Classifier Conditions Using Gene Expression Programming

Stewart W. Wilson

Prediction Dynamics, Concord MA 01742 USA
Department of Industrial and Enterprise Systems Engineering
The University of Illinois at Urbana-Champaign IL 61801 USA
wilson@prediction-dynamics.com

Abstract. The classifier system XCSF was modified to use gene expression programming for the evolution and functioning of the classifier conditions. The aim was to fit environmental regularities better than is typically possible with conventional rectilinear conditions. An initial experiment approximating a nonlinear oblique environment showed excellent fit to the regularities.

1 Introduction

A learning classifier system (LCS) [14] is a learning system that seeks to gain reinforcement from its environment via an evolving population of condition-action rules called classifiers. Broadly, each classifier has a condition, an action, and a prediction of the environmental payoff the system will receive if the system takes that action in an environmental state that satisfies its condition. Through a Darwinian process, classifiers that are useful in gaining reinforcement are selected and propagated over those less useful, leading to increasing system performance.

A classifier's condition is important to this improvement in two senses. First, the condition should contribute to the classifiers's *accuracy*: the condition should be satisfied by, or *match*, only states such that the classifier's action indeed results in the predicted payoff. Second, the condition should be *general* in the sense that the classifier should match as many such states as possible, leading to compactness of the population and, for many applications, transparency of the system's knowledge. In effect, the conditions should match the environment's *regularities*—state subsets with similar action payoffs. This depends in part on the course of the evolutionary process. But it also depends on whether the condition syntax actually permits the regularities to be represented.

Classifier system environments were initially [10] defined over binary domains. The corresponding condition syntax consisted of strings from {1,0,#}, with # a "don't care" symbol matching either 1 or 0. This syntax is effective for conjunctive regularities—ANDs of variables and their negations—but cannot express, e.g., x_1 OR x_2. Later, for real-vector environments, conditions were introduced [20] consisting of conjunctions of interval predicates, where each predicate matches if the corresponding input variable is between a pair of values.

The same logical limitation also applies—only conjuncts of intervals can be represented. But going to real values exposes the deeper limitation that accurately matchable state subsets must be hyperrectangular, whereas many environmental regularities do not have that shape and so will elude representation by single classifiers.

Attempts to match regularities more adroitly include conditions based on hyperellipsoids [2] and on convex hulls [15]. Hyperellipsoids are higher-dimensional ellipse-like structures that will evolve to align with regularity boundaries. Convex hulls—depending on the number of points available to define them—can be made to fit any convex regularity. Research on both techniques has shown positive results, but hyperellipsoids are limited by being a particular, if orientable, shape, and the number of points needed by convex hulls is exponential with dimensionality.

Further general approaches to condition syntax include neural networks (NNs) [1] and compositions of basis functions—trees of functions and terminals—such as LISP S-expressions [13]. In both cases, matching is defined by the output exceeding a threshold (or equal to 1 (`true`) in the case of S-expressions of binary operators). NNs and S-expressions are in principle both able to represent arbitrary regularities but NNs may not do so in a way that makes the regularity clear, as is desirable in some applications. Moreover, unlike its weights, the NN's connectivity is in most cases fixed in advance, so that every classifier must accept inputs from all variables, whereas this might not be necessary for some regularities. In contrast to NNs, functional conditions such as S-expressions offer greater transparency—provided their complication can be controlled—and have the ability to ignore unneeded inputs or add ones that become relevant.

This paper seeks to advance understanding of functional conditions by exploring the use of gene expression programming [7, 8] to define LCS conditions. Gene expression programming (GEP) is partially similar to genetic programming (GP) [11] in that their phenotype structures are both trees of functions and terminals. However, in GEP the phenotype results from translation of an underlying genome, a linear chromosome, which is the object of selection and genetic operators; in GP the phenotype itself acts as the genome and there is no translation step. Previous classifier system work with functional conditions has employed GP [13]. For reasons that will be explained in the following, GEP may offer more powerful learning than GP in a classifier system setting, as well as greater transparency. However, the primary aim of the paper is to test GEP in LCS and assess how well it fits environmental regularities, while leaving direct comparisons with GP for future work.

The next section examines the limits of rectilinear conditions in the context of an example landscape that will be used later on. Section 3 presents basics of GEP as they apply to defining LCS conditions and introduces our test system, XCSF-GEP. Section 4 applies XCSF-GEP to the example landscape. The paper concludes with a discussion of the promise of GEP in LCS and the challenges that have been uncovered.

2 Limits of Traditional Conditions

Classifier systems using traditional hyperrectangular conditions have trouble when the regularities of interest have boundaries that are oblique to the coordinate axes. Because classifier fitness (in current LCSs like XCS [19] and its variants) is based on accuracy, the usual consequence is evolution of a patchwork of classifiers with large and small conditions that cover the regularity, including its oblique boundary, without too much error. Although at the same time there is a pressure toward generality, the system cannot successfully cover an oblique regularity with a single large condition because due to overlap onto adjacent regularities such a classifier will not be accurate. Covering with a patchwork of classifiers is, however, undesirable because the resulting population is enlarged and little insight into the regularity or even its existence is gained.

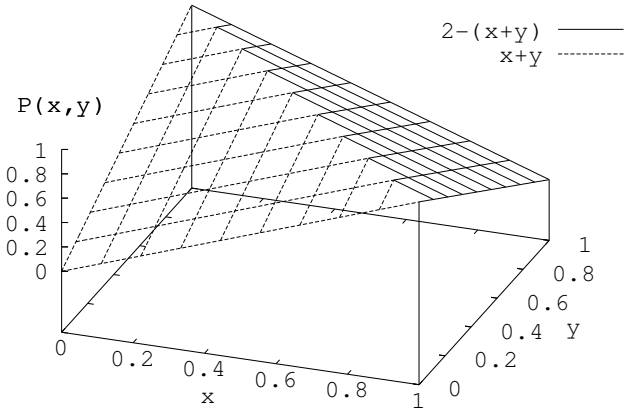

Fig. 1. "Tent" payoff landscape $P(x, y)$.

An example will make these ideas clear. Figure 1 shows a tent-like two-dimensional payoff landscape where the projection of each side of the tent onto the x-y plane is a triangle (the landscape is adapted from [25]). The two sides represent different regularities: each is a linear function of x and y but the slopes are different. The equation for the payoff function is

$$P(x,y) = \begin{cases} x+y & : \quad x+y \leq 1 \\ 2-(x+y) & : \quad x+y \geq 1 \end{cases} \tag{1}$$

The landscape of Figure 1 can be learned by XCSF [24] in its function approximation version [21, 22]. XCSF approximates non-linear functions by covering the landscape with classifiers that compute local linear approximations to

the function's value. Each such classifier will match in a certain subdomain and its prediction will be a linear function—via a weight vector—of the function's input. The classifier's condition evolves and its weight vector is adjusted by feedback until the prediction is within an error criterion of the function's value in that subdomain. At the same time, the condition will evolve to be as large as possible consistent with the error criterion. In this way XCSF forms a global piecewise-linear approximation to the function (for details of XCSF please see the references).

In the case of Figure 1, XCSF will evolve two sets of classifiers corresponding to the two sides of the tent. All the classifiers on a side will end up with nearly identical weight vectors. But their conditions will form the patchwork described earlier—a few will be larger, located in the interior of the triangle, the rest will be smaller, filling the spaces up to the triangle's diagonal. The problem is: rectangles don't fit triangles, so the system is incapable of covering each side with a single classifier. If the conditions could *be* triangles, the system would need only two classifiers, and they would clearly represent the underlying regularities.

3 Gene Expression Programming in XCSF

3.1 Some basics of GEP

As noted earlier, in GEP there is both a genome, termed *chromosome*, and a phenotype, termed *expression tree* (ET), with the expression tree derived from the chromosome by a process called *translation*. The ET's performance in the environment determines its fitness and that of the corresponding chromosome, but it is the chromosome which undergoes selection and the actions of genetic operators.

An example chromosome might be `-*ea+b/cdbbaddc`. The arithmetic symbols stand for the four arithmetic *functions*. The alphabetic symbols are called *terminals* and take on numeric values when the expression tree is evaluated. The first seven symbols of this chromosome form the *head*; the rest, all terminals, form the *tail*. Whatever the length of the chromosome, the head, consisting of functions and terminals, and tail, consisting only of terminals, must satisfy

$$t = h(n_{max} - 1) + 1, \tag{2}$$

where h is the length of the head, t the length of the tail, and n_{max} is the maximum function arity (in this case 2). The reason for the constraint implied by this equation will be explained shortly.

Many genetic operators can be employed in GEP. The simplest and, according to Ferreira [8], the most powerful, is *mutation*. It simply changes a symbol to one of the other symbols, with the proviso that, in the head, any symbol is possible, but in the tail, symbols can only be changed to other terminal symbols. Another operator is *inversion*: it picks start and termination symbols of a subsequence within the head, and reverses that subsequence. *Transposition* is

a further operator which comes in three forms [7]. The simplest, IS transposition, copies a subsequence of length n from anywhere in the chromosome and inserts it at a random point between two elements of the head; to maintain the same chromosome length, the last n elements of the head are deleted. Several *recombination* operators are used, including one- and two-point, which operate like traditional crossover on a pair of chromosomes.

The translation from chromosome to expression tree is straightforward. Proceeding from left to right in the chromosome, elements are written in a breadth-first manner to form the nodes of the tree. The basic rule is: (1) the first chromosome element forms a single node at the top level (*root*) of the tree; (2) further elements are written left-to-right on each lower level until all the arities on the level above are satisfied; (3) the process stops when a level consists entirely of terminals. Due to Equation 2, the tail is always long enough for translation to terminate before running out of symbols. Figure 2, left side, shows the ET for the example chromosome. Evaluation of the tree occurs from bottom up, in this case implementing the expression $a(b + c/d) - e$.

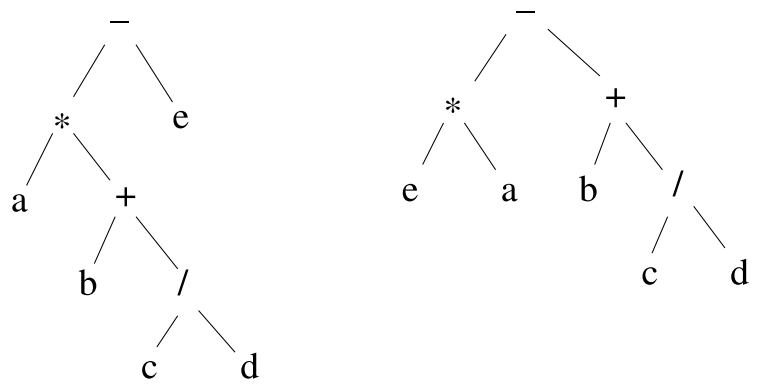

Fig. 2. Translation of chromosome `-*ea+b/cdbbaddc` into expression trees (ETs). Left, standard Karva translation. Right, prefix translation (Sec. 4.1).

The most important property of GEP translation is that every chromosome is *valid*; that is, as long as it obeys Equation 2, every possible chromosome will translate into a legal, i.e., syntactically correct, tree. The reason is that in the translation process all function arities are correctly satisfied. The validity of every chromosome has the significant consequence that the genetic operators cannot produce an illegal result. In fact, as long as it does not leave function symbols in the tail, *any* definable operator will be "safe". This is in some contrast to other functional techniques such as GP, where certain operators cannot be used without producing illegal offspring, or if used, the offspring must be either edited back to legality or discarded. Ferreira ([8], pp. 22-27, 33-34) regards this property of GEP as permitting a more thorough and therefore productive search of the problem space. However, the search issue calls for further investigation

since GEP *does* have the restriction that the tail must be free of function symbols and few direct performance comparisons have been made.

Gene expression programming has further important features, but since they were not used in the present work they will only be mentioned here. One is the ability to combine several *genes* into a single chromosome. These are chromosome segments that translate into separate expression trees. They are individually evaluated but their results are *linked* either by a predetermined operator such as addition or by a linkage structure that is itself evolved in another part of the chromosome. A second feature is that GEP has several methods for the concurrent evolution of real-valued constants that may be needed as coefficients and terms in expressions. Both features (plus others) of GEP are likely to contribute to the representational power of classifier conditions.

Finally, as in other functional approaches, GEP needs a method of dealing with undefined or otherwise undesirable results from arithmetic operators. In contrast to GP, GEP does not replace such operators with protected versions for which the result cannot occur. Instead, normal operators are retained. Then when, say, an operator like "/" receives a zero denominator argument, its tree's evaluation is aborted and the chromosome's fitness is set to zero. The philosophy is to remove (via lack of selection) such flawed chromosomes instead of further propagating their genetic material.

3.2 XCSF-GEP

The application of GEP in classifier conditions is not complicated. Basically, the condition is represented by a chromosome whose expression tree is evaluated by assigning the system's input variables to the tree's terminals, evaluating the tree, and comparing the result with a predetermined threshold. If the result exceeds the threshold, the condition matches and its classifier becomes part of the match set. From that point on, XCSF-GEP differs from XCSF only in that its genetic operators are GEP operators (as in Sec. 3.1). Covering—the case where no classifier matches the input—is handled by repeatedly generating a new classifier with a random condition (but obeying Equation 2) until one matches.

Adding GEP to XCSF would appear to have three main advantages. The first stems from use of functional (instead of rectilinear) conditions, with the consequent ability to represent a much larger class of regularities. The next two stem from GEP in particular: simplicity of genetic operations and conciseness of classifier conditions. Genetic operations are simple in GEP because they operate on the linear chromosome and not on the expression tree itself. They are also simple because offspring never have to be checked for legality, a requirement which in other functional systems can be complex and costly of computation time. However, perhaps the most attractive potential advantage of GEP is that expression tree size is limited by the fixed size of the chromosome.

As noted in the Introduction, a classifier system seeks not only to model an environment accurately, but to do so with transparency, i.e., in a way that offers insight into its characteristics and regularities. It does this by evolving

a collection of separate classifiers, each describing a part of the environment that ideally corresponds to one of the regularities, with the classifier's condition describing the part. Thus, it is important that the conditions be concise and quite easily interpretable. For this, GEP would seem to be better than other functional systems such as GP because once a chromosome size is chosen, the expression tree size is limited and very much less subject to "bloat" [17] than GP tree structures are. In GP, crossover between trees can lead to unlimited tree size unless constrained for example by deductions from fitness due to size. In GEP, the size cannot exceed a linear function ($hn_{max} + 1$) of the head length and no fitness penalty is needed.

4 An Experiment

4.1 Setup

The XCSF-GEP system was tested on the tent landscape of Figure 1. As with XCSF in its function approximation version [21, 22], XCSF-GEP was given random x, y pairs from the domain $0.0 \leq x, y \leq 1.0$, together with payoff values equal to $P(x, y)$. XCSF-GEP formed a match set [M] of classifiers matching the input, calculated its system prediction, $\hat{P}$, and the *system error* $|\hat{P} - P(x, y)|$ was recorded. Then, as in XCSF, the predictions of the classifiers in [M] were adjusted using $P(x, y)$, other classifier parameters were adjusted, and a genetic algorithm was run in [M] if called for. In a typical run of the experiment this cycle was repeated 10,000 times, for a total of 20 runs, after which the average system error was plotted and the final populations examined. Runs were started with empty populations.

For classifier conditions, XCSF-GEP used the *function set* {`+ - * / >`} and the *terminal set* {`a b`}. If the divide function "`/`" encountered a zero denominator input the result was set to 1.0 and the fitness of the associated chromosome was set to a very small value. The function "`>`" is the usual "greater than" except the output values are respectively 1 and 0 instead of `true` and `false`. To be added to [M], the evaluation of a classifier's expression tree was required to exceed a *match threshold* of 0.0. In covering and in mutation, the first (root) element of the chromosome was not allowed to be a terminal.

Partially following [21] and using the notation of Butz and Wilson [5], parameter settings for the experiment were: population size $N = 100$, learning rate $\beta = 0.4$, error threshold $\epsilon_0 = 0.01$, fitness power $\nu = 5$, GA threshold $\theta_{GA} = 12$, crossover probability (one point) $\chi = 0.3$, deletion threshold $\theta_{del} = 50$, fitness fraction for accelerated deletion $\delta = 0.1$, delta rule correction rate $\eta = 1.0$, constant x_0 augmenting the input vector $= 0.5$. Prior to the present experiment, an attempt was made to find the best settings (in terms of speed of reduction of error) for β, θ_{GA}, and χ. The settings used in the experiment were the best combination found, with changes to θ_{GA} (basically, the GA frequency) having the greatest effect.

Parameters specific to XCSF-GEP included a mutation rate $\mu = 2.0$. Following Ferreira ([8], p. 77), μ sets the average mutation rate (number of mutations)

per chromosome which, divided by the chromosome length, gives the rate per element or allele. A rate of $\mu = 2.0$ has been found by Ferreira to be near-optimal. The head length was set to 6, giving a chromosome length of 13. Inversion and transposition were not used, nor was subsumption since there is no straightforward way to determine whether one chromosome subsumes another.

Besides testing XCSF-GEP as described in this paper, the experiment also tested the same system, but with a different technique for translating the chromosome. Ferreira calls the breadth-first technique "Karva" whereas it is also possible to translate in a depth-first fashion called "prefix" (see, e.g., [16]). Like Karva, prefix has the property that every chromosome translates to a valid expression tree. Figure 2, right side, shows the translation of the chromosome of Sect. 3.1 using prefix. Looking at examples of chromosomes and their trees, it is possible to see a tendency under prefix more than under Karva for subsequences of the chromosome to translate into compact functional subtrees. This may mean (as Li et al [16] argue) that prefix preserves and propagates functional building blocks better than Karva. We therefore also implemented prefix translation.

4.2 Results

Figure 3 shows the results of an experiment using the parameter settings detailed above, in which XCSF-GEP learned to approximate the tent landscape of Figure 1. For both translation techniques, the error fell to nearly zero, with prefix falling roughly twice as fast as Karva. Still, initial learning was slower than with ordinary XCSF (also shown; relevant parameters the same as for XCSF-GEP). However, XCSF's error performance was markedly worse.

Evolved populations contained about 70 macroclassifiers [5] so that they had definitely not reduced to the ideal of just two, one for each of the tent sides. However, roughly half of the classifiers in a population were accurate and the conditions of roughly half of those precisely covered a tent-side domain. Figure 4 shows the conditions of eight high-numerosity classifiers from four runs of the experiment. The first and second pairs are from two runs in which Karva translation was used; the runs for the third and fourth pairs used prefix translation. With each chromosome is shown the algebraic equivalent of its expression tree, together with the domain in which that tree matched. The relation between the algebra and the domain is: if and only if the inputs satisfy the domain expression, the algebraic expression will compute to a value greater than zero (i.e., the classifier matches).

Many of the expressions simplify easily. For instance if the first is prepended to "> 0" (i.e., $b - (b + a)b > 0$), the result can be seen to be the same as the domain expression. However, other algebraic expressions are harder to simplify, and it seems clear that for interpretation of XCSF-GEP conditions in general, automated editing is called for. Further, it was quite remarkable how many different but correct algebraic expressions appeared in the total of forty runs of the experiment. Even though many conditions evolved to precisely delineate the two regularities of this environment, each of those regularities clearly has a multitude of algebraic descriptions. XCSF-GEP was prolific in finding these,

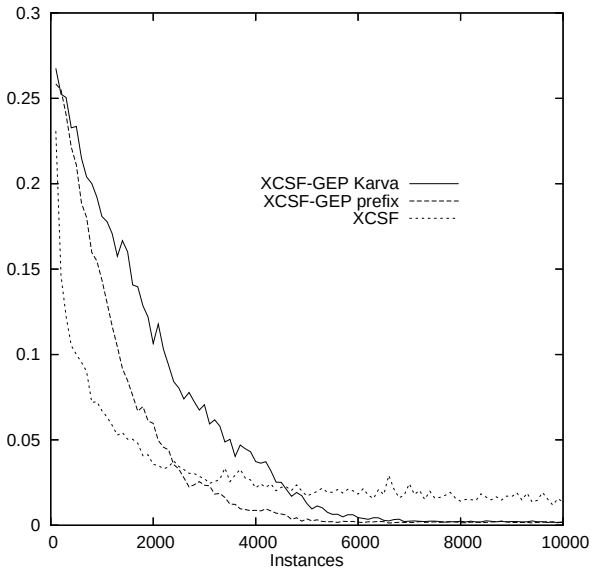

Fig. 3. XCSF-GEP system error vs. learning instances for tent landscape using Karva and prefix translation. Also, system error for XCSF. (Averages of 20 runs).

Chromosome	Algebraic equivalent	Domain
1. `(- - * b > + b b b b a b b)`	$b-(b+a)b$	$a+b<1$
2. `(* - * + / b b a b b b a b)`	$((a+b)-1)b^2$	$a+b>1$
3. `(> - - / b + a a a a a b a)`	$(1-b)>a$	$a+b<1$
4. `(> b - > a / b b a b a b a)`	$b>((b/a>b)-a)$	$a+b>1$
5. `(> - * a / / b b a b a b a)`	$(a(1/a)-b)>a$	$a+b<1$
6. `(- * * a / + b a b b a b a)`	$a((b+a)/b)b-a$	$a+b>1$
7. `(- * a + * + b a b a a b a)`	$((b+a)b+a)a-a$	$a+b>1$
8. `(- - > / / + a b b b b b a)`	$((a+b)/b^2>b)-b-a$	$a+b<1$

Fig. 4. Algebraic equivalents and domains of match for high-numerosity chromosomes evolved in the experiment of Section 4. In 1-4, expression trees were formed by Karva translation; in 5-8, by prefix translation.

but the evolutionary process, even in this simple problem, did not reduce them to two, or even a small number.

5 Discussion and Conclusion

The experiment with the tent environment showed it was possible to use GEP for the conditions of XCSF, but the learning was fairly slow and the evolved populations were not compact. However, it was the case—fulfilling one of the main objectives—that the high-numerosity classifiers, if somewhat obscurely, corresponded precisely in their conditions to this environment's oblique regularities.

Many questions ensue. On speed, it must be noted that the experimental system did not use the full panoply of genetic operators, transposition in particular, that are available in GEP, so that search was in some degree limited. Also, the conditions did not have the multigenic structure that is believed important [8] to efficiency. Nor was GEP's facility for random numerical constants used; the constants needed in the current problem were evolved algebraically, e.g., $1 = b/b$.

The match threshold may matter for speed. If it is set too low, more classifiers match, and in this sense the generality of all classifiers is increased. In effect, the generality of an XCSF-GEP classifier is defined in relation to the match threshold (which could conceivably be adaptive). Since over-generality leads to error, too low a threshold will increase the time required to evolve accurate classifiers. There is a substantial theory [3] of factors, including generality, that affect the rate of evolution in XCS-like systems; it should be applicable here.

On compactness the situation is actually much improved by the fact that regularity-fitting classifiers *do* evolve, in contrast to the poor fit of rectilinear classifiers for all but rectilinear environments. Considerable work (e.g. [23, 6, 9, 18, 4]) exists on algorithms that reduce evolved populations down to minimal classifier sets that completely and correctly cover the problem environment. If the population consists of poorly fitting classifiers, the resulting minimal sets are not very small. However, if, as with XCSF-GEP in this experiment, two classifiers are evolved that together cover the environment, compaction methods should produce sets consisting of just those two. Thus XCSF-GEP plus postprocessing compaction (plus condition editing) should go quite far toward the goals of conciseness and transparency. All this of course remains to be tested in practice.

In conclusion, defining classifier conditions using GEP appears from this initial work to lead to a slower evolution than traditional rectilinear methods, but captures and gives greater insight into the environment's regularities. Future research should include implementing more of the functionality of GEP, exploring the effect of the match threshold, testing compaction algorithms, and extending experiments to more difficult environments.

Acknowledgement

The author acknowledges helpful and enjoyable correspondence with Cândida Ferreira.

References

1. Larry Bull and Toby O'Hara. Accuracy-based neuro and neuro-fuzzy classifier systems. In Langdon et al. [12], pages 905–911.
2. Martin V. Butz. Kernel-based, ellipsoidal conditions in the real-valued XCS classifier system. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1835–1842, Washington DC, USA, 25-29 June 2005. ACM Press.
3. Martin V. Butz. *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*. Springer-Verlag, Berlin-Heidelberg, 2006.
4. Martin V. Butz, Pier Luca Lanzi, and Stewart W. Wilson. Function approximation with XCS: hyperellipsoidal conditions, recursive least squares, and compaction. *IEEE Transactions on Evolutionary Computation*, in press.
5. Martin V. Butz and Stewart W. Wilson. An Algorithmic Description of XCS. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 1996 of *LNAI*, pages 253–272. Springer-Verlag, Berlin, 2001.
6. Phillip W. Dixon, Dawid W. Corne, and Martin J. Oates. A ruleset reduction algorithm for the XCS learning classifier system. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems*, volume 2661 of *LNAI*, pages 20–29, Berlin Heidelberg, 2002. Springer.
7. Cândida Ferreira. Gene expression programming: a new algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.
8. Cândida Ferreira. *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*. 2nd Edition, Springer-Verlag, Germany, 2006.
9. Chunsheng Fu and Lawrence Davis. A modified classifier system compaction algorithm. In Langdon et al. [12], pages 920–925.
10. John H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-directed Inference Systems*. New York: Academic Press, 1978. Reprinted in: Evolutionary Computation. The Fossil Record. David B. Fogel (Ed.) IEEE Press, 1998. ISBN: 0-7803-3481-7.
11. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.
12. William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*. Morgan Kaufmann, 2002.
13. Pier Luca Lanzi. Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*, pages 345–352. Morgan Kaufmann, 1999.
14. Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Learning Classifier Systems. From Foundations to Applications*, volume 1813 of *LNAI*. Springer-Verlag, Berlin, 2000.

15. Pier Luca Lanzi and Stewart W. Wilson. Using convex hulls to represent classifier conditions. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1481–1488, Seattle, Washington, USA, 8-12 July 2006. ACM Press.
16. Xin Li, Chi Zhou, Weimin Xiao, and Peter C. Nelson. Prefix gene expression programming. In Franz Rothlauf, editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2005)*, Washington, D.C., USA, 25-29 June 2005.
17. Sean Luke. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.
18. Kreangsak Tamee, Larry Bull, and Ouen Pinngern. Towards clustering with xcs. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1854–1860, London, 7-11 July 2007. ACM Press.
19. Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.
20. Stewart W. Wilson. Get Real! XCS with Continuous-Valued Inputs. In Lanzi et al. [14], pages 209–219.
21. Stewart W. Wilson. Function approximation with a classifier system. In Lee Spector, Erik D. Goodman, Annie Wu, W.B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 974–981, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.
22. Stewart W. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2-3):211–233, 2002.
23. Stewart W. Wilson. Compact rulesets from XCSI. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 2321 of *LNAI*, pages 196–208. Springer-Verlag, Berlin, 2002.
24. Stewart W. Wilson. Classifier systems for continuous payoff environments. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 824–835, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.
25. Stewart W. Wilson. Three architectures for continuous action. In Tim Kovacs, Xavier Llora, Keiki Takadama, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems*, volume 4399 of *LNAI*, pages 239–257, Berlin Heidelberg, 2007. Springer.