Reusing Building Blocks of Extracted Knowledge to Solve Complex, Large-Scale Boolean Problems

Muhammad Iqbal, *Member, IEEE,* Will N. Browne, and Mengjie Zhang, *Senior Member, IEEE*

***Abstract*—Evolutionary computation techniques have had limited capabilities in solving large-scale problems due to the large search space demanding large memory and much longer training times. In the work presented here, a genetic programming like rich encoding scheme has been constructed to identify building blocks of knowledge in a learning classifier system. The fitter building blocks from the learning system trained against smaller problems have been utilized in a higher complexity problem in the domain in order to achieve scalable learning. The proposed system has been examined and evaluated on four different Boolean problem domains, i.e. multiplexer, majority-on, carry, and even-parity problems. The major contribution of this work is to successfully extract useful building blocks from smaller problems and reuse them to learn more complex, large-scale problems in the domain, e.g. 135-bits multiplexer problem, where the number of possible instances is $2^{135} \approx 4 \times 10^{40}$, is solved by reusing the extracted knowledge from the learnt lower level solutions in the domain. Autonomous scaling is, for the first time, shown to be possible in learning classifier systems. It improves effectiveness and reduces the number of training instances required in large problems, but requires more time due to its sequential build-up of knowledge.**

***Index Terms*—Learning Classifier Systems, Genetic Programming, Layered Learning, Scalability, Building Blocks, Code Fragments.**

I. Introduction

HUMAN beings have the ability to apply the domain knowledge learned from a smaller problem to more complex problems of the same or a related domain, but currently the vast majority of evolutionary computation techniques lack this ability. This lack of ability to apply the already learned knowledge of a domain results in consuming more resources and time to solve the more complex problems of the domain. As the problem increases in size, it becomes difficult and even sometimes impractical (if not impossible) to solve due to the needed resources and time. Therefore a system is needed that has the ability to reuse the learned knowledge of a problem domain in order to scale in the domain [1].

The main goal of the research direction is to develop a system capable of autonomous, scalable learning, from small problems to more complex problems of the same or a related domain, in a similar behavior to human beings. In order to autonomously scale in a problem domain, reusable building blocks of knowledge must be extracted. To extract and reuse building blocks of information in a problem domain, a rich encoding is needed, but the search space could then bloat, e.g. as in some forms of genetic programming (GP). Learning classifier systems (LCSs) are a well structured evolutionary computation based learning technique that have pressures to implicitly avoid bloat, such as fitness sharing through niche based reproduction [2].

Typically, 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. The rules are of the form "if *condition* then *action*". Commonly, the condition is represented by a fixed length bitstring defined over the ternary alphabet $\{0, 1, \#\}$, where '#' is the 'don't care' symbol that can be either 0 or 1, and the action is represented by a numeric constant. The LCS technique can scale in problem domains, but has to relearn from the start each time. Further, increased dimensionality of the problem, resulting in increased search space, demands large memory space and leads to much longer training times, and eventually restricts LCS to a limit in problem size. By explicitly feeding the domain knowledge to an LCS, scalability can be achieved but it adds bias and restricts use in multiple domains [3].

In the work presented here, the typically used ternary alphabet based conditions in an LCS will be replaced by code-fragment based conditions, in order to extract and reuse building blocks of knowledge. A code fragment is a tree-expression similar to a tree generated in GP (see Section II-B2). The fitter building blocks extracted from the learning system trained against smaller problems will be utilized in learning more complex, large-scale problems in the same domain, similar to transferring knowledge in a transfer learning technique (see Section II-A), in an attempt to develop a scalable classifier system.

The proposed system will be tested on four different Boolean problem domains, i.e. multiplexer, majority-on, carry, and even-parity problems. The multiplexer domain is a multimodal and epistatic problem domain. In the majority-on problem domain, the complete solution consists of strongly overlapping classifiers, which is therefore difficult to learn. Similar to the majority-on problems, the complete solution in the carry problems consists of overlapping classifiers. In addition it is a niche imbalance domain, so is more difficult to learn than the majority-on problems. Using the ternary alphabet based conditions with static numeric actions, no useful generalizations can be made for the even-parity problems. The results will be compared with the standard ternary alphabet based XCS and related layered learning GP-systems to test

Manuscript received October 5, 2012; revised February 5, 2013 and May 22, 2013; accepted June 2, 2013.

The authors are with the Evolutionary Computation Research Group (ECRG) at the School of Engineering and Computer Science, Victoria University of Wellington, PO Box 600, Wellington 6140, New Zealand.

effectiveness and efficiency of the proposed system.

Wilson's accuracy based XCS [4], [5], the most popular learning classifier system, is used to implement and test the proposed system. In XCS the genetic algorithm (GA) is applied to an action set instead of the whole population to conserve similar building blocks of information. These features of XCS make it possible to form a complete and accurate mapping from inputs and actions to payoff predictions. The ability of XCS to produce complete and accurate solutions, for a given problem, motivated its suitability for this research work. If a learning system is unable to produce a complete and accurate solution, then the extracted building blocks lack important knowledge and so may not be suitable candidates to be used to scale the system.

The rest of the paper is organised as follows. Section II describes the necessary background in transfer learning, evolutionary computation, and learning classifier systems. In Section III the novel implementation of XCS with code-fragment conditions is detailed. Section IV introduces the problem domains and parameter settings used in the experimentation. In Section V experimental results are presented and compared with the standard ternary alphabet based XCS and related layered learning GP systems. Section VI explains in detail the reuse of extracted knowledge and the messy representation in the proposed approach of XCS with code-fragment conditions. In the ending Section this work is concluded and the future work is outlined.

II. Background

This section contains only the essential background material to the novel work presented in the paper. More detailed background is available in the online appendix.

A. Transfer Learning

Transfer learning is a process to transfer knowledge learned in one or more source tasks to a related but more complex, unseen target task, in an effort to facilitate learning in the target task [6]. The source and target tasks may be from the same or different problem domains [7]. The proposed approach presented here is a form of layered learning that is a subclass of transfer learning. The source and target tasks, in each experiment conducted in this work, will be taken from the same problem domain.

Layered learning is a machine learning paradigm, formally introduced by Stone and Veloso [8] as an extension of earlier work by de Garis [9] and Asada et al. [10], where the task to be learned is decomposed into a hierarchy of subtask layers. At each layer a subtask is learned separately, commonly in sequence, by applying a suitable machine learning algorithm that is usually chosen manually according to the subtask characteristics. The knowledge learned at lower layers is used to learn the subtask at the next higher layer. Layered learning mostly applies to complex tasks for which: 1) direct learning is not tractable, and 2) a bottom-up hierarchical decomposition is possible, usually carried out manually using domain-specific knowledge. In the work presented here, each subtask will be a problem of increasing order in size and difficulty from the same problem domain. The learning algorithm to be used at each subtask layer is an extended version of XCS, proposed in this work, see Section III.

B. Evolutionary Computation

Evolutionary computation is a population-based computing paradigm [11] where each individual represents a potential solution or a part of the solution to the problem at hand. The population is evolved by applying genetic 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 of the most common evolutionary techniques, namely genetic algorithms and genetic programming, are briefly described as they are directly related to the work presented here.

1) Genetic Algorithms: The discovery component of an LCS is commonly implemented using a GA. An LCS seeks to evolve a population of co-operative rules, where each individual rule is optimized using the GA.

GAs are an evolutionary computational technique where each individual member of the population is usually represented by a bitstring of fixed length, and represents a potential problem solution [12]. Goldberg hypothesized that higher performance individuals in GAs are actually generated as a result of the combination of short-length, low-order and high-performance schemata[1] – called the *building blocks* of the system [12]. However, for a population of individuals represented by fixed length strings, the genetic operators sometimes cannot process the building blocks effectively as a random crossover point may lie within the building block. To avoid this disruption of partial solutions by the genetic operators, a probability distribution based approach, known as Estimation of Distribution Algorithm (EDA), was developed [14]. In the various forms of EDAs, the crossover and mutation operators are replaced by generating new offspring according to the probability distribution of the selected individuals [15]. Santana et al. [16] and Pelikan et al. [17] have incorporated transfer learning in the field of EDAs to transfer information between optimization problems.

Schema theory has been criticized due to its weak theoretical foundations [18]–[20], but still remains a popular tool to explain the power of GAs [21]–[23].

2) Genetic Programming: GP is an evolutionary approach to generating computer programs for solving a given problem automatically [24], and uses a much richer alphabet than GAs 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 developed here, so the GP technique is described to aid understanding.

In GP each individual is a computer program, commonly represented by a tree, that when executed generates the potential solution. The task to be solved is represented by a primitive set of operations, known as the *function set*, and a set of operands, known as the *terminal set*. The internal nodes of the tree are functions and leaves are the terminals.

[1] A schema is a similarity template for describing a set of finite-length strings defined over a finite alphabet [13].

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 [25]. The other ways to control bloat in GP include simplifying individual programs using algebraic and numerical simplification methods [26], [27], or using specific bloat control operators [28].

GP has also been implemented using non-tree representations, such as linear GP (LGP) [29] and cartesian GP (CGP) [30]. A number of GP researchers have incorporated and investigated layered learning in GP [31]–[33]. CGP and layered learning GP are briefly introduced here as they are closely related to the work presented in this paper.

a) Cartesian Genetic Programming: Cartesian genetic programming (CGP) is a flexible graph-based version of GP that allows a program to be evolved with more than one output, often using an evolution strategy [30]. In CGP, a program is represented as a directed graph that is encoded in the form of a linear string of integers. The graph-based representation has the benefit of implicitly reusing the nodes in the graph. In CGP there is a many-to-one genotype to phenotype mapping due to the presence of a large amount of redundancy [34].

Self-Modifying CGP (SMCGP) is a developmental form of CGP that allows an individual program to modify itself using a set of self-modifying functions [35]. Using SMCGP, Harding et al. [36] have evolved programs that provide general solutions to a number of problems including an n-bit parity problem and an adder to add two n-bit binary numbers.[2] To evolve these programs, they have used a set of self-modifying operators in addition to the usual computational operators.

The main aim of SMCGP was to evolve a computer program that could generate an arbitrary sequence of computer programs, each of which solves a particular problem [36], whereas in the work presented here the main aim is to extract and reuse knowledge of the domain to produce a scalable online learning system.

b) Layered Learning Genetic Programming: For complex problems, the standard monolithic GP may not find a solution due to the large search space leading to an intractable problem. In layered learning, the complex target task is decomposed into subtasks and each subtask is learned in a bottom-up fashion [8]. Gustafson and Hsu [31] implemented layered learning in GP to learn the keep-away soccer game, which is a multi-agent system problem. The main task was decomposed into two subtasks and the final population in the bottom task layer was used as the initial population for the top task layer. The layered learning GP approach evolved better solutions faster than standard GP.

Jackson and Gibbons [32] applied layered learning in GP to solve Boolean logic problems of the even-parity and the majority-on problem domains, using a two-layered approach. The solutions of the bottom layer were encapsulated as parametrized modules and reused to learn the main task in the top layer. The layered learning approach outperformed standard monolithic GP [24] and GP with automatically defined functions (ADFs) [38], albeit it did not achieve 100% success rate for the higher-order problems.

Hien et al. [39] investigated layered learning with incremental sampling in GP. They tested twelve symbolic regression problems and results were compared with standard GP [24]. The combination of incremental sampling with layered learning in GP showed improvement in terms of reducing the training time and complexity of the solutions. Later Hien and Hoai [40] incorporated parameter setting techniques derived from progressive sampling to overcome ad-hoc parameter setting issues in the incremental sampling based layered learning GP.

Hoang et al. [33] investigated interactions between evolution, development, and layered learning using tree adjoining grammar guided GP (TAG3P) [41]. The developed system, called DTAG3P, was tested in symbolic regression problems, Boolean even-parity problems, and ORDERTREE problems. The layered learning DTAG3P system produced more structured and scalable solutions to the problems as compared with two single-short learning GP systems: standard tree-based GP [24] and the pre-existing TAG3P [41]. However, the DTAG3P system introduced a number of new parameters into the TAG3P system.

A GP system produces an individual 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 [42], rather than online, reinforcement learning [43] as in LCS.

C. 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. After observing the current state of the environment, the agent performs an action, and the environment provides a reward. An LCS is an adaptive system that, using the cooperative set of rules, learns to perform the *best* action, i.e. the action that receives the maximum reward from the environment for a given input.

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.[3] 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.

XCS operates in two modes, explore and exploit [45]. In the explore mode, 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

[2]We recently developed a state-machine based XCS system that evolves general and *compact* solutions for the n-bit parity and n-bit carry problems [37].

[3]For a detailed review of different types and approaches in LCS refer to [44].

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. Overly specific classifiers may be removed by a more general and accurate classifier by performing subsumption deletion in order to reduce the number of classifiers in the final population [46]. In contrast to the explore mode, in the exploit mode the agent does not attempt to discover new information and simply performs the action with the best predicted payoff. The exploit mode is also used to test learning performance of the agent in the application.

Lanzi extended the fixed length bitstrings representation of classifier conditions in XCS to a variable-length messy coding in [47]. A messy coded string may be over- or under-specified, due to its variable-length structure [48]. In the messy coded conditions by Lanzi, environmental inputs were translated into the bitstrings that have no positional linking between the bits in a classifier condition and any feature in the environmental input. Then Lanzi and Perrucci [49] enhanced a step further from messy coding to a more complex representation in which S-expressions were used to represent the classifier conditions. Various other richer encoding schemes have been investigated to represent high level knowledge in LCS in an attempt to obtain compact classifier rules [50]–[52], to reach the optimal performance faster [53], [54], to approximate functions [55], [56], to learn problems involving a large number of actions [57], to develop useful feature extractors [58], and to identify and process building blocks of knowledge [59], [60].

1) Previous Work on Code-Fragment Based XCS: In general, a GP-like rich encoding scheme is needed in order to extract the building blocks of knowledge and to reuse them in learning complex, large-scale problems in the domain. Previously, we implemented this scheme to encode the action in a classifier rule, which produced optimal populations in discrete domain problems [52], [61], [62] as well as in continuous domain problems [63], but this did not lead to simple scaling. In our previous code-fragment conditions work [60], [64], we used a separate population of code fragments, which limited the number of available code fragments, resulting in a system that was not able to learn the complex, large-scale problems.

Our previous work introduced GP-tree like expressions to represent condition bits in a classifier rule, named *code-fragment conditions* [64].[4] This initial investigation of code fragments in XCS showed that the multiple genotypes to a single phenotype issue in feature-rich encoding disabled the subsumption deletion function. The additional methods and increased search space also led to much longer training times. This was compensated by the code fragments containing useful knowledge, such as the importance of the address bits in the multiplexer problems. The code fragments also created masks that autonomously subdivided the search space into areas of interest and uniquely, areas of no interest.

In [60], building blocks of knowledge were extracted, in the form of code fragments, from small-scale problems and reused to learn large-scale problems. The resulting code-fragment XCS system outperformed ternary alphabet based XCS in the multiplexer, carry, and even-parity problem domains in terms of improving effectiveness and reducing instances in large-scale problems. Although this was the first time such scalability had been achieved in the field of LCS, the technique could only solve problems to a scale that was previously learnable by existing XCS techniques.

In the previous work on code-fragment conditions, a separate population of code fragments was created and kept static throughout the learning process [60], [64]. This puts a limit on the number of available code fragments that can be used in the conditions of classifiers. Also, the extracted code fragments were used in a hierarchical fashion from one level to seed a population of code fragments in the next level [60], not allowing the direct reuse of the extracted knowledge in previous smaller levels, where 'level' is a single step in problem complexity, e.g. 6-bits MUX to 11-bits MUX. Further, the amount of the code fragments to be reused was set empirically.

III. XCS WITH POPULATION-BASED CODE-FRAGMENT CONDITIONS

In the new work presented here, the condition bit in a classifier is directly replaced with a code fragment instead of addressing a separate population, which is no longer used. Therefore, there is no limit on the number of available code fragments, except in the number of rules in the population. The system is allowed to reuse the extracted code fragments from all previous levels, instead of just one level. The number of code fragments to be reused from a certain level is governed by the unique code fragments in good classifiers, i.e. equal to the number of distinct code fragments in the conditions of accurate and experienced classifiers in the final population with a fitness value greater than the average fitness of the classifier population.

In the proposed XCS with code-fragment conditions, called XCSCFC, each code fragment is a binary tree of depth up to two, which was set to limit the tree size. A binary tree of depth two can have maximum seven nodes. The function set for the tree is problem dependent such as $\{+, -, *, /...\}$ for symbolic regression problems, and {AND, OR, NAND, NOR ...} for binary classification problems. The terminal set is {D0, D1, D2, ... D*n*-1}, where *n* is the length of the environmental input state. A population of classifiers having code-fragment conditions is illustrated in Table I. The symbols &, |, d, r, and $\sim$ denotes AND, OR, NAND, NOR, and NOT operators, respectively. The code fragments are shown in the postfix form.

The proposed XCSCFC system extends standard XCS, described in Section II-C, in the following cases: 1) the classifier matching procedure, the covering operation, the rule discovery operation, the subsumption deletion mechanism,

[4]In [64], the GP-tree like expressions were called automatically defined functions (ADFs), due to the resemblance with ADFs used in GP.

TABLE I
A POPULATION OF CLASSIFIERS USING CODE-FRAGMENT CONDITIONS. HERE '&', '|', 'd', '~', AND 'r' DENOTE AND, OR, NAND, NOT, AND NOR OPERATORS RESPECTIVELY. THE CODE-FRAGMENT CONDITIONS ARE SHOWN IN POSTFIX FORM.

Condition						Action
D0D0~|	D0D5d~	D1D4r~	D0D0~|	D0D0~|	D0D0~|	0
D0D0~|	D0D0~|	D0D0~|	D0	D1D4d	D4	1
D0D0~|	D0D5d~	D5D1&	D0D0~|	D0D0~|	D0D0~|	0
D3D0rD5D1dr	D0D0~|	D0D0~|	D0D0~|	D0D0~|	D0D0~|	1
D0D1dD0D4d&	D0D0~|	D1D0rD2D0|d	D3D1&~	D0D0~|	D0D0~|	0
...						...

and the procedure comparing equality of two classifiers are modified; and 2) the extracted domain knowledge is reused in the form of code fragments.

A. Classifier Matching

A classifier rule cl from the population $[P]$ is said to be matched against a problem instance s from the environment if each code fragment in its condition outputs 1. A code fragment is evaluated by loading the terminal symbols with corresponding binary bits from the observed environmental state s. Assuming the problem instance s is 110101, then the code fragment shown in Fig. 1 will give 1 as the output. This output value was generated by loading D0, D1, D2, and D5 with 1, 1, 0, and 1 respectively.

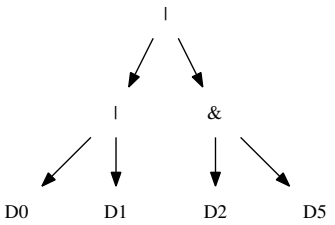


Fig. 1. An example of a code fragment.

There is a special code fragment to be used as 'don't care' symbol in the condition of a classifier rule, shown in Fig. 2. This code fragment always outputs 1.

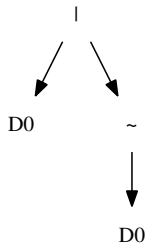


Fig. 2. A code fragment used as 'don't care' symbol in a classifier rule.

Although for simplicity there is the same number of code fragments as condition features, e.g. 6 for the 6-bits MUX problem, there is a decoupling between a code fragment and a position within the condition, i.e. unlike in standard ternary alphabet based XCS the order of code fragments is unimportant. The number of 'specific' code fragments is essentially messy as the system can choose how many 'don't care' code fragments it uses. The classifier matching procedure is described in Fig. 3.

```
1: procedure DOES_MATCH(cl, s)
2:     for i = 1 to n do
3:         cf ← the code fragment cl.cond[i]
4:         if cf ≠ 'don't care' code fragment then
5:             load terminal symbols in cf with corresponding binary bits from the state s
6:             val ← evaluate value of cf
7:             if val ≠ 1 then
8:                 return false
9:             end if
10:        end if
11:    end for
12:    return true
13: end procedure
```

Fig. 3. The procedure to match a classifier cl from the population $[P]$ against an environmental input state s. If the classifier cl matches the state s, then this procedure will return $true$ otherwise $false$. Here n is the length of condition $cond$ in a classifier rule.

B. Covering Operation

Covering occurs if an action 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 'don't care' code fragments with probability $P_{don'tCare}$. All the 'non-don't care' code fragments in this newly created classifier must output 1 against the observed state s. The covering operation is described in Fig. 4.

C. Rule Discovery Operation

In the rule discovery operation, two offspring are produced by applying the GA in the action set $[A]$. First of all, two parent classifiers are selected from $[A]$ based on fitness and the offspring are created out of them. Next, the conditions of the offspring are crossed with probability χ using a two point crossover operation[5], treating each code fragment as a single allele similar to a bit symbol in ternary alphabet based conditions. The blocks of information are essentially the code fragments so are not subjected to disruption by crossover, as shown to be beneficial by EDAs. The crossover operation is described in Fig. 5. After that, each code fragment in the conditions of the crossed over children is mutated with probability μ, such that both children match the currently observed state s. In the mutation operation, a 'non-don't care' code fragment

[5]The XCSCFC system does not depend on any specific type of crossover operation, so the interested researcher can use any type of crossover operation in XCSCFC.

```
1: procedure COVERING_OPERATION(s, a)
2:     initialize classifier cl
3:     initialize condition cl.cond with length n
4:     for i = 1 to n do
5:         if RandomNumber[0, 1) < P_don'tCare then
6:             cl.cond[i] ← 'don't care' code fragment
7:         else
8:             initialize value val to 0
9:             while val ≠ 1 do
10:                cf ← randomly create code fragment
11:                load terminal symbols in cf with corresponding binary bits from the state s
12:                val ← evaluate value of cf
13:            end while
14:            cl.cond[i] ← cf
15:        end if
16:    end for
17:    cl.action ← a
18:    return cl
19: end procedure
```

Fig. 4. The procedure to create a covering classifier cl that will match the current input state s and advocate an action a missing in the match set $[M]$. Here n is the length of condition $cond$ in a classifier rule, and $P_{don'tCare}$ is the probability of 'don't care' code fragment in condition of the newly created classifier in the covering operation.

is replaced by a 'don't care' code fragment, and a 'don't care' code fragment is replaced by a randomly generated 'non-don't care' code fragment that outputs 1 against the state s. Then, the actions of the children are mutated with probability μ. The mutation operation is described in Fig. 6. The prediction of the offspring is set to the average of the parents' values whereas the prediction error and the fitness of the offspring are set to the average of the parents' values reduced by constants *predictionErrorReduction* and *fitnessReduction* respectively, as in [65].

```
1: procedure CROSSOVER_OPERATION(cl_1, cl_2)
2:     x ← RandomNumber[0, n)
3:     y ← RandomNumber[0, n)
4:     if x > y then
5:         swap x and y
6:     end if
7:     for i = x to y do
8:         swap cl_1.cond[i] and cl_2.cond[i]
9:     end for
10: end procedure
```

Fig. 5. The procedure to perform two-point crossover operation on two offspring classifiers cl_1 and cl_2. Here n is the length of condition $cond$ in a classifier rule.

D. Subsumption Deletion

Utilizing code fragments for the matching component of the LCS removes the implicit linking between the position of a condition bit in a classifier rule and the corresponding feature in the environmental input. Although this can lead to compaction of a rule, it also places additional pressure on

```
1: procedure MUTATION_OPERATION(cl, s)
2:     for i = 1 to n do
3:         if RandomNumber[0, 1) < μ then
4:             if cl.cond[i] = 'don't care' code fragment then
5:                 initialize value val to 0
6:                 while val ≠ 1 do
7:                     cf ← randomly create code fragment
8:                     load terminal symbols in cf with corresponding binary bits from the state s
9:                     val ← evaluate value of cf
10:                end while
11:                cl.cond[i] ← cf
12:            else
13:                cl.cond[i] ← 'don't care' code fragment
14:            end if
15:        end if
16:    end for
17:    if RandomNumber[0, 1) < μ then
18:        a ← cl.action
19:        cl.action ← randomly chosen action other than a
20:    end if
21: end procedure
```

Fig. 6. The procedure to perform niche mutation on an offspring classifier cl matching the currently observed input state s. The mutated classifier cl will still match the state s. Here n is the length of condition $cond$ in a classifier rule, and μ is the mutation probability.

subsumption deletion as the reordering of the same conditions needs to be taken into account. It is to be noted that due to the multiple genotypes to a single phenotype issue caused by using tree-based code fragments in place of ternary symbols in the conditions of classifier rules, subsumption deletion is less likely to occur anyway. Subsumption deletion is still made possible, albeit problematic, by matching the code fragments on a character by character basis. The reason for a syntactic equality comparison of code fragments, instead of a semantic one, is that semantic comparison of two code fragments would require evaluation of the code fragments against each possible value of the terminal symbols in both code fragments. As the terminal symbols can be smaller level code fragments (see Section III-F), for large-scale problems, e.g. the 135-bits multiplexer problem, semantic comparison is impractical due to the amount of time needed for evaluation of the code fragments.

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. Classifier cl_1 will be more general than classifier cl_2 if cl_1 has a set of the matching environmental inputs that is a proper superset of the inputs matched by cl_2. In XCSCFC, a classifier cl_1 is said to be more general than a classifier cl_2 if: 1) the number of 'don't care' code fragments in the condition of cl_1 is larger than the number of 'don't care' code fragments in the condition of cl_2; and 2) each 'non-don't care' code fragment in the condition of cl_1 is in the condition of cl_2. This is described in Fig. 7.

```
1: procedure IS_MORE_GENERAL(cl_1, cl_2)
2:     x ← number of 'don't care' code fragments in cl_1
3:     y ← number of 'don't care' code fragments in cl_2
4:     if x ≤ y then
5:         return false
6:     end if
7:     X ← set of all 'non-don't care' code fragments in cl_1
8:     Y ← set of all 'non-don't care' code fragments in cl_2
9:     if X ⊈ Y then
10:        return false
11:    end if
12:    return true
13: end procedure
```

Fig. 7. The procedure to determine whether a classifier cl_1 is more general than another classifier cl_2. The classifier cl_1 will be more general than the classifier cl_2 if cl_1 has a set of the matching environmental inputs that is a proper superset of the inputs matched by cl_2.

E. Comparing Equality of Two Classifiers

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 action and the same code fragments in their conditions. The procedure to compare two classifiers for equality is given in Fig. 8.

```
1: procedure ARE_EQUAL(cl_1, cl_2)
2:     if cl_1.action ≠ cl_2.action then
3:         return false
4:     end if
5:     x ← number of 'non-don't care' code fragments in cl_1
6:     y ← number of 'non-don't care' code fragments in cl_2
7:     if x ≠ y then
8:         return false
9:     end if
10:    X ← set of all 'non-don't care' code fragments in cl_1
11:    Y ← set of all 'non-don't care' code fragments in cl_2
12:    if X ≠ Y then
13:        return false
14:    end if
15:    return true
16: end procedure
```

Fig. 8. The procedure to determine whether two classifiers cl_1 and cl_2 are equal. If both classifiers have the same action and the same code fragments in their conditions, then this procedure will return $true$ otherwise $false$.

F. Reusing Extracted Knowledge

The fitter code fragments, i.e. building blocks of information, from smaller problems, are used to create the code fragments in a higher level problem of the same domain. Each code fragment can be considered a module as in modular CGP [66], and each problem level can be considered a subtask layer as in layered learning [8]. In the work presented here, code fragments are kept static throughout the learning process whereas in modular CGP the modules are allowed to evolve.

The code fragments in the conditions of accurate and experienced classifiers in the final population, with a fitness value greater than the average fitness of the classifier population, are taken as the fitter code fragments and reused to learn the higher level complex problems in the domain. The code fragments from smaller problems are used as terminals in the code fragments of a higher level problem. The probability of a terminal to be a code fragment from previous levels or a condition bit from the current level problem is set to 0.5 following the ramped half and half approach of initializing a population in GP [67].

An example of code fragments in the 20-bits MUX problem is shown in Fig. 9. The code fragments in the 20-bits MUX problem contain fitter code fragments from the 6-bits MUX problem and the 11-bits MUX problem, similarly the code fragments in the 11-bits MUX problem contain code fragments from the 6-bits MUX problem. The code fragments in the 6-, 11-, and 20-bits MUX problems are named as L1_n, L2_n, and L3_n respectively, where n = 0, 1, 2, ... etc.

Multiplexer	Code Fragments	
	Name	Expression
6-bits MUX (Level 1)	L1_0	D1 D0 D4 d r
	L1_1	D5 ~ D1 D0 & &
	...	...
11-bits MUX (Level 2)	L2_0	L1_15 D2 L1_4 r &
	L2_1	L1_5 D7 | L1_11 D3 & r
	...	...
20-bits MUX (Level 3)	L3_0	L2_9 L1_7 D11 | r
	L3_1	L1_10 D17 | L2_1 D0 r &
	...	...

Fig. 9. A sample of code fragments in the 20-bits multiplexer problem. The code fragments in the 6-, 11-, and 20-bits multiplexer problems are named as L1_n, L2_n, and L3_n respectively, where n = 0, 1, 2, ... etc.

IV. THE PROBLEM DOMAINS AND EXPERIMENTAL SETUP

A. The Problem Domains

The problem domains used in the experimentation are the multiplexer problems, the majority-on problems, the carry problems and the even-parity problems.

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-bits 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. Similarly, if the input is 101101 then the output will be 0 as the third bit after the address is indexed. The multiplexer problems are considered to be interesting because they are multi-modal and epistatic, which are therefore difficult to learn. "They are non-trivial high dimensional deceptive and discrete. They have no parameters suitable for continuous gradient ascent" [68]. They

also allow generalizations and are suitable for examining the scalability of an algorithm.

In the 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 strongly overlapping classifiers, which is therefore 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.[6]

In the carry problem, two binary numbers of the same length are added. If the result triggers a carry, then the output is one otherwise zero. For example, in case of three bits numbers 101 and 010, the output is 0, whereas for the numbers 110 and 100 the output is 1. Similar to the majority-on problems, the complete solution in the carry problem domain consists of overlapping classifiers, and in addition it is a niche imbalance problem domain.

The even-parity problems are similar to the majority-on problems in that the output depends on the number of ones in the input instance. If the number of ones is even, the output will be one, or zero otherwise. Using the ternary alphabet based conditions with the static numeric actions, no useful generalizations can be made for the even-parity problems.

B. Experimental Setup

The system uses the following parameter values, commonly used in the literature, as suggested by Butz in [65], and by Butz and Wilson in [45]: learning rate $\beta = 0.2$; fitness fall-off rate $\alpha = 0.1$; prediction error threshold $\epsilon_0 = 10$; fitness exponent $\nu = 5$; threshold for GA application in the action set $\theta_{GA} = 25$; two-point crossover with probability $\chi = 0.8$; mutation probability $\mu = 0.04$; experience threshold for classifier deletion $\theta_{del} = 20$; fraction of mean fitness for deletion $\delta = 0.1$; classifier experience threshold for subsumption $\theta_{sub} = 20$; probability of 'don't care' symbol in covering $P_{don'tCare} = 0.33$; reduction of the prediction error $predictionErrorReduction = 0.25$; reduction of the fitness $fitnessReduction = 0.1$; and the selection method is tournament selection with a 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, NAND, NOR, NOT}, for all four problem domains. 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.

V. Results

In order to test the performance of XCSCFC, the results have been compared with standard XCS on the four problem domains used in experimentation. In addition, the results obtained from the related layered learning GP-systems in the even-parity and the majority-on problem domains have also been compared with XCSCFC.

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

Each result obtained 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 and the Y-axis is the classification performance measured as the moving average over the last 1000 exploit problem instances. This is different from standard supervised learning (batch processing) GP approaches, due to the online nature and descriptive purpose of LCS.

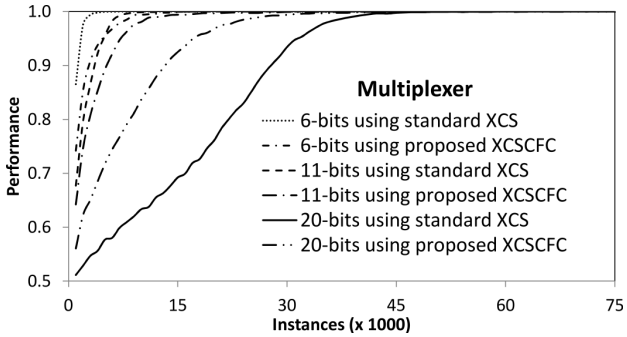
A. Results Comparison with XCS

1) The Multiplexer Problem Domain: The performance of standard XCS and XCSCFC in the multiplexer problem domain is shown in Fig. 10. The number of classifiers used, denoted by N, is 500, 1000, 2000, 5000, 10000, and 50000 for the 6-, 11-, 20-, 37-, 70-, and 135-bits multiplexer problems respectively. The number of training examples used is half a million for the 6-, 11-, and 20-bits multiplexers and one million, two million, and five million for the 37-, 70-, and 135-bits multiplexer problems respectively. Standard XCS was not able to solve the 37-bits MUX problem with $P_{don'tCare} = 0.33$ and $N = 5000$,[7] so $P_{don'tCare}$ was increased to 0.5 in Fig. 10(b). For the 70-bits and the 135-bits MUX problems, $P_{don'tCare}$ is set to 1.0 and μ is set to 0.01 in standard XCS. The condition length of a classifier rule in XCSCFC is set to $70/2 = 35$, and $135/4 = 33$ for the 70-bits and the 135-bits MUX problems respectively. Standard XCS failed 13 times out of 30 runs to solve the 70-bits MUX problem with $N = 10000$, so N was increased to 20000, Fig. 10(c). Here $p\#$ and N denote the probability of 'don't care' symbol and the number of classifiers used respectively.

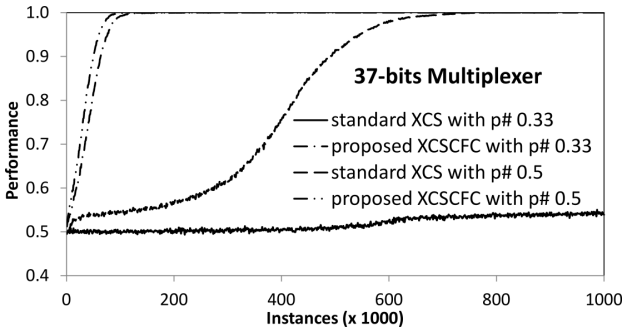
XCSCFC needs more training examples than standard XCS to learn the 6-bits and the 11-bits MUX problems, but less training examples for the 20-bits MUX problem, as shown in Fig. 10(a). Standard XCS, with parameter tuning, needs approximately 800k and 3000k problem instances to solve the 37-bits and the 70-bits MUX problems, see Fig. 10(b) and Fig. 10(c) respectively. However, XCSCFC takes approximately 200k and 500k problem instances to solve the 37-bits and the 70-bits MUX problems respectively, without parameter tuning. The performance curves for the 70-bits MUX problem using XCSCFC are almost coincident in Fig. 10(c).

Standard XCS was not able to solve the 135-bits MUX problem, either in the literature or with further parameter tuning conducted here. However, if a stepped reward function is used to guide learning [69] then the state-of-the-art in the field was to solve the 135-bits MUX problem. XCSCFC, reusing the extracted domain knowledge, successfully solved the standard 135-bits MUX problem taking approximately two million training instances, Fig. 10(d), without needing stepped reward. Considering the number of possible instances is $2^{135} \approx 4 \times 10^{40}$, and that XCSCFC takes only 2×10^6 instances (i.e. sampling only one in 10^{34} instances) to be able to solve the problem, this result is remarkable.

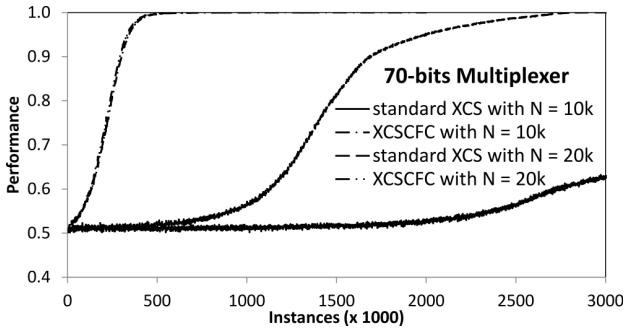
[7]In simple problems the conventional parameter values set produces robust performances, but requires adjustment in complex domains, e.g. 37-bits MUX and above.



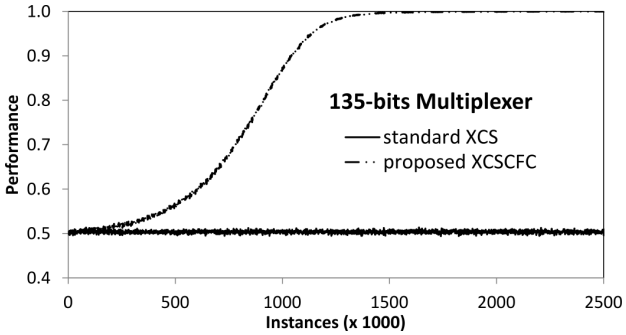
(a) Results for the 6-, 11-, and 20-bits multiplexer problems.



(b) Results for the 37-bits multiplexer problem.



(c) Results for the 70-bits multiplexer problem.



(d) Results for the 135-bits multiplexer problem.

Fig. 10. Results of the multiplexer problems using XCS and XCSCFC. The performance curves for the 70-bits MUX problem using XCSCFC are almost coincident in (c).

2) The Majority-on Problem Domain: The performance of standard XCS and XCSCFC in the majority-on problem domain is shown in Fig. 11. The number of classifiers used is 500, 1000, and 2000 for the 3-, 5-, and 7-bits majority-on problems respectively. The number of training examples used is half a million.

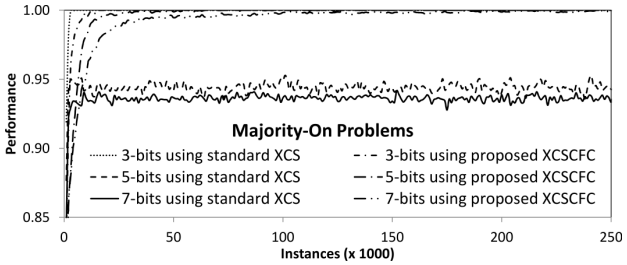


Fig. 11. Results of the majority-on problems using XCS and XCSCFC.

The complete solution of the majority-on problem domain consists of strongly overlapping classifiers. The overlapping nature of classifiers in the final solution makes it harder to learn the problem. XCSCFC successfully learned the 3-, 5-, and 7-bits majority-on problems, whereas standard XCS failed to learn the 5-bits and the 7-bits majority-on problems.

To test statistical significance of XCSCFC with comparison to standard XCS, the Wilcoxon signed rank test was conducted, see Table II. The values in column two and column three are the average performance values of the last 100 test cases along with the standard deviation. The last column shows the p-value obtained with confidence interval of 95%. The performance improvement of XCSCFC is statistically significant as for both cases the p-value is much less than 0.05.

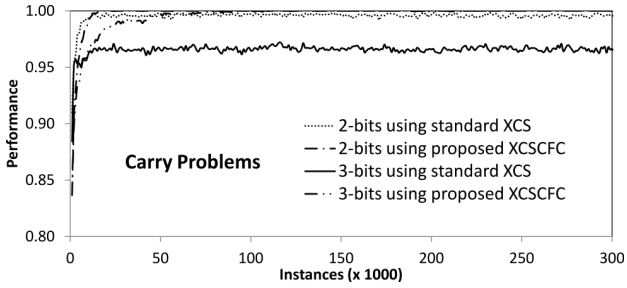
TABLE II
THE WILCOXON SIGNED RANK TEST FOR PERFORMANCE COMPARISON IN THE MAJORITY-ON PROBLEM DOMAIN.

Majority-On	XCS	XCSCFC	p-value
5-bits	95.17 $\pm$ 2.49	100.00 $\pm$ 0.00	$3.54e^{-6}$
7-bits	94.43 $\pm$ 2.65	100.00 $\pm$ 0.00	$1.66e^{-6}$

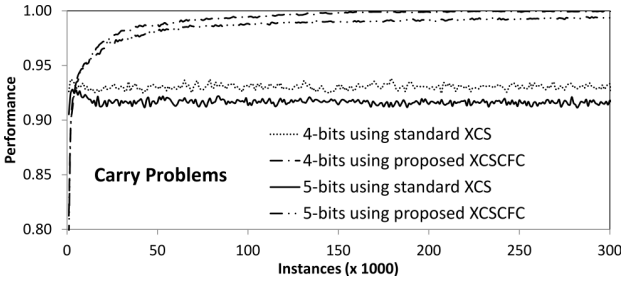
3) The Carry Problem Domain: The performance of standard XCS and XCSCFC in the carry problem domain is shown in Fig. 12. The number of classifiers used is 1000, 2000, 4000, and 6000 for the 2-, 3-, 4-, and 5-bits carry problems respectively. The number of training examples used for the 2-, and 3-bits carry problems is half a million whereas for the 4-, and 5-bits carry problems one million training examples have been used.

The complete solution in the carry problem domain consists of overlapping classifiers, in addition it is a niche imbalance domain, therefore very difficult to learn. Standard XCS was not able to reach consistent 100% performance even for the 2-bits carry problem, see Fig. 12(a), whereas XCSCFC successfully solved the 2-bits and the 3-bits carry problems. XCSCFC also learned the 4-bits carry problem and in the case of the 5-bits carry problem XCSCFC outperformed standard XCS, albeit not reaching 100% consistent and stabilized performance as shown in Fig. 12(b).

The results of the Wilcoxon signed rank test conducted to measure the statistical significance of XCSCFC compared with



(a) Results for the 2-, and 3-bits carry problems.



(b) Results for 4-, and 5-bits carry problems.

Fig. 12. Results of the carry problems using XCS and XCSCFC.

standard XCS are shown in Table III. The values in column two and column three are the average performance values of the last 100 test cases along with the standard deviation. The performance improvement of XCSCFC is statistically significant as for all cases the p-value, obtained with confidence interval of 95%, is far less than 0.05.

TABLE III
THE WILCOXON SIGNED RANK TEST FOR PERFORMANCE COMPARISON IN THE CARRY PROBLEM DOMAIN.

Carry	XCS	XCSCFC	p-value
2-bits	99.17 $\pm$ 1.19	100.00 $\pm$ 0.00	$7.80e^{-3}$
3-bits	96.30 $\pm$ 2.67	100.00 $\pm$ 0.00	$1.66e^{-6}$
4-bits	93.47 $\pm$ 2.60	100.00 $\pm$ 0.00	$1.64e^{-6}$
5-bits	92.10 $\pm$ 3.12	99.87 $\pm$ 0.43	$1.59e^{-6}$

4) The Even-Parity Problem Domain: The performance of standard XCS and XCSCFC in the even-parity problem domain is shown in Fig. 13. The number of classifiers used is 200, 300, 400, 500, 1000, and 2000 for the 2-, 3-, 4-, 5-, 6-, and 7-bits problems respectively. Each run is stopped after half a million training examples.

It is observed that XCSCFC needs more training examples than standard XCS to learn the 2-, 3-, and 4-bits even-parity problems, see Fig. 13(a). As the problem scales to 6-bits, standard XCS cannot learn the even-parity problem, see Fig. 13(b), whereas XCSCFC successfully solved up to the 7-bits even-parity problems.

The results of the Wilcoxon signed rank test conducted to measure the statistical significance of XCSCFC with comparison to standard XCS are shown in Table IV. The values in column two and column three are the average performance values of the last 100 test cases along with the standard deviation. The performance improvement of XCSCFC is statistically significant as for all the three cases the p-value, obtained with confidence interval of 95%, is far less than 0.05.

TABLE IV
THE WILCOXON SIGNED RANK TEST FOR PERFORMANCE COMPARISON IN THE EVEN-PARITY PROBLEM DOMAIN.

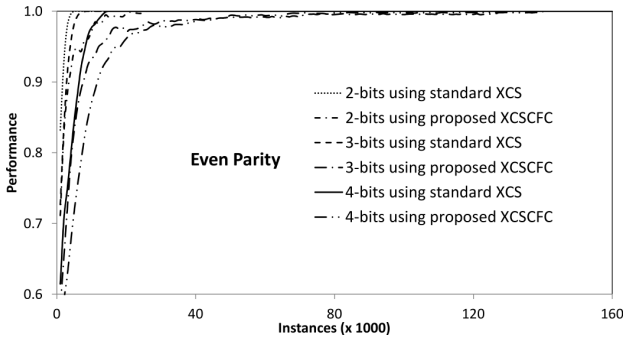
Parity	XCS	XCSCFC	p-value
5-bits	93.77 $\pm$ 9.79	100.00 $\pm$ 0.00	$4.88e^{-4}$
6-bits	79.23 $\pm$ 9.26	100.00 $\pm$ 0.00	$8.07e^{-6}$
7-bits	76.47 $\pm$ 6.18	100.00 $\pm$ 0.00	$2.49e^{-6}$

If a classifier rule is encoded using the standard ternary alphabet based conditions and the static numeric actions, then the even-parity problem domain does not allow any generalizations. Therefore, each bit must be specific for a classifier rule to be accurate in standard XCS. For small-scale problems, it is relatively easy to learn each bit, so standard XCS successfully learnt up to the 5-bits even-parity problems. As the problem scaled to 6-bits and higher levels, standard XCS was not able to solve them, having typically used XCS parameter settings where probability of 'don't care' symbol and that of mutation was set 0.33 and 0.04 respectively. However, in XCSCFC the number of 'specific' code fragments is essentially messy as the system can choose the number of 'don't care' fragments it uses. Also, utilizing code fragments for the matching component of the LCS removes the implicit linking between the position of a condition bit in a rule and the corresponding feature in the problem input. Therefore, the XCSCFC system, having the ability to generalize, has performed efficiently in the even-parity problem domain.

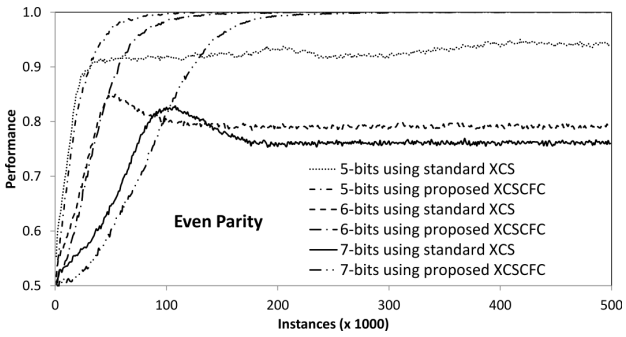
For example, consider an experienced, accurate, correct and general classifier rule 'L1_7 D2 D0D0~| : 1', taken from the final rule base of the 3-bits (i.e. Level 2) even-parity problem where L1_7 is a Level 1 (i.e. 2-bits even-parity problem) code fragment given by D1D0|D1D0d&. In XCSCFC, a classifier rule is said to be matched against an environmental input state if the computed value of all the code fragments in the classifier's condition is equal to 1. Now, L1_7 is equivalent to 'D1 XOR D0' that outputs 1 if and only if D0 and D1 have different values and D0D0~| is the 'don't care' code fragment that always outputs 1. Therefore, this rule will match an environmental state if D0 and D1 have different values and D2 is equal to 1 (as the second code fragment is just D2 in this rule) in the environmental state. So, this general rule is equivalent to two specific rules: '011 : 1' and '101 : 1'. The generalization ability of XCSCFC in the even-parity problems will be further discussed in Section VI.

B. Results Comparison with GP Systems

The LCS and GP systems are two different evolutionary techniques that solve a problem in different ways, i.e. LCS is an online reinforcement learning system whereas GP is a supervised learning batch processing approach. The primary aim of the work presented here was not to develop a competitor for the GP systems or other layered learning approaches, and it is not straightforward to compare the proposed system with a



(a) Results for the 2-, 3-, and 4-bits even-parity problems.



(b) Results for the 5-, 6-, and 7-bits even-parity problems.

Fig. 13. Results of the even-parity problems using XCS and XCSCFC.

GP system. However some attempt at comparison with layered learning GP approaches has been made to clarify the benefits of the proposed approach in terms of scalability.

The first comparison is with a layered learning GP system, called LLGP, developed by Jackson and Gibbons [32] using a two-layered approach where the solutions of the bottom layer were encapsulated as parametrized modules and reused to learn the main task in the top layer. They tested the LLGP system on the even-parity problems and the majority-on problems. The 2-bits even-parity problem was used at the bottom layer to solve the 4-, 5-, and 6-bits even-parity problems having the function set {AND, OR, NAND, NOR}. Each experiment was repeated 100 times with maximum 50 generations in each run. The population size used for the 4-bits even-parity problem was 500 and it was increased to 2000 for the 5-, and 6-bits even-parity problems. The layered learning approach outperformed the standard monolithic GP [24] and the GP with ADFs [38], albeit not achieving 100% success rate as shown in Table V.

TABLE V
PERFORMANCE OF DIFFERENT GP SYSTEMS, IN TERMS OF SUCCESS RATE OUT OF 100 RUNS, FOR THE EVEN-PARITY PROBLEMS [32].

Problem	GP	GP with ADFs	LLGP
4-bits	14	43	95
5-bits	0	32	92
6-bits	0	16	70

To compare XCSCFC with the LLGP system for the even-parity problems, the function set of XCSCFC was changed to {AND, OR, NAND, NOR}. The number of classifiers used is 200, 300, 400, 500, and 1000 for the 2-, 3-, 4-, 5-, and 6-bits even-parity problems respectively. The number of training examples used is half a million. The performance of XCSCFC for the 2-bits to 6-bits even-parity problems is shown in Fig. 14. XCSCFC solved all these problems successfully in each of the 30 conducted experiments.

The second comparison is with the LLGP system for the majority-on problems. In LLGP, the 3-bits majority-on problem was used at the bottom layer to solve the 5-, and 7-bits majority-on problems having the function set {AND, OR, NOT}. Each experiment was repeated 100 times with maximum 50 generations in each run. The population size used for

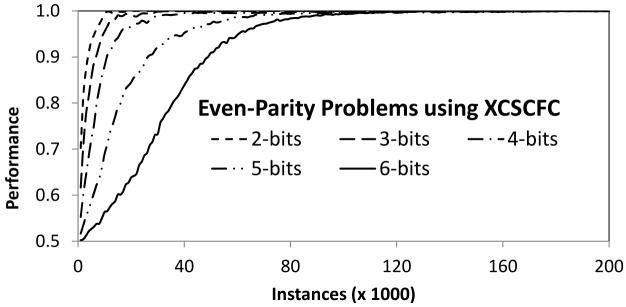


Fig. 14. Results of the even-parity problems obtained using XCSCFC with the function set {AND, OR, NAND, NOR}.

the 5-bits majority-on problem was 500 and it was increased to 1000 for the 7-bits majority-on problem. The layered learning approach outperformed the standard monolithic GP [24] and the GP with ADFs [38], albeit not achieving 100% success rate for the 7-bits majority-on problem as shown in Table VI.

TABLE VI
PERFORMANCE OF DIFFERENT GP SYSTEMS, IN TERMS OF SUCCESS RATE OUT OF 100 RUNS, FOR THE MAJORITY-ON PROBLEMS [32].

Problem	GP	GP with ADFs	LLGP
5-bits	62	7	100
7-bits	18	not attempted	90

To compare XCSCFC with LLGP for the majority-on problems, the function set of XCSCFC was changed to {AND, OR, NOT}. The number of classifiers used is 500, 1000, and 2000 for the 3-, 5-, and 7-bits majority-on problems respectively. The number of training examples used is half a million. The performance of XCSCFC for the 3-, 5-, and 7-bits majority-on problems is shown in Fig. 15. XCSCFC solved successfully all these problems in each of the conducted experiments.

The third comparison is with the DTAG3P system developed by Hoang et al. [33]. Using the DTAG3P system, the 8-bits even-parity problem was experimented in a layered learning fashion using the function set {AND, OR, NOT, XOR}, the population size $max_{pop} = 250$, and the number of maximum generations at each problem level $max_{gen} = 101$. Although DTAG3P outperformed the two single-short learning GP systems, the standard tree-GP system [24] and the TAG3P system [41], it could not achieve 100% success rate for the 8-bits

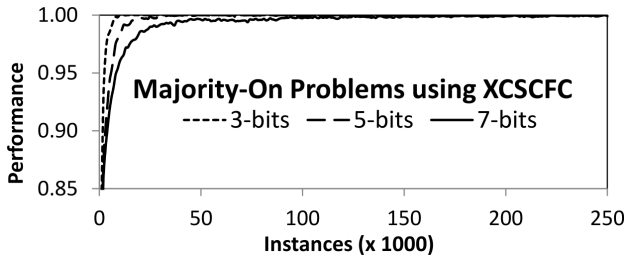


Fig. 15. Results of the majority-on problems obtained using XCSCFC with the function set {AND, OR, NOT}.

even-parity problem. The reported success rates are 6.67%, 10%, and 86.67% for the standard tree-GP, the TAG3P, and the DTAG3P systems, respectively [33].

To compare XCSCFC with DTAG3P, the function set of XCSCFC was changed to {AND, OR, NOT, XOR}. The number of classifiers used is 200, 300, 400, 500, 1000, 1500, and 2000 for the 2-, 3-, 4-, 5-, 6-, 7-, and 8-bits even-parity problems respectively. The number of training examples used is half a million. The performance of XCSCFC for the 2-bits to 8-bits even-parity problems is shown in Fig. 16. XCSCFC solved successfully all these problems in each of the conducted experiments.

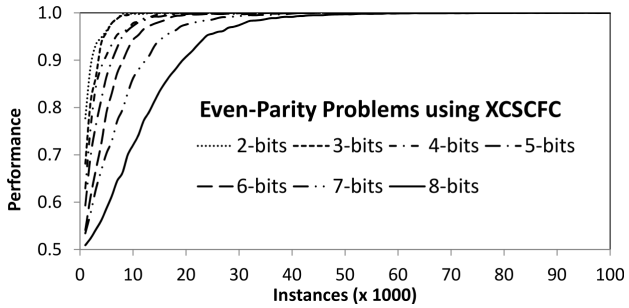


Fig. 16. Results of the even-parity problems obtained using XCSCFC with the function set {AND, OR, NOT, XOR}.

Poli and Page [70] have developed a single-short learning GP system by using smooth uniform crossover, sub-machine code GP, and distributed demes to solve higher-order even-parity problems. It is reported that the 12-, 13-, 15-, 17-, 20- and 22-bits even-parity problems were solved successfully, but they have used all the 16 Boolean operators of two variables [71] in the function set. Thus the experiment setting is very different from this paper, so a direct comparison is not very meaningful – we leave this to future work.

VI. Interpretation of Results

The XCSCFC system has solved up to and including the 135-bits multiplexer problems by extracting and reusing the building blocks of domain knowledge. The reuse of extracted knowledge has shown generalization ability in the even-parity domain problems that is not possible using the standard ternary alphabet based representation. The following subsection describes in detail the reuse of the extracted knowledge in the multiplexer and even-parity problem domains. This is followed by a discussion of messy code-fragment conditions.

A. Reuse of Extracted Knowledge

A classifier rule from the final rule base of the 20-bits multiplexer problem is depicted in Fig. 17, along with the code fragments being used by the classifier. Here A and p represent action and prediction of the classifier, respectively. It is to be noted that only specific code fragments in the condition are shown, the 16 'don't care' code fragments occurring in the condition are not shown to save space. This is a compact rule, using just four code fragments. These code fragments in the 20-bits MUX are using three building blocks of knowledge, in the form of code fragments, from the 6-bits MUX (i.e. Level 1), namely L1_29, L1_12 and L1_21, and one from the 11-bits MUX (i.e. Level 2), namely L2_3 that is further using a code fragment from the 6-bits MUX, namely L1_6.

Condition				A	p
L1_29 D1 d D4 D2 | d	L1_12 D0 d ~	L2_3 D18 & ~	L1_21	0	1000

Multiplexer	Code Fragments	
	Name	Expression
11MUX (Level 2)	L2_3	L1_6 D9 | D1 &
6MUX (Level 1)	L1_6	D0
	L1_12	D3 D1 & ~
	L1_21	D2
	L1_29	D1 D2 r ~

Fig. 17. A classifier rule from the final rule base obtained for a typical run of the 20-bits multiplexer problem.

In XCSCFC, a classifier rule is said to be matched against a problem instance if the computed values of all the code fragments in the classifier's condition are equal to 1. The fourth code fragment 'L1_21' in the classifier rule shown in Fig. 17 is just $D2$, therefore, $D2$ must be 1 in the environmental instance to be matched by this classifier.

The first code fragment 'L1_29 D1 d D4 D2 | d' is using three environmental features, i.e. $D1$, $D2$, and $D4$. Now, $D2$ must be 1 if the environmental instance is to be matched by this classifier, so the code fragment 'L1_29 D1 d D4 D2 | d' will output 1 if and only if the value of the feature $D1$ is 1, as shown in Table VII.

TABLE VII
Truth table for the code fragment 'L1_29 D1 d D4 D2 | d', where 'L1_29' is 'D1 D2 r ~'.

Sr. No.	D1	D2	D4	L1_29	L1_29D1d	D4D2|	L1_29D1dD4D2|d
1	0	1	0	1	1	1	0
2	0	1	1	1	1	1	0
3	1	1	0	1	0	1	1
4	1	1	1	1	0	1	1

The second code fragment 'L1_12 D0 d ~' is using three environmental features, i.e. $D0$, $D1$, and $D3$. Now, $D1$ must be 1 if the environmental instance is to be matched by this classifier, so the code fragment 'L1_12 D0 d ~' will output 1 if and only if $D0 = 1$ and $D3 = 0$, as shown in Table VIII.

The third code fragment 'L2_3 D18 & ~' = 'D0 D9 | D1 & D18 & ~' is using four environmental features, i.e.

TABLE VIII
TRUTH TABLE FOR THE CODE FRAGMENT 'L1_12 D0 d ∼', WHERE 'L1_12' IS 'D3 D1 & ∼'.

Sr. No.	D0	D1	D3	L1_12	L1_12D0d	L1_12D0d∼
1	0	1	0	1	1	0
2	0	1	1	0	1	0
3	1	1	0	1	0	1
4	1	1	1	0	1	0

$D0$, $D1$, $D9$, and $D18$. Now, $D0$ and $D1$ must be 1 if the environmental instance is to be matched by this classifier, so the code fragment 'L2_3 D18 & ∼' will output 1 if and only if $D18 = 0$, as shown in Table IX.

TABLE IX
TRUTH TABLE FOR THE CODE FRAGMENT 'L2_3 D18 & ∼', WHERE 'L2_3' IS 'L1_6 D9 | D1 &' AND 'L1_6' IS 'D0'.

Sr. No.	D0	D1	D9	D18	X = D0D9|	Y = XD1&	Z = YD18&	Z∼
1	1	1	0	0	1	1	0	1
2	1	1	0	1	1	1	1	0
3	1	1	1	0	1	1	0	1
4	1	1	1	1	1	1	1	0

Therefore, the classifier rule 'L1_29D1dD4D2|d L1_12D0d∼ L2_3D18&∼ L1_21: 1' will match all the problem instances having features $D0 = 1$, $D1 = 1$, $D2 = 1$, $D3 = 0$, and $D18 = 0$. This classifier is maximally general and accurate, being equivalent to the classifier '1110###############0# : 0' represented in ternary alphabet based form.

To illustrate the generalization ability of XCSCFC in the even-parity problems, a classifier rule from the final rule base of the 4-bits even-parity problem, depicted in Figure 18, is analyzed. Here A and p represent action and prediction of the classifier, respectively. It is to be noted that only specific code fragments in the condition are shown, the two 'don't care' code fragments occurring in the condition are not shown to save space. These code fragments in the 4-bits even-parity problem (i.e. 4EP) are using a code fragment from the 3EP (i.e. Level 2), namely L2_4 that is further using a code fragment from the 2EP (i.e. Level 1), namely L1_7.

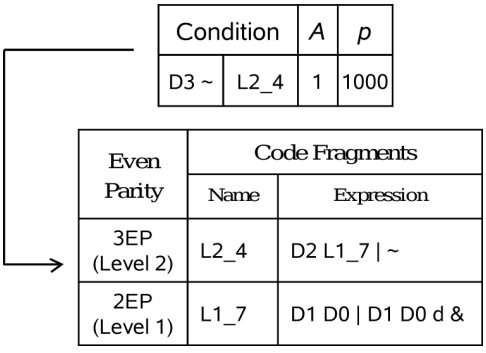


Fig. 18. A classifier rule from the final rule base obtained for a typical run of the 4-bits even-parity problem.

The first code fragment 'D3 ∼' in the classifier rule shown in Fig. 18 is just negation of $D3$, therefore, $D3$ must be 0 in the environmental instance to be matched by this classifier.

The second code fragment 'L2_4' = 'D2 L1_7 | ∼' = 'D2 D1 D0 | D1 D0 d & | ∼' uses three environmental features, i.e. $D0$, $D1$, and $D2$. The code fragment 'L1_7' is equivalent to 'D1 XOR D0' that outputs 1 if and only if $D0$ and $D1$ have different values, so the code fragment 'L2_4' will output 1 if and only if $D2 = 0$ and $D0 = D1$, as shown in Table X. The ability to consider the features' property, such as $D0 = D1$, is not expressible in the ternary alphabet based representation.

TABLE X
TRUTH TABLE FOR THE CODE FRAGMENT 'L2_4', WHERE L2_4 IS 'D2 L1_7 | ∼' AND L1_7 IS 'D1 D0 | D1 D0 d &' = 'D1 XOR D0'.

Sr. No.	D0	D1	D2	L1_7	D2L1_7|	D2L1_7|∼
1	0	0	0	0	0	1
2	0	0	1	0	1	0
3	0	1	0	1	1	0
4	0	1	1	1	1	0
5	1	0	0	1	1	0
6	1	0	1	1	1	0
7	1	1	0	0	0	1
8	1	1	1	0	1	0

Therefore, the classifier rule 'D3∼ L2_4: 1' will match all the problem instances having features $D0 = D1$, $D2 = 0$, and $D3 = 0$. This general classifier is equivalent to two specific classifiers '0000 : 1' and '1100 : 1'.

Consider another general and interesting classifier rule 'D2D2&L1_4| D0D0∼| D2L1_7d : 0', taken from the final rule base of the 3-bits even-parity problem. In this classifier rule 'L1_4' and 'L1_7' are the code fragments from the 2-bits even-parity problem given by 'D0D0rD1D1&r' and 'D1D0|D1D0d&' respectively. The code fragment 'L1_4' outputs 1 if and only if D0 is 1 and D1 is 0. To determine the subset of environmental instances being matched by this rule, consider the truth tables for the code fragments 'D2D2&L1_4|' and 'D2L1_7d' shown in Table XI and Table XII respectively.[8] This rule will match against an environmental instance if the output values for both code fragments 'D2D2&L1_4|' and 'D2L1_7d' are equal to 1. Therefore, the instances numbered 2, 5, and 8 in Table XI and Table XII constitute the matching subset of environmental instances for this rule. So, this general rule is equivalent to three specific rules: '001 : 0', '100 : 0', and '111 : 0'.

TABLE XI
TRUTH TABLE FOR THE CODE FRAGMENT 'D2 D2 & L1_4 |', WHERE 'L1_4' OUTPUTS 1 IF AND ONLY IF D0 IS 1 AND D1 IS 0.

Sr. No.	D0	D1	D2	D2D2&	L1_4	D2D2&L1_4|
1	0	0	0	0	0	0
2	0	0	1	1	0	1
3	0	1	0	0	0	0
4	0	1	1	1	0	1
5	1	0	0	0	1	1
6	1	0	1	1	1	1
7	1	1	0	0	0	0
8	1	1	1	1	0	1

B. Messy Code-Fragment Conditions

In XCSCFC, there is no linking between the position of a condition bit in a classifier rule and the corresponding feature in the environmental input state. Therefore, it is not necessary to use the same number of code fragments in a classifier's condition as the number of problem features. For example,

[8]It is to be noted that the code fragment 'D0D0∼|' in the classifier being analyzed here is the 'don't care' code fragment.

TABLE XII
TRUTH TABLE FOR THE CODE FRAGMENT 'D2 L1_7 d', WHERE 'L1_7' OUTPUTS 1 IF AND ONLY IF D0 AND D1 HAVE DIFFERENT VALUES.

Sr. No.	D0	D1	D2	L1_7	D2L1_7d
1	0	0	0	0	1
2	0	0	1	0	1
3	0	1	0	1	1
4	0	1	1	1	0
5	1	0	0	1	1
6	1	0	1	1	0
7	1	1	0	0	1
8	1	1	1	0	1

different numbers of code fragments can be used to learn the 6-bits multiplexer problem as shown in Fig. 19.

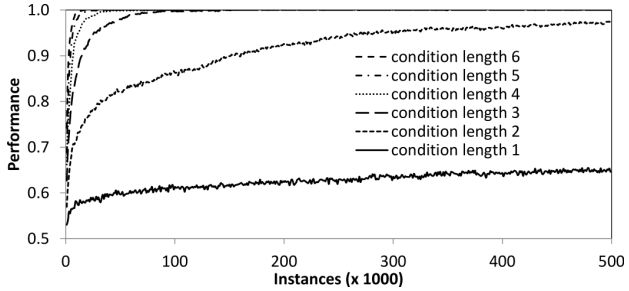


Fig. 19. The performance of XCSCFC, using different number of code fragments in the condition of a classifier rule, for the 6-bits multiplexer problem (curve order same as in legend).

It is observed that the 6-bits multiplexer problem can be solved using different numbers of code fragments in a classifier's condition, but to solve it effectively a minimum of three code fragments should be used. If more than the minimum required code fragments are used, then the performance is found to be robust for the 6-bits multiplexer problem. The minimum number of code fragments needed in any problem in a domain is not optimized currently.

VII. CONCLUSIONS

Building blocks of knowledge were successfully extracted from small-scale problems and reused to learn more complex, large-scale problems in the domain. For example, in the 135-bits multiplexer problem, where the number of possible instances is $2^{135} \approx 4 \times 10^{40}$, XCSCFC takes only 2×10^6 instances (i.e. sampling only one in 10^{34} instances) to successfully solve the problem.

The XCSCFC system, using a GP-like rich encoding scheme, has shown the generalization ability in the even-parity domain problems that is not expressible using the standard ternary alphabet-based representation.

The XCSCFC system is currently tested for only Boolean problems. It will be adapted to other problem domains such as symbolic regression, using interval based conditions in the classifier rules and appropriate operators in the function set.

The current implementation of XCSCFC uses static code fragments, extracted from smaller problems to generate code fragments in the higher level problems in the domain. A mechanism is needed to introduce plausibly better code fragments as training progresses, without disrupting existing classifiers.

XCSCFC readily solves problems of a scale that existing classifier system and genetic programming approaches cannot, e.g. the 135-bits MUX problem. However, the results obtained cannot be proved to be general due to the messy rule-based nature of the LCS approach. In the future, domain level knowledge will be extracted, instead of problem level knowledge, in the form of abstracted patterns, and reused in the function set rather than just as the terminal set at present. It is anticipated that using the extracted domain level knowledge from multiple problem domains will result in a general scalable classifier system.

REFERENCES

[1] S. Thrun, "Is Learning The n-th Thing Any Easier Than Learning The First?" in *Advances in Neural Information Processing Systems*. The MIT Press, 1996, pp. 640–646.
[2] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson, "Toward a Theory of Generalization and Learning in XCS," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 1, pp. 28–46, 2004.
[3] C. Ioannides and W. N. Browne, "Investigating Scaling of an Abstracted LCS Utilising Ternary and S-Expression Alphabets," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2007, pp. 2759–2764.
[4] S. W. Wilson, "Classifier Fitness Based on Accuracy," *Evolutionary Computation*, vol. 3, no. 2, pp. 149–175, 1995.
[5] ——, "Generalization in the XCS Classifier System," in *Proceedings of the Genetic Programming Conference*, 1998, pp. 665–674.
[6] L. Torrey and J. Shavlik, "Transfer Learning," in *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI Global, 2010, ch. 11, pp. 242–264.
[7] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.
[8] P. Stone and M. Veloso, "Layered Learning," in *Proceedings of the European Conference on Machine Learning*, 2000, pp. 369–381.
[9] H. de Garis, "GENETIC PROGRAMMING - Building Nanobrains with Genetically Programmed Neural Network Modules," in *Proceedings of the International Joint Conference on Neural Networks*, 1990, pp. 511–516.
[10] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda, "Purposive Behavior Acquisition for a Real Robot by Vision-Based Reinforcement Learning," *Machine Learning*, vol. 23, pp. 279–303, 1996.
[11] K. A. De Jong, *Evolutionary Computation: A Unified Approach*. The MIT Press, 2006.
[12] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989.
[13] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
[14] H. Mühlenbein and G. Paaß, "From Recombination of Genes to the Estimation of Distributions I. Binary Parameters," in *Proceedings of the Parallel Problem Solving from Nature*, 1996, pp. 178–187.
[15] M. Pelikan, D. E. Goldberg, and F. G. Lobo, "A Survey of Optimization by Building and Using Probabilistic Models," *Computational Optimization and Applications*, vol. 21, no. 1, pp. 5–20, 2002.
[16] R. Santana, A. Mendiburu, and J. A. Lozano, "Structural Transfer Using EDAs: An Application to Multi-Marker Tagging SNP Selection," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2012, pp. 3221–3228.
[17] M. Pelikan and M. W. Hauschild, "Learn From the Past: Improving Model-Directed Optimization by Transfer Learning Based on Distance-Based Bias," Missouri Estimation of Distribution Algorithms Laboratory, University of Missouri in St. Louis, MO, United States, Tech. Rep. 2012007, 2012.
[18] L. Altenberg, "The Schema Theorem and Price's Theorem," in *Proceedings of the Foundations of Genetic Algorithms*, 1995, pp. 23–49.
[19] H. G. Beyer, "An Alternative Explanation for the Manner in which Genetic Algorithms Operate," *BioSystems*, vol. 41, pp. 1–15, 1997.
[20] K. M. Burjorjee, "The Fundamental Problem with the Building Block Hypothesis," *CoRR*, vol. abs/0810.3356, 2008. [Online]. Available: http://arxiv.org/abs/0810.3356

[21] R. Poli and W. B. Langdon, "Schema Theory for Genetic Programming with One-point Crossover and Point Mutation," *Evolutionary Computation*, vol. 6, pp. 231–252, 1998.

[22] R. Poli, "Why the Schema Theorem is Correct also in the Presence of Stochastic Effects," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2000, pp. 487–492.

[23] J. Drugowitsch, *Design and Analysis of Learning Classifier Systems: A Probabilistic Approach*. Springer, 2008.

[24] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[25] S. Luke and L. Panait, "A Comparison of Bloat Control Methods for Genetic Programming," *Evolutionary Computation*, vol. 14, no. 3, pp. 309–344, 2006.

[26] M. Zhang and P. Wong, "Genetic Programming for Medical Classification: A Program Simplification Approach," *Genetic Programming and Evolvable Machines*, vol. 9, no. 3, pp. 229–255, 2008.

[27] D. Kinzett, M. Johnston, and M. Zhang, "Numerical Simplification for Bloat Control and Analysis of Building Blocks in Genetic Programming," *Evolutionary Intelligence*, vol. 2, no. 4, pp. 151–168, 2009.

[28] E. Alfaro-Cid, J. J. Merelo, F. F. de Vega, A. I. Esparcia-Alcázar, and K. Sharman, "Bloat Control Operators and Diversity in Genetic Programming: A Comparative Study," *Evolutionary Computation*, vol. 18, no. 2, pp. 305–332, 2010.

[29] M. Brameier and W. Banzhaf, *Linear Genetic Programming*. Springer, 2007.

[30] J. F. Miller and P. Thomson, "Cartesian Genetic Programming," in *Proceedings of the European Conference on Genetic Programming*, 2000, pp. 121–132.

[31] S. M. Gustafson and W. H. Hsu, "Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem," in *Proceedings of the European Conference on Genetic Programming*, 2001, pp. 291–301.

[32] D. Jackson and A. P. Gibbons, "Layered Learning in Boolean GP Problems," in *Proceedings of the European Conference on Genetic Programming*, 2007, pp. 148–159.

[33] T.-H. Hoang, R. I. B. McKay, D. Essam, and N. X. Hoai, "On Synergistic Interactions Between Evolution, Development and Layered Learning," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 3, pp. 287–312, 2011.

[34] J. F. Miller and S. L. Smith, "Redundancy and Computational Efficiency in Cartesian Genetic Programming," *IEEE Transaction on Evolutionary Computation*, vol. 10, no. 2, pp. 167–174, 2006.

[35] S. Harding, J. F. Miller, and W. Banzhaf, "Self-Modifying Cartesian Genetic Programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2007, pp. 1021–1028.

[36] ——, "Developments in Cartesian Genetic Programming: self-modifying CGP," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 397–439, 2010.

[37] M. Iqbal, W. N. Browne, and M. Zhang, "Extending Learning Classifier System with Cyclic Graphs for Scalability on Complex, Large-Scale Boolean Problems," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2013, to appear.

[38] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, 1994.

[39] N. T. Hien, N. X. Hoai, and B. McKay, "A Study on Genetic Programming with Layered Learning and Incremental Sampling," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2011, pp. 1179–1185.

[40] N. T. Hien and N. X. Hoai, "Learning in Stages: A Layered Learning Approach for Genetic Programming," in *Proceedings of the International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future*, 2012. [Online]. Available: http://dx.doi.org/10.1109/rivf.2012.6169838

[41] N. X. Hoai, R. I. B. McKay, and D. Essam, "Representation and Structural Difficulty in Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 2, pp. 157–166, 2006.

[42] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2011.

[43] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[44] R. J. Urbanowicz and J. H. Moore, "Learning Classifier Systems: A Complete Introduction, Review, and Roadmap," *Journal of Artificial Evolution and Applications*, vol. 2009, no. 1, pp. 1–25, 2009.

[45] M. V. Butz and S. W. Wilson, "An Algorithmic Description of XCS," *Soft Computing*, vol. 6, no. 3-4, pp. 144–153, 2002.

[46] T. Kovacs, "Evolving Optimal Populations with XCS Classifier Systems," University of Birmingham, UK, Tech. Rep. CSR-96-17 and CSRP-9617, 1996.

[47] P. L. Lanzi, "Extending the Representation of Classifier Conditions Part I: From Binary to Messy Coding," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999, pp. 337–344.

[48] D. E. Goldberg, B. Korb, and K. Deb, "Messy Genetic Algorithms: Motivation, Analysis, and First Results," *Complex Systems*, vol. 3, no. 5, pp. 493–530, 1989.

[49] P. L. Lanzi and A. Perrucci, "Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999, pp. 345–352.

[50] L. Bull and T. O'Hara, "Accuracy-based Neuro And Neuro-Fuzzy Classifier Systems," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2002, pp. 905–911.

[51] H. H. Dam, H. A. Abbass, C. Lokan, and X. Yao, "Neural-Based Learning Classifier Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 1, pp. 26–39, 2008.

[52] M. Iqbal, W. N. Browne, and M. Zhang, "Evolving Optimum Populations with XCS Classifier Systems," *Soft Computing*, vol. 17, no. 3, pp. 503–518, 2013.

[53] P. L. Lanzi, "XCS with Stack-Based Genetic Programming," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2003, pp. 1186–1191.

[54] D. Loiacono, A. Marelli, and P. Lanzi, "Support Vector Machines for Computing Action Mappings in Learning Classifier Systems," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2007, pp. 2141–2148.

[55] S. W. Wilson, "Classifiers that Approximate Functions," *Natural Computing*, vol. 1, pp. 211–233, 2002.

[56] M. V. Butz, P. L. Lanzi, and S. W. Wilson, "Function Approximation With XCS: Hyperellipsoidal Conditions, Recursive Least Squares, and Compaction," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 3, pp. 355–376, 2008.

[57] P. L. Lanzi and D. Loiacono, "Classifier Systems that Compute Action Mappings," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2007, pp. 1822–1829.

[58] M. Ahluwalia and L. Bull, "A Genetic Programming Based Classifier System," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999, pp. 11–18.

[59] M. V. Butz, M. Pelikan, X. Llorà, and D. E. Goldberg, "Automated Global Structure Extraction for Effective Local Building Block Processing in XCS," *Evolutionary Computation*, vol. 14, no. 3, pp. 345–380, 2006.

[60] M. Iqbal, W. N. Browne, and M. Zhang, "Extracting and Using Building Blocks of Knowledge in Learning Classifier Systems," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2012, pp. 863–870.

[61] ——, "Learning Overlapping Natured and Niche Imbalance Boolean Problems Using XCS Classifier Systems," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2013, to appear.

[62] ——, "Comparison of Two Methods for Computing Action Values in XCS with Code-Fragment Actions," in *Proceedings of the Genetic and Evolutionary Computation Conference (Companion)*, 2013, to appear.

[63] ——, "XCSR with Computed Continuous Action," in *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, 2012, pp. 350–361.

[64] M. Iqbal, M. Zhang, and W. N. Browne, "Automatically Defined Functions for Learning Classifier Systems," in *Proceedings of the Genetic and Evolutionary Computation Conference (Companion)*, 2011, pp. 375–382.

[65] M. V. Butz, "XCSJava 1.0: An Implementation of the XCS Classifier System in Java," Illinois Genetic Algorithms Laboratory, Tech. Rep. 2000027, 2000.

[66] J. A. Walker, J. F. Miller, P. Kaufmann, and M. Platzner, "Problem Decomposition in Cartesian Genetic Programming," in *Cartesian Genetic Programming*. Springer, 2011, ch. 3, pp. 35–99.

[67] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

[68] W. B. Langdon, "Generalisation in Genetic Programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2011, p. 205.

[69] M. V. Butz, *Rule-based Evolutionary Online Learning Systems: A Principal Approach to LCS Analysis and Design*. Springer, 2006.

[70] R. Poli and J. Page, "Solving High-Order Boolean Parity Problems with Smooth Uniform Crossover, Sub-Machine Code GP and Demes," *Genetic Programming and Evolable Machines*, vol. 1, no. 1-2, pp. 37–56, 2000.

[71] N. Nisan and S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*. The MIT Press, 2008.

Muhammad Iqbal (M'11) received a Master in Computer Science degree from the National University of Sciences and Technology (NUST), Pakistan, in 2008. He joined COMSATS Institute of Information Technology (CIIT), Pakistan, as a faculty member in the Department of Computer Science, in August 2008. He is now working with Dr Will Browne and Prof Mengjie Zhang at the School of Engineering and Computer Science, Victoria University of Wellington, New Zealand. The research is on Learning Classifier Systems with particular emphasis on extracting and using building blocks of knowledge to develop a scalable classifier system. His areas of interest include learning classifier systems and evolutionary computation.

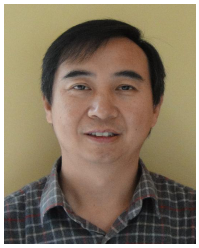
He is a member of the organizing committee for the International Workshop on Learning Classifier Systems (IWLCS) for 2013-14.

Mr Iqbal is a member of IEEE, the IEEE Computational Intelligence Society and the ACM SIGEVO Group. He has published a dozen academic papers in refereed international journals and conferences.

Will N. Browne received a BEng Mechanical Engineering, Honours degree from the University of Bath, UK in 1993, MSc in Energy (1994) and EngD (Engineering Doctorate scheme, 1999) University of Wales, Cardiff. After eight years lecturing in the Department of Cybernetics, University Redding, UK, he was appointed Senior Lecturer, School of Engineering and Computer Science, Victoria University of Wellington, NZ in 2008. Dr Browne's main area of research is Applied Cognitive Systems. This includes Learning Classifier Systems, Cognitive Robotics, Modern Heuristics for industrial application. Blue skies research includes analogues of emotions, abstraction, memories, Small-Worlds phenomenon, dissonance and machine consciousness.

He has served as Track co-chair of Genetics-based Machine Learning for Genetic and Evolutionary Computation Conference 2011-12. Organising committee of the International Workshop on Learning Classifier Systems (IWLCS) from 2009/2010. Editor-in-chief for the Australasian Conference on Robotics and Automation 2012 and organised the Cognitive Robotics Intelligence and Control EPSRC (UK)/NSF (USA) sponsored workshop 2006. Programme committee membership includes the LCS and other GBML track at Genetic and Evolutionary Computation Conference [2004-present], Congress on Evolutionary Computation [2004-present], Hybrid Intelligent Systems [2003-present], Parallel Problem Solving from Nature [2004-present] and Robot and Human Interactive Communication [2005-present]. Journal reviewer: Journal of Soft Computing, IEEE Transactions on Evolutionary Computation, IEEE Trans. Systems Man and Cybernetics, Journal of Engineering Manufacture, Journal of Pattern Analysis and Applications, Cognitive Computation.

Dr Browne is a member of IMechE, ACM and the ACM SIGEVO group. He has published over 50 academic papers in books, refereed international journals and conferences.



Mengjie Zhang (M04SM10) received a BE and an ME in 1989 and 1992 from Artificial Intelligence Research Center, Agricultural University of Hebei, China, and a PhD in computer science from RMIT University, Australia in 2000.

Since 2000, he has been working at Victoria University of Wellington, New Zealand. He is currently Professor of Computer Science and heads the Evolutionary Computation Research Group. His research is mainly focused on evolutionary computation, particularly genetic programming, particle swarm optimisation and learning classifier systems with application areas of image analysis, multi-objective optimisation, classification with unbalanced data, feature selection and reduction, and job shop scheduling. He has published over 250 academic papers in refereed international journals and conferences. He has been serving as an associated editor or editorial board member for five international journals (including IEEE Transactions on Evolutionary Computation and the Evolutionary Computation Journal) and as a reviewer of over fifteen international journals. He has been serving as a steering committee member and a program committee member for over eighty international conferences. He has supervised over thirty postgraduate research students.

Prof Zhang is a senior member of IEEE and a member of ACM. He is also a member of the IEEE CIS Evolutionary Computation Technical Committee, a member of the IEEE CIS Intelligent Systems and Applications Technical Committee, a vice-chair of the IEEE CIS Task Force on Evolutionary Computer Vision and Image Processing, and a member of IEEE CIS Task Force of Hyper-heuristics.