Evol. Intel. (2013) 6:73–91
DOI 10.1007/s12065-013-0091-1

RESEARCH PAPER

Learning complex, overlapping and niche imbalance Boolean problems using XCS-based classifier systems

Muhammad Iqbal · Will N. Browne · Mengjie Zhang

Received: 15 August 2013 / Revised: 16 September 2013 / Accepted: 17 September 2013 / Published online: 8 October 2013

Abstract XCS is an accuracy-based learning classifier system, which has been successfully applied to learn various classification and function approximation problems. Recently, it has been reported that XCS cannot learn overlapping and niche imbalance problems using the typical experimental setup. This paper describes two approaches to learn these complex problems: firstly, tune the parameters and adjust the methods of standard XCS specifically for such problems. Secondly, apply an advanced variation of XCS. Specifically, we developed previously an XCS with code-fragment actions, named XCSCFA, which has a more flexible genetic programming like encoding and explicit state-action mapping through computed actions. This approach is examined and compared with standard XCS on six complex Boolean datasets, which include overlapping and niche imbalance problems. The results indicate that to learn overlapping and niche imbalance problems using XCS, it is beneficial to either deactivate action set subsumption or use a relatively high subsumption threshold and a small error threshold. The XCSCFA approach successfully solved the tested complex, overlapping and niche imbalance problems without parameter tuning, because of the rich alphabet, inconsistent actions and especially the redundancy provided by the code-fragment actions. The major contribution of the work presented here is overcoming the identified problem in the widespread XCS technique.

Keywords Learning classifier systems · XCS · XCSCFA · Genetic programming · Code fragments · Overlapping problems · Niche imbalance

M. Iqbal (✉) · W. N. Browne · M. Zhang
Evolutionary Computation Research Group, School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand
e-mail: muhammad.iqbal@ecs.vuw.ac.nz

W. N. Browne
e-mail: will.browne@ecs.vuw.ac.nz

M. Zhang
e-mail: mengjie.zhang@ecs.vuw.ac.nz

1 Introduction

A learning classifier system (LCS) is a rule-based online learning system [8, 19] that uses an evolutionary mechanism, usually a genetic algorithm (GA), to evolve classifier rules and evaluates the utility of the rules using a machine learning technique. XCS is an accuracy-based learning classifier system, originally introduced by Wilson [51], which has been successfully used to learn various classification as well as function approximation problems [10]. Recently, Ioannides et al. [22] experimentally showed that XCS was not able to learn digital design verification (DV) problems, which are Boolean problems where the complete solution set consists of overlapping classifiers. In addition, they are niche imbalance problems, i.e. the niches covered by the classifiers in the complete solution have different sizes (see Sect. 4.1). Ioannides et al. [21] suggested two techniques to improve XCS performance in DV problems, but could not reach 100 % performance level.

Previously, we implemented code-fragment based XCS systems, where a code-fragment is a tree-expression similar to a tree generated in genetic programming (see Sect. 2.1). The main aim of using code-fragments was to identify and extract building blocks of knowledge from smaller problems in a domain and to reuse the extracted knowledge to solve complex, large-scale problems in the same domain. First, we introduced code-fragments in the condition of a

classifier rule in XCSCFC [28] and solved problems of a scale that existing classifier system and genetic programming approaches cannot, e.g. the 135-bit MUX problem. Later on, we used a code-fragment in the action of a classifier rule in XCSCFA [26], which evolved the optimum solutions for 6-bit to 37-bit multiplexer problems. The aim of the work presented here is to investigate the standard XCS and the code-fragment based XCSCFA systems in learning different complex Boolean problems, especially overlapping and niche imbalance problems.

The rest of the paper is organised as follows. Section 2 describes the necessary background in genetic programming and learning classifier systems. In Sect. 3 the implementation of XCSCFA is detailed. Section 4 introduces the problem domains and experimental setup, which is crucial to the identified XCS difficulties. In Sect. 5 experimental results are presented and compared with standard XCS as the most appropriate benchmark algorithm. Section 6 is an analysis of the evolved classifier rules. In Sect. 7 the capabilities of standard XCS in learning overlapping and niche imbalance problems are further investigated using different experimental settings. Section 8 describes the effect of action inconsistency and redundancy in the XCSCFA approach. In the ending section this work is concluded and the future work is outlined.

2 Background

Evolutionary computation is a population-based computing paradigm [30] where each individual represents a potential solution or a part of the solution to the problem at hand. The population is evolved by applying genetically inspired operations of *reproduction, elitism, crossover* and *mutation* on the selected individuals, according to their utility for the task being solved.

In the following subsections, two evolutionary techniques, namely genetic programming and learning classifier systems, are briefly described as they are directly related to the work presented here. This is followed by an introduction to code-fragment based classifier systems.

2.1 Genetic programming

Commonly, genetic programming (GP) uses a rich alphabet to encode the solution, i.e. more expressive symbols that can express functions as well as numbers. A GP like alphabet to describe the problem is used in the LCS studied here, so the GP technique is briefly described to aid understanding.

GP is an evolutionary approach where each individual is a computer program consisting of a set of functions and a set of terminals, and usually represented by a tree [45]. A tree-GP computer program may contain unnecessary bloating terms and non-optimum expressions. These problems are usually addressed by limiting maximal allowed depth for an individual tree and/or using a fitness measure that punishes excess sized individuals [42]. The other ways to control bloat in GP include simplifying individual programs using algebraic and numerical simplification methods [31, 57], or using specific bloat control operators [2]. GP has also been implemented using non-tree representations such as linear GP [6] and cartesian GP [44].

A GP system produces a computer program as a 'single' solution, rather than a co-operative set of rules as in an LCS. It generally requires supervised learning with the whole training set, rather than online, reinforcement learning as in LCS.

2.2 Learning classifier systems

Traditionally, an LCS represents a rule-based agent that incorporates evolutionary computing and machine learning to solve a given task by interacting with an unknown environment via a set of sensors for input and a set of effectors for actions [8, 19]. After observing the current state of the environment, the agent performs an action, and the environment provides a reward. The goal of an LCS is to evolve a set of classifier rules that collectively solve the problem. The generalization property in LCS allows a single rule to cover more than one environmental state provided that the action-reward mapping is similar. Traditionally, generalization in an LCS is achieved by using a special 'don't care' symbol (#) in classifier conditions, which matches any value of a specified attribute in the vector describing the environmental state. LCS can be applied to a wide range of problems including data mining, control, modeling and optimization problems [3, 7, 46].

2.2.1 XCS

XCS is a formulation of LCS that uses accuracy-based fitness to learn the problem by forming a complete mapping of states and actions to rewards. [1] In XCS, the learning agent evolves a population [*P*] of classifiers, where each classifier consists of a rule and a set of associated parameters estimating the quality of the rule. Each rule is of the form 'if *condition* then *action*', having two parts: a condition and the corresponding action. Commonly, the condition is represented by a fixed-length bitstring defined over the ternary alphabet {0, 1, #}, and the action is represented by a numeric constant. Each classifier has three

[1] For a detailed review of different types and approaches in LCS refer to [36, 48].

main parameters: (1) prediction p, an estimate of the payoff expected from the environment if its action is executed; (2) prediction error ϵ, an estimate of the errors between the predicted payoff and the actually received reward; and (3) fitness F, an estimate of the classifier's utility. In addition, each classifier keeps an experience parameter *exp*, which is a count of the number of times it has been updated, and a numerosity parameter n, which is a count of the number of copies of each unique classifier.

In the following, XCS operations are concisely described. For a complete description, the interested reader is referred to the original XCS papers by Wilson [51, 52], and to the algorithmic details by Butz and Wilson [14].

In XCS, on receiving the environmental input state s, a match set $[M]$ is formed consisting of the classifiers from the population $[P]$ that have conditions matching the input s. For every action a_i in the set of all possible actions, if a_i is not represented in $[M]$ then a covering classifier is randomly generated. After that an action a is selected to be performed on the environment and an action set $[A]$ is formed, which consists of the classifiers in $[M]$ that advocate a. After receiving an environmental reward, the associated parameters of all classifiers in $[A]$ are updated. When appropriate, new classifiers are produced using an evolutionary mechanism, usually a GA.

Additionally, in XCS overly specific classifiers may be subsumed by any more general and accurate classifiers. There are two subsumption procedures: (a) *GA subsumption*, and (b) *action set subsumption*. If *GA subsumption* is being used and an offspring generated by the GA has the same action as that of the parents, then its parents are examined to see if either of them: (1) has an experience value greater than a threshold, (2) is accurate, and (3) is more general than the offspring, i.e. has a set of the matching environmental inputs that is a proper superset of the inputs matched by the offspring. If this test is satisfied, the offspring is discarded and the numerosity of the subsuming parent is incremented by one. If the offspring is not subsumed by its parents, then it can be checked if it is subsumed by other classifiers in the action set.

In *action set subsumption*, any less general classifiers in an action set $[A]$ are subsumed by the most general subsumer (i.e. accurate and sufficiently experienced) classifier in the set $[A]$. Subsumption deletion is a way of biasing the genetic search towards more general, but still accurate, classifiers [13, 52]. It also effectively reduces the number of classifier rules in the final population [32].

2.2.2 LCS with rich encoding schemes

Various rich encoding schemes have been investigated in LCS to represent high level knowledge in an attempt to improve the generalization, to obtain compact classifier rules, to reach the optimal performance faster, to generate feature extractors, and to investigate scalability of the learning system. Most of these schemes have been implemented on Wilson's XCS, which is a well-tested LCS model.

A GP-based rich encoding has been used by Ahluwalia and Bull [1] within a simplified strength-based LCS [50]. They used binary strings to represent the condition and an S-expression to represent the action of a classifier rule. This GP-based LCS generates filters for feature extraction, rather than performing classification directly. The extracted features are used by the k-nearest neighbor algorithm to perform classification.

Bull and O'Hara [9] developed XCS-based neuro and neuro-fuzzy classifier systems (named X-NCS and X-NFCS), where a condition-action rule was represented by a small neural network and the action value of a classifier rule was computed by feedforwarding the environmental state to the neural network. Experimental results indicate that neural network based classifier systems are able to learn single-step as well as multi-step problems [9, 20].

Dam et al. [15] implemented a UCS-based classifier system[2] by replacing the typically used numeric action in a classifier rule with a neural network. The developed system, known as NLCS, resulted in better generalization, more compact solutions, and the same or better classification accuracy than a numeric action based UCS.

Lanzi extended the fixed-length bitstrings representation of classifier conditions to a variable-length messy coding in [34]. A messy coded string may be over- or under-specified due to its variable-length structure [18]. In the messy coded conditions by Lanzi, environmental inputs were translated into bitstrings that have no positional linking between a bit in a classifier condition and any feature in the environmental input. Then Lanzi and Perrucci [40] enhanced a step further from messy coding to a more complex representation in which S-expressions were used to represent classifier conditions.

In 2000, Wilson introduced XCSR, a version of XCS taking real inputs [53]. In XCSR, the classifier condition is represented as a concatenation of "interval predicates", $int_i = (c_i, s_i)$, where c_i and s_i are reals. A classifier matches an input x if and only if $c_i - s_i \leq x_i < c_i + s_i$, for all x_i. Wilson also introduced an integer based version of XCS, known as XCSI [54]. In XCSI, the classifier condition is represented as a concatenation of "interval predicates", $int_i = (l_i, u_i)$, where l_i and u_i are integers defining the lower and upper bounds of the interval. A classifier matches an input x if and only if $l_i \leq x_i \leq u_i$, for all x_i. Behdad

[2] UCS (sUpervised Classifier System) [5] is a variant of XCS, which is specifically designed for supervised learning problems.

et al. [4] combined XCSR with principal component analysis to reduce the computational time and the population size required by XCSR in learning various high-dimensional real valued problems.

Wilson introduced the idea of computed prediction, as a function of the input matched by classifier condition and a weight vector, to learn approximations to functions [55]. In the implemented system, known as XCSF, classifier condition was changed from ternary alphabet string to a concatenation of interval based numeric values. Stalph et al. [47] compared the function approximation performance of XCSF with the locally weighted projection regression algorithm (LWPR) [49], which is a statistics-based machine learning technique mostly used for function approximation tasks in robotics. The experimental results show that both methods achieve a suitable performance, but the evolutionary structuring capability of XCSF is more powerful than LWPR's stochastic gradient descent. In 2005, Lanzi et al. [38] used XCSF for the learning of Boolean functions. They have shown that XCSF can produce more compact classifier rules as compared to XCS since the use of computed prediction allows more general solutions [39].

Lanzi developed an XCS with stack-based genetic programming [35] where the classifier conditions were represented by mathematical expressions using reverse polish notation (RPN). The system did not restrict the generation of syntactically incorrect conditions, therefore the search space was unnecessarily large. Even then, it is reported that the system was able to learn the 6-bit multiplexer, the 11-bit multiplexer and the woods1 problems.

Lanzi and Loiacono [37] introduced a version of XCS with computed actions, named XCSCA, to be used for problem domains involving a large number of actions. The classifier action was computed using a parametrized function in a supervised fashion. They have shown that XCSCA can evolve accurate and compact representations of binary functions which would be difficult to solve using standard XCS. Then, Loiacono et al. [41] extended XCSCA using support vector machines to compute the classifier action, which resulted in reaching optimal performance faster than the original XCSCA.

In 2008, Wilson implemented classifier conditions using gene expression programming (GEP) [56]. GEP based conditions have captured and shown greater insight into the environment's regularities, albeit the evolution of the system was slower, than traditional methods.

2.2.3 *Code-fragment based classifier systems*

The main goal of this research direction is to develop a scalable classifier system. To achieve this goal, we implemented a GP-like rich encoding scheme, in the conditions of classifier rules, in an XCS based system to identify building blocks of knowledge [29]. Then the fitter building blocks of knowledge were extracted, in the form of tree-like code fragments, from low dimensional problems and reused in learning more complex, large-scale problems in the domain [23, 28].

Sr. No.	Condition						Action
	D0	D1	D2	D3	D4	D5	
1	0	0	1	#	#	0	D4D0&D2|
2	0	1	#	0	0	#	D2D5&D0D3|d
3	0	0	#	1	0	1	D0D1|D2D5&|
4	0	#	1	0	1	0	D2D0d
5	1	0	0	#	1	1	D5~D1r
6	0	0	1	0	#	#	D3D1rD0D3&d
...	...						...

Fig. 1 Classifier population using code-fragment actions

Subsequently, we implemented the code-fragment encoding scheme in the action of a classifier rule in XCSCFA [26], which evolved optimum populations for 6-bit to 37-bit multiplexer problems. We also implemented code-fragment based action in real-valued XCSRCFA to compute continuous action [24].

The aim of the work presented here is to investigate the standard XCS and the code-fragment based XCSCFA systems in learning different complex Boolean problems, especially overlapping and niche imbalance problems.[3] The next section describes in detail, with algorithmic descriptions, the XCSCFA approach.

3 XCS with code-fragment actions

In XCSCFA, the typically used numeric action is replaced by a GP-tree like code fragment [26]. Each code fragment is a binary tree and to limit the tree size a code fragment can have maximum seven nodes. The function set for the tree is problem dependent such as $\{\text{AND}, \text{OR}, \text{NOT}\ldots\}$ for binary classification problems and $\{+, -, *, /\ldots\}$ for symbolic regression problems. The terminal set is $\{\text{D0}, \text{D1}, \text{D2}, \ldots \text{D}n-1\}$ where n is the length of an environmental input message. A population of classifiers having code-fragment actions is illustrated in Fig. 1. The symbols &, |, ~, d, and r denote AND, OR, NOT, NAND, and NOR operators, respectively. The code-fragment trees are shown in postfix form.

The XCSCFA approach extends standard XCS [14] in the following aspects: the action value, the covering operation, the rule discovery operation, the procedure comparing equality of two code-fragment actions, and the subsumption deletion mechanism.

[3] This work is an extension of our previous work on XCSCFA published in [25, 27].

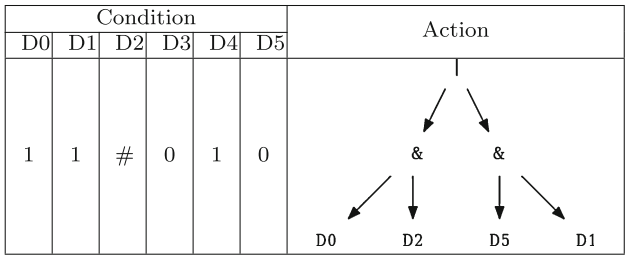


Fig. 2 A classifier with code-fragment action

3.1 Code-fragment action value

In XCSCFA, the action value of a classifier is usually computed by loading the terminal symbols in the action tree with corresponding binary values from the condition in the classifier rule, and a 'don't care' symbol in the condition is randomly treated as 0 or 1. For example, consider the classifier rule shown in Fig. 2 and the environmental input message '110010'. In the condition of this classifier rule, D2 is a '#' symbol. To compute the action value in this classifier rule, D2 will be loaded with 0 or 1 randomly. Now, if the binary value taken for D2 is 0 then the action will be 0, otherwise 1. Hence, the action value in XCSCFA may vary, even for the same environmental instance, at different times during the training process; unlike standard XCS.

In this work, a more intuitive implementation of XCSCFA is presented where the action value is computed by loading the terminal symbols with the corresponding binary values from the environmental input instead of the classifier condition. For sake of readability, the two implementations of XCSCFA are named XCSCFA$_c$ (i.e. classifier based) and XCSCFA$_e$ (i.e. environment based). It is to be noted that in XCSCFA$_c$ the action value is independent of the input, as it is in standard XCS, whereas in XCSCFA$_e$ the action value depends upon the input matched by the classifier condition. A classifier rule in XCSCFA$_e$ can have different action values for different environmental instances, but for the same instance it will have the same action value every time, unlike XCSCFA$_c$ where this can vary.

3.2 Covering operation

Covering occurs if an action a is missing in the match set $[M]$. In the covering operation, a random classifier is created whose condition matches the current environmental state s and contains '#' symbols with probability $P_{\#}$. The code-fragment action tree is randomly generated until its output is a. The covering operation is described in Algorithm 1. Here n is the length of condition *cond* in a classifier rule, and $P_{\#}$ is the probability of the 'don't care' symbol '#' in condition of the newly created classifier in the covering operation.

Algorithm 1: Covering_Operation

Data: The currently observed input state s and an action a missing in the match set $[M]$.
Result: A newly created classifier cl matching the state s and advocating the action a.

```
initialize classifier cl
initialize condition cl.cond with length n
for i = 1 to n do
    if RandomNumber[0, 1) < P# then
        cl.cond[i] ← #
    else
        cl.cond[i] ← s[i]
    end
end
repeat
    cf ← randomly create code fragment
    val ← evaluate value of cf
until val = a
cl.action ← cf
return cl
```

3.3 Rule discovery operation

In the rule discovery operation, two offspring are created using the following steps. First of all, two parent classifiers are selected from the action set $[A]$ based on fitness and the offspring are created from them. Next, the conditions and action trees of the offspring are crossed with probability χ by applying GA- and GP-based crossover operations respectively.

Then each symbol in the conditions of the crossed over children is mutated with probability μ, such that both children match the currently observed state s. Then, the action trees of the children are mutated with probability μ, using GP-based mutation, by replacing a subtree of the action with a randomly generated subtree of depth up to 1.

The prediction and prediction error of the offspring are set to the average of the parents' values whereas the fitness of the offspring is set to the average of the parents' values multiplied by the constant *fitnessReduction*, as suggested by Butz and Wilson in [14].

For example, consider the rule discovery operation graphically summarized in Fig. 3. Figure 3a shows the action set containing three classifiers with action value 1, formed from the classifier population $[P]$ shown in Fig. 1, against the environmental input $s = 001010$. First of all, two parent classifiers are selected, Fig. 3b, from the action set based on fitness F, and two children are created from them, Fig. 3c. Then, Fig. 3d, the conditions of the reproduced children are crossed over by applying two-point GA-crossover operation at the two marked points, and the action trees of the children are crossed over by applying

Condition						Action	F
0	0	1	#	#	0	D4D0&D2|	0.45
0	#	1	0	1	0	D2D0d	0.09
0	0	1	0	#	#	D3D1rD0D3&d	0.35

(a) action set

Condition						Action	F
0	0	1	#	#	0	D4D0&D2|	0.45
0	0	1	0	#	#	D3D1rD0D3&d	0.35

(b) selected parents

Condition						Action	F
0	0	1	#	#	0	D4D0&D2|	0.45
0	0	1	0	#	#	D3D1rD0D3&d	0.35

(c) reproduced children

Condition						Action	F
0	0	1	0	#	0	D4D0&D3D1r|	0.04
0	0	1	#	#	#	D2D0D3&d	0.04

(d) crossed over children

Condition						Action	F
0	#	1	0	#	0	D5~D3D1r|	0.04
0	0	1	#	1	#	D2D0D1&d	0.04

(e) mutated children

Condition						Action	F
0	#	1	0	#	0	D5~D3D1r|	0.04
0	0	1	#	1	#	D2D0D1&d	0.04

(f) final children

Fig. 3 Rule discovery by applying GA- and GP-based operations in an action set

GP-crossover operation. The fitness value of crossed over children is set to the average of parents' fitness values multiplied by 0.1, as suggested by Butz and Wilson in [14]. After that, Fig. 3e, the conditions of the crossed over children are mutated by applying GA-mutation such that both children match the currently observed state $s = 001010$, and the action trees of the children are mutated by applying GP-mutation. The final children generated in the rule discovery operation are shown in Fig. 3f.

3.4 Comparing two code-fragment actions

If a newly created classifier in the rule discovery operation is not subsumed (either by the parents or in the action set) and there is no classifier equal to it in the population, then it will be added to the population. Two classifiers are considered to be equal if and only if both have the same conditions and the genotypically same code fragment in their actions. The code fragment *genotype* is its formal expression as seen in the classifier action, and the *phenotype* is the value that the action computes to with a given input. The procedure to compare two code-fragment actions for equality is given in Algorithm 2.

Algorithm 2: Are_Equal_Actions

Data: Two code-fragment actions cf_1 and cf_2.
Result: If the actions cf_1 and cf_2 are genotypically the same, then this algorithm will return *true* otherwise *false*.

```
m1 ← Length of code-fragment action cf1
m2 ← Length of code-fragment action cf2
if m1 ≠ m2 then
    return false
end
for i = 1 to m1 do
    if cf1[i] ≠ cf2[i] then
        return false
    end
end
return true
```

3.5 Subsumption deletion

A classifier cl_1 can subsume another classifier cl_2 if both have the same action and cl_1 is accurate, sufficiently experienced, and more general than cl_2 [14].

It is to be noted that due to the multiple genotypes to a single phenotype mapping of code-fragment actions in classifier rules, subsumption deletion is less likely to occur in the XCSCFA systems. Subsumption deletion is still made possible by matching the code-fragment actions on a character by character basis. Originally in XCSCFA [26], it was necessary for a subsumer classifier to have a consistent action value, i.e. the same action value at any time for every matched input message, as a precaution to avoid any volatility in the performance of the classifier system. In the work presented here, this restriction has been removed in order to perform fair comparison with XCS; and to determine whether the XCSCFA$_c$ and XCSCFA$_e$ systems can produce more general classifiers by incorporating generalization in the action of a classifier rule.

4 The experimental design

4.1 The problem domains

The problem domains used in the experimentation are the multiplexer, the majority-on, the count ones, the digital design verification, the carry, and the even-parity.

A multiplexer is an electronic circuit that accepts input strings of length $n = k + 2^k$, and gives one output. The value encoded by the k address bits is used to select one of the 2^k remaining data bits to be given as output. For

example in the 6-bit multiplexer, if the input is 011101 then the output will be 1 as the first two bits 01 represent the index 1 (in base ten), which is the second bit following the address. Multiplexer problems are highly non-linear, multi-modal and have epistasis, i.e. importance of data bits is dependent on address bits.

In majority-on problems, the output depends on the number of ones in the input instance. If the number of ones is greater than the number of zeros, the problem instance is of class one, otherwise class zero. In the majority-on problem domain, the complete solution consists of over-lapping classifiers, so is difficult to learn. For example, '1##11:1' and '11#1#:1' are two maximally general and accurate classifiers, but they overlap in the "11*11" subspace.[4]

Count ones problems are similar to majority-on problems in that the output depends on the number of ones in the input instance. In count ones problems only k bits are relevant in an input instance of length l [10]. If the number of ones in the k relevant positions is greater than half k, the problem instance is of class one, otherwise class zero. For example, consider a count ones problem of length $l = 7$ with the first $k = 5$ relevant bits. In this problem, input string 1010110 would be in class one whereas input string 1001011 would be class zero. Similar to majority-on problems, the complete solution for a count ones problem consists of overlapping classifiers.

Digital design verification is a real world problem domain, where a digital design is verified before manufacturing in order to discover as many bugs in the design as possible. The design verification problem experimented in this work is a 7-bit Boolean example of a simulation-based DV problem, named DV1, originally introduced by Ioannides et al. [22]. A Boolean function can be represented compactly in the Sigma notation by listing each onset row from the truth table of the function [16]. For example, the function $\bar{x}y + xy$ can be represented in Sigma notation as $\Sigma(1, 3)$. The DV1 problem is denoted by the following Sigma notation: $\Sigma(1, 2, 3, 8, 9, 10, 11, 13, 14, 24, 25, 26, 27, 28, 30, 40, 41, 42, 43, 46, 47, 56, 57, 58, 59, 61, 65, 66, 67, 69, 70, 71, 72, 73, 74, 75, 77, 78, 79, 81, 82, 83, 85, 86, 88, 89, 90, 91, 93, 94, 95, 97, 98, 99, 101, 102, 103, 104, 105, 106, 107, 109, 110, 113, 114, 115, 117, 118, 121, 122, 123, 125, 126, 127)$. Similar to majority-on and count ones problems, the complete solution for the DV1 problem consists of overlapping classifiers, and in addition it is a niche imbalance problem. For example, '#00#0#1 : 1' and '#00#01# : 1' are two maximally general and accurate classifiers, but they overlap in the "*00*011" subspace. Similarly, '#111#01 : 1' and '1####01 : 1' are two maximally general and accurate classifiers, but the size of the niche covered by the latter is twice that of the former. The optimum ternary encoded solution set for the DV1 problem consists of 30 maximally general and accurate classifiers [22], shown in Table 1. It is to be noted that there are different numbers of '#' symbols in the conditions of these overlapping classifiers, which indicate that these classifiers cover different sized niches.

Table 1 The optimum ternary encoded rule set for the DV1 problem [22]

No.	Input	Output	No.	Input	Output
1	###0#00	0	16	1####10	1
2	010#10#	0	17	#00#01#	1
3	0##01##	0	18	#0#10##	1
4	##10111	0	19	#001#01	1
5	##0#100	0	20	##010##	1
6	1###100	0	21	0011##0	1
7	00##111	0	22	#0#1#10	1
8	1101111	0	23	0101#1#	1
9	0#10###	0	24	100###1	1
10	111##00	0	25	0##10##	1
11	001#1#1	0	26	1#00##1	1
12	011#1#0	0	27	#111#01	1
13	01#0###	0	28	1#11##1	1
14	0#1#111	0	29	1####01	1
15	#00#0#1	1	30	1###0#1	1

In carry problems, two binary numbers of the same length are added. If the addition triggers a carry, then the class is one otherwise zero. For example, in case of 3-bit numbers 110 and 101, the class is one, whereas for the numbers 010 and 011 the class is zero. The complete solution in the carry problem domain consists of strongly overlapping classifiers, and in addition it is a niche imbalance problem domain. For example, '##000# : 0' and '###000 : 0' are two maximally general and accurate classifiers, but they overlap in the "**0000" subspace. Similarly, '1##1## : 1' and '1#1#11 : 1' are two maximally general and accurate classifiers, but the size of the niche covered by the former is twice that of the latter. The optimum ternary encoded solution set for the 3+3 bit carry problem consists of 18 maximally general and accurate classifiers, shown in Table 2. It is to be noted that there are different numbers of '#' symbols in the conditions of these overlapping classifiers, which indicate that these classifiers cover different sized niches.

In even-parity problems, the output depends on the number of ones in the input instance. If the number of ones is even, the output will be one, and zero otherwise. Using the ternary alphabet based conditions with the static numeric action, no useful generalizations can be made for even-parity problems.

[4] Here, * can be 0, 1, or #.

Table 2 The optimum ternary encoded rule set for the 3+3 bit carry problem

No.	Input	Output	No.	Input	Output
1	##000#	0	10	#0#00#	0
2	###000	0	11	#000##	0
3	00##0#	0	12	1##1##	1
4	000###	0	13	11##1#	1
5	0##0##	0	14	#1#11#	1
6	0###00	0	15	111##1	1
7	00###0	0	16	1#1#11	1
8	0#0#0#	0	17	#111#1	1
9	#0#0#0	0	18	##1111	1

Table 3 Properties of different problem domains

Problem Domain	Properties
Multiplexer	Multi-modal and epistatic
Majority-on	Overlapping
Count ones	Overlapping
Design verification	Overlapping and niche imbalance
Carry	Overlapping and niche imbalance
Even-parity	Hard to generalize

The properties of different problem domains experimented in the this work are summarized in Table 3.

4.2 Experimental setup

The system uses the following parameter values, commonly used in the literature, as suggested by Butz and Wilson in [14]: learning rate $\beta = 0.2$; fitness fall-off rate $\alpha = 0.1$; fitness exponent $\nu = 5$; prediction error threshold $\epsilon_0 = 10$; two-point crossover with probability $\chi = 0.8$; mutation probability $\mu = 0.04$; fraction of mean fitness for deletion $\delta = 0.1$; experience threshold for classifier deletion $\theta_{del} = 20$; threshold for GA application in the action set $\theta_{GA} = 25$; classifier experience threshold for subsumption $\theta_{sub} = 20$; probability of 'don't care' symbol in covering $P_{\#} = 0.33$; initial prediction $p_I = 10.0$; initial fitness $F_I = 0.01$; initial prediction error $\epsilon_I = 0.0$; reduction of the fitness *fitnessReduction* $= 0.1$; and the selection method is tournament selection with tournament size ratio 0.4. Both GA subsumption and action set subsumption are activated. The function set for the code fragments used is {AND, OR, NOT, NAND, NOR}, denoted by {&, |, ∼, d, r}, for all the problem domains experimented in this work. The number of training examples used is two million for all the experiments. Explore and exploit problem instances are alternated. The reward scheme used is 1000 for a correct classification and 0 otherwise. All the experiments have been repeated 30 times with a different seed in each run.

5 Results

In order to test the performance of the XCSCFA approach, results have been compared with standard XCS on the six problem domains used. Each result reported in this work is the average of the 30 independent runs.

In all graphs presented here, the X-axis is the number of problem instances used as training examples, the Y-axis is the classification performance measured as the moving average over the last 1000 exploit problem instances, and the error bars show the standard deviation in the 30 runs. All graphs in the paper are in color for better readability.

5.1 The multiplexer problem domain

The performance of standard XCS and the XCSCFA methods in learning multiplexer problems is shown in Fig. 4. The number of classifiers used is 500, 1000, 2000, and 6000 for the 6-, 11-, 20-, and 37-bit multiplexer problems respectively. The value of $P_{\#}$ was increased to 0.5 for the 37-bit multiplexer problem, as both the standard XCS and XCSCFA methods were not able to solve it using $P_{\#} = 0.33$.

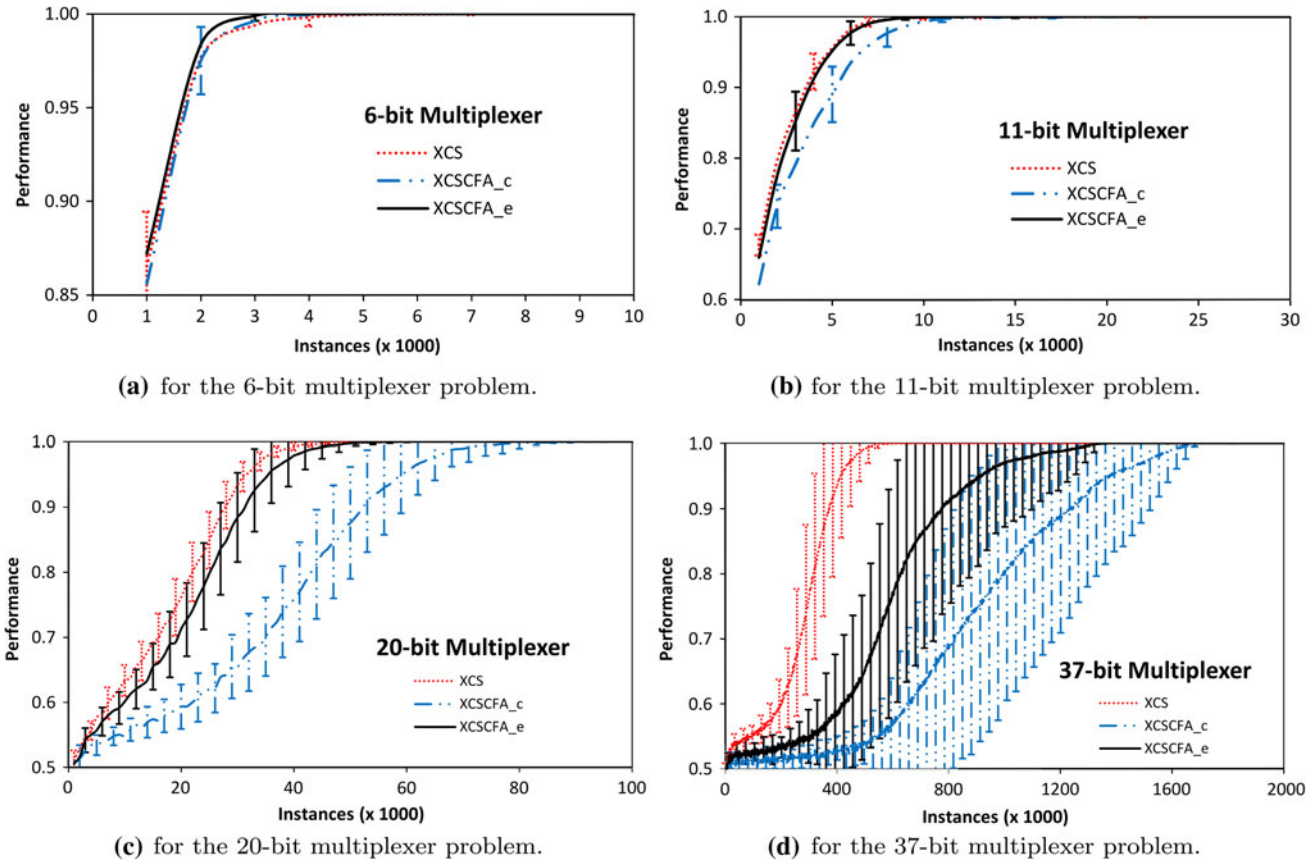
The multiplexer is a niche balanced problem domain and there exists a complete solution that does not contain any overlapping classifier rules. Hence, the standard XCS effectively solved the 6-, 11-, 20-, and 37-bit multiplexer problems. The XCSCFA methods used more training examples, to reach a similar performance level, due to the increased search space.

5.2 The majority-on problem domain

The majority-on problem used in this work is of length $l = 7$. The number of classifiers used is $N = 3{,}000$. The performance of standard XCS and the XCSCFA methods in learning the 7-bit majority-on problem is shown in Fig. 5. It is observed that standard XCS reached approximately 94 % performance level, but could not completely solve the 7-bit majority-on problem, whereas the XCSCFA methods solved it using approximately 20,000 training examples.

5.3 The count ones problem domain

The count ones problem used in this work is of length $l = 20$ with the first $k = 7$ relevant bits. The number of classifiers used is $N = 4000$. The performance of standard XCS and the XCSCFA methods is shown in Fig. 6. It is observed that standard XCS reached approximately 95 %



(a) for the 6-bit multiplexer problem.

(b) for the 11-bit multiplexer problem.

(c) for the 20-bit multiplexer problem.

(d) for the 37-bit multiplexer problem.

Fig. 4 Results of multiplexer problems

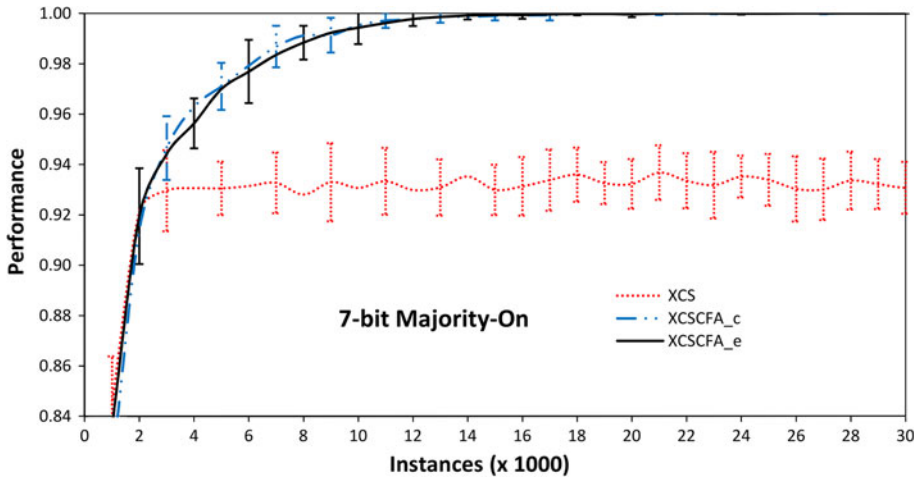


Fig. 5 Results of the 7-bit majority-on problem

performance level, but could not completely solve the 7-bit count ones problem, whereas the XCSCFA methods solved it using approximately 50,000 training examples.

5.4 The design verification problem domain

The performance of standard XCS and the XCSCFA methods in learning the DV1 problem is shown in Fig. 7. The number of classifiers used is $N = 3{,}000$. It is observed that standard XCS reached approximately 97 % performance level, but could not completely solve the DV1 problem, whereas the XCSCFA methods successfully solved it. Ioannides et al. [21] improved the performance of XCS in learning the DV1 problem to 99.76 %, by modifying the standard fitness update procedure and using an individually computed learning rate for each classifier[5], but could not completely solve it.

[5] They also used a larger value for action set subsumption threshold, i.e. 100 instead of the commonly used value of 20.

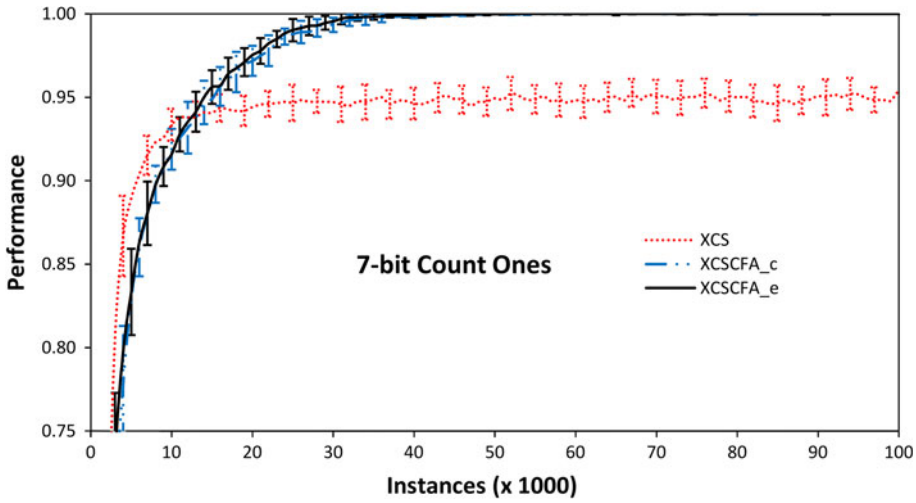


Fig. 6 Results of the 7-bit count ones problem. The performance curves using the XCSCFA methods are coincident

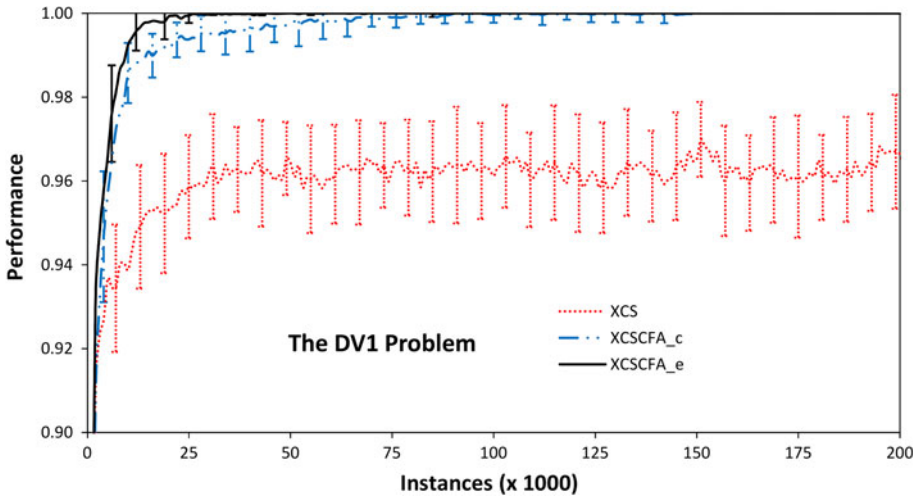


Fig. 7 Results of the DV1 problem

5.5 The carry problem domain

The carry problem used in this work is the 4+4 bit carry problem. The number of classifiers used is $N = 4{,}000$. The performance of standard XCS and the XCSCFA methods is shown in Fig. 8. It is observed that standard XCS reached approximately 93 % performance level, but could not completely solve the 4+4 bit carry problem, whereas the XCSCFA methods successfully solved it.

5.6 The even-parity problem domain

The performance of standard XCS and the XCSCFA methods in learning the 7-bit even-parity problem is shown in Fig. 9. The number of classifiers used is $N = 2{,}000$. It is observed that standard XCS and XCSCFA$_c$ could not learn the 7-bit even-parity problem, whereas XCSCFA$_e$ successfully solved it.

The even-parity problem domain does not allow generalizations if the standard ternary alphabet based encoding scheme is used with static numeric action. So each bit must be specific for a rule to be accurate, requiring 2^8 such rules for the 7-bit even-parity problem. The standard XCS and XCSCFA$_c$ methods were not able to evolve enough accurate rules using the experimental setup given in Sect. 4.2 and 2000 classifier rules.[6] However, the XCSCFA$_e$ method solved the 7-bit even-parity problem by producing generalized classifier rules, see Sect. 6.3

6 Analysis of evolved classifiers

In this section, the evolved rules in different problem domains are analyzed. It is to be noted that from the

[6] These methods need larger population size to solve even-parity problems.

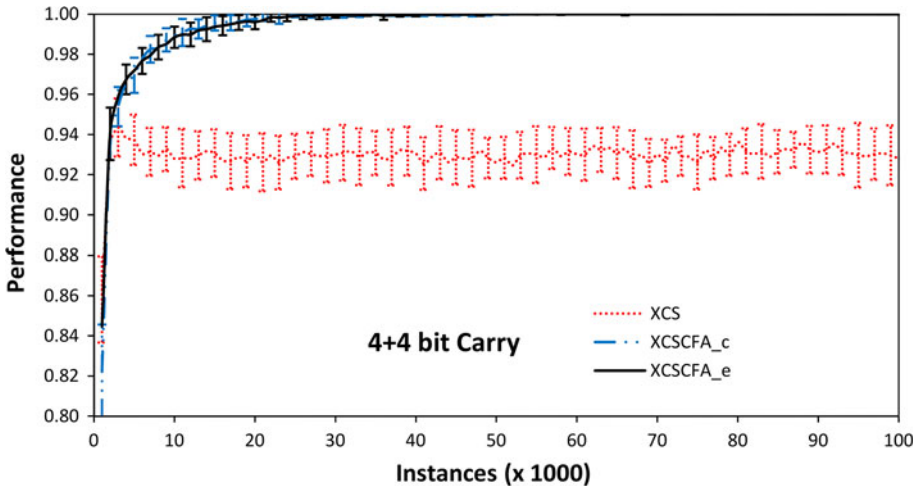


Fig. 8 Results of the 4+4 bit carry problem. The performance curves using the XCSCFA methods are coincident

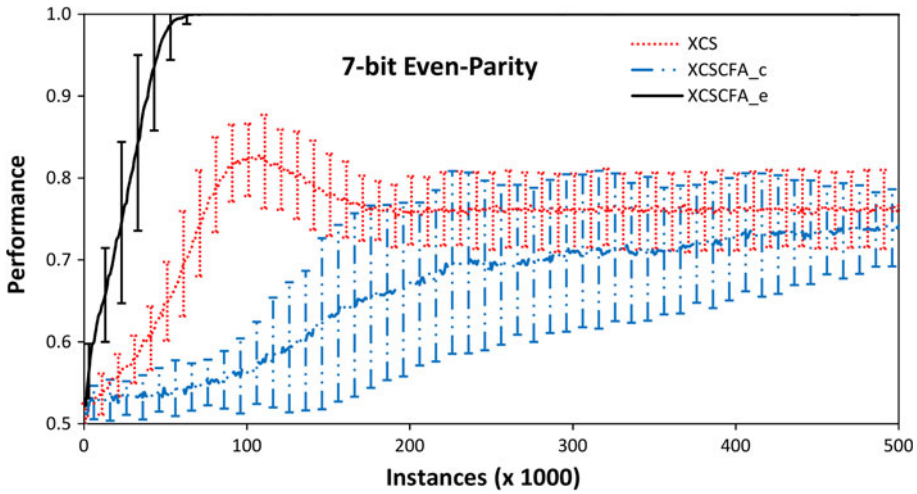


Fig. 9 Results of the 7-bit even-parity problem

overlapping and niche imbalance problems only the 7-bit majority-on problem is chosen for rules analysis to save space, and other problems follow a similar pattern.

6.1 The multiplexer problem domain

Every optimal classifier rule in a multiplexer problem, encoded in ternary representation, contains $k + 1$ specific bits in the condition (i.e. k address bits and the corresponding data bit), see Sect. 4.1, e.g. '10##1# : 1' is an optimal rule for the 6-bit MUX. The solutions obtained using XCS contain both optimal and sub-optimal classifier rules, which have some form of grouping, but often no distinct separation of optimal and sub-optimal classifiers. To obtain the desired optimum rule set in binary action based XCS, extensive processing is needed [32], e.g. condensation or compaction algorithm. The XCSCFA$_c$ system has the advantage of producing the optimal classifiers separated from the sub-optimal ones with respect to the numerosity values. When a very simple numerosity-based condensation mechanism, introduced in our previous work [26], was applied to the final solutions obtained for the 6-, 11-, 20-, and 37-bit multiplexer problems, XCSCFA$_c$ consistently produced the optimum rule sets containing 8, 16, 32, and 64 classifiers respectively, for all 30 runs of each experiment. The standard XCS approach failed one, four, 13, and 25 times to produce the optimum rule sets for the 6-, 11-, 20-, and 37-bit MUX problems respectively.

The XCSCFA$_e$ method produced more general classifiers than the other two methods containing only k specific bits in conditions, e.g. '11#### : D5' is a generated rule for the 6-bit MUX. These may appear overgenerals as they have more '#' symbols than the optimal ternary representation classifiers. However, they are perfectly accurate for all the matching environmental instances due to the referencing of specific bits in the code-fragment action. For example, the classifier '11#### : D5' does not contain any specific data bit in the condition, but the addressed data bit is in the action, i.e. D5. Now, the action value of this classifier is 1 if the environmental bit D5 is 1 and 0 otherwise. Therefore, it is correct for all the matching

instances. It is to be noted that the optimal classifiers produced in XCSCFA$_e$ were not clearly separated from the sub-optimal ones; unlike XCSCFA$_c$.

6.2 The majority-on problem domain

We believe that majority-on problems are hard to learn because the complete solution for a majority-on problem consists of overlapping classifiers. It is observed that standard XCS produced overgeneral classifiers in the 7-bit majority-on problem, see Table 4. For example, the 10$^{\text{th}}$ classifier rule '1###1#1 : 1' is an overgeneral classifier which matches 16 environmental instances. It is a "dangerous" classifier because it is correct for all the matching instances except '1000101', so likely to be considered accurate ($\epsilon < \epsilon_0$) and therefore, it is highly likely that it will subsume the specific classifier '1000101 : 1' in the training process, resulting in decreased chances for production of the desired classifier '1000101 : 0'.

It is observed that almost all the classifiers in the final solutions obtained using XCS for the 7-bit majority-on problem were overgeneral. This indicates that the overgeneral classifiers having $\epsilon < \epsilon_0$, such as the 2^{nd}, 10^{th} and 15^{th} classifiers in Table 4, would have subsumed the otherwise needed accurate classifiers, resulting in the poor performance of the system.

The XCSCFA systems managed to avoid overgeneral classifier rules in the final solutions, see Table 5 and Table 6. The XCSCFA$_e$ method produced more general (and still accurate) classifiers than the other two methods, similar to the multiplexer domain. For example, the 15$^{\text{th}}$ classifier in Table 6 '0##1#11 : D4 D1 | D2 D1 | |' matches eight environmental instances. Now, the action value of this classifier is 1 if any of the environmental bits D1, D2, or D4 is 1. Therefore, it is correct for all the matching instances. It is to be noted that the classifier number eight in Table 6 is an erroneous classifier, therefore, it has relatively low fitness value. The reason for it being in the final solution is that it is newly created (cf. low experience value), otherwise it would have been deleted in the training process.

6.3 The even-parity problem domain

In standard XCS with numeric action, it is not possible for a classifier rule to have a '#' symbol in the condition and still be accurate for an even parity problem. However, it is interesting to note that XCSCFA$_e$ was able to produce accurate general classifier rules having a '#' symbol in the condition for the 7-bit even parity problem. For example, the classifier rule '111000# : D6' matches two environmental instances, '1110000' and '1110001'; and the action value is the binary value of the last symbol in the environmental instance, i.e. D6. Therefore the advocated action for the problem instance '1110000' is 0 and for '1110001' is 1, and both are accurate.

As a result of the generalized classifiers in XCSCFA$_e$, the number of classifiers required in the final solution set reduced to half of the number of specific classifiers needed otherwise and it successfully solved the 7-bit even parity problem. However, standard XCS as well as XCSCFA$_c$ could not produce accurate general classifier rules in the 7-bit even parity problem, and failed to solve it

Table 4 A sample of classifiers from a final solution obtained using XCS in learning the 7-bit majority-on problem

No.	Condition	Action	n	F	ϵ	p	exp
1	####11#	0	83	0.01	166.85	82.28	194
2	#0#0##0	1	73	0.51	3.71	0.45	1508
3	##00###	0	67	0.06	139.84	964.81	360
4	0###00#	1	61	0.21	73.40	18.67	830
5	####111	1	58	0.10	238.67	792.05	921
6	#1##11#	0	57	0.01	16.67	2.60	974
7	#0#0#0#	1	55	0.08	284.91	233.67	537
8	##0#00#	1	54	0.21	168.94	82.68	216
9	##0##0#	0	53	0.06	470.90	681.82	201
10	1###1#1	1	50	0.44	6.82	999.14	499
11	#111###	0	48	0.09	301.47	203.09	504
12	###00##	0	47	0.01	296.17	893.83	447
13	##1##11	0	47	0.00	89.27	27.08	392
14	0##0#0#	1	47	0.16	183.70	72.53	598
15	#00###0	1	47	0.56	2.03	0.23	1019

Table 5 A sample of classifiers from a final solution obtained using XCSCFA$_c$ in learning the 7-bit majority-on problem

No.	Condition	Code-Fragment Action	n	F	ϵ	p	exp
1	110#000	D0	12	0.52	0	0	201
2	110#000	D0 D2 &	11	0.37	0	1,000	69
3	1001011	D5 D5 D6 & |	10	0.39	0	1,000	36
4	0101010	D0 D3 & D0 D3 & r	10	0.46	0	0	59
5	##00001	D6 D3 r	9	0.25	0	1,000	74
6	1#10011	D6 D5 d D5 r	9	0.34	0	0	104
7	0110110	D4 D2 d D6 D0 & |	9	0.39	0	0	61
8	0010011	D2 D5 & D6 &	9	0.46	0	0	399
9	11#1#10	D6 D6 r	8	0.12	0	1,000	41
10	1#11##1	D1 D2 r	8	0.33	0	0	157
11	111#001	D2 ~ D2 ~ |	8	0.34	0	0	57
12	0011110	D1 D6 | D1 D6 | |	8	0.29	0	0	45
13	0011011	D5 ~	8	0.33	0	0	106
14	1100#11	D2 D3 | D5 &	8	0.32	0	0	258
15	0#11000	D3 D2 d D3 &	8	0.26	0	1,000	238

Table 6 A sample of classifiers from a final solution obtained using XCSCFA$_e$ in learning the 7-bit majority-on problem

No.	Condition	Code-Fragment Action	n	F	ϵ	p	exp
1	0##010#	D1 D6 & D2 D1 & &	13	0.21	0	1,000	3103
2	1##0011	D1 D2 | D1 D2 | &	13	0.26	0	1,000	1594
3	0##00#1	D2 D1 & D5 D1 & &	11	0.20	0	1,000	570
4	01##0#0	D2 D3 D5 & d	11	0.17	0	0	1380
5	#1011#0	D5 D0 r D5 D0 r d	11	0.19	0	1,000	1787
6	11#00#0	D2 D5 & D2 D5 & &	11	0.20	0	1,000	2096
7	1#1#001	D1 D3 r D1 D3 r |	11	0.24	0	0	4077
8	1##0#11	D2 D1 | D2 D1 | &	10	0.04	306.69	806.46	47
9	##011#1	D5 D5 & D1 D0 | |	10	0.19	0	1,000	1209
10	1#110#0	D1 D5 | D2 D5 | d	10	0.18	0	0	655
11	001#00#	D2 D2 & D2 D2 & &	10	0.16	0	0	653
12	11#00#0	D2 D2 D5 & &	10	0.15	0	1,000	947
13	##01101	D3 D0 D1 | d	10	0.19	0	0	1375
14	0##010#	D2 D6 & D2 D1 & &	9	0.14	0	1,000	673
15	0##1#11	D4 D1 | D2 D1 | |	9	0.16	0	1,000	672

with this setup. It is to be noted that there is just one '#' symbol in the condition of a general classifier produced in XCSCFA$_e$ for the 7-bit even-parity problem. In XCSCFA$_c$ the classifiers having just one '#' symbol cannot be consistently accurate in the even-parity domain because the '#' symbol is randomly treated 0 or 1 in XCSCFA$_c$ to compute the action value of a classifier.

7 XCS's bias against overlapping rules

It has been reported that XCS is biased against the overlapping classifiers in a rule population [12, 22, 33]. The solutions obtained using XCS for the overlapping and niche imbalance problems experimented here consist of overgeneral classifiers, e.g. see Table 4. Some of them are only incorrect for very few instances, therefore, it is anticipated that they would have subsumed the otherwise needed accurate classifiers, resulting in poor performance of the system. Therefore, two more sets of experiments were conducted using XCS: (1) deactivating the action set subsumption, and (2) tuning the parameters, i.e. increasing the classifier experience threshold for subsumption θ_{sub} from 20 to 200 and decreasing the prediction error threshold ϵ_0 from 10 to 0.01. The performance of XCS improved in both of these experimental setups and it successfully solved the DV1, the 7-bit count ones, and the 4+4 bit carry problems, see Fig. 10. This is consistent with previous findings as Butz et al. [11] have solved count ones problems by using an error-based fitness approach in XCS and deactivating the action set subsumption.

It is observed that XCS with tuned parameters setting required approximately 400,000 problem instances to learn the DV1 problem whereas XCS without action set subsumption needed approximately 16,00,000 problem instances to reach the 100 % performance level for the DV1 problem, see Fig. 10c. In the 7-bit majority-on problem, the performance improved to more than 99 %, but did not reach the stabilized 100 % performance level, as shown in Fig. 10a.

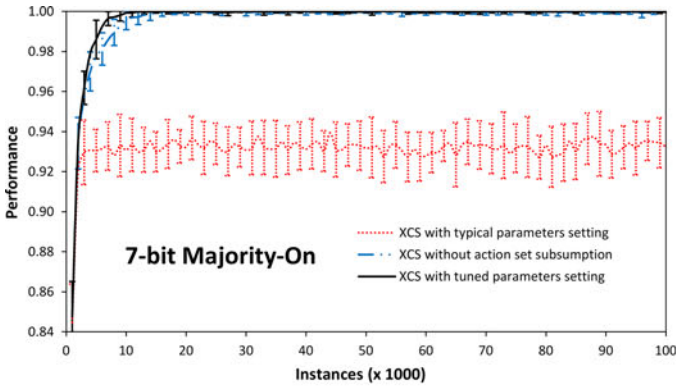
It is to be noted that usually ϵ_0 is set to 10 to handle noise in the environment and θ_{sub} is set to 20 to compact the population using the subsumption deletion. By tuning ϵ_0 and θ_{sub} for overlapping and niche imbalance problems, the classification performance is improved, but it is anticipated to compromise the robustness to noise and the compactness of the population.

It is observed that in the multiplexer problem domain, parameter tuning in this way degraded the performance of XCS in terms of the required instances to reach the optimum performance level, e.g. see Fig. 11. The reason for this degradation is the unnecessarily large value of θ_{sub} and very small value of ϵ_0 for the non-overlapping and niche balanced multiplexer domain.

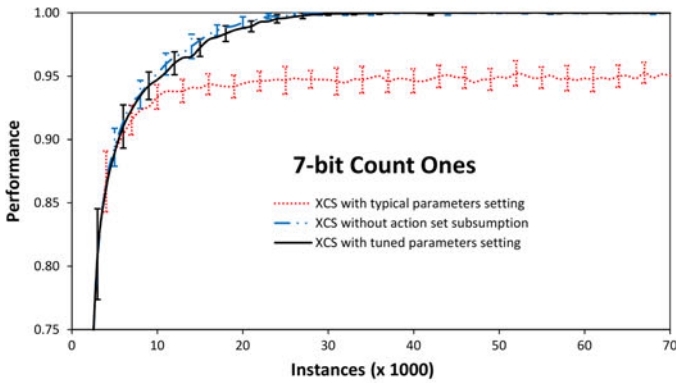
8 Action inconsistency and redundancy in XCSCFA

XCS keeps a complete map, i.e. the classifiers advocating consistently correct classification (and hence predicting an accurate, maximum environmental reward of say 1000) as well as the classifiers advocating consistently incorrect classification (and hence predicting an accurate, minimum

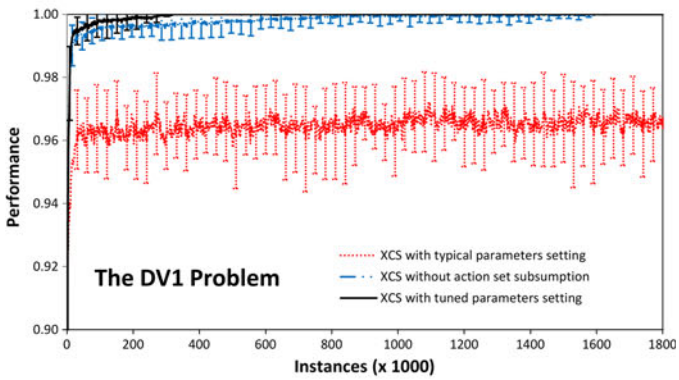
Fig. 10 Results of overlapping and niche imbalance problems using XCS



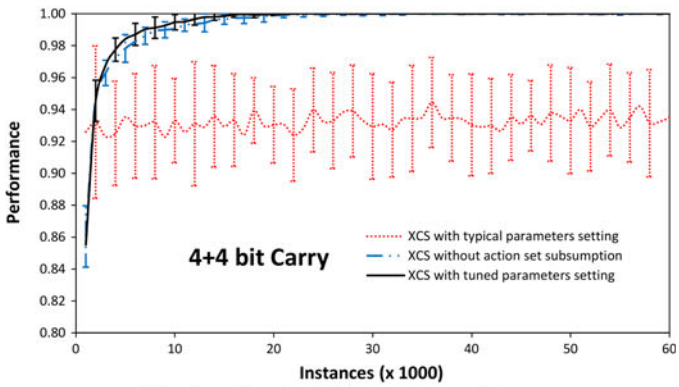
(a) for the 7-bit majority-on problem.



(b) for the 7-bit count ones problem.



(c) for the DV1 problem.



(d) for the 4+4 bit carry problem.

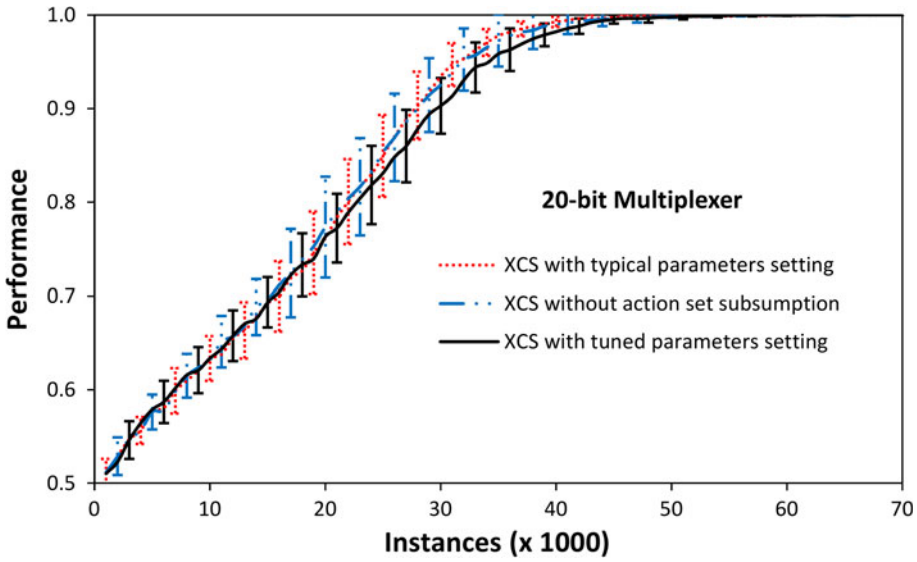


Fig. 11 Result of the 20-bit multiplexer problem using XCS

environmental reward of say 0). It is noted that the building blocks of information in the condition of an accurate incorrect classifier are exactly the same as in the counterpart correct classifier. For example, in case of the 6-bit MUX '000### : 1 → 0' is an accurate incorrect classifier which has the same condition as that in the counterpart accurate correct classifier '000### : 0 → 1000'. The rule discovery operation is applied in the action set, which is formed by the classifiers advocating a *certain* action, commonly selected at random, and covering the currently observed environmental input. As all the classifiers in an action set advocate the same action, the correct and incorrect classifiers cannot occur in the same action set, so cannot be simultaneously used in the breeding of new classifiers. Therefore, in an XCS-based system, although both correct and incorrect classifiers are kept throughout the learning of the system, the building blocks of information in them are not efficiently exploited as they are not allowed to take part in the same breeding operation. In the XCSCFA approach, due to inconsistent action values, the incorrect classifiers can occur in the same action set as correct classifiers so can be used for the production of good classifiers.

Further, the multiple genotypes to a single phenotype mapping of the GP-like code fragments in the XCSCFA approach provides redundancy and diversity [2, 43], which may increase the robustness of the system, especially in learning overlapping and niche imbalance problems. In order to understand the effect of redundancy in the XCSCFA approach, consider the 7-bit majority-on problem and the hypothetical classifiers shown in Table 7. Here 'V' is the computed action value of a classifier rule.

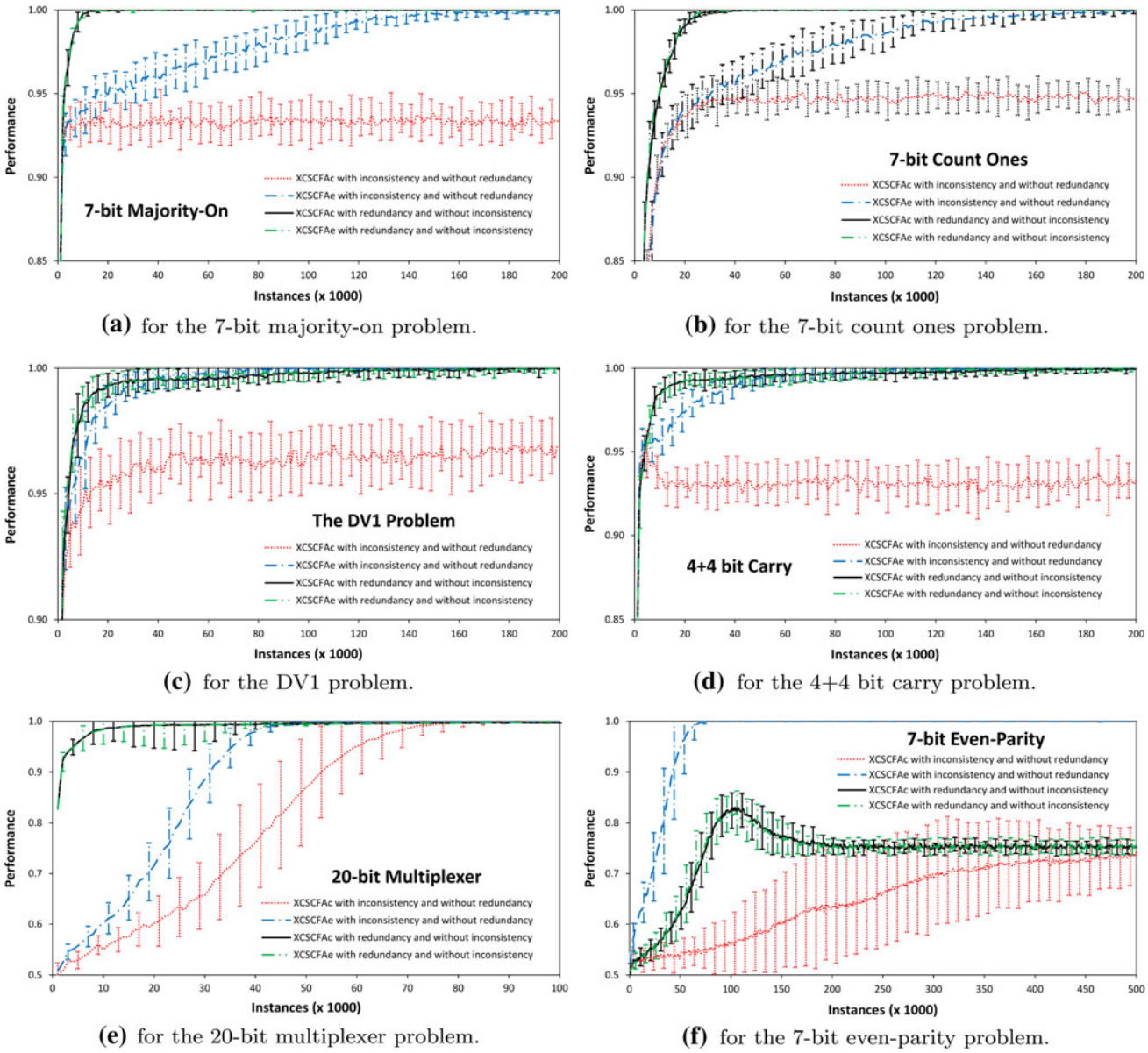
Classifier 1 is more general than classifier 2 and classifier 3, where the latter two classifiers are completely accurate

Table 7 Effect of redundancy in the XCSCFA approach

No.	Condition	Action	V	*p*	ε
1	1###1#1	D4 D6 &	1	999.14	6.82
2	1000101	D4 D6 &	1	0	0
3	1000101	D3 D5 D3 & d	1	0	0

(although incorrect), but the former is incorrect once out of the 16 matched input messages. Classifier 1 and classifier 2 have the same genotypic code-fragment action, therefore, the former can subsume the latter, resulting in reduced chances of production of the required correct classifier '1000101 : 0', similar to standard XCS as discussed in Sect. 6.2 However, classifier 3 cannot be subsumed by classifier 1 because of having different genotypic code-fragment actions. This redundant classifier can be used in the rule discovery operation to produce the required correct classifier '1000101 : 0'. In XCSCFA there can be more than one genotypic classifier equivalent to the classifier '1000101 : 0' in XCS, e.g. '1000101 : D4 D2 | D2 D2 | &' and '1000101 : D6 D0 D2 d d', which further increase the chances of producing the phenotypic classifier '1000101 : 0' in XCSCFA. The XCSCFA$_e$ approach, in addition to this redundancy, has ability to incorporate generalization in actions making it more robust than XCS and XCSCFA$_c$.

It is considered that the XCSCFA systems outperformed standard XCS, using typical experimental setup, due to the following two properties of code-fragment based actions: (1) the ability in all accuracy-based systems to keep a complete map combined with inconsistent actions in the XCSCFA approach preserved important building blocks of information, and (2) the multiple genotypes to a single phenotype mapping of the GP-like code fragments



(a) for the 7-bit majority-on problem. **(b)** for the 7-bit count ones problem.

(c) for the DV1 problem. **(d)** for the 4+4 bit carry problem.

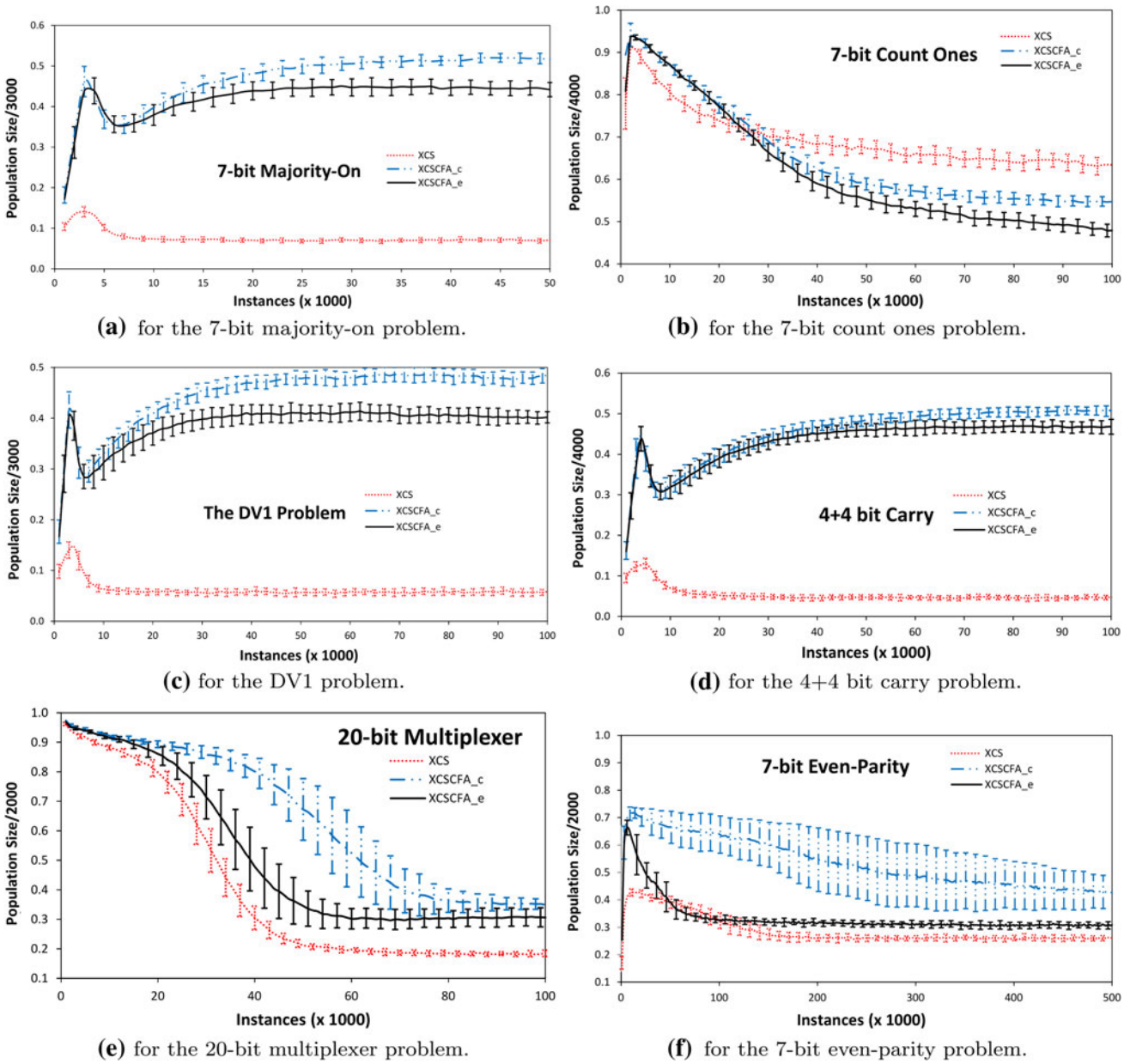
(e) for the 20-bit multiplexer problem. **(f)** for the 7-bit even-parity problem.

Fig. 12 Effect of action inconsistency and redundancy in the XCSCFA approach. The performance curves are coincident for both XCSCFA$_c$ and XCSCFA$_e$ on disallowing the inconsistency while retaining the redundancy

provided redundancy and diversity. To verify this, we repeated all the experiments by disallowing the inconsistent and redundant actions. The obtained results were similar to the results obtained using XCS with the typical experimental setup. Then, to investigate the effect of each property separately, we conducted two more sets of experiments: (1) allowing inconsistency but no redundancy, and (2) allowing redundancy but no inconsistency. The obtained results are shown in Fig. 12.

It is observed that by disallowing the redundancy while retaining the inconsistency, the performance of XCSCFA$_c$ degraded and it failed in learning all the problems except multiplexer problems. The XCSCFA$_e$ method still solved all the problems because of its ability to incorporate generalization in the actions of classifier rules, but in learning the overlapping and niche imbalance problems it needed relatively large number of problem instances. On the other hand, by disallowing the inconsistency while retaining the redundancy, both XCSCFA$_c$ and XCSCFA$_e$ behaved similarly and successfully solved the overlapping and niche imbalance problems, again more slowly, but failed in learning the 7-bit even-parity problem.

It is to be noted that due to the multiple genotypes to a single phenotype mapping of code-fragment actions in the XCSCFA systems, the subsumption deletion mechanism is not fully enabled and there are multiple classifiers in the XCSCFA systems for a corresponding single classifier in XCS. Therefore, the XCSCFA systems have produced larger



(a) for the 7-bit majority-on problem.

(b) for the 7-bit count ones problem.

(c) for the DV1 problem.

(d) for the 4+4 bit carry problem.

(e) for the 20-bit multiplexer problem.

(f) for the 7-bit even-parity problem.

Fig. 13 Population size in terms of macro-classifiers in learning different problems using XCS and the XCSCFA systems

macro-classifier populations than XCS, except in learning the 7-bit count ones problem, as shown in Fig. 13. The percentage of the number of macro-classifiers in the final solutions evolved using XCS and the XCSCFA systems are shown in Table 8, where entries in bold denotes more compact solution than others. The last column in Table 8 shows the number of macro-classifiers in XCSCFA$_c$ after removing the classifiers that have inconsistent action values and then treating two classifiers equivalent if they have the same condition and the same action value. It can be seen that this simplification of the solutions obtained using XCSCFA$_c$ significantly reduced the population size, which is smaller than standard XCS in most of the problems.

9 Conclusions

The main goal of the work presented here was to investigate the standard XCS and code-fragment based XCSCFA systems in learning different complex Boolean problems, especially overlapping and niche imbalance problems. The obtained experimental results indicate that XCS has strong bias against the overlapping rules and it produced overgeneral classifiers for overlapping and niche imbalance problems. To learn such problems, it is beneficial to either deactivate the *action set subsumption* or use a relatively high subsumption threshold and a small error threshold so that the less general, but

Table 8 The percentage of the number of macro-classifiers, along with standard deviation, in the final solutions obtained using XCS and the XCSCFA systems for different problems

Problem	XCS	$XCSCFA_c$	$XCSCFA_e$	Simplified $XCSCFA_c$
6-bit MUX	9 ± 1	35 ± 2	21 ± 2	$\mathbf{7 \pm 1}$
11-bit MUX	13 ± 1	38 ± 2	25 ± 1	$\mathbf{10 \pm 1}$
20-bit MUX	19 ± 1	40 ± 1	29 ± 1	$\mathbf{16 \pm 1}$
37-bit MUX	22 ± 1	43 ± 1	37 ± 1	$\mathbf{21 \pm 1}$
7-bit majority-on	$\mathbf{7 \pm 1}$	50 ± 1	37 ± 1	25 ± 1
7-bit count ones	62 ± 2	50 ± 1	37 ± 1	$\mathbf{29 \pm 1}$
DV1	$\mathbf{6 \pm 1}$	45 ± 1	35 ± 1	15 ± 1
4+4 bit carry	$\mathbf{5 \pm 0}$	49 ± 1	39 ± 1	20 ± 1
7-bit even-parity	26 ± 1	41 ± 1	30 ± 1	$\mathbf{20 \pm 1}$

necessary classifiers covering small niches are retained in the population.

The XCSCFA systems successfully solved overlapping and niche imbalance problems because of the inconsistent actions and especially the redundancy provided by the GP-like code-fragment actions. $XCSCFA_e$ produced more generalized classifiers, but $XCSCFA_c$ has the advantage of producing the optimal classifiers separated from the sub-optimal ones in certain domains. The evolved classifiers using $XCSCFA_c$ are easy to interpret, as are in standard XCS, whereas in $XCSCFA_e$ they are slightly harder to interpret as their actions now depend on the environmental state.

In future, the XCSCFA approach will be tested in learning the k-Disjunctive Normal Form (k-DNF) family of Boolean functions [17] and the results will be compared with other machine learning systems. It would also be interesting to extend and test the XCSCFA approach for multi-step problems and the problems involving more than two actions.

References

1. Ahluwalia M, Bull L (1999) A genetic programming based classifier System. In proceedings of the genetic and evolutionary computation conference, pp 11–18
2. Alfaro-Cid E, Merelo JJ, de Vega FF, Esparcia-Alcázar AI, Sharman K (2010) Bloat control operators and diversity in genetic programming: a comparative study. Evol Comput 18(2):305–332
3. Behdad M, Barone L, French T, Bennamoun M (2012) On XCSR for electronic fraud detection. Evol Intell 5(2):139–150
4. Behdad M, French T, Barone L, Bennamoun M (2012) On principal component analysis for high-dimensional XCSR. Evol Intell 5(2):129–138
5. Bernadó-Mansilla E, Garrell-Guiu JM (2003) Accuracy-based learning classifier systems: models, analysis and applications to classification tasks. Evol Comput 11(3):209–238
6. Brameier M, Banzhaf W (2007) Linear genetic programming. Springer, New York
7. Bull L (2004) Applications of learning classifier systems. Springer, New York
8. Bull L, Kovacs T (2005) Foundations of learning classifier systems: an introduction. Springer, New York
9. Bull L, O'Hara T (2002) Accuracy-based neuro and neuro-fuzzy classifier systems. In proceedings of the genetic and evolutionary computation conference, pp 905–911
10. Butz MV (2006) Rule-based evolutionary online learning systems: a principal approach to LCS analysis and design. Springer, New York
11. Butz MV, Goldberg DE, Lanzi PL (2007) Effect of pure error-based fitness in XCS. In learning classifier systems, vol 4399. Springer, New York, pp 104–114
12. Butz MV, Goldberg DE, Tharakunnel K (2003) Analysis and improvement of fitness exploitation in XCS: bounding models, tournament selection, and bilateral accuracy. Evol Comput 11(3):239–277
13. Butz MV, Kovacs T, Lanzi PL, Wilson SW (2004) Toward a theory of generalization and learning in XCS. IEEE Trans Evol Comput 8(1):28–46
14. Butz MV, Wilson SW (2002) An algorithmic description of XCS. Soft Comput 6(3-4):144–153
15. Dam HH, Abbass HA, Lokan C, Yao X (2008) Neural-based learning classifier systems. IEEE Trans Knowl Data Eng 20(1):26–39
16. Dandamudi SP (2003) Fundamentals of computer organization and design. Springer, New York
17. Franco MA, Krasnogor N, Bacardit J (2012) Analysing BioHEL using challenging Boolean functions. Evol Intell 5(2):87–102
18. Goldberg DE, Korb B, Deb K (1989) Messy genetic algorithms: motivation, analysis, and first results. Complex Syst 3(5):493–530
19. Holland JH, Booker LB, Colombetti M, Dorigo M, Goldberg DE, Forrest S, Riolo RL, Smith RE, Lanzi PL, Stolzmann W, Wilson SW (2000) What is a learning classifier system? In learning classifier systems, from foundations to applications. Springer, New York, pp 3–32
20. Hurst J, Bull L (2006) A neural learning classifier system with self-adaptive constructivism for mobile robot control. Artif Life 12(3):353–380
21. Ioannides C, Barrett G, Eder K (2011) Improving XCS performance on overlapping binary problems. In proceedings of the IEEE congress on evolutionary computation, pp 1420–1427
22. Ioannides C, Barrett G, Eder K (2011) XCS Cannot learn all Boolean functions. In proceedings of the genetic and evolutionary computation conference, pp 1283–1290
23. Iqbal M, Browne WN, Zhang M (2012) Extracting and using building blocks of knowledge in learning classifier systems. In proceedings of the genetic and evolutionary computation conference, pp 863–870
24. Iqbal M, Browne WN, Zhang M (2012) XCSR with computed continuous Action. In proceedings of the Australasian joint conference on artificial intelligence, pp 350–361
25. Iqbal M, Browne WN, Zhang M (2013) Comparison of two methods for computing action values in XCS with code-fragment actions. In proceedings of the genetic and evolutionary computation conference (companion), pp 1235–1242
26. Iqbal M, Browne WN, Zhang M (2013) Evolving optimum populations with XCS classifier systems. Soft Comput 17(3):503–518
27. Iqbal M, Browne WN, Zhang M (2013) Learning overlapping natured and niche imbalance Boolean problems using XCS classifier systems. In proceedings of the IEEE congress on evolutionary computation, pp 825
28. Iqbal M, Browne WN, Zhang M (2013) Using building blocks of extracted knowledge to solve complex, large-scale Boolean

problems. IEEE transactions on evolutionary computation. (to appear)
29. Iqbal M, Zhang M, Browne WN (2011) Automatically defined functions for learning classifier systems. In proceedings of the genetic and evolutionary computation conference (companion), pp 375–382
30. De Jong KA (2006) Evolutionary computation: a unified approach. The MIT Press, Cambridge
31. Kinzett D, Johnston M, Zhang M (2009) Numerical simplification for bloat control and analysis of building blocks in genetic programming. Evol Intell 2(4):151–168
32. Kovacs T (1996) Evolving optimal populations with XCS classifier systems. Technical report CSR-96-17 and CSRP-9617, University of Birmingham, UK
33. Kovacs T (2002) What should a classifier system learn and how should we measure it? Soft Comput 6(3–4):171–182
34. Lanzi PL (1999) Extending the representation of classifier conditions part I: from binary to messy coding. In proceedings of the genetic and evolutionary computation conference, pp 337–344
35. Lanzi PL (2003) XCS with stack-based genetic programming. In proceedings of the IEEE congress on evolutionary computation, pp 1186–1191
36. Lanzi PL (2008) Learning classifier systems: then and now. Evol Intell 1(1):63–82
37. Lanzi PL, Loiacono D (2007) Classifier systems that compute action mappings. In proceedings of the genetic and evolutionary computation conference, pp 1822–1829
38. Lanzi PL, Loiacono D, Wilson SW, Goldberg DE (2005) XCS with computed prediction for the learning of Boolean functions. Technical Report 2005007, Illinois Genetic Algorithms Laboratory
39. Lanzi PL, Loiacono D, Wilson SW, Goldberg DE (2007) Generalization in the XCSF classifier system: analysis, improvement, and extension. Evol Comput 15(2):133–168
40. Lanzi PL, Perrucci A (1999) Extending the representation of classifier conditions part II: from messy coding to S-expressions. In proceedings of the genetic and evolutionary computation conference, pp 345–352
41. Loiacono D, Marelli A, Lanzi PL (2007) Support vector machines for computing action mappings in learning classifier systems. In proceedings of the IEEE congress on evolutionary computation, pp 2141–2148
42. Luke S, Panait L (2006) A comparison of bloat control methods for genetic programming. Evol Comput 14(3):309–344
43. Miller JF, Smith SL (2006) Redundancy and computational efficiency in cartesian genetic programming. IEEE Trans Evol Comput 10(2):167–174
44. Miller JF, Thomson P (2000) Cartesian genetic programming. In proceedings of the European conference on genetic programming, pp 121–132
45. Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk. (With contributions by Koza JR)
46. Shafi K, Kovacs T, Abbass HA, Zhu W (2009) Intrusion detection with evolutionary learning classifier systems. Nat Comput 8(1):3–27
47. Stalph PO, Rubinsztajn J, Sigaud O, Butz MV (2012) Function approximation with LWPR and XCSF: a comparative study. Evol Intell 5(2):103–116
48. Urbanowicz RJ, Moore JH (2009) Learning classifier systems: a complete introduction, review, and roadmap. J Artif Evol Appl 2009(1):1–25
49. Vijayakumar S, Schaal S (2000) Locally weighted projection regression: an o(n) algorithm for incremental real time learning in high dimensional space. In proceedings of the international conference on machine learning, pp 1079–1086
50. Wilson SW (1994) ZCS: A zeroth level classifier system. Evol Comput 2(1):1–18
51. Wilson SW (1995) Classifier fitness based on accuracy. Evol Comput (2):149–175
52. Wilson SW (1998) Generalization in the XCS classifier system. In proceedings of the genetic programming conference, pp 65–674
53. Wilson SW (2000) Get real! XCS with continuous-valued inputs. In learning classifier systems. Springer, New York, pp 209–219
54. Wilson SW (2000) Mining oblique data with XCS. In proceedings of the genetic and evolutionary computation conference (companion), pp 158–174
55. Wilson SW (2002) Classifiers that approximate functions. Nat Comput 1:211–233
56. Wilson SW (2008) Classifier conditions using gene expression programming. In learning classifier systems, Springer, New York, pp 206–217
57. Wong P, Zhang M (2006) Algebraic simplification of GP programs during evolution. In proceedings of the genetic and evolutionary computation conference, pp 927–934