Extending Learning Classifier System with Cyclic Graphs for Scalability on Complex, Large-Scale Boolean Problems

Muhammad Iqbal  
Victoria University of Wellington, NZ  
muhammad.iqbal@vuw.ac.nz

Will N. Browne  
Victoria University of Wellington, NZ  
will.browne@vuw.ac.nz

Mengjie Zhang  
Victoria University of Wellington, NZ  
mengjie.zhang@ecs.vuw.ac.nz

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. Recently work has begun on autonomously reusing learnt building blocks of knowledge to scale from low dimensional problems to large-scale ones. An XCS-based classifier system has been shown to be scalable, through the addition of tree-like code fragments, to a limit beyond standard learning classifier systems. Self-modifying cartesian genetic programming (SMCGP) can provide general solutions to a number of problems, but the obtained solutions for large-scale problems are not easily interpretable. A limitation in both techniques is the lack of a cyclic representation, which is inherent in finite state machines. Hence this work introduces a state-machine based encoding scheme into scalable XCS, for the first time, in an attempt to develop a general scalable classifier system producing easily interpretable classifier rules. The proposed system has been tested on four different Boolean problem domains, i.e. even-parity, majority-on, carry, and multiplexer problems. The proposed approach outperformed standard XCS in three of the four problem domains. In addition, the evolved machines provide general solutions to the even-parity and carry problems that are easily interpretable as compared with the solutions obtained using SMCGP.

Categories and Subject Descriptors
F.1.1 [Models of Computation]: Genetics-Based Machine Learning, Learning Classifier Systems

General Terms
Algorithms, Performance

Keywords
Learning Classifier Systems, XCS, Finite State Machines, Genetic Programming, Scalability, Pattern Recognition

1. INTRODUCTION
A learning classifier system (LCS) is a rule-based learning system that adaptively learns a task by interacting with an unknown environment and uses evolutionary computing to evolve the rules according to the reinforcement received. The LCS technique can scale in a problem domain, but as the problem increases in dimensionality, resulting in increased search space, it becomes difficult to solve due to the needed resources. By explicitly feeding domain knowledge to an LCS, scalability can be achieved, but it adds bias and restricts use in multiple domains [1].

A main research direction is to develop an autonomously scalable classifier system that can provide general solutions, i.e. solutions that can be scaled to any problem size in the domain. A code-fragment based XCS system has been developed by Iqbal et al. [15], using a genetic programming (GP) like rich encoding scheme and a layered learning approach. Although the system has shown scalability beyond standard XCS [25], the solutions found were not scalable in general at the domain level due to the large sized nested tree structures.

The self-modifying cartesian genetic programming (SMCGP) is a developmental graph-based GP system where a computer program is evolved that can generate an arbitrary sequence of computer programs (see section 2.3). The graph-based representation has benefits of implicitly reusing the nodes in the graph. Harding et al. [10] have evolved programs using SMCGP that were proved to provide general solutions for a number of problems. The issue with SMCGP is that the size of the obtained graphical solutions to Boolean problems can increase for every next level problem in the domain. This makes the visualization and interpretation of the solution difficult for large scale problems in such domains.

A finite-state-machine (FSM) is a cyclic mathematical model that can be used to represent any finite-state system (see section 2.1). The advantage of cyclic representations is that they can map repeated patterns in a compact form, so this work introduces a state-machine based rich encoding scheme into scalable XCS, for the first time, in an attempt to develop a general scalable classifier system producing concise and easily interpretable classifier rules. The proposed system is to be tested on four different Boolean problem domains, i.e. even-parity, majority-on, carry, and multiplexer problems. These are complex problem domains having overlapping, niche imbalance, and epistatic properties (see section 4). The results are to be compared...
with standard numeric action based XCS and SMCGP to
test the scalability of the proposed system.

2. BACKGROUND

2.1 Finite State Machines

A finite state machine (FSM), also known as finite state
automaton (FSA), is a mathematical model of computation
that can be used to model any finite-state system. FSMs
have been used to design sequential logic circuits as well
as algorithms for different computational tasks such as pat-
tern matching, sequence prediction, communication proto-
cols, and parsing [2]. In general, a finite state machine con-
sists of a set of finite states and can be in only one state at
any given time, called the current state. On receiving an
input, the machine can change its current state and/or cause
an action or output to take place for any given change. One
of the states is labeled as start state, which is used as the
current state in the beginning while processing the input.

There are many state machine modeling techniques, e.g.
deterministic finite automata (DFA), non-deterministic fi-
nite automata (NFA), Mealy machines, Moore machines,
pushdown automata, and Turing machines. The state ma-
chine used in the work presented here is the Moore ma-
chine [23], as it is simple and suitable for modeling classifi-
cation problems. A Moore machine is formally defined as a
six-tuple \( M = (Q, \Sigma, \Delta, \delta, \lambda, q_0) \) where \( Q \) is a finite set of
states, \( \Sigma \) is a finite set of input symbols, \( \Delta \) is a finite set of
output symbols, \( \delta \) is a transition function from a state \( p \in Q \)
and an input \( x \in \Sigma \) to a next state \( q \in Q \), \( \lambda \) is an output
function from a state \( p \in Q \) to an output \( y \in \Delta \), and \( q_0 \in Q \)
is the start state.

For example, the state machine shown in Fig. 1 is de-
scribed as \( Q = \{q_0, q_1, q_2\}, \Sigma = \{0, 1\}, \Delta = \{0, 1\}, \) start
state \( q_0 = 0 \), \( \lambda \) defined as: \( \lambda(q_0) = 1, \lambda(q_1) = 1, \) and \( \lambda(q_2) = 0, \)
and \( \delta \) defined as: \( \delta(q_2, 0) = q_2, \) \( \delta(q_0, 1) = q_1, \) \( \delta(q_1, 0) = q_1, \) \( \delta(q_1, 1) = q_0, \) \( \delta(q_2, 0) = q_1, \) and \( \delta(q_2, 1) = q_2. \) Here a circle
denotes a state along with the corresponding output value,
and an arc represents a transition.

![A Moore state machine with three states](image)

Figure 1: A Moore state machine with three states
\( q_0, q_1, \) and \( q_2 \) where \( q_0 \) is the start state.

Usually, a Moore machine outputs a string of symbols
from \( \Delta \) on receiving an input string of symbols from \( \Sigma, \) e.g.
if the input is ‘10100’ then the Moore machine depicted in
Fig. 1 will produce ‘11101’ as the output string. In the work
presented here, it is adapted for classification problems. To
use a Moore machine for classification, the value of the last
state visited while processing the input string is taken as
the only output, instead of a whole output string. Therefore
the class of input string ‘10100’ will be 1 whereas the string
‘10101’ will belong to class 0, if processed via the machine
shown in Fig. 1.

2.2 Learning Classifier Systems

Traditionally, an LCS represents a rule-based agent that
incorporates evolutionary computing and machine learning
to solve a given task by interacting with an unknown en-
vironment. After observing the current state of the envi-
rone
ment, the agent performs an action and the environment
provides a reward.

XCS [23] 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. In XCS, the learning agent
evolves a population \( P \) of classifiers, where each classifier
consists of a rule and a set of associated parameters esti-
mating 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 rep-
resented by a fixed length bitstring defined over the ternary
alphabet \( \{0, 1, \#\} \) where ‘\#’ is the ‘don’t care’ symbol which

XCS operates in two modes, explore and exploit. 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 \( s \) of the environment, 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 gen-
  eralization probability such that it matches \( s \) and ad-
  vocates \( a_i \), and added to the set \( M \) as well as to the
  population \( P \).
- 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 state \( s \).
- selects an action \( a \) to explore (probabilistically or ran-
  domly) and selects all the classifiers in \( M \) that advo-
  cated \( a \) to form the action set \( A \).
- performs the action \( a \), records the reward \( r \) received
  from the environment, and uses \( r \) to update the asso-
  ciated parameters of all classifiers in \( A \).
- when appropriate, implements rule discovery by appli-
  cing an evolutionary mechanism (commonly a genetic
  algorithm) in the action set \( A \), to introduce new clas-
  sifiers to the population.

Additionally, the explore mode may perform subsumption
deletion to merge specific classifiers into any more general
and accurate classifiers. Subsumption deletion is a way of
biasing the genetic search towards more general, but still ac-
curate, classifiers [4]. It also effectively reduces the number
of classifier rules in the final population [18].

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 pay-
off. The exploit mode is also used to test learning per-
formance of the agent in the application. Various richer
encoding schemes have been investigated to represent high
level knowledge in LCS in an attempt to obtain compact

1046
classifier rules [8,13,14], to reach the optimal performance faster [19,20], to approximate functions [5,20], to develop useful feature extractors [1], and to identify and process building blocks of knowledge [6,15].

2.3 Scalable Evolutionary Computation

A main research direction is to develop an autonomously scalable classifier system to obtain general solutions for any problem size in the domain. To achieve this goal, Iqbal et al. [15] used GP-like rich encoding scheme to identify building blocks of knowledge. Then they extracted building blocks of knowledge, in the form of tree-like code fragments, from smaller problems and reused them to learn more complex problems of the domain [12]. The system having ability to reuse the extracted domain knowledge [15] has shown scalability beyond the standard XCS, but the solutions found were not scalable in general at the domain level. The reason behind this limitation is that the code fragments extracted from smaller problems were reused as terminals in the code fragments of larger problems of the domain, resulting in large sized acyclic nested tree structures that increased the search space and eventually restricted the system to a limit in problem size.

Cartesian genetic programming (CGP) is a flexible graph-based version of GP that allows a program to be evolved with more than one output [22]. The graph-based representation has benefit of implicitly reusing the nodes in the graph. SMCGP is a developmental form of CGP where an evolved individual computer program can be iterated to produce an infinite sequence of computer programs using a set of self-modifying functions [9]. It allows programs to take more inputs and give more outputs during each iteration so that each of the generated programs solves a particular problem in the domain. Using SMCGP, Harding et al. [10] have evolved programs that can provide general solutions to a number of problems including an n-bits parity problem and an adder to add two n-bits binary numbers. The limitation of SMCGP is that cycles were not allowed in the graphical programs to avoid infinite loops. Therefore, the size of the obtained solutions to Boolean problems increases for every next level problem in the domain (e.g. see Fig. 3), due to the application of self-modifying operators, especially the duplication operator that duplicates a part of the graph in it. So for very large scale Boolean problems, solutions are very long making visualization and interpretation difficult.

3. XCS WITH STATE-MACHINE ACTION

In the work presented here, the typical static numeric action in XCS will be replaced by an FSM in an attempt to develop a general scalable learning classifier system having concise and easily interpretable classifier rules.

In the proposed approach of XCS with state-machine action, called XCSSMA, the static binary action is replaced by a Moore state machine [22] retaining the ternary alphabet in the condition of a classifier rule. Each state machine consists of n states, where some of the states may be deactivated. A linear representation is used to encode a machine. Each state p ∈ Q is encoded as a three-tuple {v, active, T} where v ∈ Δ is the output value of the state p, active is a Boolean flag determining whether the state p is activated, and T is a transition function from an input symbol x ∈ Δ to a next state q ∈ Q.

For example the state machine shown in Fig. 2 where a deactivated state q1 is represented by a dashed circle, will be encoded as ‘11q1q20q2q10q2q0’. Here Q = {q0, q1, q2}, Σ = {0, 1}, Δ = {0, 1}, startstate = q0. λ is defined as: λ(q0) = 1, λ(q1) = 1, and λ(q2) = 0, and δ is defined as: δ(q0, 0) = q1, δ(q0, 1) = q2, δ(q1, 0) = q2, δ(q1, 1) = q1, δ(q2, 0) = q2, and δ(q2, 1) = q0.

![Figure 2: A Moore state machine with a deactivated state q₁. This machine will be encoded as ‘11q₁q₂0q₂q₁0q₂q₀’](image)

If a state qₙ has been deactivated and an activated state qₙ has a transition to qₙ, then that transition will be changed to any activated state in the machine, chosen uniformly randomly, as suggested by Spears and Gordon-Spears [24]. e.g. in Fig. 2 the transition δ(q₀, 0) = q₁ will be set to δ(q₀, 0) = q₀ or δ(q₀, 0) = q₂. For simplicity, the first state in the state-machine action is set to be the start state. To avoid the creation of any junk machine in the evolutionary process, the start state will always be activated.

The proposed XCSSMA approach extends standard XCS, described in section 2.2, in the following cases: the action value, the covering operation, the rule discovery operation, the procedure comparing equality of two classifiers, and the subsumption deletion mechanism.

3.1 State-Machine Action Value

The action value of a classifier is determined by processing the current input string s via the state-machine action in the classifier. The processing starts from the start state in the state-machine action, and the value of the last state visited is taken as the action value. For example, consider the classifier shown in Fig. 3. If the input string s is ‘100101’ then the action value will be 0.

![Figure 3: A classifier rule with state-machine action.](image)

3.2 Covering Operation

Covering occurs if an action a is missing in the match set [M]. If so, a random classifier is created whose condition matches the current environmental state s and contains...
‘don’t care’ symbols with probability $P_{\text{d}}$. The state-machine action is randomly generated until the output is a.

### 3.3 Rule Discovery Operation

In the rule discovery operation, a genetic algorithm is applied in the action set $[A]$ to produce two offspring. First of all, two parent classifiers are selected from $[A]$ based on fitness and the offspring are created out of them. Next, the conditions and state-machine actions of the offspring are separately crossed, with probability $\chi$, by applying GA crossover operation. After that, each symbol in the conditions of the resulted children by crossover are mutated with probability $\mu$, such that both children match the currently observed state $s$. In the mutation of conditions, a ‘non-don’t care’ symbol is replaced by the ‘don’t care’ symbol ‘#’, and a ‘don’t care’ symbol is replaced by the corresponding bit in the state $s$. Then, the state-machine actions of the children are mutated with probability $\mu$.

### 3.4 Comparing Equality of Two Classifiers

If a newly created classifier in the rule discovery operation is not subsumed and there is no classifier equal to it in the population, then it will be added to the population. Two classifiers are considered to be equal if and only if both have the same conditions and genotypically the same state-machine (i.e. identically the same state-machine encoding) in their actions.

### 3.5 Subsumption Deletion

A classifier $cl_1$ can subsume another classifier $cl_2$ if both have the same action and $cl_1$ is accurate, sufficiently experienced, and more general than $cl_2 [7]$. It is to be noted that due to the multiple genotypes to a single phenotype issue caused by using state-machines in place of numeric values in the actions of classifier rules, subsumption deletion is less likely to occur. Subsumption deletion is still made possible, albeit limited, by matching the state-machine descriptions on a character by character basis.

### 4. EXPERIMENTAL DESIGN

#### 4.1 The Problem Domains

The problem domains used in the experimentation are the even-parity, majority-on, carry, and multiplexer problems which have been often studied using the LCS and GP approaches.

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

The majority-on problems are similar to even-parity problems in that 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, so is therefore difficult to learn. For example, ‘1##1:1’ and ‘11#1:#1’ are two maximally general and accurate classifiers, but they overlap in the “11*11” subspace[8].

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.

A multiplexer is an electronic circuit that accepts input strings of length $n = k + 2^k$, and gives one output. The values of $k$ so-called address bits are used to select one of the $2^k$ remaining 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. Multiplexer problems are highly non-linear and have epistasis, i.e. importance of data bits is dependent on address bits.

#### 4.2 Experimental Setup

The system uses the following parameter values, commonly used in the literature, as suggested by Butz and Wilson in [7]: 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 $\theta_{cA} = 25$; 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 $P_{\text{d}} = 0.5$; reduction of the fitness $\text{FitnessReduction} = 0.1$; number of states $n = 5$; input alphabet $\Sigma = \{0, 1\}$; output alphabet $\Delta = \{0, 1\}$; and the selection method is tournament selection with tournament size ratio 0.4. Both GA subsumption and action set subsumption are activated. The number of classifiers used is 2000 and the number of training examples is half a million, in all the experiments conducted here. 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 known different seed in each run. Each result reported in this work is average of the 30 runs.

In all graphs presented here, the X-axis is the number of problem instances used as training examples, the Y-axis is the performance measured as the percentage of correct classification during the last 100 exploit problem instances, and the error bars show standard deviation in the 30 runs.

#### 5. RESULTS

##### 5.1 The Even-Parity Problem Domain

The largest solved parity problem, directly from training data, reported in literature is 24-bits even-parity problem, by Harding et al. [10] using SMCGP. The performance of standard XCS and XCSSMA in learning 24-bits even-parity problem is shown in Fig. 4. It is observed that standard XCS cannot learn the 24-bits even-parity problem, whereas XCSSMA has solved it using approximately 5,000 training examples.

The even-parity domain does not allow generalizations if the standard ternary alphabet based encoding scheme is used with static numeric action. So each bit must be specific for a rule to be accurate, requiring $2^{25}$ such rules for the 24-

---

[8] Here, * can be 0, 1, or #.
bits even-parity problem. The standard XCS technique was not able to evolve enough accurate rules, even using a larger population of classifiers. However, the proposed XCSSMA approach not only solved the 24-bits even-parity problem, but the obtained solutions are compact, easily understandable and general for any $n$-bits even-parity problem.

One of the classifier rules from the final solution obtained using XCSSMA is shown in Fig. 5. This is a maximally general and accurate classifier covering the whole problem space. The FSM action in this rule is general to solve any $n$-bits even-parity problem. It is to be noted that state $q_3$ is not active, so there is no transition from any active state to $q_3$. The state $q_4$ is active, but not reachable from the start state $q_0$. It means only three states, i.e. $q_0$, $q_1$ and $q_2$, are the working states in this FSM action.

5.2 The Majority-on Problem Domain

The complete solution of the majority-on problem domain consists of strongly overlapping classifiers, which makes it a hard problem to learn. The largest attempted majority-on problem in literature is a 7-bits problem, by Jackson and Gibbons [17] using layered learning in genetic programing and the reported success rate is 90%. The performance of standard XCS and XCSSMA in learning 7-bits majority-on problem is shown in Fig. 7. It is observed that standard XCS reached approximately 94% performance, but could not completely solve the 7-bits majority-on problem, whereas XCSSMA has solved it using approximately 25,000 training examples.

Figure 5: A sample classifier rule from final solution of 24-bits even-parity problem in XCSSMA.

5.3 The Carry Problem Domain

The largest solved carry problem, directly from training data, reported in literature is 6+6 bits carry problem, by Harding et al. [10] using SMCGP. The performance of standard XCS and XCSSMA in learning 6+6 bits carry problem is shown in Fig. 9. The complete solution in the carry problem domain consists of overlapping classifiers, in addition it is a niche imbalance domain, which makes it very difficult to learn. However, XCSSMA successfully learned the 6+6 bits carry problem, whereas standard XCS failed. In addition, the obtained solutions in XCSSMA are compact, easily understandable, and general for any $n+n$ bits carry problem.

Figure 7: Results of 7-bits majority-on problem.
One of the classifier rules from the final solution obtained using XCSSMA is shown in Fig. 10. This is a maximally general and accurate classifier covering the whole problem space. The FSM action in this rule is general to solve any \( n+n \) bits carry problem. It is to be noted that state \( q_1 \) and state \( q_2 \) are not active, so there is no transition from any active state to these deactivated states. The state \( q_3 \) is active, but not reachable from the start state \( q_0 \). It means only two states, \( q_0 \) and \( q_4 \), are the working states in this FSM action.

The solutions found by Harding et al. [10] using SMCGP were also general, but not as compact and easily interpretable as the classifier rules obtained using XCSSMA here. For example, an evolved program using SMCGP is shown in Fig. 11 that provides solution for the 3+3 bits carry problem after two iterations. Here \((x_0, y_0), (x_1, y_1)\), and \((x_2, y_2)\) represent input-pair symbols; \(c_0\) and \(c_1\) denote the carry bits; INPP and OUTPUT are input and output nodes respectively; BF0, BF1, BF2, ... BF15 are the 16 Boolean operators of two variables; and DUP is the duplication operator. To obtain the solution for 6+6 bits carry problem, three more iterations will be performed resulting in a long chaining phenotype.

### 5.4 The Multiplexer Problem Domain

The results of standard XCS and XCSSMA for the 20-bits multiplexer problem are shown in Fig. 12. XCSSMA successfully solved it, but took more instances as compared with the standard XCS to reach a similar performance level.

The multiplexer is a niche balanced problem domain and there exists a complete solution for multiplexer problems that does not contain any overlapping classifier rules. So, standard XCS effectively solved the 20-bits multiplexer problem. The XCSSMA approach used more training examples, due to the increased search space as a result of the state-machine based actions.

A sample classifier rule for 20-bits multiplexer problem is shown in Fig. 13. This rule is equivalent to numeric action based XCS rule ‘0011###0############ : 0’. The states \( q_1 \) and \( q_4 \) are not reachable from the start state \( q_0 \) in the FSM action of this rule. The processing of any matched input message ends at the state \( q_3 \) that have output value 0, therefore action value of this rule will be 0.

### 6. DISCUSSION

Although SMCGP solutions are large, they compute parity in a conceptually parallel fashion. Standard XCS computes condition matching in parallel, but the action is a
Figure 11: An evolved genotype, using SMCGP [10], iterated twice that provides solution for the 3+3 bits carry problem. Here \((x_0, y_0), (x_1, y_1), \) and \((x_2, y_2)\) represent input-pair symbols; \(y_0\) and \(c_1\) denote the carry bits. This is a colored figure and is adapted by kind permission of the author.

Figure 13: A sample classifier rule from final solution of 20-bits multiplexer problem in XCSSMA.

single scalar. By introducing XCSSMA the action is computed by a sequential process which trades compactness for execution speed, which is still fast. It is to be noted that condition matching is still in parallel so that the division of search space may be computed efficiently.

The developed system, XCSSMA, can be viewed from two different perspectives. First, it can be regarded as a system to evolve finite state machines. If a general state machine \(M\) can be created for the problem at hand with the given machine configuration parameters (i.e. the number of states, input alphabet, output alphabet, etc.), then the system produces a rule of the form ‘### : M’. This rule will cover the whole problem space and classify each input instance accurately using the machine \(M\) as the action, e.g. the classifier rules shown in Fig. 5 and Fig. 10 for 24-bits even-parity problem and 6+6 bits carry problem respectively. Second, if a single general FSM cannot be created for the problem then the system behaves like a typical XCS and evolves a set of rules that collectively solve the problem, e.g. in majority-on and multiplexer problem domains.

An FSM is an abstract model that can represent a finite-state system in a compact form, but the evolution of FSMs is a hard task. Usually, FSMs are evolved using supervised learning so for large scale problems some form of subsampling and/or incremental testing is needed [21][24]. The online learning, niche based breeding, and generalization properties of XCS implicitly provides incremental testing and subsampling of the training data set. So, the developed system, as a combination of XCS and FSMs, rapidly evolved the general FSMs for the even-parity and carry problems.

It is to be noted that using an FSM as the action of a classifier rule, in place of a static numeric action, the size of the search space increases. This was compensated by improving the generalization ability of standard XCS in even-parity, carry, and majority-on problems. This improvement is obvious in the accurate rules shown in Fig. 5, Fig. 10, and Fig. 6 that match all the problem instances from 24-bits even-parity, all the problem instances from 6+6 bits carry, and eight problem instances from 7-bits majority-on problems respectively. The generalization to this level is beyond the ability of standard XCS using ternary alphabet based conditions along with the numeric action.

The FSM based action could not improve the generalization beyond numeric action based XCS for the multiplexer domain because the state machines needed for this domain are more complex than the other domains. The inherent property of indexing to a certain position according to the address bits in the input problem makes the creation of a state machine really hard in the multiplexer domain. So in the multiplexer problem domain, XCSSMA takes more training examples, due to the increased search space, to reach a performance level similar to standard XCS. It is to be noted that the code-fragment based XCS techniques [12][14][15] can scale in the multiplexer domain beyond 20-bits MUX, and using XCS with code-fragment conditions [15] 135-bits MUX have been solved.

In summary, if a proper representation scheme is used in XCS to encode the classifier rules, then the system can evolve maximally general and accurate classifiers, possibly just one classifier rule covering the whole problem space, e.g. XCSSMA for the even-parity and carry problems here and a code-fragment based XCS for the frog problem in [13]. It is also possible that the obtained solutions will be general to solve any problem from the domain, e.g. the solutions obtained in XCSSMA for the even-parity and carry problems.

The suite of scalable-XCS techniques and SMCGP are very promising systems that use rich encoding schemes to address the problem of scalable learning, but in different ways. SMCGP produces an individual as a ‘single’ solution, whereas the XCS classifier systems evolve a co-operative set of rules. SMCGP generally requires supervised learning with the whole training set, rather than on-line, reinforcement learning as in an XCS classifier system. Scalable-XCS can produce more compact solutions, but needs human selection for encoding schemes, i.e. code fragments or FSMs. SMCGP is more flexible as it evolves a computer program that can generate a sequence of programs, each of which solves a particular problem in the domain. However, XCS
can divide up problem space using co-operative classifiers, which is advantageous in certain domains.

7. CONCLUSIONS

The proposed XCSSMA technique successfully solved four difficult Boolean problems, i.e. even-parity, majority-on, carry, and multiplexer problems. Further XCSSMA evolved, for the first time, compact and easily interpretable general solutions for the even-parity and carry problem domains. The XCSSMA technique is expected to be most useful when problems contain cyclic regularity in the problem input that is useful for deciding the answer to the problem.

Future work includes simplifying the final solution obtained by removing the deactivated and non-reachable states. Also two FSMs could be compared semantically, instead of syntactically as present, to fully enable the subsumption deletion mechanism in XCSSMA.

In the work presented here, only binary classification problems were tested. It would be interesting to apply the proposed system to the problems having more than two classes. It is anticipated that the implemented system can be extended to learn sequence prediction and multi-step tasks. There are advanced forms of state machines that have memory component such as pushdown automata and Turing machines, which could be used in XCSSMA for problems needing memory.

8. REFERENCES