Production system rules as protein complexes from genetic regulatory networks: an initial study

Larry Bull

Abstract This short paper introduces a new way by which to design production system rules. An indirect encoding scheme is presented which views such rules as protein complexes produced by the temporal behaviour of an artificial genetic regulatory network. This initial study begins by using a simple Boolean regulatory network to produce traditional ternary-encoded rules before moving to a fuzzy variant to produce real-valued rules. Competitive performance is shown with related genetic regulatory networks and rule-based systems on benchmark problems.

Keywords Fuzzy logic · Imitation · Indirect encoding · Random Boolean Networks · Rule-based systems

1 Introduction

The genetic regulatory networks (GRN) within cells synthesize proteins, some of which have regulatory functions, some have intra-cellular function, and some pass through the cell membrane. A growing body of work incorporates increasing levels of detail from the natural phenomena for computational intelligence but very few have considered proteins explicitly, and only two are known to do so whilst also exploiting an underlying GRN [3, 18]. In this paper, a well-known Boolean GRN model, Random Boolean Networks (RBN) [15], is extended to consider the role of non-regulatory proteins in a simple way, in particular their formation of complexes. Such protein aggregations are viewed as production system rules and hence a new, indirect encoding for Learning Classifier Systems (LCS) [14] is presented. Whilst a number of indirect encodings have been presented for artificial neural networks (e.g., see [11, 32] for overviews) no prior work for rules is known. The anticipated advantage of indirect encodings is one of scaling, i.e., when a large number of rules are required to solve a given task. Initial results indicate that increases in performance are possible through the extra layer of abstraction in comparison to the traditional RBN model but not to the related Pittsburgh-style LCS [26]. The Boolean logic GRN model is then altered to one using simple fuzzy logic operations [19] to enable the design of rules for continuous-valued problems. Again using versions of well-known benchmark problems, it is shown that improved performance is obtained in comparison to equivalent Pittsburgh-style LCS even on the relatively small tasks tried.

It has recently been shown [5] that a population-based algorithm which uses imitation as inspiration for its search mechanisms, as opposed to genetic evolution, is highly effective in the design of (dynamical) networks and so is used here.

2 Background

2.1 Random Boolean Networks: a simple GRN model

Within the traditional form of Random Boolean Networks there is a network of $R$ nodes, each with $B$ directed connections randomly assigned from other nodes in the network. All nodes update synchronously based upon the current state of those $B$ nodes. Hence those $B$ nodes are seen to have a regulatory effect upon the given node, specified by the given Boolean function randomly attributed to it. Nodes can also be self-connected. Since they
have a finite number of possible states and they are deterministic, such networks eventually fall into an attractor. It is well-established that the value of $B$ affects the emergent behaviour of RBN wherein attractors typically contain an increasing number of states with increasing $B$. Three phases of behaviour were originally suggested through observation: ordered when $B = 1$, with attractors consisting of one or a few states, i.e., point or small cyclic attractors; chaotic when $B > 3$, with a very large number of states per attractor; and, a critical regime around $1 < B < 4$, where similar states lie on trajectories that tend to neither diverge nor converge and so small cyclic attractors are typically seen (see [16] for discussions of this critical regime, e.g., with respect to perturbations). Subsequent formal analysis using an annealed approximation of behaviour identified $B = 2$ as the critical value of connectivity for behaviour change [9]. Figure 1 shows examples of typical behaviour.

There is a small amount of prior work exploring the use of computational intelligence to design RBN. Van den Broeck and Kawai [30] used a simulated annealing-type approach to design feedforward RBN for the four-bit parity problem. Kauffman [1993, p.211] evolved RBN to match a given attractor (see also Lemke et al. [20]). The same approach has been used to explore attractor stability [12] and to model real regulatory network data, e.g., see [29]. Sipper and Ruppin [25] evolved RBN for the well-known density task. Most closely related is work on the use of RBN to represent the rules within modern forms of Michigan-style LCS (e.g., [4, 22]). Where it has been allowed to vary, $B$ has been shown to evolve to $\sim 2$ in these studies (e.g., [4, 16, 23]). In contrast to all this work, the approach presented here does not use the RBN to provide outputs from given inputs directly.

2.2 Protein-inspired computation

The idea to consider cytoplasm proteins as a significant component in cellular information processing is not new [2, 21] but little work in the area of computational intelligence appears to exist. This is, of course, not the same as the work on protein structure prediction, e.g., using computational intelligence, or the more specific modelling work of many aspects of cells typically undertaken in Systems Biology. Fisher et al. [10] viewed proteins in signalling pathways as agents capable of rudimentary pattern recognition, memory, signal integration, etc. More recently, Qadir et al. [24] have used a protein metaphor to realize relatively fault-tolerant associative memory in evolvable hardware. Most similar to the work presented here, and the only other known abstract models of genome and proteome interaction which may potentially be used as a general representation, is that by Bentley [3] and Knibbe et al. [18]. Bentley [3] presents a model whereby products from a GRN interact by merging to form protein complexes. Knibbe et al. [18] used a fuzzy representation wherein proteins of overlapping membership functions interact: proteins are produced by an underlying Gray encoded GRN and contribute fractionally to phenotypic traits based upon the degree of overlap. This paper uses the simple RBN model of GRN to define proteins and uses them as rule components, as explained below.

2.3 Imitation programming: culture-inspired search

The basic principle of imitation computation is that individuals alter themselves based upon another individual(s), typically with some error in the process. Individuals are not replaced with the descendants of other individuals as

![Fig. 1](#) Typical behaviour of a traditional RBN with $R = 1500$ nodes: on the left, showing example temporal dynamics; and on the right, the average behaviour (100 runs) after 100 update cycles. Nodes were initialized at random. Error bars show max and min behaviour

[Springer]
in evolutionary search; individuals persist through time, altering their solutions via imitation. Thus imitation may be cast as a directed stochastic search process, thereby combining aspects of both recombination and mutation used in evolutionary computation. Imitation Programming (IP) [5] is such a population-based stochastic search process shown to be competitive with related evolutionary search for the design of highly recurrent, dynamical networks:

As noted above, in all the aforementioned previous work using RBN for computation and most artificial GRN research (e.g., [6]), it is the slower response mechanism that is considered. That is, much like a form of neural network, problem inputs are encoded and applied to the GRN before it is updated to determine a response. In this paper, the formation of protein complexes and their use to provide responses to problem inputs has been considered through a simple extension, essentially resulting in an indirect encoding scheme for Pittsburgh-style LCS.

Within RBN, the expression of genes, and hence implicitly the formation of the protein they each encode, is considered a binary event. Thus if a gene node within a network has a logical value ‘1’ it is considered ‘on’ and to have therefore produced a protein, and vice versa. To add a layer of protein complexes in a relatively simple but computationally useful form, in keeping with the RBN model, ternary-encoded production system rules are initially used. Here, for a set of predefined nodes in an RBN, the presence or absence of the protein expressed by a given gene is used to specify one part of a rule/complex. The series of states of the set of RBN nodes can therefore be used to specify multiple rules—for $T$ update cycles, $T$ rules can be determined.

Each gene node of a given network is set to its defined start state. Nodes then update synchronously for $T$ cycles. On each cycle, the value of a set of $L$ identified nodes, e.g., the first $L$ nodes in the RBN as here, is joined into a binary string which is subsequently turned into a traditional ternary LCS rule. Thus if the given problem has two binary-encoded inputs $I$ and one binary output $O$, then $L = 5$. The value of each of the first $2 \times L$ nodes are used to specify the rule condition. Each pair of bits is interpreted as either ‘0’, ‘1’ or the generalization symbol ‘#’: ‘11’ = ‘1’, ‘00’ = ‘0’, otherwise ‘#’. Actions are interpreted directly. Figure 2 shows an example. Once the RBN has been iterated for $T$ cycles and the $T$ rules determined, the fitness of the RBN is ascertained by evaluating the rules on the given problem.

In this paper, similar to Differential Evolution [28], each individual in the population $p$ creates one variant of itself and it is adopted if better per iteration. Other schemes are, of course, possible, e.g., Particle Swarm Optimization (PSO) [17] always accepts new solutions but then also “imitates” from the given individual’s best ever solution per learning cycle. This aspect of the approach, like many others, is open to future investigation. The individual to imitate is chosen using a roulette-wheel scheme based on proportional solution utility, i.e., the traditional reproduction selection scheme used in Genetic Algorithms (GA) [13]. Again, other schemes, such as the spatial networks of PSO, could be used. Further details of how IP is used to design RBN are given below.

3 Rules from GRN: protein complexes

The RBN model is an abstraction of gene regulation and does not explicitly consider the role of proteins which are maintained within the cell body. It is those proteins which determine the primary response of the cell to its environment. Typically, such proteins also form multi-protein complexes of two or more proteins which can, for example, increase their catalytic capabilities. Of course, the environment has causal effects on gene expression through a series of protein–protein interactions but the timescale is significantly longer, typically around 5 min. That is, cells often respond to a given stimulus through protein complexes which were created by their GRN before the event.
To avoid the issue of multiple rules matching a given input, the rules are ordered based upon the RBN cycle that created them. For example, if the rule created on RBN cycle 2 and that on cycle 8 both match a given input the action of the rule from cycle 2 will be used as the output. Of course, other schemes are possible but not considered. A failure to match an input means no output is given here.

4 Experimentation I: Boolean logic

4.1 Imitation programming RBN

For RBN design, IP here utilizes a variable-length representation of pairs of integers defining node inputs, each with an accompanying single bit defining the node’s start state, and an integer to define the node function, and there is an integer per RBN to define $T$. Five imitation operators are used: copy a node connection, copy a node start state, copy a node function, copy a network cycle count $T$, and change size through copying. In this paper, each operator can occur with or without error, with equal probability, such that an individual performs one of the ten during the imitation process as follows:

To copy a node connection, a randomly chosen node has one of its randomly chosen connections set to the same value as the corresponding node and its same connection in the individual it is imitating. When an error occurs, the connection is set to the next or previous node in the individual being imitated (equal probability, bounded by solution size). Imitation can also copy the start state for a randomly chosen node from the corresponding node, or do it with error (bit flip here). Size is altered by adding or deleting nodes and depends upon whether the two individuals are the same size. If the individual being imitated is larger than the copier, the connections and node start state of the first extra node are copied to the imitator, a randomly chosen node being connected to it. If the individual being imitated is smaller than the copied, the last added node is cut from the imitator and any/all connections to it reassigned at random. If the two individuals are the same size, either event can occur (with equal probability). Node addition adds a randomly chosen node from the individual being imitated onto the end of the copier and it is randomly connected into the network. The operation can also occur with errors such that copied connections are either incremented or decremented. Deletion is as before. For a problem with a given number of inputs $I$ and outputs $O$, the node deletion operator has no effect if the parent consists of $O + (2^*I) = L$ nodes. Similarly, there is a maximum size (100) defined beyond which the growth operator has no effect. The number of cycles $T$ can also be imitated, its value being incremented or decremented by 1 in the allowed range. The RBN contain $L$ nodes initially here. Adoption/replacement is based on quality, or $R$ if that is equal, or $T$ if $R$ is also equal, the smaller adopted in both cases, or finally the decision is random if all three values are equal.
4.2 Results

The well-known benchmark multiplexer task is used in this paper since they can be used to build many other logic circuits, including larger multiplexers. These scalable Boolean functions are defined for binary input strings of length \( l = x + 2^x \), under which the \( x \) bits index into the remaining \( 2^x \) bits, returning the value of the indexed bit. The inverse task, i.e., the demultiplexer, is also used where input strings are of length \( l = x + 1 \). Here there are \( 2^x \) output bits, all of which must be zero except for the bit indexed by the \( x \) input bits that must be set to the value of the other input bit. Hence there is significant scope for generalization in the multiplexer task but very little in the demultiplexer, the former having been used widely also motivating its use here.

Upon each evaluation, each node in an RBN has its state set to its specified start state. The RBN is then executed for \( T \) cycles, as encoded alongside the RBN topology. The value on the \( L \) output nodes is recorded on each cycle and the production system rule formed and placed in an ordered list as described above. Using the rules produced by the RBN, the correct response to an input results in a quality increment of 1, with all possible \( 2^{2^l} \) binary inputs being presented per evaluation. All results presented are the average of 20 runs, with a population/society of \( \mu = 50 \) and \( B = 2 \) (see Sect. 2.1). Nodes functions are from the allowed set \{AND, NAND, OR, NOR, XOR, XNOR\} and the number of rules allowed is \( 1 \leq T \leq 32 \), initialized uniformly.

Figure 3 (left) shows the performance of the approach on the 6-bit \((x = 2)\) multiplexer problem. Optimal performance (64) is obtained around 20,000 iterations, with 20 rules per individual. Figure 3 (right) shows performance of the same algorithm for an \( x = 2 \) demultiplexer, i.e., one with three inputs and four outputs. It can be seen that optimal performance (8) is reached around 8,000 iterations, with 7–8 rules. It can be noted that, as previously found [5], performance is significantly slower on both tasks (\( T \) test, \( p \leq 0.05 \)) if the imitation process is removed and only random alterations (mutations) are made to solutions (not shown).

As noted in Sect. 2.1, RBN are discrete dynamical systems with a finite number of possible states and they are deterministic, hence such networks eventually fall into a basin of attraction. As Fig. 1 (left) shows, for low \( B \) it typically takes around 10–15 update cycles for an attractor to be reached. This is potentially significant for the scheme described so far since it implies there is a finite set of different rules an RBN can produce, i.e., those from the states encountered into an attractor and then those produced (repeatedly) within it, regardless of the value of \( T \). For example, for \( B = 2 \), as used here, the typical length of an attractor is the square-root of the size of the RBN \((\sqrt{R})\) [15]—a potentially small number for the RBN evolved here.

It should be noted that it is not an issue for the two tasks used above since they can be solved optimally with fewer than 10–15 rules. To reduce this general limitation, the RBN can be supplied with a changing external input to (potentially) keep it out of attractors. Figure 4 shows results for the tasks above but with the first five nodes of an RBN having their first connection receive the corresponding bit of a binary-encoded “clock” input. That is, since \( 1 \leq T \leq 32 \), on each update cycle of the RBN, the corresponding binary pattern from the \( 2^5 \) possible inputs is applied. As can be seen, in both cases there is no significant difference \( (T \) test, \( p > 0.05 \)) in time taken to reach the optimum or in the size of the RBN, but there is a significant \( (T \) test, \( p \leq 0.05 \)) increase in the number of rules produced.

![Fig. 3 Performance on multiplexer (left) and demultiplexer (right)](image-url)
on the multiplexer (not the demux). This is perhaps a somewhat expected potential side-effect of keeping the RBN out of attractors.

As noted above, the standard use of RBN considers them as a form of recurrent neural network with inputs applied at some nodes, as is the clock input in the approach here, and outputs taken from other nodes, as the rule components are here. The performance of this approach has been explored using the same IP mechanisms as before (Sect. 4.1). Figure 5 shows how it is significantly slower ($T_{\text{test}}, p \leq 0.05$) to solve the multiplexer than the protein-based scheme (Fig. 4). The same was true for the demultiplexer (not shown). And this is the case whether initialization gives the networks the minimum required $O + I$ nodes for the approach or $O + 2I$ as above (the former is shown).

Figure 5 also shows the performance of using IP to design the standard, directly encoded rule-concatenation, ternary representation of Pittsburgh-style LCS. Here imitation can copy a rule component, or copy/delete a whole rule causing a change in size using the same general scheme described in Sect. 4.1. Each can occur with error and one of the four possible operations is applied with equal probability per iteration. Each individual was seeded with a number of rules uniform randomly in the same range as $T$ above. As can be see, the standard representation is approximately ten times faster at finding the optimum on the multiplexer than the protein approach (Fig. 4) using the same number of rules, and a similar result was found for the demultiplexer (not shown). However, this is perhaps not very surprising because the tasks do not need a large number of rules, and given the complexity and size of the RBN representation compared to the standard representation here: an RBN is defined by $\omega$ nodes, each encoded as 3 integers of various ranges and a bit, plus an extra integer.
for $T$; and, a directly encoded ruleset consists of $\psi$ rules, each encoded as $I$ ternary numbers and $O$ binary bits. Note also that $(2I + O) \leq \omega \leq 100$ and $1 \leq \psi \leq 100$.

The following section shows how the indirect encoding can be beneficial when even relatively small real-valued tasks are considered.

## 5 Experimentation II: fuzzy logic

### 5.1 Fuzzy logic networks

The continuous-valued dynamical systems known as Fuzzy Logic Networks (FLN) [19] are an extension of RBN where the Boolean functions are replaced with simple fuzzy logic functions. Kok and Wang explored 3-gene regulation networks using FLN and found that not only were FLN able to represent the varying degrees of gene expression but also that the dynamics of the networks were able to mimic a cell’s irreversible changes into an invariant state or progress through a periodic cycle. A number of different fuzzy logic sets have been introduced since the original Max/Min method was proposed. Table 1 shows the six functions used here, again with $B = 2$. Most closely related is work on the use of FLN to represent the rules within modern forms of Michigan-style LCS [23] also using the same function set. All other aspects remain the same as above, except start states are seeded uniform randomly [0.0, 1.0] and their error is from the range $[-0.1, 0.1]$ under imitation. The binary clock input is used.

Hence FLN enable the protein approach to produce real-valued rules from a mostly integer-based encoding. The GRN again requires $O + 2I$ nodes as a minimum, where the two real-valued numbers per problem input variable represent the upper and lower bound of an (ordered) interval, e.g., see [27].

### 5.2 Results

The “real multiplexer” problem [31] is used here which is an extension of the Boolean multiplexer: the binary input strings are replaced as real-valued vectors in the range [0, 1]. Each value in the vector is then interpreted as 0 if greater than a threshold value, $\theta$, else 1, where typically $\theta = 0.5$ as here, before being treated in the same way as the Boolean case for evaluation. Since there is no longer a finite set of possible inputs, a training set of 1000 randomly created vectors of length $l$ was created per experiment, together with another 1000 randomly created examples to act as the test set. The same approach was used to create a real-valued demultiplexer problem. All other parameters/details remained the same above. Figure 6 (left) shows the performance of the approach on the $x = 2$ real multiplexer problem. Over the allowed time, performance is around 95% on the training set and 94% on the unseen test set, with around 13 rules and 17 nodes per individual, which is competitive with other reported results [31]. Figure 6 (right) shows performance of the same algorithm for an $x = 2$ real demultiplexer. It can be seen that performance is

| ID | Function (CFMQVS and MV) | Logic |
|----|-------------------------|-------|
| 0  | Fuzzy AND (max/min)     | Max(x,y) |
| 1  | Fuzzy AND (CFMQVS and probabilistic) | $x \cdot y$ |
| 2  | Fuzzy OR (max/min)      | min(x,y) |
| 3  | Fuzzy OR (CFMQVS and MV) | min(1, $x + y$) |
| 4  | Fuzzy NOT               | $1 - x$ |
| 5  | Identity                | $x$ |

![Fig. 6](https://example.com/fig6.png) Performance on real multiplexer (left) and real demultiplexer (right)
around 97% on the training set and 87% on the test set, with typically 11 rules and nodes per individual.

Figure 7 shows the performance of the equivalent direct encoding interval scheme on the same two tasks, with mutation operating over the same range as the imitation error, and all other details the same as before. As can be seen, performance on the test set is not significantly different for the demultiplexer ($T$ test, $p \geq 0.05$) but significantly worse for the real multiplexer ($T$ test, $p \leq 0.05$). More rules were used in both cases ($T$ test, $p \leq 0.05$). Figure 8 shows the performance of the protein approach on the $x = 3$ real multiplexer and demultiplexer. Performance with the direct encoding was significantly worse in both cases ($T$ test, $p \leq 0.05$), with little or no learning emerging over random behaviour for either (not shown): the benefits of the indirect encoding become clear as the task difficulty increases, not least since the search space of the traditional, direct encoding increases more rapidly with real-valued variables.

6 Conclusions

This paper has introduced a method for learning production system rules using an indirect encoding, more specifically exploiting the dynamical behaviour of artificial genetic regulatory networks. Such general efforts exploiting temporally dynamic representations have previously been termed dynamical genetic programming [4]. It is suggested here that the rules may be seen as loosely analogous to the protein complexes which form within cells and process the internal and external stimuli it receives. It has been shown that the approach can be more effective than the standard way of using GRN for computation but less effective over
the traditional direct encoding in well-known binary problems. However, in continuous-valued problems the use of a restricted form of fuzzy logic within the GRN has been shown to be more effective than the direct encoding. Parameter sensitivity and other data sets are currently being explored.

One of the potential advantages of the traditional rule concatenation representation is the ability to incorporate problem knowledge into the search process and directly manipulating rules during learning to aid the search, e.g., through the creation and insertion of matching rules into solutions. Such procedures are more difficult with an indirect encoding, of course. However, hybrids which also maintain a small list of traditionally encoded rules can be envisaged, for example. Future work will explore this approach.

Future work should also consider comparison with other approaches to the evolution of rules (e.g., [1]) and data mining algorithms, particularly on tasks with large numbers of real-valued variables and/or which require large numbers of rules. It can be noted that the approach described here requires the GRN to grow in initial size as the number of variables increases since a pair of nodes is needed per variable. The construction of individual rules through more than one iteration of the regulatory network, e.g., one iteration = one variable, can be envisaged for rules needing large numbers of variables thereby somewhat negating the complexity increase.

Other future work will consider the incorporation of structural dynamism (after [8]), application to multi-step problems, and allowing the protein complexes to affect the behaviour of the GRN during use (after [7]), e.g., in non-Markov domains.

References

1. Bacardit J, Garrell J (2003) Evolving multiple discretizations with Adaptive intervals for a pittsburgh reul-based learning classifier system. In: Cantu-Paz E et al. (eds) Proceedings of the 2003 genetic and evolutionary computing conference. Springer, pp 1818–1831
2. Bray D (1995) Protein molecules as computational elements in living cells. Nature 376:307–312
3. Bentley P (2004) Adaptive fractal gene regulatory networks for robot control. In: Workshop proceedings of the genetic and evolutionary computation conference. Gecco (CD)
4. Bull L (2009) On dynamical generic programming: simple Boolean Networks in learning classifier systems. Int J Parallel Emergent Distrib Syst 24(5):421–442
5. Bull L (2012a) Using genetical and cultural search to design unorganised machines. Evolut Intell (in press)
6. Bull L (2012b) Evolving Boolean Networks on tunable fitness landscapes. IEEE Trans Evolut Compu (in press)
7. Bull L (2012c) A simple computational cell: coupling Boolean gene and protein networks. Artif Life (in press)
8. Bull L (2012d) On natural genetic engineering: structural dynamism in random Boolean Networks. Arxiv: http://arxiv.org/abs/1201.3545
9. Derrida B, Pomeau Y (1986) Random networks of automata: a simple annealed approximation. Europhys Lett 1:45–49
10. Fisher M, Paton R, Matuno K (1999) Intracellular signalling proteins as ‘smart’ agents in parallel distributed processes. BioSystems 50:159–171
11. Floreano D, Dürr P, Mattiussi C (2008) Neuroevolution: from architectures to learning. Evol Intel 1(1):47–62
12. Fretter C, Szejka A, Drossel B (2009) Perturbation propagation in random and evolved Boolean networks. New J Phys 11(3):033005
13. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor
14. Holland JH (1976) Adaptation. In: Rosen R, Snell FM (eds) Progress in theoretical biology 4. Plenum, New York, pp 263–293
15. Kauffman SA (1969) Metabolic stability and epigenesis in randomly constructed genetic nets. J Theor Biol 22:437–467
16. Kauffman SA (1993) The origins of order. Oxford University Press, Oxford
17. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of IEEE international conference on neural networks. IEEE Press, pp 1942–1948
18. Knibble C, Fayard J-M, Beslon G (2008) The topology of the protein network influences the dynamics of gene order: from systems biology to a systemic understanding of evolution. Artif Life 14:149–156
19. Kok T, Wang P (2006) A study of 3-gene regulation networks using NK-Boolean network model and fuzzy logic networking. In: Fuzzy applications in industrial engineering. Studies in fuzziness and soft computing, vol 201. Springer, Berlin, pp 119–151
20. Lemke N, Mombach J, Bodmann B (2001) A numerical investigation of adaptation in populations of random Boolean Networks. Phys A 301:589–600
21. Paton R (1993) Some computational models at the cellular level. BioSystems 29:63–75
22. Preen R, Bull L (2009) Discrete dynamical genetic programming in XCS. In: Proceedings of the 2009 genetic and evolutionary computation conference. ACM Press, pp 1299–1306
23. Preen R, Bull L (2011) Fuzzy dynamical genetic programming in XCSF. In: Proceedings of the 2011 genetic and evolutionary computation conference. ACM Press, pp 167–168
24. Qadir O, Liu J, Timmis J, Tempesti G, Tyrrell AM (2010) Principles of protein processing for a self-organising associative memory. In: Proceedings of the 12th IEEE congress on evolutionary computation. IEEE Press
25. Sipper M, Ruppin E (1997) Co-evolving architectures for cellular machines. Physica D 99:428–441
26. Smith SF (1980) A learning system based on genetic adaptive algorithms. PhD Thesis, University of Pittsburgh
27. Stone C, Bull L (2003) For Real! XCS with continuous-valued inputs. Evol Comput 11(3):299–336
28. Storn R, Price K (1997) Differential evolution: a simple and efficient heuristic for global optimization over continuous spaces. J Glob Optim 11:341–359
29. Tan P, Tay J (2006) Evolving Boolean Networks to find intervention points in dengue pathogenesis. In: Keijzer M et al. (eds) Proceedings of the 2006 genetic and evolutionary computation conference. ACM Press, pp 307–308
30. Van den Broeck C, Kawai R (1990) Learning in feedforward neural networks. Phys Rev A 42:6210–6218
31. Wilson S (2000) Get real! XCS with continuous-valued inputs. In: Lanzi P-L et al. (eds) Learning classifier systems: from foundations to applications. Springer, Berlin, pp 209–222
32. Yao X (1999) Evolving artificial neural networks. Proc IEEE 87(9):1423–1447