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 computational techniques have had limited capabilities in solving large-scale problems due to the large search space demanding large memory and much longer training time. 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 a 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 shown to be possible for the first time 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 computational 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 autonomously 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, enacting in 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 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 typical ternary alphabet based conditions in LCS will be replaced by code fragmented 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-B). The fitter building blocks from a learning system trained against smaller problems will be utilized in a higher complex problem in the same domain, similar to transferring knowledge in a transfer learning technique (see section II-A), in an attempt to achieve scalable learning.

The proposed system will be tested on four different Boolean problem domains, i.e. multiplexer, majority-on, carry, and even-parity problems. The multiplexer problem domain is highly non-linear and has epistasis property. In majority-on problem domain, the complete solution consists of strongly overlapping classifiers, which is therefore difficult to learn. Similar to 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 along with static numeric action, 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 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 con-
serve 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 using code fragmented conditions is detailed. Section IV introduces the problem domains and experimental setup 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 code fragmented XCS. In the ending section this work is concluded and the future work is outlined.

II. BACKGROUND

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 level 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 the 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, hypothetically the best, individuals selected 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 [12] where each individual member of the population is usually represented by a bitstring of fixed length, and represents a potential problem solution. The evolutionary process in GAs has been described using the concept of schema. A schema is a similarity template for describing a set of finite-length strings defined over a finite alphabet. For example, if the alphabet is \{0, 1, *\} then the schema “10**1” is describing all strings of length five that start with symbols ‘10’ and end with symbol ‘1’ such as ‘10001’, ‘10011’, ‘10101’, and ‘10111’. It is to be noted that ‘*’ is treated as ‘don’t care’ symbol here, meaning it can be either ‘0’ or ‘1’. The distance between first and last specific string positions in a schema \( H \) is called its defining length, denoted by \( \delta(H) \), and the number of specific positions in it is called its order, denoted by \( o(H) \). For example the defining length of the schema “10**1” is 4 and the order is 3.

Goldberg hypothesized that higher performance individuals are actually generated as a result of the combination of short-length, low-order and high-performance schemata [12]. These schemata are called the building blocks of the system. These building blocks are likely to be selected and combined via crossover to produce longer and fit individuals in a GA. These building blocks are also relatively less affected by mutation. The assumption by Goldberg that this is the way a GA works, is termed the building block hypothesis.

However, for a population of individuals represented by fixed length strings, the genetic operators sometimes can not 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 [13]. 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 [14]. Santana et al. [15] and Pelikan et al. [16] have incorporated transfer learning in the field of EDAs to transfer information between optimization problems.

The schema theory has been criticized due to its weak theoretical foundations [17–19], but still remains a popular tool to explain the power of GAs [20–22].

2) Genetic Programming: GP is an evolutionary approach to generating computer programs for solving a given problem automatically [23–25], and uses a much richer alphabet than GA 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.

To generate a computer program for regression and classification using GP, a set of (input, output) pairs is needed for training the candidate solutions along with sets of functions and terminals. GP attempts to construct a computer program that maps each of the (input, output) pairs correctly. For example, if the (input, output) pairs set is \{\{(0,1), (1,3), (2,7), (3,13), (4,21)\}\} and \{+, -, *, /\} are the function set and terminal set respectively, then the optimal corresponding GP generated program is as shown in Fig. 1. This GP tree is equivalent to the output expression \((x \times x) + (x + 1)\), where \(x\) is the input.

![Fig. 1. A GP generated tree program to map the set of (input, output) pairs \{\{(0,1), (1,3), (2,7), (3,13), (4,21)\}\}.](image)

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 \[26\]. The other ways to control bloat in genetic programming include simplifying individual programs using algebraic and numerical simplification methods \[27\], 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 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 \[34\]. 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 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. 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 \[3\]. 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 even-parity and majority-on problem domains, using a two-layered approach. The solutions of the bottom layer were encapsulated as parameterized modules and reused to learn the main task in the top layer. The layered learning approach outperformed standard monolithic GP \[23\] and GP with automatically defined functions (ADFs) \[37\], albeit it did not achieve 100\% success rate for the higher-order problems.

Hien et al. \[38\] investigated layered learning with incremental sampling in GP. They tested twelve symbolic regression problems and results were compared with standard GP \[23\]. 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 Hooi \[39\] 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. \[40\] investigated interactions between evolution, development, and layered learning using tree adjoining grammar guided GP (TAG3P) \[40\]. 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 compared with two single-short learning GP systems: standard tree-based GP \[23\] and the pre-existing TAG3P \[40\]. 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 \[41\], rather than on-line, reinforcement learning \[42\] 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, enacting in an unknown environment via a set of sensors for input and a set of effectors for actions. The rules are of the form “if state then action”. After observing the current state of the environment, the agent performs an action, and the environment provides a reward, as depicted in Fig. 2. An LCS is an adaptive system that, using the cooperative set of rules, learns to perform the best action receiving maximum reward from the environment for a given input.

There are two important families of LCSs: the Pittsburgh [43] and Michigan [44] approaches. Michigan classifier systems can be used for online learning as well as offline learning whereas the Pittsburgh classifier systems can only be applied to offline learning [45]. The Michigan approach is considered in this work, because an online learning system is potentially more suitable for scalability as it does not require the whole training set or any sub-sampling of the training set.

In a Michigan-style LCS, the population consists of a single set of co-operative rules, i.e. each individual represents a unique, distinct rule. The goal here is to find the best set of classifier rules that, when applied, gain an optimum result for the problem to be solved. Michigan-style LCSs have two main types of fitness definitions: strength-based, e.g. ZCS [46] and accuracy-based, e.g. XCS [4]. In ZCS the fitness of classifier rules depends on the magnitude of predicted rewards, whereas in XCS the fitness depends on the accuracy of predicted rewards. Due to the strength-based fitness, ZCS proliferates overgeneral classifier rules [47] in certain domains resulting in unsatisfactory performance. XCS is used to implement and test the proposed system as XCS evolves maximally general and accurate classifiers, attributed to different evolutionary pressures in it [2]. The resulting complete map of states to rewards is considered to contain more building blocks of knowledge than ZCS.

1) Accuracy Based Learning Classifier System: 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 [4]. 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 main parameters: 1) prediction $p$, an estimate of the payoff expected from the environment if its action is executed; 2) prediction error $e$, 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.

The agent has two modes of operation, explore (training) and exploit (application). In the following, XCS operations are concisely described. For a complete description, the interested reader is referred to the original XCS papers by Wilson [4], [5], and to the algorithmic details by Butz and Wilson [49].

In the explore mode, the agent attempts to obtain information about the environment and describes it by creating the decision rules, using the following steps:

- observes the current state of the environment, $s \in S$ where $S$ is the set of all possible environmental states. The current state $s$ is usually represented by a fixed length bitstring defined over the binary alphabet $\{0, 1\}$.
- selects classifiers from the classifier population $[P]$ that have conditions matching the state $s$, to form the match set $[M]$.
- performs covering: for every action $a_i \in A$ in the set of all possible actions, if $a_i$ is not represented in $[M]$ then a random classifier is generated with a given generalization probability such that it matches $s$ and advocates $a_i$, and added to the set $[M]$ as well as to the population $[P]$.
- The prediction, prediction error, and fitness of the generated classifier are set to very small initial values.
- forms a system prediction array, $P(a_i)$ for every $a_i \in A$ that represents the system’s best estimate of the payoff should the action $a_i$ be performed in the current state $s$. Commonly, $P(a_i)$ is a fitness weighted average of the payoff predictions of all classifiers advocating $a_i$.
- selects an action $a$ to explore (probabilistically or randomly) and selects all the classifiers in $[M]$ that advocated $a$ to form the action set $[A]$.
- performs the action $a$, records the reward $r$ received from the environment, and uses $r$ to update the associated parameters of all classifiers in $[A]$.

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

If the classifier population size grows larger than the specified limit, then one of the classifier rules has to be deleted so that the new rule can be inserted.
On receiving the environmental reward $r$, the parameters of each classifier $j$ in the action set $[A]$ are updated as follows. First of all, the experience $exp_j$ is increased by one. Then, the prediction error $\epsilon_j$ is updated: $\epsilon_j \leftarrow \epsilon_j + \beta(|r - p_j| - \epsilon_j)$ for $exp_j > 1/\beta$, otherwise $\epsilon_j \leftarrow [\epsilon_j * (exp_j - 1) + |r - p_j|]/exp_j$, where $0 \leq \beta \leq 1$ is the learning rate and $p_j$ is the prediction of the classifier $j$. Next, the prediction $p_j$ is adjusted: $p_j \leftarrow p_j + \beta(r - p_j)$ for $exp_j > 1/\beta$, otherwise $p_j \leftarrow [p_j * (exp_j - 1) + r]/exp_j$. After that, the classifier’s accuracy is computed: $k_j = \alpha(\epsilon_j/\epsilon_0)^{-\nu}$ for $\epsilon_j \geq \epsilon_0$, otherwise 1. The parameter $\epsilon_0(\epsilon_0 > 0)$ determines the threshold error under which a classifier is considered to be accurate, providing robustness to noise. The parameters $\alpha(0 < \alpha < 1)$ and $\nu(\nu > 0)$ are used to handle the degree of decline in accuracy.\footnote{Currently only single step problems are under investigation so the parameter updates being described here are for single step problems. For multi-step problems, parameter updates occur in the previous action set $[A]_{-1}$, as described in [2, 5].} The parameter $\nu$ separates rules of similar fitness to increase the probability for selection of better rules.

Then, the relative accuracy $k_j$ is computed by dividing the accuracy $k_j$ by the total amount of accuracies in the action set. Finally, the fitness $F_j$ is updated according to the classifier’s relative accuracy: $F_j \leftarrow F_j + \beta(k_j - F_j)$. The relative accuracies based fitness update mechanism implicitly exhibits fitness sharing among the classifiers in an action set. Fitness sharing allocates resources to niches evenly, i.e. unbalanced classes or complex classes do not get ignored.

- when appropriate, implements rule discovery by applying an evolutionary mechanism (commonly a GA) in the action set $[A]$, to introduce new classifiers to the population.

Additionally, the explore mode may perform subsumption to merge specific classifiers into any more general and accurate classifiers. There are two subsumption procedures in XCS: (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: (i) has an experience value greater than a threshold, (ii) is accurate, and (iii) 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 [2, 5]. It also effectively reduces the number of classifier rules in the final population [51].

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.

The generalization property in LCS allows a single rule to cover more than one state provided that the action-reward mapping is similar. Traditionally, generalization in LCS classifier conditions is achieved by the use of a special ‘don’t care’ symbol ($) in the ternary representation, which matches any value of a specified attribute in the vector describing the state $s$. The next section presents various other rich representations that have been successfully used in LCS.

2) LCS with Rich Encoding Schemes: Various richer encoding schemes have been investigated in the LCS research community 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 the most tested and often best performing model of learning classifier systems.

A GP-based rich encoding has been used by Ahluwalia et al. [52] within a simplified strength-based learning classifier system [46]. They used binary strings to represent 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.

Lanzi extended the fixed length bitstrings representation of classifier conditions to a variable length messy coding in [53]. A messy coded string may be over- or under-specified, due to its variable length structure [54]. 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 [55] enhanced a step further from messy coding to a more complex representation in which S-expressions were used to represent general classifier conditions.

Lanzi presented an XCS with stack-based genetic programming [56] 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 redundant. Even then, it is reported that the system was able to learn multiplexer and woods1 problems. In the work presented here, only syntactically correct conditions are allowed to be produced by the system.

Butz et al. incorporated the EDA mechanism in XCS to identify and process building blocks for solving hierarchical decomposable binary classification problems [57]. They have used extended compact GA (ECGA) and the Bayesian optimization algorithm (BOA) to estimate the probability of distribution. In domains containing building blocks, this approach has shown the benefits of not using the potentially destructive crossover operation.

Charalambos and Browne [3] investigated scaling of an abstracted LCS by implementing classifier conditions as a...
combination of ternary and S-expression alphabets, and using pre-constructed functions for a specific problem domain. By using domain-relevant functions the scalability of XCS was shown to be improved, but without the domain knowledge the appropriate functions for a problem need to be automatically discovered [3].

Lanzi and Loiacono [58] 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 typical LCS. Then, Loiacono et al. [59] extended XCSCA using support vector machines to compute the classifier action that resulted in reaching optimal performance faster than the original XCSCA.

All of the above mentioned encoding schemes, to represent a classifier’s knowledge, have their own advantages and limitations. Thus, we investigated the introduction of code fragments into LCS, which has culminated in the work presented in this paper.

3) Previous Work on Code Fragmented XCS: In summary, a GP-like rich encoding scheme is needed in an attempt to extract the building blocks of knowledge and to reuse them to learn complex problems in the domain to achieve scalability. Previously, we implemented this scheme to encode the action of a classifier in order to successfully evolve optimal populations in discrete domain problems [60] as well as in continuous domain problems [61], but this did not lead to simple scaling. In our previous code fragmented conditions work [62], [63], we used a separate population of code fragments, which was inefficient and limited the number of available code fragments, resulting in a system not learning the highly complex problems.

In detail, our previous work introduced GP-tree like expressions to represent condition bits in a classifier rule, named code fragmented conditions [62] [63]. This initial investigation of code fragments in XCS showed that the multiple genotypes to a phenotype issue in feature rich encodings disabled the subsumption deletion function. The additional methods and increased search space also lead 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 problem. The code fragments also created masks that autonomously subdivided the search space into areas of interest and uniquely, areas of not interest.

Next, building blocks of knowledge were extracted, in the form of code fragments, from small scale problems and reused to learn large scale problems [63]. The proposed approach of code fragmented XCS reusing the extracted knowledge outperformed ternary alphabet based XCS in multiplexer, carry, and even-parity problem domains in terms of improving effectiveness and reducing instances in large 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 a separate population of code fragments was created and kept static throughout the learning process [62], [63]. This puts a limit on the number of available code fragments to be used in 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 [63], 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 FRAGMENTED CONDITIONS

In the 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 fitness value greater than the average fitness of the classifier population.

In the proposed XCS with code fragmented 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, ... Dn-1\} where n is the length of condition in a classifier rule. A population of classifiers having code fragmented conditions is illustrated in Table I. The symbols \&., |, ., r and ~ denotes AND, OR, NAND, NOR, and NOT operators respectively. The code fragments are shown in postfix form.

The proposed code fragmented XCS extends the standard XCS, described in section II-C1, in the following cases: 1) the classifier matching procedure, the covering operation, the rule discovery operation, the subsumption deletion mechanism, 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. Suppose the problem instance s is 110101 then the code fragment shown in Fig. 3 will give 1 as the output. This

---

4In [62], the GP-tree like expressions were called automatically defined functions (ADFs), due to the resemblance with ADFs used in GP domain.
output value was generated by loading D0, D1, D2, and D5 with 1, 1, 0, and 1 respectively.

![Diagram of code fragments](image)

Fig. 3. 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. 4. This code fragment always outputs 1.

![Diagram of 'don’t care' symbol](image)

Fig. 4. 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 position within the condition, i.e. unlike 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. 5.

B. Covering Operation

Covering occurs if an action is missing in the match set \( \mathcal{M} \). 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 \( P_{\text{don’tCare}} \). 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. 6.
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 \(\chi\) using a two point crossover operation, 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. 7. After that, each code fragment in the conditions of the crossed over children is mutated with probability \(\mu\), such that both children match the currently observed state \(s\). In mutation operation, a ‘non-don’t care’ code fragment 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 \(\mu\). The mutation operation is described in Fig. 8. The prediction of the offspring is set to the average of the parents’ values whereas the prediction error and fitness of the offspring are set to the average of the parents’ values reduced by constants \(\text{predictionErrorReduction}\) and \(\text{fitnessReduction}\) respectively, as in [64].

\begin{verbatim}
1: procedure Crossover_Operation(cl1, cl2)
  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 cl1.cond[i] and cl2.cond[i]
  9:     end for
10: end procedure
\end{verbatim}

Fig. 7. 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 in a rule and the corresponding feature in the problem pair. Although this can lead to compaction of a rule, it also places additional pressure on 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 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. 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 will 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. 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 code fragmented XCS, 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 \(cl_1\) are greater 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. 9.

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. 10.
Fig. 9. 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 $cl_2$.

```plaintext
1: procedure IS_MORE_GENERAL($cl_1, cl_2$)
2:   $x \leftarrow$ number of ‘don’t care’ code fragments in $cl_1$
3:   $y \leftarrow$ number of ‘don’t care’ code fragments in $cl_2$
4:   if $x \leq y$ then
5:     return false
6:   end if
7:   $X \leftarrow$ set of all ‘non-don’t care’ code fragments in $cl_1$
8:   $Y \leftarrow$ set of all ‘non-don’t care’ code fragments in $cl_2$
9:   if $X \not\subseteq Y$ then
10:      return false
11:   end if
12: return true
13: end procedure
```

Fig. 10. 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.

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

Fig. 11. A sample of the code fragments for 20-bits multiplexer problem. The code fragments of 6-, 11-, and 20-bits multiplexer problems are named as $L_{1,n}$, $L_{2,n}$, and $L_{3,n}$ 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, majority-on problems, carry problems, and even-parity problems.

A multiplexer is an electronic circuit that accepts $n$ inputs, of the form $k + 2^k$, and gives one output. The values of $k$ address bits are used to select one of the $2^k$ data bits to be given as output. For example in 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. Multiplexer problems are considered to be interesting because they are highly non-linear and have epistasis properties, therefore, relatively difficult to learn. “They are non-trivial high dimensional deceptive and discrete. They have no parameters suitable for continuous gradient ascent” [66]. They also allow generalizations and are suitable for examining the scalability of an algorithm.

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 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.\footnote{Here, * can be 0, 1, or #.}

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 majority-on problems, the complete solution in carry problem domain consists of overlapping classifiers, and in addition it is a niche imbalance problem domain.

The even-parity problems are similar to 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, and zero otherwise. Using the ternary alphabet based conditions with the static numeric action, no useful generalizations can be made for the even-parity problems.

\section{B. Experimental Setup}

The system uses the following parameter values, which are commonly used in the literature, as suggested by Butz in \footnote{In simple problems the conventional parameter setting produce robust performances, but require adjustment in complex domains, e.g. 37-bits MUX and above.}, and by Butz and Wilson in \cite{49}: 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 $P_{don'tC} = 0.33$; initial prediction $p_{t1} = 10.0$; initial prediction error $e_{t1} = 0.0$; initial fitness $F_{t1} = 0.01$; reduction of the prediction error $\text{predictionErrorReduction} = 0.25$; reduction of the fitness $\text{fitnessReduction} = 0.1$; 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 \{ $\text{AND, OR, NAND, NOR, NOT}$ \}, 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.

\section{V. Results}

In order to test the performance of the code fragmented alphabet, results of the proposed scheme have been compared with the standard ternary alphabet based implementation of XCS on the four problem domains used in experimentation. In addition, results obtained from some of the GP-systems in even-parity and multiplexer problem domains have been compared with the proposed system.

Each result obtained in this work is the average of the 30 independent runs, with a different seed in each run. In all graphs presented here, the X-axis is the number of problem instances used as training examples and the Y-axis is the performance measured as the percentage of correct classification during the last 100 exploit problem instances. This is different to standard supervised learning batch processing GP approaches, due to the online nature and descriptive purpose of LCS.

\subsection{A. Results Comparison with XCS}

1) The Multiplexer Problem Domain: The performance of standard XCS and code fragmented XCS methods in the multiplexer problem domain is compared as the problem scales, see Fig. 12. The number of classifiers used, denoted by $N$, is 500, 1000, 2000, 5000, 10000, and 50000 for 6-, 11-, 20-, 37-, 70-, and 135-bits multiplexer problems respectively. The number of training examples used is half a million for 6-, 11-, and 20-bits multiplexers and one million, two million, and five million for 37-, 70-, and 135-bits multiplexer problems respectively. The standard XCS was not able to solve 37-bits MUX problem with $P_{don'tC} = 0.33$ and $N = 5000$, so $P_{don'tC}$ was increased to 0.5 in Fig. 12(b). For 70-bits and 135-bits MUX problems, $P_{don'tC}$ is set to 1.0 and $\mu$ is set to 0.01 in the standard XCS. The condition length of a classifier in the proposed XSCFC is set to $70/2 = 35$, and $135/4 = 33$ for 70-bits and 135-bits MUX problems respectively. The standard XCS failed 13 times out of 30 runs to solve 70-bits MUX problem with $N = 10000$, so $N$ was increased to 20000, Fig. 12(c). Here p# and N denote the probability of ‘don’t care’ symbol and the number of classifiers used respectively.

The code fragmented XCS needs more training examples in comparison with standard XCS to learn 6-bits and 11-bits MUX problems, but less training examples for 20-bits MUX problem, as shown in Fig. 12(a). The standard XCS, with parameter tuning, needs approximately 800k and 3000k problem instances to solve 37-bits and 70-bits MUX problems, see Fig. 12(b) and Fig. 12(c) respectively. Whereas the proposed code fragmented XCS takes approximately 200k and 500k problem instances to solve 37-bits and 70-bits MUX problems respectively, without parameter tuning. The performance curves for 70-bits MUX problem using the proposed XSCFC approach are almost coincident in Fig. 12(c).

The standard XCS was not able to solve the 135-bits MUX problem, either in literature or with further parameter tuning conducted here. However, if a stepped reward function is used to guide learning \footnote{In simple problems the conventional parameter setting produce robust performances, but require adjustment in complex domains, e.g. 37-bits MUX and above.} then the state-of-the-art in the field was to solve the 135-bits MUX problem. The code fragmented XCS using the extracted domain knowledge has successfully solved the standard 135-bits MUX problem taking approximately two million training instances Fig. 12(d) without needing stepped reward. Considering the number of possible instances is $2^{135} \approx 4 \times 10^{40}$, and that the proposed approach XSCFC takes only $2 \times 10^6$ instances (i.e. sampling only one in $10^{34}$ instances) to be able to solve the problem, this result is remarkable.

2) The Majority-on Problem Domain: The performance of standard XCS and code fragmented XCS methods in the majority-on problem domain is shown in Fig. 13. The number
of classifiers used is 500, 1000, and 2000 for 3-, 5-, and 7-bits majority-on problems respectively. The number of training examples used is half a million.

**3) The Carry Problem Domain:** The performance of standard XCS and code fragmented XCS methods in the carry problem domain is shown in Fig. 14. The number of classifiers used is 1000, 2000, 4000, and 6000 for 2-, 3-, 4-, and 5-bits carry problems respectively. The number of training examples used for 2-, and 3-bits carry problems is half a million whereas for 4-, and 5-bits carry problems one million training examples have been used.

To test statistical significance of the proposed approach with comparison to the 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 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 code fragmented approach is statistically significant as in both cases the p-value is much less than 0.05.

### Table II

<table>
<thead>
<tr>
<th>Majority-On</th>
<th>Standard XCS</th>
<th>Code Fragmented XCS</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-bits</td>
<td>95.17 ± 2.49</td>
<td>100.00 ± 0.00</td>
<td>3.54e-06</td>
</tr>
<tr>
<td>7-bits</td>
<td>94.43 ± 2.65</td>
<td>100.00 ± 0.00</td>
<td>1.66e-06</td>
</tr>
</tbody>
</table>

The Wilcoxon signed rank test for performance comparison in the majority-on problem domain.

![Fig. 12. Results of the multiplexer problems using the standard XCS and the proposed code fragmented XCS. The performance curves for 70-bits MUX problem using the proposed XCSCFC approach are almost coincident in (c).](image)

![Fig. 13. Results of the majority-on problems using the standard XCS and the proposed code fragmented XCS.](image)
The results of the Wilcoxon signed rank test conducted to measure the statistical significance of the proposed approach with comparison to the standard XCS are shown in Table III. The values in column two and column three are the average performance values of last 100 test cases along with the standard deviation. The performance improvement of code fragmented approach is statistically significant as in all cases the p-value, obtained with confidence interval of 95%, is far less than 0.05.

**TABLE III**

<table>
<thead>
<tr>
<th>Carry</th>
<th>Standard XCS</th>
<th>Code Fragmented XCS</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-bits</td>
<td>99.17 ± 1.19</td>
<td>100.00 ± 0.00</td>
<td>7.80e−3</td>
</tr>
<tr>
<td>3-bits</td>
<td>96.30 ± 2.67</td>
<td>100.00 ± 0.00</td>
<td>1.66e−3</td>
</tr>
<tr>
<td>4-bits</td>
<td>93.47 ± 2.60</td>
<td>100.00 ± 0.00</td>
<td>1.64e−10</td>
</tr>
<tr>
<td>5-bits</td>
<td>92.10 ± 3.12</td>
<td>99.87 ± 0.43</td>
<td>1.59e−3</td>
</tr>
</tbody>
</table>

4) The Even-Parity Problem Domain: The performance of standard XCS and code fragmented XCS methods in even-parity problem domain is shown in Fig. 15(a). The number of classifiers used is 200, 300, 400, 500, 1000, and 2000 for 2-, 3-, 4-, 5-, 6-, and 7-bits problems respectively. Each run is stopped after half a million training examples.

It is observed that the code fragmented XCS needs more training examples in comparison with standard XCS to learn 2-, 3-, and 4-bits even-parity problems Fig. 15(a) As the problem scales to 6-bits, standard XCS cannot learn the even-parity problem Fig. 15(b) using the experimental setup given in section V-B whereas the code fragmented XCS reusing the extracted domain knowledge has solved up to 7-bits even-parity problems successfully.

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

The even-parity problem domain does not allow generalizations if the standard ternary alphabet based encoding scheme is used along with the numeric action. So each bit must be specific for a rule to be accurate. For small scale problems, it is relatively easy to learn each bit. 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 the code fragmented rich encoding scheme, the number of ‘specific’ code fragments is essentially messy as the system can choose how many ‘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 in a rule and the corresponding feature in the problem pair. Therefore, the proposed code fragmented XCS, 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 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 code fragmented XCS, a classifier rule is said to be matched against an environmental instance 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 instance 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 instance. So, this general rule is equivalent to two specific rules: ‘011 : 1’ and ‘101 : 1’. The generalization ability of the proposed approach 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 a different way, 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 GP systems or other layered learning approaches, and it is not straightforward to compare the proposed system with a GP system. However, some attempt at comparisons 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 parameterized 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 4-, 5-, and 6-bits even-parity problems having the function set \{AND, OR, NAND, NOR\}. Each experimented was repeated 100 times with maximum 50 generations in each run. The population size used for 4-bits even-parity problem was 500 and it was increased to 2000 for 5-, and 6-bits even-parity problems. The layered learning approach outperformed the standard monolithic GP [23] and the GP with ADFs [37], 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].

<table>
<thead>
<tr>
<th>Problem</th>
<th>GP</th>
<th>GP with ADFs</th>
<th>LLGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bits</td>
<td>14</td>
<td>43</td>
<td>93</td>
</tr>
<tr>
<td>5-bits</td>
<td>0</td>
<td>32</td>
<td>92</td>
</tr>
<tr>
<td>6-bits</td>
<td>16</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

To compare the proposed approach 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 2-, 3-, 4-, 5-, and 6-bits even-parity problems respectively. The number of training examples used is half a million. The performance of the XCSCFC system for 2-bits to 6-bits even-parity problems is shown in Fig. 16. The XCSCFC solved all these problems successfully in each of the conducted experiments.

The second comparison is with the LLGP system for the majority-on problems. The 3-bits majority-on problem was used at the bottom layer to solve 5-, and 7-bits majority-on problems having the function set \{AND, OR, NOT\}. Each experimented was repeated 100 times with maximum 50 generations in each run. The population size used for 5-bits majority-on problem was 500 and it was increased to 1000 for the 7-bits problem. The layered learning approach outperformed the standard monolithic GP [23] and the GP with ADFs [37], 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].

<table>
<thead>
<tr>
<th>Problem</th>
<th>GP</th>
<th>GP with ADFs</th>
<th>LLGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-bits</td>
<td>62</td>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>7-bits</td>
<td>18</td>
<td>not attempted</td>
<td>90</td>
</tr>
</tbody>
</table>

To compare the proposed approach XCSCFC with the LLGP system for the majority-on problems, the function set was changed to \{AND, OR, NOT\}. The number of classifiers used is 500, 1000, and 2000 for 3-, 5-, and 7-bits majority-on problems respectively. The number of training examples used is half a million. The performance of the XCSCFC system for 3-, 5-, and 7-bits majority-on problems is shown in Fig. 17. The XCSCFC solved all these problems successfully in each of the conducted experiments.

The third comparison is with the DTAG3P system developed by Hoang et al. [33]. Using the DTAG3P system, 8-bits even-parity problem was experimented in a layered learning approach, but it was not able to solve the problem successfully.
Fig. 17. Results of the majority-on problems obtained using the proposed approach XCSCFC with function set \{AND, OR, NOT\}.

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 it outperformed the two single-short learning GP systems, the standard tree-GP system \[23\] and the TAG3P system \[40\], it could not achieve 100% success rate for the 8-bits even-parity problem. The reported success rates are 6.67%, 10%, and 86.67% for the tree-GP, the TAG3P, and the DTAG3P systems respectively \[33\].

To compare the proposed approach XCSCFC with the DTAG3P system, its function set was changed to \{AND, OR, NOT, XOR\}. The number of classifiers used is 200, 300, 400, 500, 1000, 1500, and 2000 for 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 the XCSCFC system for 2-bits to 8-bits even-parity problems is shown in Fig. 18. The XCSCFC solved all these problems successfully in each of the conducted experiments.

Poli and Page \[68\] 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 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 \[69\] in the function set. 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 proposed approach of code fragmented conditions has solved up to 135-bits multiplexer problems by extracting and reusing the building blocks of the domain knowledge. The reuse of extracted knowledge has shown generalization ability in even-parity domain problems that is not possible using standard ternary alphabet-based representation. The following section describes in detail the reuse of the extracted knowledge in multiplexer and even-parity problem domains. This is followed by a discussion of messy code fragmented conditions.

A. Reuse of Extracted Knowledge

A classifier rule from the final rule base of 20-bits multiplexer problem is depicted in Fig. [19] 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 20MUX code fragments are using three building blocks of knowledge, in the form of code fragments, from 6MUX (i.e. Level 1), namely $L_1\_29$, $L_1\_12$, and $L_1\_21$, and one from 11MUX (i.e. Level 2), namely $L_2\_3$ that is further using a code fragment from 6MUX, namely $L_1\_6$.

Table VII.

<table>
<thead>
<tr>
<th>Condition</th>
<th>$A$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1_29$</td>
<td>$D_1$</td>
<td>$D_2$</td>
</tr>
<tr>
<td>$L_1_12$</td>
<td>$D_0$</td>
<td>$D_2$</td>
</tr>
<tr>
<td>$L_1_21$</td>
<td>$D_3$</td>
<td>$D_1$</td>
</tr>
<tr>
<td>$L_1_29$</td>
<td>$D_1$</td>
<td>$D_2$</td>
</tr>
</tbody>
</table>

Fig. 18. Results of the even-parity problems obtained using the proposed approach XCSCFC with function set \{AND, OR, NOT, XOR\}.

Fig. 19. A classifier rule from the final rule base obtained for a typical run of 20-bits multiplexer problem.

In code fragmented XCS, a classifier rule is said to be matched against a problem instance if the computed value of all the code fragments in the classifier’s condition is equal to 1. Fourth code fragment ‘$L_1\_21$’ in the classifier rule shown in Fig. [19] is just $D_2$, therefore, $D_2$ must be 1 in the environmental instance to be matched by this classifier.

The first code fragment ‘$L_1\_29$’ is using three environmental features, i.e. $D_1$, $D_2$, and $D_4$. Now, $D_2$ is necessarily to be 1 in the environmental instance to be matched by this classifier, so the code fragment ‘$L_1\_29$’ will output 1 if and only if the value of the feature $D_1$ is 1, as shown in Table VII.

The second code fragment ‘$L_1\_12$’ is using three environmental features, i.e. $D_0$, $D_1$, and $D_3$. Now, $D_1$ is necessarily to be 1 in the environmental instance to be matched by this classifier, so the code fragment ‘$L_1\_12$’ will output 1 if and only if $D_0 = 1$ and $D_3 = 0$, as shown in Table VIII.
The third code fragment 'L2_3 D18 & ∼ = D0 D9 | D1 & D18 & ∼' is using four environmental features, i.e. D0, D1, D9, and D18. Now, D0 and D1 are necessarily to be 1 in the environmental instance 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>
<thead>
<tr>
<th>Sr. No.</th>
<th>D0</th>
<th>D1</th>
<th>D9</th>
<th>D18</th>
<th>L1_12</th>
<th>L1_12D0d</th>
<th>L1_12D0d~</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Therefore, the classifier rule 'L1_29D1dD4D2 | d' is using 4 code fragments occurring in the condition in the new alphabet of ternary representation. This classifier is maximally general and accurate, equivalent to the classifier '1110' represented in ternary alphabet-based form.

To illustrate the generalization ability of the proposed approach in even-parity problems, a classifier rule from the final rule base of 4-bits even-parity problem, depicted in Figure 20, 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. This is a compact rule, using just two code fragments. These 4EP code fragments are using one code fragment from 3EP (i.e. Level 2), namely L2_4 that is further using a code fragment from 2EP (i.e. Level 1), namely L1_7.

In code fragmented XCS, a classifier rule is said to be matched against a problem instance if the computed value of all the code fragments in the classifier’s condition is equal to 1. First code fragment 'D3 ~' in the classifier rule shown in Fig. 20 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 & | ∼' is using 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 like D0 = D1 is not expressible in ternary alphabet-based representation.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>D0</th>
<th>D1</th>
<th>D9</th>
<th>L1_7</th>
<th>L1_7D0</th>
<th>L1_7D0d</th>
<th>L1_7D0d~</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Therefore, the classifier rule 'D3 ~ L2_4: l' 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 3-bits even-parity problem where L1_4 and L1_7 are 2-bits even-parity problem code fragments given by D0D0rD1D1& and D1D0|D1D0d respectively. In this rule, 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. This rule will match against an environmental instance if the corresponding output value for both code fragments D2D2|L1_4| and D2L1_7d is 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'.

### B. Messy Code Fragmented Conditions

In the proposed approach of code fragmented conditions, there is no linking between the position of a condition in a rule and the corresponding feature in the problem pair. Therefore, it is not necessary for a classifier’s condition to have the same number of code fragments as problem features, i.e. different number of code fragments can be used for 6-bits multiplexer problem as shown in Fig. 21.
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.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D2 &amp; L1</th>
<th>L1_4</th>
<th>D2 &amp; D1 &amp; L1_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE XII

Truth table for the code fragment 'D2 L1_7 |', where 'L1_7' outputs 1 if and only if D0 and D1 have different values.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>L1_7</th>
<th>D2 &amp; L1_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 21. Performance of XCS with code fragmented conditions, using different number of code fragments in the condition of a classifier rule, for 6-bits multiplexer problem (curve order same as in legend).

It is observed that 6-bits multiplexer problem can be addressed using different number of code fragments in a classifier’s condition, as shown in Fig. 21 but to solve it effectively a minimum of three code fragments should be used in the classifier’s condition. If more than the minimum required code fragments are used in the condition, then performance is robust for 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 problems in the domain. For example, in the 135-bits multiplexer problem where the number of possible instances is \(2^{135} \approx 4 \times 10^{66}\), the proposed approach takes only \(2 \times 10^6\) instances (i.e. sampling only one in \(10^{44}\) instances) to solve the problem.

The proposed approach of code fragmented XCS, using a GP-like rich encoding scheme, has shown the generalization ability in even-parity domain problems that is not expressible using the standard ternary alphabet-based representation.

The proposed approach of XCS with code fragmented conditions is currently tested for only Boolean problem domains. 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 the proposed approach uses static code fragments, extracted from smaller problems to generate code fragments in the higher level complex problem in the domain. A mechanism is needed to introduce plausibly better code fragments as training progresses, without disrupting existing classifiers.

Although previously not learnable by evolutionary computing techniques, the 135-bits multiplexer problem is solved using the proposed code fragmented XCS. However, the results obtained cannot be mathematically 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 the terminal set as present. It is anticipated that using the extracted domain level knowledge from multiple problem domains will result in a general scalable classifier system.

REFERENCES
