Growing and Evolving Bitstreams in a Fixed Configuration Space

R. Timothy Edwards

Johns Hopkins University Applied Physics Lab,
11100 Johns Hopkins Road,
Laurel, MD 20723-6099, USA
tim@stravinsky.jhuapl.edu

Abstract. An indirect-mapping approach to evolvable algorithms is presented that is specifically intended for evolvable hardware. The conventional wisdom is that because hardware is configured by downloading a bitstream of a fixed size and hardware-specific format, hardware is not suited to genetic algorithms in which the size of the chromosome may vary. The algorithm presented herein departs from the traditional representation of a genome as the configuration bitstream of the hardware itself. Instead, the genome represents a set of rules from which the configuration bitstream is built, in a process analogous to biological growth, or ontogeny. Chromosomes are generated largely through a process of self-organization that determines both the length and content. Evolved circuits grow to a state of maturity, and demonstrate an effective model of hardware self-repair.

1 Introduction

The purpose of this research is to demonstrate the plausibility and practicability of an algorithm for evolvable hardware which has an *indirect mapping* between the genotype and the phenotype. The conventional wisdom is that because configuration bitstreams are of fixed size and format, the genome must also be of fixed size and format. This work shows that algorithms exist for which this restriction is lifted. The algorithm presented has unconstrained chromosome size. The size of a chromosome may vary from individual to individual in the population, even though every individual has the same target output configuration bitstream. For this system to be stable from generation to generation, genes must necessarily be clustered properly on either side of the splitpoint. Proper clustering results from a combination of *self-organization* and natural selection, as opposed to *mutation* and natural selection, an idea espoused by complexity theorists [1]. The indirect mapping makes the space of the genotype huge with respect to the space of the phenotype. Thus, many different genotypes may generate the phenotype or phenotypes with maximum fitness.

2 Requirements for an Ontogenic (Growth) Algorithm

We wish to define the requirements for a growth algorithm implementing the indirect mapping between a genome of no fixed size and a target application (assumed to be a

hardware circuit) having a configuration bitstream (binary vector) $\mathbf{b}$ of fixed length B bits.

In a "first level of indirection," the bitstream $\mathbf{b}$ is replaced by a symbol stream $\mathbf{s}$ containing symbols q that comprise a larger set (total of $Q + 1$ symbols):

$$\mathbf{b} = b_1, b_2, \ldots, b_i, \ldots, b_B \tag{1}$$

$$b_i \in [0, 1] \tag{2}$$

$$\mathbf{s} = s_1, s_2, \ldots, s_i, \ldots, s_S \tag{3}$$

$$s_i \in [q_1, \ldots, q_Q, q_X] \tag{4}$$

The length of the symbol stream, S, may differ from the length of the bitstream, B, as it may be desirable to map a symbol to groups of bits. Also, a symbol can map differently depending on its position in the bitstream. This tactic allows the hardware bitstream to be grouped in a meaningful way. Most hardware bitstreams are grouped into sets of bits corresponding to the configuration of each core module in the array, interspersed with other sets corresponding to the configuration of the interconnect, and so forth. For example, the symbol '**a**' might map to the bits defining a 4-input NOR gate when its position in the bitstream corresponds to a LUT, but it might map to a completely different 20-bit value when its position in the bitstream corresponds to an interconnect crossbar switch.

The general concept is that the target symbol stream starts in a known initial state, and the growth algorithm modifies it iteratively until convergence (defined below). The genome contains the rules for the modification, in the form of a table of (*key:value*) pairs representing, respectively, (*context:result*). A *context* is defined as any recognizable pattern of one or more symbols. The context should be defined in terms of the *relative* position of symbols to some (variable) reference point, but not an *absolute* position. The *result* is defined as a pattern of one or more symbols, relative to the same reference point.

The table of rules should cover the entire space of possible contexts. That is, every position in the symbol stream should be covered by at least one rule. It is important that the iterative algorithm converges because the application of all rules to the current bitstream results in the same bitstream, not because no rules were applicable!

In the general case, each iteration operates as follows:

1. Loop through each position in the current symbol stream. For each position:
   (a) Loop through each rule in the table. For each rule:
      i. Determine the context of the stream position as defined by the rule.
      ii. Measure the "distance" between the current context and the context defined in the rule.
   (b) Determine the rule that is the best match for the position (minimum distance).
   (c) Apply the result of the rule to a temporary bitstream.
2. Arbitrate where more than one result applies to a single stream position to obtain a single, unique symbol stream result.
3. If the new result is the same as the current symbol stream, then the growth algorithm has converged.
4. Update the current symbol stream with the new result.

Note that this "general algorithm" leaves a number of specifics undefined or loosely defined. The distance between contexts, for instance, may be any reasonable distance metric. However, if the size of a context differs from rule to rule, then the distance metric will require normalizing between rules to unambiguously determine the minimum distance over all rules. Arbitration (also undefined) may be a fixed rule, such as selecting as winner the result that is closest to the current result. Or, arbitration may be encoded into the rule table and evolved along with the rest of the system.

The general-purpose definition of context removes the need to define whether the bitstream should be considered 1-dimensional or 2-dimensional (to match the physical layout of the hardware). However, when working with hardware in a 2-dimensional array, the algorithm should act on the bitstream in an arrangement that preserves the order and direction of rows and columns. This will necessitate another re-mapping of the bitstream to obtain the correct bit sequence for the hardware.

Item 3 purposefully does not specify a "break" in the iterative loop, but only notes the condition of convergence. If the growth algorithm itself is embedded in hardware, the algorithm preferably runs forever, with a number of interesting conseqences (discussed below).

3 A Simplified Implementation of the General Algorithm

We define a simple growth algorithm meeting the requirements defined in the section above. The growth algorithm is essentially a state machine and the *genome* is the state transition table. As a simplification of the general algorithm, the *context* is of a fixed length for every position in the symbol stream. The context vector $\mathbf{v}$ is defined as a windowed portion of the symbol stream centered around position i and including a fixed *kernel* of K values to the right and left. The window slides from left to right, resulting in S contexts, one for each position in the symbol stream.

$$\mathbf{v} = v_1, v_2, \ldots, v_i, \ldots, v_S \tag{5}$$

$$v_i = s_{(i-K)}, \ldots, s_{(i+K)} \tag{6}$$

Endpoints of the symbol stream are handled by defining a symbol q_X (outside of the set $[q_0, \ldots, q_Q]$) representing the symbol stream boundary, and letting

$$s_i = q_X, \qquad i < 1, \quad i > S \tag{7}$$

The state machine table, or genome, $\mathbf{G}$ is a chromosome having M genes, or rules. In this work the genome is equated with the chromosome, although it is straightforward to expand this to a multiple chromosome model (space limitations prevent the multiple chromosome model from being presented here). Each rule is a (*context:result*) pair. The rule's context vector $\mathbf{r}_i$ is the same length as the symbol stream context vector $\mathbf{v}$, making direct comparison between the two simple. The result is an output value o_i representing a state to which the system transitions on the next iteration. The symbol set of o_i does not include q_X. Again, the general case is simplified by specifying only one symbol for the result, which eliminates the need for arbitration.

$$\mathbf{G} = (\mathbf{r}_1 : o_1), (\mathbf{r}_2 : o_2), \ldots, (\mathbf{r}_i : o_i), \ldots, (\mathbf{r}_M : o_M) \tag{8}$$

$\mathbf{s} = \mathbf{s}_0$
repeat {
  for $(1 \leq i \leq B)$ {
    $j^* = \underset{1 \leq j \leq M}{\text{minidx}} (\textbf{dist}(v_i, g_j))$
    $s'_i = o_{j^*}$
  }
  if$(\mathbf{s}' = \mathbf{s})$ **break**
  $\mathbf{s} = \mathbf{s}'$
  $b_i = \mathrm{m}(s_i) \qquad 1 \leq i \leq B$
}

Table 1. The iterative growth algoritm.

$$\mathbf{r}_i = r_{(i,-K)}, \ldots, r_{(i,+K)} \tag{9}$$

$$o_i \in [q_1, \ldots, q_Q] \tag{10}$$

The system has a mapping function $\mathrm{m}()$, called the system *chemistry*, which maps the space of the symbol set $\mathbf{s}$ onto the space of the bitstream $\mathbf{b}$:

$$\mathrm{m}(q_i) \in [0, 1] \qquad 1 \leq i \leq Q \tag{11}$$

In this simplified implementation of the general algorithm, each symbol maps to a single bit, and maps to the same bit regardless of the position of the symbol in the stream.

Every state machine needs an initial state. The initial state vector of this system is $\mathbf{s}_0$ and is randomly selected at the outset, but remains the same for every individual of every generation.

The growth algorithm is shown in Table 1. For each context vector v_i, find the gene (rule) having the minimum distance to the context vector. Set the next-state symbol s'_i to be the output symbol specified for this rule. After the entire next state vector $\mathbf{s}'$ has been determined, replace the current state vector $\mathbf{s}$ with the computed next state vector $\mathbf{s}'$, and repeat.

The distance function **dist**() can be any reasonable distance metric. One choice is a Hamming distance metric. The metric used for this proof-of-concept research was a simple linear distance between symbols, *e.g.*, $\textbf{dist}(q_1, q_3) = |q_1 - q_3| = |1 - 3| = 2$, with the boundary symbol treated separately as $\textbf{dist}(q_X, q_X) = 0$, and distance to all other symbols being a constant, large distance (a system parameter, but normally set to $2Q$).

Breaking out of the iterative loop upon condition of convergence is required when evolving the system. Because there is no guarantee that array $\mathbf{s}$ will converge to a static value, it is useful to force a break after some predetermined number of iterations. Individuals for which the growth algorithm does not converge are assigned a low fitness value.

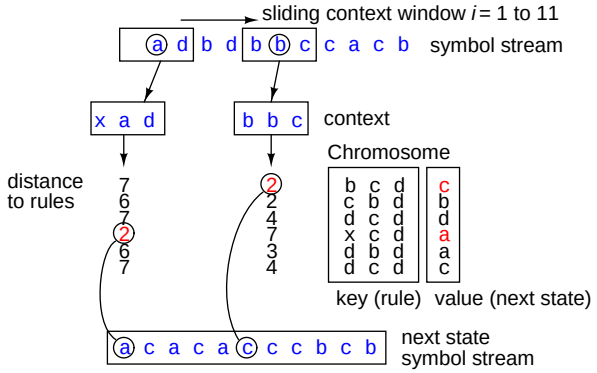

Fig. 1. Diagram of operations performed by the ontogenic algorithm. The values in the figure correspond to the example in the text.

4 Example

To visualize the operation of the ontogenic algorithm, consider an example using parameters sequence length $B = S = 11$ and symbol space $Q = 4$ with assignments q_1 = 'a', q_2 = 'b', q_3 = 'c', q_4 = 'd', and q_X = 'x' for the boundary symbol. The randomly-selected initial state $\mathbf{s}_0$ is the symbol string "adbdbbccacb". The mapping from symbols to bitstream bits is $\mathrm{m}(a) = 0$, $\mathrm{m}(b) = 0$, $\mathrm{m}(c) = 1$, and $\mathrm{m}(d) = 1$.

Figure 1 shows a representation of the (single) chromosome, containing six genes, or rules, with (*key:value*) pairs. It also shows the computations for positions 1 and 6 in the symbol stream on the first iteration, and the resulting symbol stream. If followed through, the growth sequence of the symbol stream is **adbdbbccacb**, **acacacccbcb**, **acbcbcccbcb**. The symbol streams of iterations 3 and 4 are identical, indicating that further iterations will not change the result. The symbol stream has converged, and the final bitstream value can be mapped. Applying the mapping given above, the final bitstream is **01010111010**.

Only rules 1, 2, 4, and 5 were used in the growth algorithm. Rules 3 and 6, then, are "junk DNA" as far as this individual is concerned. That does not, however, indicate that they are not used elsewhere in the population, or that they will not be used to advantage by this individual's offspring.

5 Biologically Inspired?

Apart from the fact that the algorithm defines an indirect mapping from genotype to phenotype, several key aspects of the algorithm are "biologically plausible." Each position in the symbol stream (*i.e.*, each cell in the organism) sees the same genetic code. How that genetic code acts in each case is determined by context; in each position, certain genes are relevant ("active") and some are not ("inactive"). This is both a parallel and a localized operation: It is independent of the ordering of genes in the table, and it is independent of the absolute position of any symbol in the stream. But the fitness of

an organism is highly dependent on the absolute position of all symbols in the stream, and the evolution of the system is highly dependent on the ordering of genes!

In biology, the same genetic code is presented to each cell by giving each cell a copy of the genetic code. This is not feasible in electronic hardware. However, the growth algorithm in electronic hardware is many orders of magnitude faster than cell growth in a biological system, so it is possible to remap the massively parallel operation of DNA into a sequential operation (the loop through each symbol in the stream), trading density for speed.

6 The Evolutionary Algorithm

The evolutionary algorithm built around the ontogenic algorithm is fairly pedestrian, and operates as follows. Assuming convergence of the above iterative growth algorithm, the final, mapped configuration bitstream $\mathbf{b}$ is loaded into hardware, tested for the desired behavior, and assigned a fitness. After all individuals in the population are tested and assigned a fitness, pairs are chosen as parents of the next generation. The method used for parent selection is not important; for this proof-of-concept research, a tournament-style selection was used, and no individuals were held over from one generation to the next. The method prohibits the pairing of any two individuals with the same chromosome, as that would be equivalent to cloning, or allowing an individual from one generation to appear in the next one.

Each genome consists of a set of M rules and declares a split point position p between 1 and M inclusive. Both M and p may differ from individual to individual in the population (this can only be true in an indirectly-mapped algorithm). Each new individual Z is created from parents X and Y by one-point crossover combination, copying all rules from 1 to $p(X)$ followed by all rules $p(Y)$ to $M(Y)$, or vice versa.

$$G(Z) = \begin{cases} (\mathbf{r}_i : o_i)\,(X) & 1 \le i \le (p(X) - 1) \\ (\mathbf{r}_i : o_i)\,(Y) & p(Y) \le i \le M(Y) \end{cases} \tag{12}$$

$$p(Z) = p(X) \tag{13}$$

$$M(Z) = M(Y) - p(Y) + p(X) \tag{14}$$

Note that a "null genome," that is, a chromosome without any genes ($M = 0$) is not disallowed by the algorithm; as it results always in the initial state $\mathbf{s}_0$ being the final state, normally it would have a low fitness and be eliminated quickly from the population.

7 Prior Research

Research in indirect maps between genotype and phenotype has been ongoing in several groups for a few years [2–4]. Prior research by the author [5] was designed to investigate growth of circuits in an unconstrained medium, based on simple principles of self-organization. That algorithm (similar to that presented in [2]), differs from the algorithm presented in Section 3 by assuming that the modules in the underlying hardware array have an "off" state, and requiring a fixed rule (not part of the genome) that a context of

all “off” states must result in an “off” state. The initial state of the system is to have all modules in the “off” state except for one module in the center of the array, defined as the “seed” module. Upon application of the iterative algorithm, a circuit, defined as a group of configured modules, grows from the “seed”, and subsumes as many modules as it requires. While intriguing from a theoretical standpoint, this method has several major drawbacks:

- Hardware resources (array size) are usually quite limited, but the algorithm assumes unlimited resources.
- Hardware input and output is usually fixed or has limited configurability with respect to the core module array. The original algorithm has no way to reliably map inputs and outputs to the final, grown state of the system.
- There is no known growth algorithm that “naturally” limits outward expansion during growth. Constructing a ruleset that does so is highly contrived and impossible to evolve reliably.

The idea of “naturally limited” growth is based directly on biological principles. However, growth in biological systems is usually limited by a process of first forming a membrane, then growing the system inward. Thus, an algorithm which tends to produce growth with unlimited outward expansion is not biologically realistic. Instead, it makes more sense to treat the boundary of the hardware as a “membrane,” and grow the circuit under the assumption that it will naturally fill the available space. Inputs and outputs in a hardware system are connected to sensors and actuators. It is just as unrealistic to assume these resources are unlimited, and they will tend to have further restrictions of placement based on mechanical design. Because the circuit array boundary is usually where input and output connections are located, growing the circuit into the fixed space allows system inputs and outputs to be predefined in specific locations, which is not possible in the system with unconstrained growth.

Another questionable aspect of the original research was the use of the configuration bitstream $\mathbf{b}$ of the module itself as the symbol stream $\mathbf{s}$. Equating them has the advantage of a simpler hardware implementation when incorporating the growth algorithm directly into the hardware. However, it has not been established whether or not this method overconstrains the system, preventing evolution toward the desired fitness goal. Until this issue is resolved, the algorithm has been modified such that the symbolic value of a module used during the growth algorithm is mapped to the configuration bitstream value via a simple table. The table can be considered, in biological terms, to be the “chemistry” of the system, and is fixed.

Apart from the issues of unconstrained growth and mapping of inputs and outputs, the main unresolved issue from the original research was the question of whether the ontogenic algorithm can be incorporated in an evolvable system. Prior work showed the ability to grow circuits demonstrating interesting (usually chaotic) computational behavior, but did not attempt to measure fitness or evolve a specific function or behavior.

8 Proof-of-Concept Simulation and Observations

To prove evolvability within the described biologically-inspired architecture, both the problem space and architecture were simplified. First, the bitstream was considered

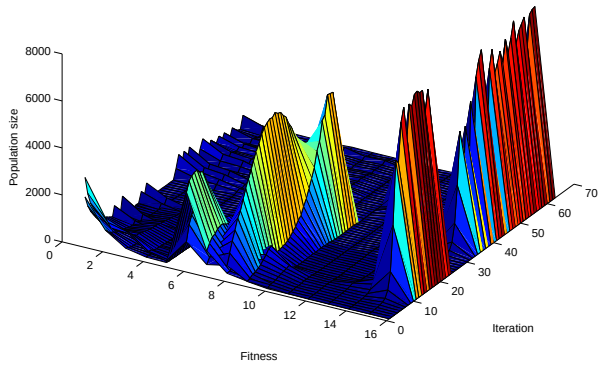

Fig. 2. Evolution of a randomly-generated ontogenic system toward a higher fitness value. Around iteration 25, a solution of fitness 9 briefly gains dominance due to the robustness of its growth pattern.

only as a one-dimensional vector. The "context" for a module is the symbolic value of the module concatenated with the symbolic values of one or more modules to its right and left. To further simplify, the configuration of each module was made a single bit.

An artificial fitness function was constructed as a function of the bitstream. A fitness was assigned to each bitstream value, in a random power law distribution, with approximately half the bitstreams having zero fitness, a quarter having fitness value 1, and so forth, but ensuring that at least one bitstream had the maximum fitness value. This artificially constructed bitstream is designed to mimic a typical digital circuit function, where fitness is defined as the number of input-output values matching those of the desired function. As is typical for an electronic circuit, fitness is not a direct function of any definable variable space. In fact, this random fitness function presents a "needle in a haystack" problem for the evolutionary algorithm. Solutions cannot be sought in any deterministic fashion other than a prohibitively costly exhaustive search of the problem space. It cannot be expected that this, or any other evolutionary algorithm, can do any better than another on such a problem. However, it suffices to show that the system is capable of evolving from an initial random, low fitness to a high fitness.

A simulation of the system was constructed to show the statistics of interest, which included the distribution of fitness throughout the population (Figure 2). The system was executed repeatedly while varying these key parameters: The "chemistry" mapping, the initial state, the number of symbols, and the rate of mutation.

What distinguishes the indirect mapping approach from the direct mapping approach is the tendency to fill the population not with the highest-fitness solutions, but with the solutions which are most stable as the result of self-organization. This can be much stronger than the selection of the most fit solution; solutions with higher fitness can even disappear from the population. In biology, this is known as "regression toward the mean," indicating that the fitness of an offspring will tend to be closer to the average fitness of the population as a whole, regardless of the fitness of the parents. The modal solution, that is, the solution represented by the majority of the population, is quite stable. This appears to mimic the process of speciation, and may explain why species tend

to be stable over long periods of time, and support the idea of "punctuated equilibrium." To force evolution toward a specific goal requires some extra effort, such as scaling the fitness such that individuals with a fitness higher than the modal fitness will have a larger-than-proportional representation in the succeeding generation. As shown in Figure 2, the fitness scaling can successfully overcome the stability of the self-organization and nudge the population into an overall higher fitness.

A novel aspect of the indirect mapping approach is the effect of random mutation on the solution. Crossover combination suffices to evolve the system. However, the addition of a small amount of random mutation tends to speed up convergence and stabilize the solution. This is because changes to the indirect map do not always produce changes to the solution. Some mutations will affect the offspring of some pairs of parents but not others, which leads to the rapid selection of the most robust genomes. This effect is impossible to achieve with direct mapping, since any change to the chromosome will alter the solution of, on average, half of the offspring generated from that chromosome. The allowance of multiple chromosomes magnifies this effect.

Lack of convergence of the algorithms has been observed to be a problem only in a very small minority of cases, due to pathological combinations of chemistry mapping, starting vector, and the randomly-defined solution. Generally, the number of genes converges within 50 generations.

The self-organizing process tends to fill the population with chromosomes of a constant size, but this is not necessarily the case. Indeed, it is quite possible to end up with two specific chromosomes in the population having one side distinctly different; call them 'X' and 'Y' (for obvious reasons). This happens regularly as a result of preventing offspring of two individuals with exactly the same genome (intended to promote genetic diversity by preventing a single genome from dominating the population). In other instances, the population is made "trivially diverse" by the presence of *junk DNA*. In this algorithm, junk DNA can be defined as any rule in the associative lookup table that is not exercised during the growth process.

9 Discussion and Future Research

The indirect mapping allows the genome to act as a coded, backup definition of the individual. It is anticipated that for very large systems, the genome is much smaller than the bitstream, resulting in an efficient coding. While the mapping is not reversible, it is possible to use the genome to repair damage to the individual.

If the indirect-map growth algoritm is viewed as a nonlinear dynamic system, then naturally limited growth can be seen as an "attractor state" of the system. The final system state can be fed back to the growth algorithm and produces the same state.

Damage to the individual can then be viewed as pushing the dynamic system away from this attractor. Small, temporary damage like single-event upsets (SEUs) do not push the system out of the capture zone of an attractor. When the growth algorithm operates continuously, the system returns to the attraction state. Most single-bit changes and even many multiple-bit changes of the configuration state of the individual will be corrected in one iteration.

With a properly evolved chromosome, the system can have multiple attractor states, each leading to a (possibly different) fit solution. Some massive faults or permanent

damage can be repaired. Massive, permanent damage will push the system out of the capture zone and into another attractor. It is not clear how well this can be made to work in practice, but it has the potential to allow evolved circuits or systems to continue to operate uninterrupted while sustaining damage.

An interesting line of future investigation is to look at system implementations in which the ontogenic algorithm runs continuously, and the target circuit *may change the value of the symbol stream that created it*. Due to the "self-healing" mechanism mentioned above, many such changes will have no affect on the circuit. However, specific changes may permanently change the state of the system, a form of "memory" in the system hardware. Such changes are volatile; they cannot be recovered when the system is reset and regrown.

10 Conclusions

This research demonstrates that it is possible to "grow" a circuit from a non-reversible, coded representation, that the growth is stable, and that a population of individuals thus coded can, through directed evolution, achieve a specific target function. Unlike previous work, the algorithm supports the growth of circuits onto hardware of a fixed size, with system inputs and outputs able to be specified prior to executing the growth algorithm. These aspects are critical for the ability to evolve any system on real hardware, such as an FPGA, where the size of the system is fixed, and positioning of inputs and outputs may be fixed or less flexible than the positioning of general-purpose logic modules.

The algorithm described here is a conveniently simple and straightforward way of creating an indirect mapping between genotype and phenotype under the primary constraint that the phenotype must be of fixed length but that the genotype need not be. It is certainly not the only possible indirect-mapping algorithm to satisfy those constraints, and it likely is not the best. Hopefully this work will interest others in the indirect mapping approach, and encourage a search for alternative and better algorithms.

References

1. Kauffman, S. A.: Investigations. Oxford University Press New York (2000)
2. Miller, J. F.: Evolving developmental programs for adaptation, morphogenesis, and self-repair. In: Lecture Notes in Artificial Intelligence, W. Banzhaf, T. Christaller, P. Dittrich, J. T. Kim, J. Ziegler (eds.) 7th European Conference on Artificial Life, Vol. 2801 (2003) 256–265
3. Haddow, P. C., and Tufte, G.: Bridging the Genotype-Phenotype Mapping for Digital FPGAs. In: Proceedings of the Third NASA/DoD Workshop on Evolvable Hardware. IEEE Computer Society Press (2001) 109–115
4. Gordon, T. G. W., and Bentley, P. J.: Towards Development in Evolvable Hardware. In: Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware. IEEE Computer Society Press (2002) 241–250
5. Edwards, R. T.: Circuit Morphologies and Ontogenies. In: Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware. IEEE Computer Society Press (2002) 251–260