then the node connects 2 nodes back and so on. All such genes are constrained to be greater than 0, to avoid nodes referring directly or indirectly to themselves.

If a gene specifies a connection pointing outside of the graph, i.e. with a larger relative address than there are nodes to connect to, then this is treated as connecting to zero value. Inputs arise in the graph through special functions. This is described in section III-C.

This encoding is demonstrated visually in Figure 2. The relative addressing used here attempts to allow for sub-graphs to be placed or duplicated in the graph whilst retaining their semantic validitity. This means that sub-graphs could represent the same sub-function, but acting on different inputs.

Each node in the SMCGP graph is defined by a function that is represented internally as an integer. Associated with each function are genes denoting connected nodes and also a set of parameters that influence the function's behavior. These parameters are primarily used by functions that perform modification to the phenotype's graph. In the genotype they are represented as real numbers but certain functions can require that they be cast (truncated) to integers.

Section V details the available functions and any associated parameters.

C. Inputs and outputs

The way we handled inputs in our original paper on SMCGP was flawed. It did not scale well as sub-graphs became disconnected from inputs, because self-modifying

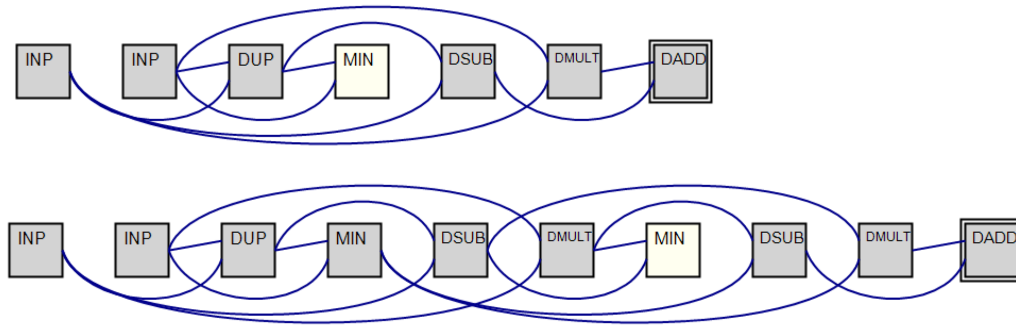

Fig. 2. Example program execution. Showing the DUP(licate) operator being activated, and inserting a copy of a section of the graph (itself and a neighboring functions on either side) elsewhere in the graph in next iteration. Each node is labeled with a function, the relative address of the nodes to connect to and the parameters for the function (see Section III-D).

functions moved them away from the beginning of the graph causing them to lose their semantic validity. The new input strategy we devised, required two simple changes from conventional CGP and our previous work in SMCGP.

The first, was to make all negative addressing return false (or 0 for non-binary versions of SMCGP). In previous work [7] , we used negative addresses to connect nodes to input values.

The second was to change how the INPUT function works. When a node is of type INP (shorthand for INPUT), each successive call gets the next input from the available set of inputs. If the INP node is called more times than there are inputs, the counting starts from the beginning again, and the first node is used.

Outputs are handled slightly differently to inputs. We added another gene to the SMCGP node that defines if the phenotype should attempt to use that node as an output. In previous work, we used the last n-nodes in the graph to represent the n-outputs. However, as with the inputs, we felt this approach did not scale as the graph changes size. When an individual is evaluated, the first stage is to identify the nodes in the graph that have their output gene set to 1. Once these are found, the graph can be evaluated from each of these nodes in a recursive manner.

If no nodes are flagged as outputs, the last n nodes in the graph are used as the n-outputs. Essentially, this reverts the system back to the previous approach. If there are more nodes flagged as outputs than are required, then the leftmost nodes that have flagged outputs are used until the required number of outputs is reached. If there are fewer nodes in the graph than required outputs, the individual is deemed to be corrupt and it is not evaluated (it is given a bad fitness score to ensure that it is not selected for).

D. Evaluation of the SMCGP graph

From a high level perspective, when a genotype is evaluated the process is as follows. The initial phenotype is a copy of the genotype. This graph is then executed, and if there are any modifications to be made, they alter the phenotype graph.

Technically, we consider the genotype invariant during the entire evaluation of the individual and perform all modifications on the phenotype which started out as a copy of the genotype. In subsequent iterations, the phenotype will usually gradually diverge from the genotype.

The encoded graph is executed in the same manner as standard CGP, but with changes to allow for self-modification. The graph is executed by recursion, starting from the output nodes down through the functions, to the input nodes. In this way, nodes that are unconnected are not processed and do not effect the behavior of the graph at that stage.

For function nodes (e.g. AND, OR, XOR) the output value is the result of the mathematical operation on input values.

Each active (non-junk) graph manipulation function (starting on the leftmost node of the graph) is added to a “To Do” list of pending modifications. After each iteration, the “To Do” list is parsed, and all manipulations are performed (provided they do not exceed the number of operations specified in the user defined “To Do” list length). The parsing is done in order of the instructions being appended to the list, i.e. first in is first to be executed.

The length of the list can be limited as manipulations are relatively computationally expensive to perform. Here we limit the length to just 2 instructions. There is a single “To Do” list for evaluation of each individual, and hence subprocedures also share the same list. All graph manipulation functions require a number of parameters, as described in section V. These parameters are encoded in the genotype, and the necessary casts are made when the “To Do” list is parsed.

IV. Evolutionary algorithm and parameters

We use an (1+4) evolutionary strategy for the experiments in this paper. We bootstrap the process by testing a population of 50 random individuals. We then select the best individual and generate four offspring. We test these new individuals, and use the best of these to generate the next population.

We have used a relatively high (for CGP) mutation rate of 0.1. This means that each gene has a probability of 0.1 of being mutated. SMCGP, like normal CGP, allows for different mutation rates to effect different parts of the genotype (for example functions and connections could have

different mutation rates). In these experiments, for simplicity, we chose to make all the rates the same. Mutations for the function type and relative addresses themselves are unbiased; a gene can be mutated to any other valid value.

For the real-valued genes, the mutation operator can choose to randomize the value (with probability 0.1) or add noise (normally distributed, sigma 20).

Evolution is limited to 10,000,000 evaluations. Trials that fail to find a solution in this time are considered to have failed.

The evolutionary parameter values have not been optimized, and we would expect performance increases if more suitable values were used.

V. FUNCTION SET

The function set is defined in two parts. The first is a set of modification operators, as described in section V. These are common to all data types used in SMCGP. The remainder of the set are the computational operations. The data type these functions manipulate is determined by the problem definition. Here, the data is binary strings. The complete set of available binary operators are defined in table I. Depending on the experiment, different sub-sets of this set are used.

Self modifying functions are typically defined by 4 variables. The genotype (and phenotype) nodes contain three double precision numbers, called "parameters". In the following discussion we denote these P_0,P_1,P_2. The other variable is the integer position of the node in the phenotype graph that contained the self modifying function (i.e. the leftmost node is position 0), we denote this x. In the definitions of the SM functions we often need to refer to the values taken by node connection genes (which are all relative addresses). We denote the jth connection gene on node at position i, by c_{ij}.
There are several rules that decide how addresses and parameters are treated:

- When P_i are added to the x, the result is treated as an integer.
- Address indexes are corrected if they are not within bounds. Addresses below 0 are treated as 0. Addresses that reach beyond the end of the graph are truncated to the graph length.
- Start and end indexes are sorted into ascending order (if appropriate).
- Operations that are redundant (e.g. copying 0 nodes) are ignored, however they are taken into account in the ToDo list length.

The functions (with the short-hand name) are defined as follows:

Duplicate and scale addresses (DU4) Starting from position $(P_0 + x)$ copy (P_1) nodes and insert after the node at position $(P_0 + x + P_1)$. During the copy, c_{ij} of copied nodes are multiplied by P_2.

Shift Connections (SHIFTCONNECTION) Starting at node index $(P_0 + x)$, add P_2 to the values of the c_{ij} of next P_1.

ShiftByMultConnections (MULTCONNECTION) Starting at node index (P_0 + x), multiply the c_{ij} of the next P_1 nodes by P_2.

Move (MOV) Move the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.

Duplication (DUP) Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$.

DuplicatePreservingConnections (DU3) Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ and insert after $(P_0 + x + P_2)$. When copying, this function modifies the c_{ij} of the copied nodes so that they continue to point to the original nodes.

Delete (DEL) Delete the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$.

Add (ADD) Add P_1 new random nodes after $(P_0 + x)$.

Change Function (CHF) Change the function of node P_0 to the function associated with P_1.

Change Connection (CHC) Change the $(P_1 mod 3)$th connection of node P_0 to P_2.

Change Parameter (CHP) Change the $(P_1 mod 3)$th parameter of node P_0 to P_2.

Overwrite (OVR) Copy the nodes between $(P_0 + x)$ and $(P_0 + x + P_1)$ to position $(P_0 + x + P_2)$, replacing existing nodes in the target position.

Copy To Stop (COPYTOSTOP) Copy from x to the next "COPYTOSTOP" function node, "STOP" node or the end of the graph. Nodes are inserted at the position the operator stops at.

VI. EXPERIMENTAL SETUP

A. Fitness function

Fitness is computed as the number of correctly predicted bits over all test cases. The fitness function used here tests the program to produce various sized parity circuits during development. In the first iteration, it tests for 2 input parity, then 3 input parity and continues up to a maximum number of inputs. If the candidate solution fails to find a totally correct solution for a given input size, it is not tested on other input sizes - allowing the process to abort development and save CPU time. We define each of these circuit sizes to be one test case. We evolve for 18 test cases (2 inputs to 20 inputs).

The fitness function is designed to force the SMCGP to find a solution that grows through each test case to the next. In this way, the chance of finding a general solution is maximised.
The fitness function can be summarized as:

- For each individual:
  - For each test case (2 to 20 inputs):
    - Take the genotype.
    - Iterate it $(inputs - 2$ times)
    - Apply input bit patterns
    - Count incorrect outputs, and add to fitness sum
    - If fails to solve test case, continue to next individual.

Function	Operation
BAND	a AND b
BOR	a OR b
BNAND	NOT (a AND b)
BXOR	a XOR B
BNOR	NOT (a OR b)
BNOT	NOT a
BIAND	(NOT a) AND b
BF0	FALSE
BF1	(a AND b)
BF2	a AND (NOT b)
BF3	(a AND (NOT b)) or (a AND b)
BF4	(NOT a) AND b
BF5	((NOT a) AND b) OR (a AND b)
BF6	((NOT a) AND b) OR (a AND (NOT b))
BF7	((NOT a) AND b) OR (a AND NOT(b)) OR (a AND b)
BF8	((NOT a) AND (NOT b))
BF9	((NOT a) AND (NOT b)) OR (a AND b)
BF10	((NOT a) AND (NOT b)) OR (a AND NOT (b))
BF11	((NOT a) AND (NOT b) OR a AND (NOT b) OR a AND b)
BF12	((NOT a) AND (NOT b) OR (NOT a) AND b)
BF13	((NOT a) AND (NOT b) OR (NOT a) AND b OR a AND b)
BF14	((NOT a) AND (NOT b) OR (NOT a) AND b OR a AND (NOT b))
BF15	((NOT a) AND (NOT b) OR (NOT a) AND b OR a AND (NOT b) OR a AND b)

TABLE I

BINARY FUNCTIONS

VII. RESULTS

A. *Restricted function set*

First, we evolved even parity functions from 2 inputs to 20 inputs using Boolean functions: AND, OR, NAND and NOR.

Table II shows the average number of evaluations required. We obtained 100% success rate for evolving to 18 test cases (up to 20 inputs). This is a substantial improvement over our earlier published results where, after 5 inputs the success rate dropped below 100% [7]. From the results, it can be seen that the number of evaluations required to solve for a given size stablises to approximately 318,000 evaluations. This is because "general" solutions are often discovered by the time even 6 parity is solved, and therefore solutions for larger circuits do not need to be found by evolution.

Table III shows a comparison between other CGP implementations. The speed up values show the relative performance in terms of evaluations. A value above 1 indicates SMCGP performs better. The results show that for smaller sizes, SMCGP is slower to evolve than the previous techniques. For parity problems with more than 6 inputs, SMCGP starts to perform better. It is important to note that these other techniques do not report results beyond 8 inputs and that the fitness function is different. We believe that the task of finding a program that grows these circuits should be substantially more difficult.

No. Of Inputs	Average Evaluations
2	126,095
3	289,824
4	308,643
5	309,990
6	311,022
7	313,489
8	313,978
9	314,056
10	317,700
11	317,712
12	317,931
13	317,936
14	317,941
15	317,950
16	317,960
17	317,965
18	317,979
19	317,994
20	317,999

TABLE II

EVALUATIONS REQUIRED TO EVOLVE TO EACH SIZE, USING THE RESTRICTED FUNCTION SET OF AND, OR, NAND AND NOR.

B. *Full function set*

We repeated the experiment but with a full function set (BF0 to BF15) so that we could compare with [10].

Table IV shows the number of evaluations required to evolve a given sized even parity circuit. Care should be taken when comparing to other results in the literature, as we are solving a subtly different problem. Our aim is to evolve a program that can produce a parity circuit for a given size - *and all smaller sizes*. Other approaches typically aim to find a circuit for a given number of inputs; which probably makes the problem easier, as their goals are a proper subset of ours.

Next, we investigated the scaling properties of the parity circuit. We evolved to 20 bits of input and then tested exhaustively to 24 bits of input. We find that solutions are often able to scale to solve larger problems beyond what they were evolved to solve. Table V shows the percentage of solutions that were able to solve these additional problems. See section VII-C for a more detailed investigation into the generality of an individual.

We investigated the stage at which solutions start to act as "general" solutions. By general we mean that they can solve all subsequent test sizes up to 20 bits of input. We see that some solutions can generalise after solving for just 2 inputs cases (i.e. have been evolved to solve both 2 and 3 bits). Table VI shows the percentage of solutions that generalise to all 18 test sizes both at a given point, and cumulatively. We see that the majority of the "general" solutions are found between 5 and 7 bits of input. This explains why table IV shows a stablisation in the number of evaluations required to find a solution. On average, it takes until 12 inputs for general solutions to emerge.

Results presented here are based on 251 runs.

We found that for the majority of the runs, the graph size consistently increases in size. However, for some runs the graph size does not maintain constant growth and either

Inputs	Average Evaluations				Speedup		
	SMCGP	SMCGP 2007	CGP	ECGP	Vs. SMCGP2007	Vs.CGP	Vs. ECGP
4	308,643	28,811	81,728	65,296	0.09	0.26	0.21
5	309,990	58,194	293,572	181,920	0.19	0.95	0.59
6	311,022	199,256	972,420	287,764	0.64	3.13	0.93
7	313,489	410,128	3,499,532	311,940	1.31	11.16	1.00
8	313,978	1,080,656	10,949,256	540,224	3.44	34.87	1.72

TABLE III

COMPARISON TO PREVIOUS CGP BASED TECHNIQUES, USING THE RESTRICTED FUNCTION SET OF AND, OR, NAND AND NOR. SMCGP2007 = FIRST SMCGP PAPER [7]. ECGP = EMBEDDED CGP [19]. CGP = NORMAL CGP [7]

reduces in size or remains constant. Figure 3 shows the behaviour trends as a whole.

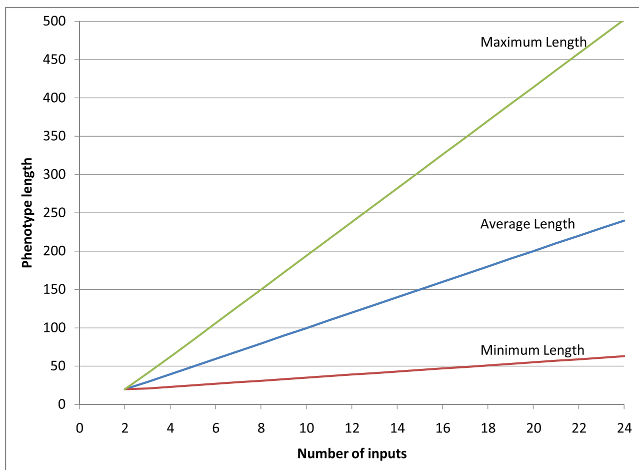

Fig. 3. Plot showing the average, maximum and minimum graph length of the SMCGP phenotype as it iterates with number of inputs (for the parity problem).

C. Generality

It is computationally very expensive to evaluate individuals greater than 24-inputs, therefore a method is needed to either prove generality, or, provide a high degree of confidence that a solution is general. A general solution is defined as a program that when iterated will always produce the next sized parity solution. We show that this is indeed the case for one example.

Consider the individual illustrated in figure 4. Here we prove that it is indeed the general solution to even-parity. To begin with we need to show that the first case at the top of the figure computes even-2 parity. In 5(a) we provide an annotated version showing the outputs of the active nodes. The leftmost active node is an INP which returns x_0, the next call to INP returns x_1. These both connect to the leftmost BXOR node which computes the binary EXOR of these inputs. Two identical outputs from this BXOR are provided as input to the function BNOR, this inverts the input (we have denoted this as carrying out EXORing with 1). This output is then provided as the first input to BXOR. We have denoted the second input to this function as v (which in this case is zero, as the connection reaches 14 nodes back, which is beyond the first node of the graph). Thus the output is now $x_0 \oplus x_1 \oplus 1 \oplus v$. The DUP function is a self-modification instruction and is defined to return the first input, hence it outputs this quantity. The final (rightmost) active node is also BXOR and this EXORs with input u (which is 22 nodes back, which is again beyond the end of the graph so returns zero). Thus the graph outputs $B(x_0, x_1, u, v) = x_0 \oplus x_1 \oplus 1 \oplus v \oplus u$. In this case this reduces to $x_0 \oplus x_1 \oplus 1$, which is the even-2 parity function.

No. Of Inputs	Average Evaluations
2	1,429
3	4,013
4	8,656
5	19,894
6	43,817
7	71,857
8	82,936
9	102,868
10	107,586
11	104,343
12	108,356
13	118,790
14	121,835
15	118,477
16	114,116
17	110,216
18	110,223
19	110,255
20	110,262

TABLE IV

EVALUATIONS REQUIRED TO EVOLVE TO EACH SIZE.

No. Of Inputs	% Solution
21	97.1
22	96.1
23	96.1
24	96.0

TABLE V

PERCENTAGE OF SOLUTIONS THAT GENERALISE TO VARIOUS, UN-EVOLVED INPUT LENGTHS.

Now we discuss the effect of the DUP operation, which happens, in this case, to be the only active self modifying node. This copies the 9 nodes on its left and itself (shown within the box shown in figure 5(a). It inserts these nodes twelve nodes back from itself. That is just before the node

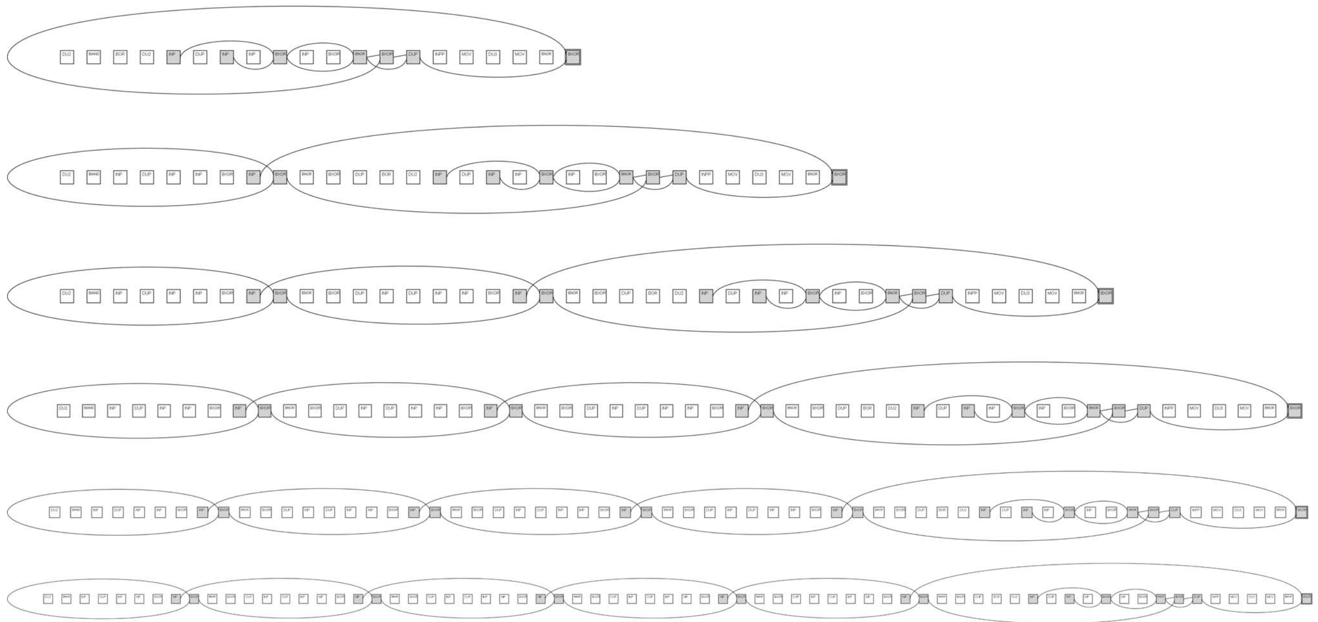

Fig. 4. An example of the development of a parity circuit. Each line shows the phenotype graph at a given time step. The first graph solves the 2-input parity, the second solves 3-input and continues to 7-bits. The graph has been tested to generalise through to 24 inputs. This pattern of growth is typical of the programs investigated.

No. Of Inputs	% that generalise at this step	Total perc. that generalise by this stage
2	0.00	0.00
3	0.97	0.97
4	10.68	11.65
5	23.30	34.95
6	24.27	59.22
7	14.56	73.79
8	7.77	81.55
9	3.88	85.44
10	4.85	90.29
11	1.94	92.23
12	0.97	93.20
13	1.94	95.15
14	0.00	95.15
15	0.00	95.15
16	0.00	95.15
17	1.94	97.09
18	0.00	97.09
19	1.94	99.03
20	0.97	100.00

TABLE VI

PERCENTAGE OF SOLUTIONS THAT GENERALISE AT A GIVEN INPUT SIZE.

BOR (the third node from the left). Now consider what happens when we carry out this operation (shown in Figure 5(b)). The input denoted u of the even-2 parity function now activates the INP node of the duplicated code and the v input activates the BXOR input (immediately on its right). The input makes u equal to x_0. The inputs to the BXOR function are actually 11 nodes back (first) and ten nodes back (second), in this case they both reach beyond the end of the graph and are therefore zero. Thus the v input to B is zero and we obtain even-3 parity. We denote the sub-function of copied nodes, after insertion as $M(y_0, y_1)$, where y_0 and y_1 denote its inputs. In this case of even-3 parity this "module" has two outputs, x_0 and 0. This is shown in figure 5(b). It is apparent that the structure of the leftmost ith copied modules is given by $M_i(y_0, y_1) = x_i : y_0 \oplus y_1$, where the colon denotes the two outputs of the module. We can see this because the INP function inside provides a new program input each time it is called (we begin at 0) and the BXOR inside provides the EXOR of the two inputs. The general case has the form of a series of $n - 2$ copied modules, M, connected to the even-2 parity function B. This is shown in 5(c). Thus the general parity solution is obtained.

VIII. CONCLUSIONS

We argue that self-modification is a unifying view of development. Multi or single cellular systems and re-writing systems can all be seen as forms of self-modification. In multi-cellular systems, the cells modify the collection of cells (i.e. through self-replication and differentiation). In single-celled systems running a GRN, the genetic code changes which pieces of code are expressed over time so that the "genetic program" being run changes over time.

Self modifying Cartesian Genetic Programming has a number of virtues. Cartesian GP is a general method for program evolution and SMCGP builds on that because at each iteration it represents a CGP graph. So in that sense SMCGP is a general computational developmental system. This is, in our view, one of the key strengths of the approach. We have presented an improved version of SMCGP and demonstrated that it can solve arbitrary parity problems. It was also shown to require less computational effort to solve

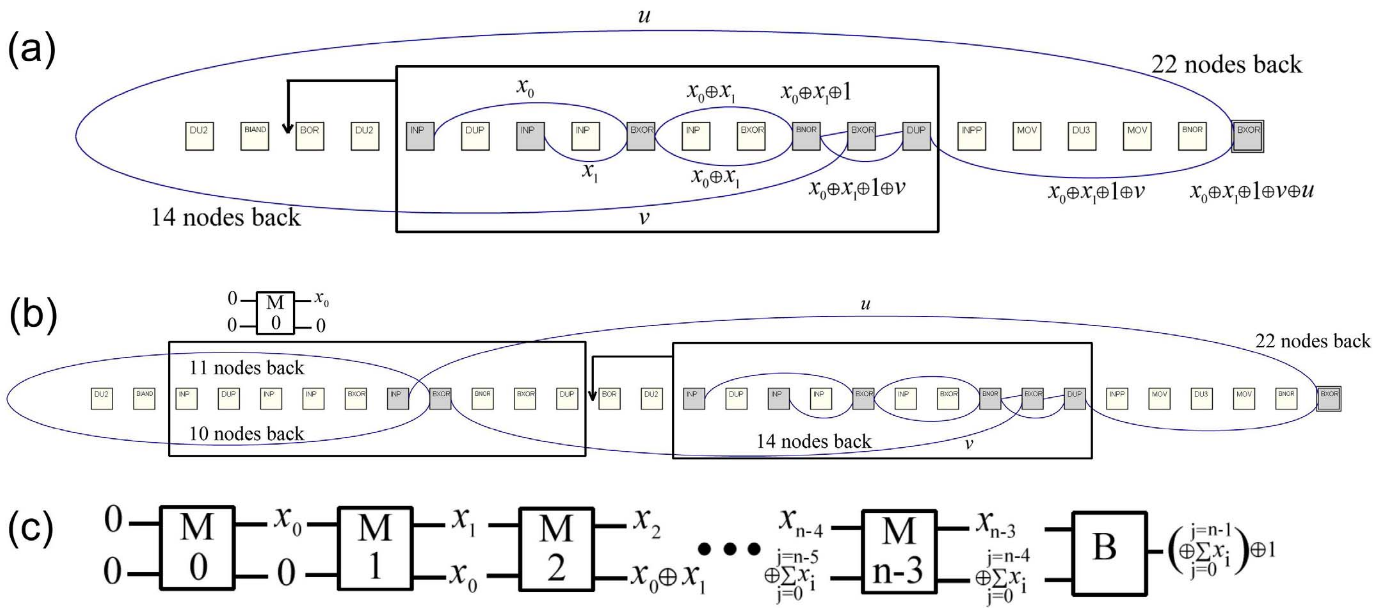

Fig. 5. The general solution for n even parity. See section VII-C for discussion.

parity functions than either CGP or modular CGP. We have applied it to a number of other problems with excellent results and these will be published in due course.

REFERENCES

[1] W. Banzhaf, G. Beslon, S. Christensen, J. A. Foster, F. Kps, V. Lefort, J. F. Miller, M. Radman, and J. J. Ramsden, "From artificial evolution to computational evolution: A research agenda," *Nature Reviews Genetics*, vol. 7, pp. 729–735, 2006.

[2] G. Kampis, *Self-modifying Systems in Biology and Cognitive Science*. Pergamon Press, 1991.

[3] L. Spector and K. Stoffel, "Ontogenetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Stanford University, CA, USA: MIT Press, 28–31 1996, pp. 394–399.

[4] F. Gruau, "Neural network synthesis using cellular encoding and the genetic algorithm." Ph.D. dissertation, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994.

[5] J. F. Miller and P. Thomson, "A developmental method for growing graphs and circuits." in *ICES*, ser. Lecture Notes in Computer Science, A. M. Tyrrell, P. C. Haddow, and J. Torresen, Eds., vol. 2606. Springer, 2003, pp. 93–104.

[6] S. Kumar and P. Bentley, *On Growth, Form and Computers*. Academic Press, 2003.

[7] S. L. Harding, J. F. Miller, and W. Banzhaf, "Self-modifying cartesian genetic programming," in *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, D. Thierens, H.-G. Beyer, and et al, Eds., vol. 1. London: ACM Press, 7-11 Jul. 2007, pp. 1021–1028.

[8] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge Massachusetts: MIT Press, 1994.

[9] J. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*. Cambridge, Massachusetts, USA: MIT Press, 1992.

[10] R. Poli and J. Page, "Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code gp and demes," *Genetic Programming and Evolvable Machines*, vol. 1, no. 1-2, pp. 37–56, 2000.

[11] L. Huelsbergen, "Finding general solutions to the parity problem by evolving machine-language representations," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, and et al., Eds. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann, 22-25 Jul. 1998, pp. 158–166.

[12] M. L. Wong and K. S. Leung, "Evolving recursive functions for the even-parity problem using genetic programming," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. E. Kinnear, Jr., Eds. Cambridge, MA, USA: MIT Press, 1996, ch. 11, pp. 221–240.

[13] M. L. Wong and T. Mun, "Evolving recursive programs by using adaptive grammar based genetic programming," *Genetic Programming and Evolvable Machines*, vol. 6, no. 4, pp. 421–455, 2005.

[14] T.-H. Hoang, R. McKay, D. Essam, and X. H. Nguyen, "Developmental evaluation in genetic programming: A position paper," *Frontiers in the Convergence of Bioscience and Information Technologies, 2007. FBIT 2007*, pp. 773–778, Oct. 2007.

[15] T. G. Gordon and P. J. Bentley, "Development brings scalability to hardware evolution," in *EH '05: Proceedings of the 2005 NASA/DoD Conference on Evolvable Hardware*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 272–279.

[16] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. of EuroGP 2000*, ser. LNCS, R. Poli and W. Banzhaf, et al.,, Eds., vol. 1802. Springer-Verlag, 2000, pp. 121–132.

[17] V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," in *Proc. of ICES*. Springer-Verlag, 2000, vol. 1801, pp. 252–263.

[18] T. Yu and J. Miller, "Neutrality and the evolvability of boolean function landscape," in *Proc. of EuroGP 2001*, ser. LNCS, J. F. Miller and M. T. et al., Eds., vol. 2038. Springer-Verlag, 2001, pp. 204–217.

[19] J. A. Walker and J. F. Miller, "Investigating the performance of module acquisition in cartesian genetic programming," in *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2005, pp. 1649–1656.