Towards Development in Evolvable Hardware

Timothy G. W. Gordon and Peter J. Bentley
Department of Computer Science
University College London
Gower Street, London
UK, WC1E 6BT
{t.gordon, p.bentley}@cs.ucl.ac.uk

Abstract

Mapping between genotype and phenotype using a model of biological development has been widely touted as a technique for evolving solutions to large, complex problems [1-3]. Here we describe two test-bed developmental systems for evolvable hardware problems, and compare each to a naive mapping system. We find that designing evolvable developmental systems is not a trivial problem, however early analysis of the evolved structures demonstrates the potential of the generative processes behind development. We also account for the differences between the results of the two systems, highlighting the importance of search space evolvability over size.

1. Introduction

As a problem becomes more complex it generally becomes increasingly difficult for an evolutionary algorithm (EA) to find acceptable solutions. This is known as the scalability problem, and has been reported by many evolvable hardware researchers as a serious issue that prevents the evolution of large, complex circuits [4-7]. Consequently finding mechanisms to increase the evolvability of such design spaces is crucial to furthering the field.

Approaches to the scalability problem for evolvable hardware include Function Level Evolution [5], Cartesian Genetic Programming [4] and Incremental Learning [8]. A new approach that is gaining popularity is to use a model of biological development to map between genotype and phenotype. Not only does this allow evolution to search for useful representations, it can introduce the types of features that may aid evolvability, such as redundancy, modularity and local learning. In the field of evolvable hardware, Koza et al. [2], Hemmi et al. [9] and Lohn and Columbano [10] have pioneered use of such systems. However there is still much to be understood as to which features of developmental systems aid evolvability and scalability. This paper presents the progress we have made in developing test-bed developmental systems to explore evolvability and scalability, along with early results from two of these systems, and analysis that demonstrates their potential.

The rest of the paper is structured as follows: section two discusses why development is such a good candidate for tackling the scalability problem. Section three describes and presents results from the first of two developmental systems discussed in this paper, along with analysis of the evolved circuits. Section four describes the second developmental system with results and analysis. Conclusions and plans for future work are given in section five.

2. Improving Scalability

In general a solution found by an inductive learning algorithm does not follow deductively from the information provided by the training data and the solution description. The set of additional assumptions needed to ensure that the solution follows deductively is called the inductive bias [11], and can be further categorised into representational bias and algorithmic bias [12]. Solving a learning problem is a matter of discovering and implementing a set of inductive biases that are suitable for the problem at hand. However this is often extremely hard.

For example, one common approach to simplifying large evolvable hardware problems has been to reduce the search to a smaller space. The most common tactic is to choose primitives that impose a stronger bias to hopefully limit the search to useful areas of space (but do so without ruling out interesting areas of space). This

modification of the representational bias is the approach of function level evolution, which was developed at ETL [5,13]. The difficulty here is in choosing the correct structures to use in the representation. Any abstraction used makes assumptions about the type of problem. Therefore problem-dependent components may have to be developed again and again. Once this trade-off has been made, evolution is now limited to search the space of this abstraction, and any innovative solutions at the lower abstraction will be unattainable. In addition, modules must be chosen that provide a space tractable to evolution, otherwise we may be even worse off than before. It may be that in some cases abstractions that transform the space into something of the same size (or bigger) that is more tractable to evolution are more useful than abstractions that humans find useful, applied blindly to reduce the size of the space. However the problem of how to find and impose useful biases still remains.

At this point let us consider how the complex features that we possess came about. Certainly it is extremely unlikely that such highly complex and well-optimised characteristics should arise overnight by chance. Many features tend to be simpler in organisms that evolved early in the history of life. For instance mechanisms of gene expression are far less complex, and seem to provide fewer avenues for evolvability in lower organisms. Such reasoning resulted in Dawkins [14] suggesting that evolvability itself has evolved. In more recent years this idea has become widely accepted [15,16], and experimental evidence of mechanisms to evolve evolvability have been demonstrated [17]. In terms of a learning algorithm, we can think of evolution as altering its inductive bias in the hope of finding one more appropriate to the current environment. So natural evolutionary systems not only possess biases that aid evolvability, but they possess the ability to shift their bias, thereby performing a meta-search of bias space. Hence they adapt their search, in addition to their phenotypes, to the prevailing conditions.

2.1 Development

Development can be thought of as one of biology's representational bias search mechanisms. Biological development essentially maps genotype to phenotype through a complex process of regulated gene expression. As the developmental process itself results merely from expression of genes in a given environment, it is also under the control of evolution. But development's basic mechanism, gene expression, allows the formation of complex gene regulatory networks (GRNs). These are networks of gene products that regulate the expression of each other. Such networks form modular, iterative and recursive patterns. When some of these gene products are involved in (or trigger) the formation of some biological function or structures, modular, iterative and recursive morphologies can arise [18]. Variation of genes within a module allows space to be searched in "leaps" involving this module, rather than the small steps of its components. If a master control gene is expressed multiple times, the module is reused [16]. Multiple iterations across time are possible through feedback loops within a module or between modules.

Another form of iteration is across space. In multicellular organisms these processes occur in a distributed manner. The environment of particular cells results in different genes, hence perhaps different modules, being activated. This allows the generation of regular, iterative patterns across space as cells differentiate according to their environment. Recursive patterns across both space and time are also possible, through the same processes.

In this way modules can impose a strong representational bias, biasing evolution to work with more complex primitives, but allowing re-use, iteration and recursion of these primitives, either across the structure of the organism, or across time to allow regulatory patterns to form. It is this re-use that is of most interest in any study that centres on improving scalability. A developmental module that is re-used more and more often in good solutions as the problem scales leaves less and less of the problem that has to be learned from scratch. So as problem complexity scales, finding a solution becomes increasingly easier relative to a system that cannot re-use modules.

2.2 Development in Evolutionary Algorithms

Developmental systems have been used in EAs since Dawkins [14] demonstrated that an explicit genotype-phenotype map could easily be used to affect evolvability. He presented several handcrafted developmental systems that altered the morphology of geometrical patterns. He also noted that these mappings could be evolved, and were likely to be evolved in natural systems. Since then the idea of evolving a developmental system rather than the phenotype itself has been explored in a number of directions. Bentley and Kumar [19] noted that these can be divided into two approaches, explicit and implicit.

2.2.1 Explicit Approaches

Explicit approaches use a mapping that explicitly provides the properties of hierarchical modularity, iteration and recursion that make development so useful. Such is the approach taken by Cellular Encoding, which was first developed by Gruau to develop ANN architectures [20] but is perhaps better known through

Koza et al.'s evolution of analogue circuits [2]. The basic technique is to evolve trees of developmental steps using genetic programming. Each developmental step, encoded as a GP node, explicitly codes for a phenotype modification. A fixed 'embryonic' phenotype is 'grown' by applying a tree of rules to it. Automatically Defined Functions can explicitly provide modularity, and Automatically Defined Loops/Iterations can provide iteration. Lohn and Columbano use a similar approach [10], but with a linear mapping representation which is applied to an embryonic circuit in an unfolding manner, rather than a circuit modifying one.

2.2.2 Implicit Approaches

The perhaps more elegant implicit approach uses sets of production rules rather than explicit mechanisms for generating modularity, iteration and recursion. The idea behind these is that complex objects can be defined by successively rewriting a symbolic description of a simple object according to the set of production (or rewriting) rules. A program to map between genotype and phenotype is specified by a start symbol (or set of symbols) for the rule rewriting process. Usually this is fixed and the grammar is evolved. Hence the fixed program is implicitly evolved though alteration of its grammar. The dynamics of an evolutionary system working on a set of grammar rules is clearly quite different from that of a program-modifying system, and there is empirical evidence to suggest that such an approach may be more scalable for at least a limited set of problems [19,21]. One example of a system that uses production rules to evolve hardware is [9]. Here rules were evolved to generate hardware description language (HDL) descriptions of circuits. It is an interesting approach that has the potential to evolve extremely large circuits, albeit at a very high-level abstraction that leaves little room for low-level hardware innovations.

Many of these systems are based around class of production rules called Lindenmayer systems (L-systems), which were proposed specifically to model plant development [22]. This class of systems is defined by the parallel application of the complete set of rules at each rewriting timestep, rather than the sequential application used by more traditional production rule systems [23]. Such an approach models the parallel division of cells in nature more closely. Thus L-systems achieve the complexity, iteration and recursion inherent to hierarchical mapping systems, that has been identified as a key component of biological development. In light of this, they have been applied to a number of evolutionary design problems [24,25] and have been proposed for evolving circuits [1,26]. A system designed specifically to evolve circuits have recently been reported [7]. However the results do not provide much insight into their viability for large circuit design problems owing to their preliminary nature. Another recent approach of particular interest is the use of parametric context-free L-systems (P0L-systems) that allow external environmental parameters to guide development [24], unlike traditional context-free systems. Work on these is at an early stage, but this type of L-system looks promising.

A number of similar approaches have also arisen from the study of biological systems. Some of the earliest developmental systems used cellular automata (CA) model the interaction between cells. The CA rules specify how each cell should react to the states of the surrounding cells. This sort of approach has many similarities with context driven L-systems. However there are some differences. First, the product of the rule interaction is usually not a program to generate the solution, but the solution itself. This models biological systems more closely. Secondly, and perhaps most importantly, CA is inherently driven by spatial context. Several researchers have reported promising results using CA-based biological techniques. For instance, de Garis [27] used a CA model of biological development to 'grow' ANN architectures. However rather than evolving the CA rules he used hand-coded rules, and evolved the CA starting conditions. The use of CA to evolve hardware has also been advocated by those at EPFL [28]. Our work focuses on this approach rather than a more mathematically rooted L-system approach.

3. Differentiation-based Development

The first developmental system explored here aims to model features of cellular differentiation in order to generate circuits. Cellular differentiation is an aspect of biological development exhibited by all Metazoan organisms [29], and is one of nature's key methods of generating complex iterative structures. At the heart of the differentiation process is DNA transcription.

Transcription from DNA involves the following steps [30]. First a protein called RNA polymerase binds to a site at the start of the gene sequence called the promoter. Once bound, the RNA polymerase travels along the sequence, generating the RNA. The rate at which genes are transcribed (hence expressed) is controlled by the presence of more proteins called transcription factors. These are called activators or repressors, depending on whether they increase or decrease the rate of gene transcription. They work by binding to specific sequences of DNA upstream of the gene, and then modulate the ability of RNA polymerase to bind to the promoter. Typically many transcription factors are needed to stabilise the binding of RNA polymerase to the promoter. All the transcription factors are proteins that are coded

for by other genes. Thus a dynamic network of gene products specifies which genes are expressed.

3.1 The Rule Design

This interaction of genes and proteins has been modelled in the developmental system presented here. Proteins are modelled as binary state variables - they are either present or not. The element corresponding to a biological gene is a rule, and a chromosome is essentially just a set of rules. An example rule is shown in Fig. 1. Each rule has two parts, a precondition and a postcondition. The precondition determines which of a number of proteins are transcription factors for the rule, i.e., which proteins must be present or absent for the rule to be activated. The precondition of the example rule in Fig. 1 codes for five proteins. (Five proteins were used in all experiments reported here.) Within the precondition each protein is coded by a two bit locus. If the protein must be present for the rule to activate, the locus has the binary value 11. In the example in figure 1, proteins A and E are coded as 11. We can think of these proteins as activators. If the protein must not be present for the rule to activate, the locus has the value 00. (For instance protein D in the example.) In this case we can think of the protein acting as a repressor. If a protein is coded as 01 or 10, for instance proteins B and C in the example below, they are "don't care" terms where the protein has no effect. The postcondition is simply a key to a lookup table that determines the effect that gene expression has on the phenotype. These can either generate a protein as the example below does, or indirectly alter the structure of the circuit as described sections 3.2 and 3.3. Protein concentration is not modelled - a protein is either generated or not, and in turn a protein is either detected or not.

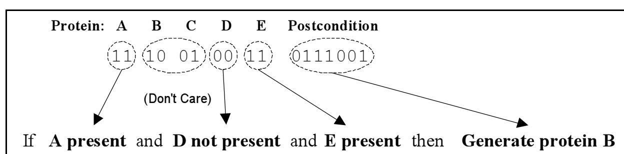

Fig. 1. An example developmental rule.

With this model of transcription, developmental modules can be formed through the dynamic network of gene expression. It is up to evolution to generate functional modules through the network if it so needs. Each individual contains a fixed number of rules, but evolution is free to evolve all the bits within the rules. Therefore several rules may have identical preconditions or postconditions, allowing for redundancy at the genetic level. Genes are expressed at each of a series of developmental timesteps. The effects of gene expression can be split into two classes, those that generate more proteins and those that affect the structure of the circuit, as described below. The mechanism by which changes are made to the circuit structure is based on the process of cellular differentiation.

3.2 The Cell Design - Protein Development

To demonstrate iteration across space we modelled the circuit design space as a cellular structure, consisting of a number of identical functional units. The circuits were evaluated using a Field Programmable Gate Array (FPGA). As these consist of an array of identical functional units called configurable logic blocks (CLBs), it was decided that each cell to be evolved should map to a distinct CLB. Each cell has five types of components - inputs, function generators, outputs, a protein detector and a protein generator. The protein detector records what proteins are present in the cell's environment at timestep t. (If the majority of the neighbouring cells are producing a protein, it is detected by the cell's detector.) This is achieved by querying the protein generators of the surrounding cells. The protein generators simply record what proteins will be produced by a cell at timestep $t+1$ through the activation of rules that have postconditions corresponding to production of one of the proteins. The interaction between protein generators and detectors is shown in Fig. 2.

We can think of this as a type of environmental interaction - there is an environment of cells and proteins. A cell is affected by its *context*, the nature of the surrounding cells, as a result of the protein interactions.

3.3 The Cell Design - Mapping to Virtex

The remaining three components of the cell are functional - inputs, LUTs and outputs, and are based on the Xilinx Virtex architecture [31]. Previous work used the design shown in Fig. 3a. Each cell had four inputs, which correspond to the four inputs of a Virtex LUT. Each input could be connected to the cell output from the north, east, south or west. However the Virtex CLB is split into two slices, each containing two LUTs labelled the F and G LUTs. To make better use of the hardware the cell design was altered to incorporate two LUTs, each with an independent output as shown in Fig. 3b. Each cell maps directly to a Virtex CLB, and these cells are arranged as an array to make the evolved area, as shown in Fig. 3c. Where possible, the north, south, east and west connections are each mapped to a manually selected single line leading in one of these directions that can be routed to the each of the four inputs. [1] The LUT outputs

[1] For full details of lines used, contact the authors.

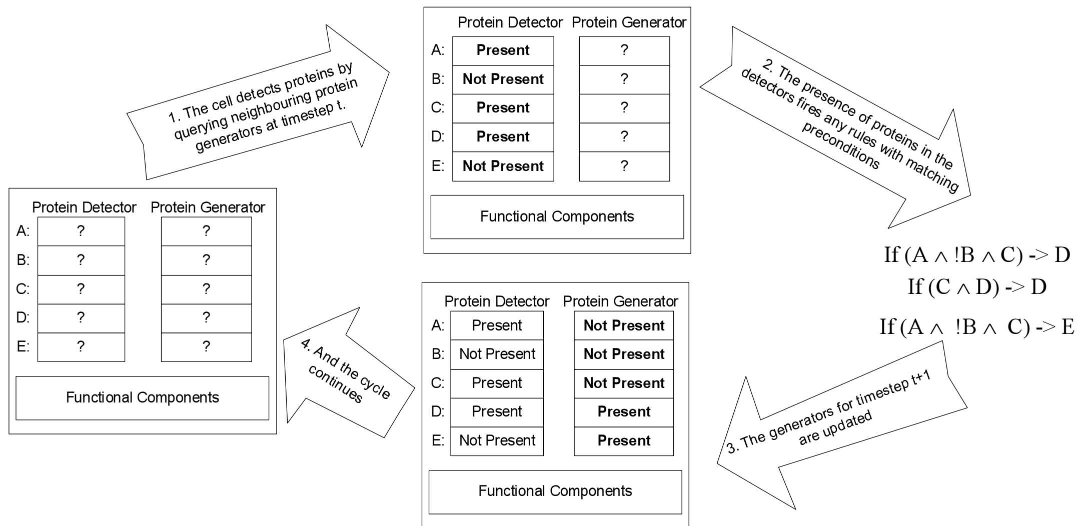

Fig. 2. The protein cycle undergone by each cell.

are each mapped through four of the eight Virtex CLB output multiplexers to single lines manually selected to carry signals in the required directions without causing contention with neighbouring outputs. The entire second slice of the CLBs, and the carry logic and flipflops from the first slice are not evolved.

As no registers are used, unclocked feedback loops can occur between the LUTs of several cells. This potentially allows for the evolution of circuits outside conventional design spaces as the digital design rules of synchrony have been relaxed. In addition no clock has been provided. (Although the signal used to clock the configuration and data transfer interface circuits pervades the FPGA and may be sequestered by unconventional means to allow synchronisation.)

3.4 Circuit Development

We have discussed how protein generation is controlled by development, and how the functional elements of the cell (the inputs, LUTs and outputs) map to the Virtex architecture. Now we discuss how these functional elements are controlled by development.

In the Virtex architecture, inputs from different directions should not be routed to a given LUT input concurrently. Because of this a rule postcondition cannot be allowed to select an input source directly. Instead the source of each cell's four inputs is determined by a competition between its eight possible inputs, two from each of the four surrounding cells, as shown in Fig. 4. So the only rule postconditions required to control input selection are ones that increase the score of a particular input in this competition. This totals 32 postconditions (4 inputs x 8 sources). The element with the highest score at any timestep is selected as the current input. In the event of a draw the winner is selected arbitrarily. The two LUTs share the same four winning cell inputs.

The LUT functions are controlled by a related scoring method, also again shown in Fig. 4. Postcondition keys exist for each of the 32 P-terms (16 in each LUT). A score of how many times a particular P-term postcondition has been activated is kept. At any developmental timestep the state of each LUT P-term entry is determined by the value of the corresponding P-term score. If the score is above a threshold value, the LUT entry is set to true. If it is below a threshold value the LUT entry is set to false. The threshold value for both LUTs is set to the expected P-term score if a set of random rules were fired. Thus it is dependent on the number of proteins and rules used. Unlike the inputs, the Virtex architecture allows each of the two outputs of the cell to connect to all four output directions simultaneously. Hence its configuration is dealt with in a similar manner to the LUTs - i.e. a threshold value rather than a competition. In this way the cell can be mapped to the Virtex architecture at any point during its development, using the Xilinx JBits API.

3.5 Experiments - Evolving Two Bit Adders

The evolution of adders has been well studied in the past [32-35] and the adder design space is very well understood, particularly in the digital domain. We have suggested that one of the primary uses of development is the learning of a bias that embodies a useful design abstraction. It is known that this problem can be described using such a design abstraction - as a combinational circuit that makes no use of feedback or memory. So while a naive evolutionary system working at a low design abstraction may struggle to solve this

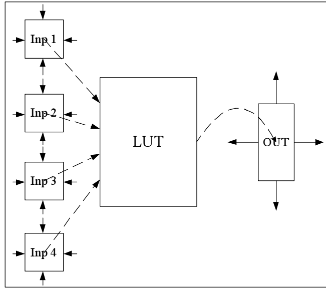

Fig 3a. The original developmental cell. Dashed connections are fixed.

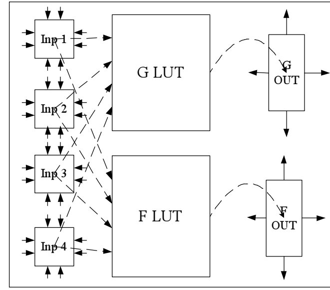

Fig 3b. The final developmental cell. Dashed connections are fixed.

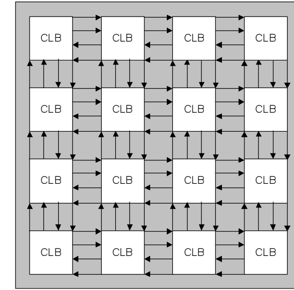

Fig 3c. An array of the final cells each mapped to a Virtex CLB

problem (especially as the size of the problem is scaled), if a suitable developmental process learns to incorporate this abstraction into its bias it may be able to simplify its search. (It may be the case that evolution does not find this abstraction as useful as humans do, but it is a useful starting point.) So we have focused on the evolution of adders for our initial experiments. The experiments presented here set out to evolve two bit adders with carry using a genetic algorithm (GA).

Fitness evaluation was carried out on a Xilinx Virtex XCV400 mounted on an Alpha Data ADM-XRC PCI card [36]. A 2x5 cell area of the chip was selected for evolution using the current developmental system. This area was selected to expose a similar reconfiguration area as used successfully in [37].

Additionally a naive genotype/phenotype mapping system was tested on the same problem using the same circuit components. The naive 52 bit representation for one cell is shown in Table 1. (The concatenation of 10 cells yields a naive chromosome length of 520 bits.)

For both systems, in order to prevent contention and reduce noise, inputs and outputs on the edges of the evolved area other than those specified by the problem were forced off. The five cells on the west edge were provided with five input signals of the two bit adder with carry problem, A0, A1, B0, B1 and CIn, as their eastbound signals. The task was to evolve a circuit that mapped inputs to the three outputs Sum0, Sum1 and COut in accordance with the two bit adder with carry truth table. Fitness was measured as the total correct output bits across all input combinations, which in this case gives a maximum fitness of 96. The current developmental system uses 5 distinct proteins. Hence each rule precondition is coded in 10 bits. All circuit-modifying rule postconditions can be coded in 7 bits. 80 rules are used, yielding 1360 bits for the rule section of the chromosome. Initial protein states are also evolved.

A bit signifying the initial concentration of each protein for each cell yields an additional 5x10 = 50 bits, making a total chromosome length of 1410 bits. Each individual was allowed to develop for 30 steps.

Table 1. Naive representation for the 2 bit problem.

Locus	Component	Bits (Representation)
0-2	Input 1	3 (GN,GS,GE,GW, FN,FS,FE,FW)
3-5	Input 2	2 (GN,GS,GE,GW, FN,FS,FE,FW)
6-8	Input 3	2 (GN,GS,GE,GW, FN,FS,FE,FW)
9-11	Input 4	2 (GN,GS,GE,GW, FN,FS,FE,FW)
12-27	GLUT	16 (16 P-terms)
28-43	FLUT	16 (16 P-terms)
44	Output GN on	1
45	Output GS on	1
46	Output GE on	1
47	Output GW on	1
48	Output FN on	1
49	Output FS on	1
50	Output FE on	1
51	Output FW on	1

These values have been chosen as the observed best parameters during informal experimentation with a range of settings.

The genetic parameters, held constant for both representations followed earlier experiments to evolve a two bit adders [37]. Two-member tournament selection was used. A tournament selection pressure as described by Miller [33] was also introduced, and set to 0.8, i.e. the winner of each tournament was selected only with 80% probability. One-point crossover and simple mutation were used. The population size was set to 100 and the mutation rate was set to an expected five mutations per individual. The first generation of each run was randomly generated. Evolution was halted after 2500 generations.

In order to ensure generalisation across any order of input sequences, the order of sequence presentation to the circuit under evaluation was randomised. The genetic representation allows for circuits that may not generate the same fitness when evaluated twice - the outputs may exhibit dynamical variations unrelated to the inputs.

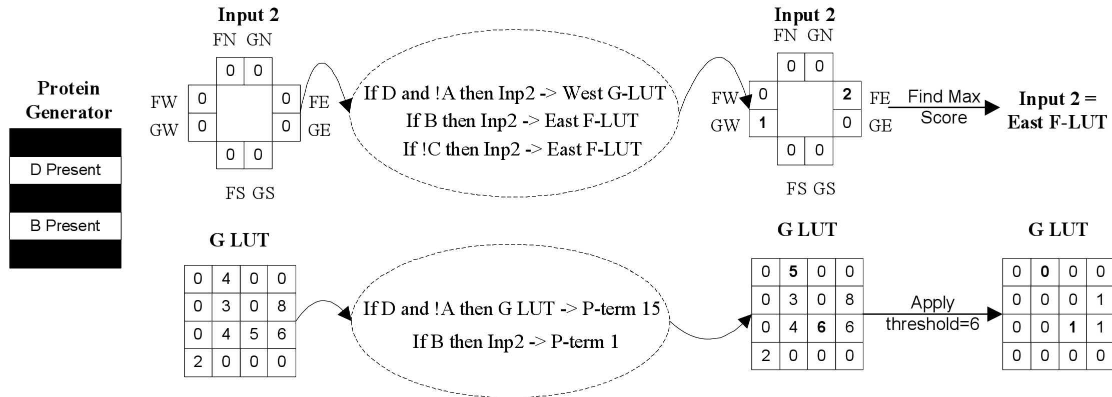

Fig. 4. An input differentiates towards two possible sources, but the East F-LUT finally wins. Also a LUT differentiates, and then is thresholded to yield its final function.

Although such circuits are not useful final solutions they may contain valuable information about how to solve part of the problem, or how to traverse the fitness landscape. Because of this each individual was evaluated five times, and its worst fitness selected.

Early results for both the developmental system and the naive representation are shown in Table 2. Runs using random search across the same number of evaluations are also presented for both representations.

Table 2. Results from 5 evolutionary runs with and without a developmental systems, & 5 random searches with and without a developmental system.

System	Mean Fitness of Best Solns.	Std. Dev. of Best Solns.	Best Soln.
Developmental Evolution	80.20	3.30	84
Naive Evolution	91.60	4.10	96
Developmental Random Search	69.2	3.34	72
Naive Random Search	63.4	1.34	64

The results show that the developmental system does not evolve as fit solutions as the naive mapping system, and unlike the naive system, it has not succeeded in finding a fully functional two bit adder with carry. Note the difference between the improvement in mean fitness of the best solutions ($\overline{f}_{best}$) as we move from random search to the GA search. When using a naive representation this increase is large. When using the developmental system, this is much smaller. This suggests that the reason for the low performance of the developmental system is that it is not as evolvable as the naive mapping system on the two bit adder problem. This is not completely surprising, as there are some disadvantages to this kind of system. Firstly, any additional layer of mapping from genotype to phenotype is likely to bring about increased epistatic interactions if a phenotypic feature relies on the presence of more than one gene product. In addition, although rule-based systems provide an advantage in robustness through the distribution of computation and the resultant possibility for redundancy, the fixed lengths of the rules in this system ensure that building blocks composed of more than one gene product will have defining lengths of at least two rules. Hence if the rule is long, they have a fixed, and reasonably high, minimum probability of disruption by crossover. But the most important impediment to evolvability is that we are introducing extra work for the GA by allowing it to search bias space. This means that evolution must find a representation that is evolvable before evolution can operate to find a successful solution. The results suggest that at least for this problem, evolution may not have done so before the stopping conditions are reached.

Also random search on the developmental search space outperforms the random search on the naive search space by some margin. This suggests that even though the developmental space is not as evolvable as the naive system, the density of better solutions may be higher in the developmental space than the naive space.

Despite lower mean fitnesses, the developmental system did demonstrate significant results because of the way it solved the problem. Fig. 5 shows various elements of the best circuit discovered with the first run, in various states of the development. The fully developed F-LUTs for each cell are shown in Fig. 5a. Each LUT is represented by a Karnaugh map (K-map), with white representing a true state and black a false. Two different K-map configurations can be identified, laid out in a regular, symmetrical pattern. These are the kinds of patterns one might expect to see if we had hand-designed K-maps for a traditional ripple-carry adder. However in this case the patterns have been created with a handful of rules activated by the presence or absence of proteins. In

fact with such a set of rules it is possible to generate patterns that repeat indefinitely if the number of cells available to development were increased. This is precisely what would be needed if we wished to design a large ripple-carry adder. Hence here we see one of the key advantages of generative processes in action - the ability to generate large iterative structures. With this in mind it could be argued that generating larger adders would not tax evolution much more than our two-bit with carry example. We anticipate that the ratio of best evolved fitness to perfect fitness may increase as we move to larger adders, because the proportion of the circuit where symmetry must be broken to achieve a perfect score (in this case the top and the bottom) is reduced.

The two distinct patterns shown in the final K-maps suggest that the process has indeed differentiated the cells, which were all originally identical in function, to two different functions. We can corroborate this by examining the development of the cell protein concentrations. Fig. 5b shows the initial starting conditions, where each horizontal stripe corresponds to one of the proteins, a white stripe denoting the protein is present in the cell. Little regularity can be picked out amongst the plethora of different cell states. Fig. 5c shows the final concentrations. Again two distinct states can be seen. The distribution of these matches the two distinct LUT functions shown in Fig. 5a. This reaffirms the suggestion that the process of development has differentiated all the cells into two distinct types, and no other precursor cell types remain.

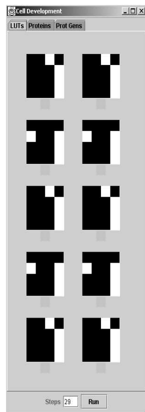

Fig. 5a F-LUTs of the best evolved adder

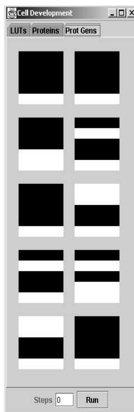

Fig. 5b Proteins of the undifferentiated cells

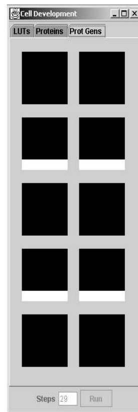

Fig. 5c Proteins of the final differentiated cells

4. Incorporating Aspects of Growth

One possible factor contributing to the evolvability of the developmental system we discussed above is the length of the rules. There were three possible methods of reducing rule length. Firstly the precondition size could be reduced, either by reducing the number of proteins, or encoding the proteins in less than two bits each. Informal experiments with the first cell representation suggested that these options had little effect on the performance of the system. Secondly the postcondition size could be reduced, by reducing the number of postcondition keys. To investigate this possibility, a new LUT development representation was devised. The original development system used a unique rule postcondition for each P-term of each LUT. Hence 32 rules were needed for LUT rules alone. This system had been selected to employ as little bias as possible in the patterns of LUT P-terms that development used. As rules had to fire many times before a P-term became active, it also provided a mechanism for gradual change in the state of the LUT, modelling natural differentiation processes.

The new postcondition system more closely resembles growth. Rules corresponding to individual P-term activation were removed. Instead the LUT was modelled as a K-map with only one active P-term at any given time. Rules were introduced to change the active P-term by moving it up, down, left or right by one step on the K-map. A final rule sets the active P-term as true in the final, developed circuit. (At developmental step 0 the active P-term always begins at P-term 0.) An example growth step is given in Fig. 6.

With these rules development can generate a function by navigating the K-map, setting P-terms as it goes. The reduction from 32 rules to 10 rules (5 for each LUT) allow the complete set of postcondition keys to be coded in six bits rather than the previous seven, reducing the rule length by one bit to 16b and the chromosome length from 1410b to 1330b. (Note that this system introduces an initial bias towards functions represented by P-terms set near the edges of a K-map, centred at P-term 0.) The experiments to evolve two bit adders were repeated with this change. The results are shown in the Table 3.

Table 3: Results from 5 runs of 2 bit adder evolution with LUT growth rather than LUT differentiation

Mean Fitness of Best Solns	Std. Dev. of Best Solutions	Best Solution Found
75.6	2.61	80.0

The results suggest that the introduction of the growth system has resulted in a small drop in performance. (A t-test revealed 97% probability that there was a significant difference between the mean best fitnesses.) This demonstrates that reducing the search space does not necessarily increase performance - a more important factor is the evolvability of the landscape as defined by the biases we use for the search. Here the additional bias

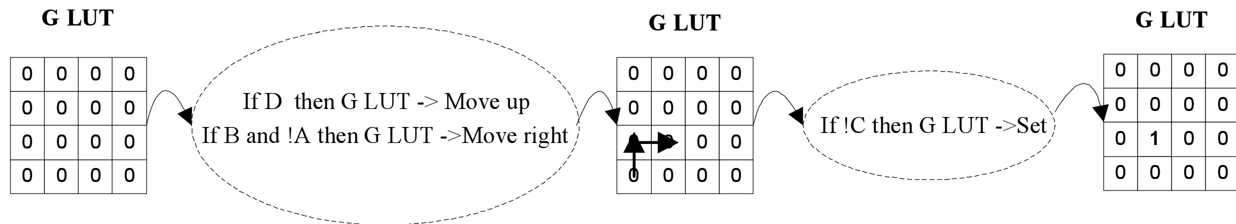

Fig. 6. Two rules fire to move to P-term 13, then a rule fires to set it in the final K-Map

we have introduced does not seem conducive to discovery of adder functions.

It is worth taking a subjective look at the patterns generated by the two different systems. Fig. 7 shows K-maps of the F-LUTs of typical circuits generated by both systems. We see that the pattern for the differentiation LUTs (Fig. 5a and Fig. 7a) there is no immediately evident trend in the LUT patterns generated by various individuals. In the case of the growth LUTs it is interesting to note that more regular patterns are evident, in particular continuous lines, as in the example shown in Fig. 7b. These lines are quite easily generated through the reuse of 'move' and 'set' rules throughout development. Such patterns are typical of the K-maps generated by the growth system. The ability to generate regular patterns like these is potentially useful. For instance if we wished to evolve logic that is to be mapped to traditional gate technologies or gate-based programmable logic, we may find that lower fanin, more gate-efficient solutions evolve when using these types of rules, as evolution will favour more minimised forms of logic.

5. Conclusions

The problem of scalability is one of great importance if evolvable hardware is ever to achieve much impact in the real world. In this paper we have focused on the approach of evolving developmental systems.
This approach allows evolution to search for good inductive biases for solving large-scale complex problems as it generates inherently modular, iterative structures that exist in many real-world circuit designs, but at the same time allows evolution to search innovative areas of space where it sees fit. We have presented two developmental systems that allow us to model the features of modularity and local interaction in biological developmental systems, but for an engineering use. The results have demonstrated the ability of a developmental process to generate coherent and useful patterns of differentiated cells in circuit designs. There still remain problems of evolvability of these systems, which current work is addressing. Nevertheless, this work provides some crucial steps towards applying the generative power of development to solving large circuit design problems in the future.

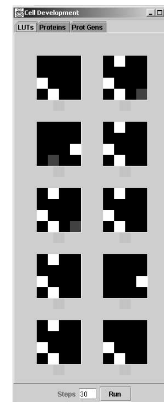

Fig. 7a. F-LUTs generated by a typical run of the differentiation-based system

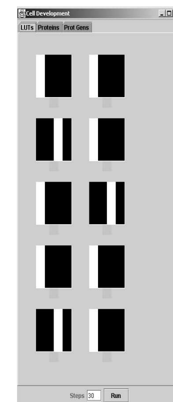

Fig. 7b. F-LUTs generated by a typical run of the growth-based system

References

[1] P. C. Haddow and G. Tufte, "Bridging the Genotype-Phenotype Mapping for Digital FPGAs," Proc. of the 3rd NASA / DoD Workshop on Evolvable Hardware, Pasadena, California, U.S.A., 2001.

[2] J. Koza, F. H. I. Bennett, D. Andre, and M. A. Keane, *Genetic Programming III*. San Francisco, California, U.S.A.: Morgan-Kaufmann, 1999.

[3] A. Thompson, *Hardware Evolution*. London, U.K.: Springer Verlag, 1998.

[4] V. K. Vassilev and J. F. Miller, "Scalability Problems of Digital Circuit Evolution," Proc. of the 2nd NASA/DOD Workshop on Evolvable Hardware, Los Alamitos, California, U.S.A., 2000.

[5] W. X. Liu, M. Murakawa, and T. Higuchi, "ATM cell scheduling by function level evolvable hardware," in *Evolvable Systems:*

From Biology to Hardware, vol. 1259, *Lecture Notes in Computer Science*, 1997, pp. 180-192.
[6] I. Kajitani, T. Hoshino, M. Iwata, and T. Higuchi, "Variable length chromosome GA for Evolvable Hardware," Proc. of the 3rd Int. Conf. on Evolutionary Computation, Nagoya, Japan., 1996.
[7] P. C. Haddow, G. Tufte, and P. van Remortel, "Shrinking the Genotype: L-systems for EHW?," The 4th Int. Conf. on Evolvable Systems: From Biology to Hardware, Tokyo, Japan, 2001.
[8] J. Torresen, "Scalable Evolvable Hardware Applied to Road Image Recognition," Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, Silicon Valley, USA., 2000.
[9] H. Hemmi, J. Mizoguchi, and K. Shimohara, "Evolving Large Scale Digital Circuits," Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems, Nara, Japan, 1996.
[10] J. D. Lohn and C. S.P., "Automated Analog Circuit Synthesis Using a Linear Representation," Proc. of the 2nd Int. Conf. on Evolvable Systems, Lausanne, Switzerland., 1998.
[11] T. M. Mitchell, *Machine Learning*. London: McGraw-Hill, 1997.
[12] D. Gordon and M. des Jardins, "Evaluation and selection of biases in machine learning," *Machine Learning J.*, pp. 5-22, 1995.
[13] M. Murakawa, S. Yoshizawa, T. Adachi, S. Suzuki, K. Takasuka, M. Iwata, and T. Higuchi, "Analogue EHW chip for intermediate frequency filters," in *Evolvable Systems: From Biology to Hardware*, vol. 1478, *Lecture Notes in Computer Science*, 1998, pp. 134-143.
[14] R. Dawkins, "The evolution of evolvability," Proc. of Artificial Life: The Quest for a New Creation, Santa Fe, U.S.A., 1989.
[15] P. Marrow, "Evolvability: Evolution, Computation, Biology," Proc. of the 1999 Genetic and Evolutionary Computation Conf. Workshop Program, Orlando, FL, U.S.A., 1999.
[16] G. Wagner and L. Altenberg, "Perspective--complex adaptations and the evolution of evolvability," *Evolution*, vol. 50, pp. 967-976, 1996.
[17] P. D. Turney, "Increasing evolvability considered as a large-scale trend in evolution," Proc. of the Genetic and Evolutionary Computation Conf. Workshop Program, Orlando, Florida USA, 1999.
[18] J. M. W. Slack, *From Egg to Embryo*, 2nd ed. Cambridge: Cambridge University Press, 1991.
[19] P. J. Bentley and S. Kumar, "Three Ways to Grow Designs: A Comparison of Embryogenies for an Evolutionary Design Problem.," Proceeding of the Genetic and Evolutionary Computation Conf. (GECCO '99), Orlando, Florida USA,, 1999.
[20] F. Gruau, *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm* Doctoral Thesis Ecole Normale Supirieure de Lyon, 1994.
[21] S. Kumar and P. J. Bentley, "The ABCs of Evolutionary Design: Investigating the Evolvability of Embryogenies for Morphogenesis.," Genetic and Evolutionary Computation Conf. (GECCO '99) Late Breakers, Orlando, Florida USA,, 1999.
[22] A. Lindenmayer, "Mathematical models for cellular interactions in development I Filaments with one-sided inputs," *J. Theor. Biol..*, vol. 18, pp. 280-289, 1968.
[23] N. Chomsky, *Syntactic Structures*. The Hague: Moutin and Co., 1957.
[24] G. S. Hornby and J. B. Pollack, "The advantages of generative grammatical encodings for physical design," Proc. of the Congress on Evolutionary Computation., Seoul, South Korea, 2001.
[25] E. J. W. Boers and H. Kuiper, *Biological metaphors and the design of modular artificial neural networks* Masters Thesis Leiden University, 1992.
[26] H. Kitano, "Challenges of evolvable systems: Analysis and future directions," in *Evolvable Systems: From Biology to Hardware*, vol. 1259, *Lecture Notes in Computer Science*, 1997, pp. 125-135.
[27] H. de Garis, L. S. Kang, Q. M. He, Z. J. Pan, M. Ootani, and E. Ronald, "Million module neural systems evolution - The next step in ATR's billion neuron artificial brain ("CAM-Brain") Project," in *Artificial Evolution*, vol. 1363, *Lecture Notes in Computer Science*, 1998, pp. 335-347.
[28] M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Heidelberg: Springer-Verlag, 1997.
[29] M. Kirschner and J. Gerhart, "Evolvability," *Proc. of the National Acadamy of Science*, vol. 95, pp. 420-8427, 1998.
[30] P. Raven and G. Johnson, *Biology*, 6th ed: McGraw-Hill Higher Education, 2001.
[31] Xilinx_Inc., *Virtex 2.5 V Field Programmable Gate Arrays Data Sheet*: http://direct.xilinx.com/partinfo/ds003.pdf, 2001.
[32] S. J. Louis and G. J. E. Rawlins, "Designer Genetic Algorithms: Genetic Algorithms in Structure Design," Proc. of the 4th Int. Conf. on Genetic Algorithms, San Diego, CA, U.S.A, 1991.
[33] J. F. Miller, P. Thomson, and T. C. Fogarty, "Designing Electronic Circuits using Evolutionary Algorithms. Arithmetic Circuits: A Case Study," in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications*, D. Quagliarella, J. Periaux, C. Poloni, and G. Winter, Eds. London, U.K.: Wiley, 1997.
[34] C. A. C. Coello, A. D. Christiansen, and A. H. Aguirre, "Towards automated evolutionary design of combinational circuits," *Comput. Electr. Eng.*, vol. 27, pp. 1-28, 2001.
[35] G. Hollingworth, S. Smith, and A. Tyrrell, "The Safe Intrinsic Evolution of Virtex Devices," Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, Palo Alto, CA, U.S.A., 2000.
[36] Alpha_Data_Parallel_Systems_Ltd., "ADM-XRC User Manual," 1.2 ed, 1999.
[37] T. G. W. Gordon and P. J. Bentley, "On Evolvable Hardware," in *Soft Computing in Industrial Electronics*, S. Ovaska and L. Sztandera, Eds. Heidelberg, Germany.: Physica-Verlag, 2002, pp. To appear.