An Algorithmic Description of XCS

Martin V. Butz[1] and Stewart W. Wilson[2]

[1] Institute for Psychology III & Department of Computer Science
University of Würzburg, Germany
butz@psychologie.uni-wuerzburg.de
[2] University of Illinois at Urbana-Champaign
Prediction Dynamics, Concord, MA 01742
wilson@prediction-dynamics.com

Abstract. A concise description of the XCS classifier system's parameters, structures, and algorithms is presented as an aid to research. The algorithms are written in modularly structured pseudo code with accompanying explanations.

1 Introduction

XCS is a recently developed learning classifier system (LCS) that differs in several ways from more traditional LCSs. In XCS, classifier fitness is based on the *accuracy* of a classifier's payoff prediction instead of the prediction itself. Second, the genetic algorithm (GA) takes place in the action sets instead of the population as a whole. Finally, unlike the traditional LCS, XCS has no message list and so is only suitable for learning in Markov environments (XCS extensions using an internal-state register have shown promise in non-Markov environments).

XCS's fitness definition and GA locus together result in a strong tendency for the system to evolve accurate, maximally general classifiers that efficiently cover the state-action space of the problem and allow the system's 'knowledge' to be readily seen. As a result of these properties, there has been considerable interest in further investigation and potential extension of XCS and its principles. We therefore thought it would be useful to provide a basic algorithmic description of XCS, both as a core definition of the system and as a common framework from which new variants and research directions could spring.

We first present XCS's relation to the problem environment, followed by the system's structures and parameters. The rest of the paper consists of a top-down modular description of the XCS algorithm, written in pseudo-code accompanied by explanatory notes. We hope the result will be useful, and we encourage researchers to give us feedback regarding potential problems and clarifications. This document should definitely be read in conjunction with some of the basic XCS literature, for example Wilson (1995), Kovacs (1997), and Wilson (1998). Additional papers on XCS and other LCSs, together with a complete LCS bibliography, are found in Lanzi, Stolzmann, and Wilson (2000).

2 Environment Interaction, Structures, and Parameters

2.1 Interaction with the Environment

In keeping with the typical LCS model, the environment provides as input to the system a series of sensory situations $\sigma(t) \in \{0,1\}^L$, where L is the number of bits in each situation. In response, the system executes actions $\alpha(t) \in \{a_1, ..., a_n\}$ upon the environment. Each action results in a scalar reward $\rho(t)$ (possibly zero). The interaction is divided into *problems*, which may be either single-step or multi-step. A flag *eop* indicates the end of a problem. While $\sigma(t)$ and $\alpha(t)$ are interactions with the environment itself, the reward $\rho(t)$ and the flag are normally provided by another component which, following Dorigo and Colombetti (1998), we term the *reinforcement program rp*. The reinforcement program determines the reward according to the current environmental input and the action that was executed. The separation of environment and reinforcement components is useful and natural because reinforcement is often not an inherent aspect of the environment, but may be due, e.g., to a trainer or to the system's own priorities. Figure 1 illustrates the interaction of the environment and *rp* with XCS.

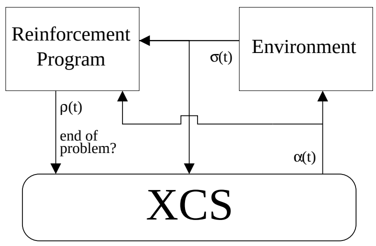

Fig. 1. XCS interacts with an environment and a reinforcement program.

In a single-step problem such as the Boolean multiplexer, the successive situations are not related to each other. After execution of an action the *rp* provides the appropriate reward ρ and signals with the flag *eop* that the problem has ended. In a multi-step problem such as a maze, the successive situations are related to each other. Reward is only provided in certain situations (e.g. the food position(s) in mazes). The point at which a problem ends in a multi-step environment must be defined by the *rp* according to the task that is to be solved.

2.2 A Classifier in XCS

XCS keeps a population of classifiers which represent its knowledge about the problem. Each classifier is a condition–action–prediction rule having the following parts:

- The condition $C \in \{0, 1, \#\}^L$ specifies the input states (sensory situations) in which the classifier can be applied (matches).

- The action $A \in \{a_1, ..., a_n\}$ specifies the action (possibly a classification) that the classifier proposes.
- The prediction p estimates (keeps an average of) the payoff expected if the classifier matches and its action is taken by the system.

Moreover, each classifier keeps certain additional parameters:

- The prediction error ϵ estimates the errors made in the predictions.
- The fitness f denotes the classifier's fitness.
- The experience exp counts the number of times since its creation that the classifier has belonged to an action set.
- The time stamp ts denotes the time-step of the last occurrence of a GA in an action set to which this classifier belonged.
- The action set size as estimates the average size of the action sets this classifier has belonged to.
- The numerosity num reflects the number of micro-classifiers (ordinary classifiers) this classifier—which is technically called a *macroclassifier*—represents.

To refer to one of the attributes of a classifier cl we will write x_{cl} where x can be any of the above attributes (i.e. $x \in \{C, A, p, \epsilon, f, exp, ts, as, num\}$).

Important in our notation is the term *payoff*. Due to the Q-learning-like reinforcement learning in XCS, payoff does not refer solely to the expected reward ρ but is a combination of ρ and the payoff prediction of the best possible action in the next state. However, in the case of a single-step problem payoff reduces to the reward produced by the proposed action.

Note that ϵ measures the error of the predictions in units of payoff. The same is true of the parameter ϵ_0 that is defined in section 2.4.

Classifiers in XCS are *macroclassifiers*, i.e., each classifier represents num traditional or *micro-* classifiers having identical conditions and actions. Algorithms covering creation, deletion, and adjustment of the numerosity of macroclassifiers are given in section 3.9, 3.10, and 3.11. In XCS, macroclassifiers are always handled *as though* they consist of num micro-classifiers.

2.3 The Different Sets

There are four different sets that need to be considered in XCS.

- The population $[P]$ consists of all classifiers that exist in XCS at any time t.
- The match set $[M]$ is formed out of the current $[P]$. It includes all classifiers that match the current situation $\sigma(t)$.
- The action set $[A]$ is formed out of the current $[M]$. It includes all classifiers of $[M]$ that propose the executed action.
- The previous action set $[A]_{-1}$ is the action set that was active in the last execution cycle.

2.4 Learning Parameters in XCS

In order to control the learning process in XCS the following parameters are used:

- N specifies the maximum size of the population (in micro-classifiers, i.e., N is the sum of the classifier numerosities).
- β is the learning rate for p, ϵ, f, and as.
- α, ϵ_0, and ν are used in calculating the fitness of a classifier.
- γ is the discount factor used—in multi-step problems—in updating classifier predictions.
- θ_{GA} is the GA threshold. The GA is applied in a set when the average time since the last GA in the set is greater than θ_{GA}.
- χ is the probability of applying crossover in the GA.
- μ specifies the probability of mutating an allele in the offspring.
- θ_{del} is the deletion threshold. If the experience of a classifier is greater than θ_{del}, its fitness may be considered in its probability of deletion.
- δ specifies the fraction of the mean fitness in $[P]$ below which the fitness of a classifier may be considered in its probability of deletion.
- θ_{sub} is the subsumption threshold. The experience of a classifier must be greater than θ_{sub} in order to be able to subsume another classifier.
- $P_{\#}$ is the probability of using a # in one attribute in C when covering.
- p_I, ϵ_I, and f_I are used as initial values in new classifiers.
- p_{explr} specifies the probability during action selection of choosing the action uniform randomly.
- θ_{mna} specifies the minimal number of actions that must be present in a match set $[M]$, or else covering will occur.
- *doGASubsumption* is a Boolean parameter that specifies if offspring are to be tested for possible logical subsumption by parents.
- *doActionSetSubsumption* is a Boolean parameter that specifies if action sets are to be tested for subsuming classifiers.

2.5 Commonly Used Parameter Settings

For parameter settings, it is best to check the literature for a similar experiment. In some cases, the following suggestions could be taken as starting points. The population size, N, should be large enough so that, starting from an empty population, covering occurs only at the very beginning of a run. The learning rate, β, could be in the range 0.1-0.2. The parameter α is normally 0.1. The parameter ϵ_0 is the error below which classifiers are considered to have equal accuracy; a typical value would be about one percent of the maximum value of ρ, e.g., 10 if the maximum value is 1000. The power parameter ν is typically 5. The discount factor γ has been 0.71 in many problems in the literature, but larger or smaller values could certainly work, depending on the environment. The threshold θ_{GA} is often in the range 25-50. Crossover probabilities χ in the range 0.5-1.0 have been used. Mutation probabilities μ in the range 0.01-0.05 are often used.

The deletion threshold θ_{del} could be about 20. δ is often taken to be 0.1. The subsumption threshold θ_{sub} could be about 20, though larger values (more experience) are important in some problems. $P_{\#}$ could be around 0.33. Larger values reduce the need for covering, but may make it harder to evolve accurate classifiers. The initialization parameters p_I, ϵ_I, and f_I should be taken very small—essentially zero. The exploration probability p_{explr} could be 0.5, but this depends on the type of experiment contemplated. To cause covering to provide classifiers for every action, choose θ_{mna} equal to the number of available actions.

The setting of *doGASubsumption* and *doActionSetSubsumption* depends on the problem. In general, subsumption is used to eliminate classifiers that clearly add nothing to the system's decision capability. In the case of GA subsumption, these are offspring classifiers whose conditions are logically subsumed by a parent's condition, given that the parent is both accurate and sufficiently experienced. In the case of action set subsumption, a general, accurate, and experienced classifier in an action set eliminates classifiers in the set that its condition logically subsumes. The two subsumption methods are independent and different in their effects. Broadly, action set subsumption appears to be 'stronger': it causes greater 'condensation' of the population. Subsumption is useful in problems where there is a well-defined underlying target function, such as the Boolean multiplexer.

Note, however, that subsumption is *not* necessary for XCS to form accurate maximal generalizations (see Wilson (1995) for the basic XCS *generalization hypothesis*). Subsumption tends to result in accurate classifiers that are as formally general as possible without being contradicted by any actually occurring input. Without subsumption, the system produces these as well as accurate but more-specific classifiers that match the same inputs as the general ones do (Wilson, 1998). Subsumption produces smaller final populations, but because the more-specific classifiers are not present, the system is more vulnerable to environmental changes—it does not have the more-specific classifiers to "fall back on".

3 An Algorithmic Description of XCS

This section presents the algorithms used in XCS. The description starts from the top level. First, the overall execution cycle is described. The following subsections specify the single parts in more detail. Each specified sub-procedure in this description is written in capital letters. When referring to a module other than XCS (e.g. the environment *env*), a colon is used.

3.1 Initialization

When XCS is started, the modules must first of all be initialized. The parameters in the environment must be set and, e.g., a maze must be read in. Also, the reinforcement program *rp* must be initialized. Finally, XCS itself must be initialized. Apart from the parameter settings and the start of the time-step counter referred to as `actual time` t, the population $[P]$ needs to be initialized. The population $[P]$ can either be left empty or can be filled with the maximal number of classifiers N, generating each classifier with a random condition and action and the initial parameters. The two methods differ only slightly in their effect on performance. Thus the simpler way of leaving the population empty in the beginning is commonly used. After the initialization, the main loop is called.

XCS:

```
initialize environment env
initialize reinforcement program rp
initialize XCS
RUN EXPERIMENT
```

3.2 The Main Loop

In the main loop *RUN EXPERIMENT*, the current situation is first sensed (received as input). Second, the match set is formed from all classifiers that match the situation. Third, the prediction array PA is formed based on the classifiers in the match set. PA predicts for each possible action a_i the resulting payoff. Based on PA, one action is chosen for execution and the action set $[A]$ is formed, which includes all classifiers of $[M]$ that propose the chosen action. Next, the winning action is executed. Then the previous action set $[A]_{-1}$ (if this is a multi-step problem and there is a previous action set) is modified using the Q-learning-like payoff quantity `P` which is a combination of the previous reward ρ_{-1} and the largest action prediction in the prediction array PA. Moreover, the GA may be applied to $[A]_{-1}$. If a problem ends on the current time-step (single-step problem or last step of a multi-step problem), $[A]$ is modified according to the current reward ρ and the GA may be applied to $[A]$. The main loop is executed as long as the termination criterion is not met. A termination criterion is, e.g., a certain number of trials or a 100% performance level.

The main loop specifies many sub-procedures essential for learning in XCS. Some of the procedures are more or less trivial while others are complex and themselves call other sub-procedures. The following sections describe all procedures specified in the main loop, covering all relevant processes. Each of them tries to specify the general idea and the overall process and then gives a more detailed description of single parts in successive paragraphs.

3.3 Formation of the Match Set

The *GENERATE MATCH SET* procedure gets as input the current population $[P]$ and the current situation σ. Although the procedure sounds trivial, it has within it a covering process. Covering is called when the number of different actions represented by matching classifiers is less than the parameter θ_{mna}. Thus, *GENERATE MATCH SET* first looks for the classifiers in $[P]$ that match σ and next, checks if covering is required. A classifier generated by covering can be

RUN EXPERIMENT():

```
ρ₋₁ ← 0
do{
    σ ← env: get situation
    GENERATE MATCH SET [M] out of [P] using σ
    GENERATE PREDICTION ARRAY PA out of [M]
    act ← SELECT ACTION according to PA
    GENERATE ACTION SET [A] out of [M] according to act
    env: execute action act
    ρ ← rp: get reward
    if([A]₋₁ is not empty)
        P ← ρ₋₁ + γ * max(PA)
        UPDATE SET [A]₋₁ using P possibly deleting in [P]
        RUN GA in [A]₋₁ considering σ₋₁ inserting and
            possibly deleting in [P]
    if(rp: eop)
        P ← ρ
        UPDATE SET [A] using P possibly deleting in [P]
        RUN GA in [A] considering σ inserting and
            possibly deleting in [P]
        empty [A]₋₁
    else
        [A]₋₁ ← [A]
        ρ₋₁ ← ρ
        σ₋₁ ← σ
}while(termination criteria are not met)
```

directly added to the population since it must differ from all current classifiers. Note that the while loop at step 2 is executed as long as covering is required.

In the following paragraphs we will describe the sub-procedures included in the *GENERATE MATCH SET* algorithm. The sub-procedure *DELETE FROM POPULATION* however can be found in section 3.10.

Classifier Matching The matching procedure is that commonly used in LCSs. A 'don't care'-symbol # in C matches any symbol in the corresponding position of σ. A 'care' or non-# symbol only matches with the exact same symbol at that position. The basic XCS relies on binary coding and thus the care symbols are $\in \{0, 1\}$. Recently, Lanzi (1999b) and Lanzi (1999c) introduced an extension of the conditions in XCS involving messy coding and Lisp s-expressions. Wilson (2000) introduced the first XCS extension that is able to handle real coded inputs. However, here we will consider only the basic kind of classifier condition.

The *DOES MATCH* procedure checks each component in the classifier's condition C. If a component is specified (i.e. is not a don't care symbol), it is compared with the corresponding attribute in the current situation σ. Only if all comparisons hold does the classifier match σ and the procedure return *true*.

GENERATE MATCH SET([P], σ):

```
initialize empty set [M]
while([M] is empty)
   for each classifier cl in [P]
      if(DOES MATCH classifier cl in situation σ)
         add classifier cl to set [M]
   if(the number of different actions in [M] < θ_mna)
      GENERATE COVERING CLASSIFIER cl_c considering [M] and σ
      add classifier cl_c to set [P]
      DELETE FROM POPULATION [P]
      empty [M]
return [M]
```

DOES MATCH(cl, σ):

```
for each attribute x in C_cl
   if(x ≠ # and x ≠ the corresponding attribute in σ)
      return false
return true
```

Covering Covering occurs if the number of actions present in $[M]$ is $< \theta_{mna}$. A classifier is created whose condition matches $\sigma(t)$ and contains don't cares with probability $P_{\#}$. The classifier's action is chosen randomly from among those not present in $[M]$.

GENERATE COVERING CLASSIFIER([M], σ):

```
initialize classifier cl
initialize condition C_cl with the length of σ
for each attribute x in C_cl
   if(RandomNumber[0, 1) < P_#)
      x ← #
   else
      x ← the corresponding attribute in σ
A_cl ← random action not present in [M]
p_cl ← p_I
ε_cl ← ε_I
f_cl ← f_I
exp_cl ← 0
ts_cl ← actual time t
as_cl ← 1
num_cl ← 1
```

3.4 The Prediction Array

Given an input, XCS makes a "best guess" prediction of the payoff to be expected for each possible action. These *system predictions* are stored in a vector called the Prediction Array PA. The system prediction for an action is a fitness-weighted average of the predictions of all classifiers in $[M]$ that advocate that action. If no classifiers in $[M]$ advocate a certain action, its system prediction is not defined, symbolized by *null*.

The *GENERATE PREDICTION ARRAY* procedure considers each classifier in $[M]$ and adds its prediction multiplied by its fitness to the prediction value total for that action. The total for each action is then divided by the sum of the fitnesses for that action to yield the system prediction.

GENERATE PREDICTION ARRAY($[M]$):

```
initialize prediction array PA to all null
initialize fitness sum array FSA to all 0.0
for each classifier cl in [M]
   if(PA[A_cl] = null)
      PA[A_cl] ← p_cl * f_cl
   else
      PA[A_cl] ← PA[A_cl] + p_cl * f_cl
   FSA[A_cl] ← FSA[A_cl] + f_cl
for each possible action A
   if(FSA[A] is not zero)
      PA[A] ← PA[A] / FSA[A]
return PA
```

3.5 Choosing an Action

XCS does not prescribe any particular action-selection method, and any of a great variety can be employed. For example, actions may be selected randomly, independent of the system predictions, or the selection may be based on those predictions—using, e.g., roulette-wheel selection or simply picking the action with the highest system prediction.

In our *SELECT ACTION* procedure we illustrate a combination of *pure exploration*—choosing the action randomly—and *pure exploitation*—choosing the best one. Lanzi (1999a) published first experiments with so-called *biased exploration* where pure exploration is chosen with a probability p_{explr} and otherwise pure exploitation is chosen. This action selection method is known as *ϵ-greedy* selection in the reinforcement learning literature (Sutton & Barto, 1998) where the ϵ has identical meaning to our p_{explr} parameter. As an aside, it appears better to perform the GA only on exploration steps, especially if most steps are exploitation.

SELECT ACTION(PA):

```
if(RandomNumber[0,1) < p_explr)
   //Do pure exploration here
   return a randomly chosen action from those not null in PA
else
   //Do pure exploitation here
   return the best action in PA
```

3.6 Formation of the Action Set

After the match set $[M]$ is formed and an action is chosen for execution, the *GENERATE ACTION SET* procedure forms the action set out of the match set. It includes all classifiers that propose the chosen action for execution.

GENERATE ACTION SET($[M]$, *act*):

```
initialize empty set [A]
for each classifier cl in [M]
   if(A_cl = act)
      add classifier cl to set [A]
```

3.7 Updating Classifier Parameters

Although Wilson (1995) applied the update procedures as well as the GA originally in $[M]$ and only later (Wilson, 1998) was this switched to $[A]$, application in $[A]$ is now commonly used and gives better performance in all cases known to us. Thus we will use the $[A]$ notation in the following procedures although they are basically independent of the classifier set involved.

The reinforcement portion of the update procedure follows the pattern of Q-learning (Sutton & Barto, 1998). Classifier predictions are updated using the immediate reward and the discounted maximum payoff anticipated on the next time-step. The difference is that in XCS it is the prediction of a possibly general *rule* that is updated, whereas in Q-learning it is the prediction associated with an environmental *state-action* pair. Updates of classifier parameters other than prediction are unique to XCS. Note that in single-step problems, the prediction is updated using the immediate reward alone.

Each time a classifier enters the set $[A]$, its parameters are modified in the order: exp, p, ϵ, as, and f. Variations in the order are possible. The principle one is to exchange the p and ϵ updates. If prediction comes before error, the prediction of a classifier in its very first update immediately predicts the correct payoff and consequently the prediction error is set to zero. This can lead to faster learning in simpler problems but can be misleading in more complex ones. A more conservative strategy which puts the error update first, seems to work better on harder problems. The update of the action set size estimate is independent from the other updates and consequently can be executed at any point in time. While the updates of exp, p, ϵ, and as are straightforward, the update of f is more complex and requires more computational steps. Thus, we refer to another sub-procedure. Finally, if the program is using action set subsumption, the procedure calls the *DO ACTION SET SUBSUMPTION* procedure. This procedure is very powerful and is able to eliminate a large number of classifiers in the action set in one step. Section 3.11 describes the procedure in detail.

UPDATE SET($[A]$, P, $[P]$):

```
for each classifier cl in [A]
    exp_cl++
    //update prediction p_cl
    if(exp_cl < 1/β)
        p_cl ← p_cl + (P - p_cl) / exp_cl
    else
        p_cl ← p_cl + β * (P - p_cl)
    //update prediction error ε_cl
    if(exp_cl < 1/β)
        ε_cl ← ε_cl + (|P - p_cl| - ε_cl) / exp_cl
    else
        ε_cl ← ε_cl + β * (|P - p_cl| - ε_cl)
    //update action set size estimate as_cl
    if(exp_cl < 1/β)
        as_cl ← as_cl + (Σ_{c∈[A]} num_c - as_cl) / exp_cl
    else
        as_cl ← as_cl + β * (Σ_{c∈[A]} num_c - as_cl)
UPDATE FITNESS in set [A]
if(doActionSetSubsumption)
    DO ACTION SET SUBSUMPTION in [A] updating [P]
```

Fitness Update The fitness of a classifier in XCS is based on the *accuracy* of its predictions. The *UPDATE FITNESS* procedure first calculates the classifier's accuracy κ using the classifier's prediction error ϵ. Then the classifier's fitness is updated using the *normalized accuracy* computed in lines 8-10.

UPDATE FITNESS($[A]$):

```
 1 accuracySum ← 0
 2 initialize accuracy vector κ
 3 for each classifier cl in [A]
 4     if(ε_cl < ε_0)
 5         κ(cl) ← 1
 6     else
 7         κ(cl) ← α * (ε_cl / ε_0)^(-ν)
 8     accuracySum ← accuracySum + κ(cl) * num_cl
 9 for each classifier cl in [A]
10     f_cl ← f_cl + β * (κ(cl) * num_cl / accuracySum - f_cl)
```

3.8 The Genetic Algorithm in XCS

The final sub-procedure in the main loop, *RUN GA*, is also the most complex. First of all, the action set is checked to see if the GA should be applied at all. In order to apply a GA the average time period since the last GA application in the set must be greater than the threshold θ_{GA}. Next, two classifiers (i.e. the parents) are selected by roulette wheel selection based on fitness and the offspring are created out of them. After that, the offspring are possibly crossed and mutated. If the offspring are crossed, their prediction, error, and fitness values are set to the average of the parents' values. Finally, the offspring are inserted in the population, followed by corresponding deletions. However, if GA subsumption is being used, each offspring is first tested to see if it is subsumed by either of its parents; if so, that offspring is not inserted in the population, and the subsuming parent's numerosity is increased. (Besides checking if an offspring is subsumed by a parent, one could also check if it is subsumed by other classifiers in the action set, or even the population as a whole.)

RUN GA([A], σ, [P]):

```
if(actual time $t$ - $\sum_{cl \in [A]} ts_{cl} * num_{cl}$ / $\sum_{cl \in [A]} num_{cl}$ > $\theta_{GA}$)
    for each classifier $cl$ in $[A]$
        $ts_{cl} \leftarrow$ actual time $t$
    $parent_1 \leftarrow$ SELECT OFFSPRING in $[A]$
    $parent_2 \leftarrow$ SELECT OFFSPRING in $[A]$
    $child_1 \leftarrow$ copy classifier $parent_1$
    $child_2 \leftarrow$ copy classifier $parent_2$
    $num_{child_1} = num_{child_2} \leftarrow$ 1
    $exp_{child_1} = exp_{child_2} \leftarrow$ 0
    if(RandomNumber$[0, 1)$ < $\chi$)
        APPLY CROSSOVER on $child_1$ and $child_2$
        $p_{child_1} \leftarrow$ ($p_{parent_1}$ + $p_{parent_2}$) / 2
        $\epsilon_{child_1} \leftarrow$ ($\epsilon_{parent_1}$ + $\epsilon_{parent_2}$) / 2
        $f_{child_1} \leftarrow$ ($f_{parent_1}$ + $f_{parent_2}$) / 2
        $p_{child_2} \leftarrow p_{child_1}$
        $\epsilon_{child_2} \leftarrow \epsilon_{child_1}$
        $f_{child_2} \leftarrow f_{child_1}$
    $f_{child_1} \leftarrow f_{child_1}$ * 0.1
    $f_{child_2} \leftarrow f_{child_2}$ * 0.1
    for both children $child$
        APPLY MUTATION on $child$ according to $\sigma$
        if($doGASubsumption$)
            if(DOES SUBSUME $parent_1, child$)
                $num_{parent_1}$++
            else if(DOES SUBSUME $parent_2, child$)
                $num_{parent_2}$++
            else
                INSERT $child$ IN POPULATION
        else
            INSERT $child$ IN POPULATION
        DELETE FROM POPULATION $[P]$
```

While the sub-procedures *INSERT IN POPULATION* and *DELETE FROM POPULATION* are defined in section 3.9 and 3.10, respectively, the others are described in the following paragraphs.

Roulette-Wheel Selection The Roulette-Wheel Selection chooses a classifier for reproduction proportional to the fitness of the classifiers in set $[A]$. First, the sum of all the fitnesses in the set $[A]$ is computed. Next, the roulette-wheel is spun. Finally, the classifier is chosen according to the roulette-wheel result.

SELECT OFFSPRING([A]):

```
fitnessSum ← 0
for each classifier cl in [A]
  fitnessSum ← fitnessSum + f_cl
choicePoint ← RandomNumber[0, 1) * fitnessSum
fitnessSum ← 0
for each classifier cl in [A]
   fitnessSum ← fitnessSum + f_cl
   if(fitnessSum > choicePoint)
      return cl
```

Crossover The crossover procedure is similar to the standard crossover procedure in GAs. Implementations of XCS with one point and two point crossover were tested and resulted in approximately identical results. In the *APPLY CROSSOVER* procedure we show two-point crossover. The actions are not affected by crossover.

APPLY CROSSOVER(c_1, c_2):

```
x ← RandomNumber[0, 1) * (LENGTH of C_c1 +1)
y ← RandomNumber[0, 1) * (LENGTH of C_c1 +1)
if(x > y)
   switch x and y
i ← 0
do{
   if(x ≤ i and i < y)
      switch C_cl1[i] and C_cl2[i]
   i++
}while(i < y)
```

A more sophisticated crossover could include assurance that x and y are different. However, even without such special checks the algorithm serves its purpose.

Mutation While crossover does not affect the action, mutation takes place in both the condition and the action. A mutation in the condition flips the attribute to one of the other possibilities. Mutation in the action changes it equiprobably to one of the other actions. Though more time-efficient methods are possible, we present here the simple one in which a die is flipped for each attribute. Since in XCS most of the time is spent in processes that operate on the whole population such as matching and deletion, the type of algorithm used for mutation only slightly affects efficiency. Note that our mutation is somewhat restricted. Rather than allowing an attribute of the condition to change to any other attribute, we only allow changes either to # or to the specific value that matches the corresponding component of $\sigma(t)$. Thus the resulting condition still matches the current input. The effect (if the action is unchanged) is to search the current action set niche along the axis of specificity vs. generality. Less restricted mutation schemes are of course possible.

APPLY MUTATION(cl, σ):

```
i ← 0
do{
    if(RandomNumber[0, 1) < μ)
        if(C_cl[i] = #)
            C_cl[i] ← σ[i]
        else
            C_cl[i] ← #
    i++
}while(i < length of C_cl)
if(RandomNumber[0, 1) < μ)
    A_cl ← a randomly chosen other possible action
```

3.9 Insertion in the Population

The *INSERT IN POPULATION* procedure checks to see if the classifier to be inserted is identical in condition and action with a classifier already in the population. If so, the latter's numerosity is incremented; if not, the new classifier is added to the population.

INSERT IN POPULATION(cl, [P]):

```
for all c in [P]
    if(c is equal to cl in condition and action)
        num_c++
        return
add cl to set [P]
```

3.10 Deletion from the Population

The deletion procedure realizes two ideas at the same time: (1) it assures an approximately equal number of classifiers in each action set, or environmental 'niche'; (2) it removes low-fitness individuals from the population. The following paragraphs describe the *DELETE FROM POPULATION* procedure and the *DELETION VOTE* sub-procedure.

Roulette-Wheel Deletion Like the selection procedure, the deletion procedure *DELETE FROM POPULATION* chooses individuals (for deletion) by roulette-wheel selection. But first, the procedure checks to see if the sum of classifier numerosities in $[P]$ is less than or equal to N. If so, the procedure exits, as deletion is unnecessary. Otherwise, the sum of all deletion votes is calculated, and a classifier is chosen for deletion. If the classifier is a macroclassifier and currently represents more than one classifier, then its numerosity is merely decreased by one. Otherwise, the classifier is completely removed from the population.

DELETE FROM POPULATION($[P]$):

`if(`$\sum_{c\in[P]} num_c \leq N$`)`
   `return`
$avFitnessInPopulation \leftarrow \sum_{c\in[P]} f_c$ ` / ` $\sum_{c\in[P]} num_c$
$voteSum \leftarrow$ `0`
`for each classifier` c `in` $[P]$
   $voteSum \leftarrow voteSum$ ` + DELETION VOTE of ` c ` with ` $avFitnessInPopulation$
$choicePoint \leftarrow$ `RandomNumber`$[0,1)$ ` * ` $voteSum$
$voteSum \leftarrow$ `0`
`for each classifier` c `in` $[P]$
   $voteSum \leftarrow voteSum$ ` + DELETION VOTE of ` c ` with ` $avFitnessInPopulation$
   `if(`$voteSum > choicePoint$`)`
      `if(`$num_c >$ `1)`
         num_c`--`
      `else`
         `remove classifier` c `from set` $[P]$
      `return`

The Deletion Vote As mentioned above, the deletion vote realizes niching as well as removal of the lowest fitness classifiers. The deletion vote of each classifier is based on the action set size estimate as. Moreover, if the classifier has sufficient experience and its fitness is significantly lower than the average fitness in the population, the vote is increased in inverse proportion to the fitness. In this calculation, since we are deleting one micro-classifier at a time, we need to use as fitness the (macro)classifier's fitness divided by its numerosity. The following *DELETION VOTE* procedure realizes all this.

DELETION VOTE(cl, $avFitnessInPopulation$):

$vote \leftarrow as_{cl}$ ` * ` num_{cl}
`if(`$exp_{cl} > \theta_{del}$ ` and ` f_{cl} ` / ` $num_{cl} < \delta$ ` * ` $avFitnessInPopulation$`)`
   $vote \leftarrow vote$ ` * ` $avFitnessInPopulation$ ` / ( ` f_{cl} ` / ` num_{cl}`)`
`return` $vote$

3.11 Subsumption

Two subsumption procedures were introduced into XCS in Wilson (1998). The first, 'GA subsumption', checks an offspring classifier to see if its condition is logically subsumed by the condition of an accurate and sufficiently experienced parent. If so, the offspring is not added to the population, but the parent's numerosity is incremented. The idea is that such an offspring cannot improve the system's performance, since everything it accomplishes is accomplished just as well by the subsuming parent. GA subsumption, if enabled, occurs within the procedure *RUN GA*. It is detailed within that procedure, and is not called as a sub-procedure (though it could be). However, the sub-procedure *DOES SUBSUME* is called, and is described in this section.

The second subsumption procedure, 'action set subsumption', if enabled, takes place in every action set. It has a purpose similar to GA subsumption but is different and independent of it. The action set is searched for the most general classifier that is both accurate and sufficiently experienced. Then all other classifiers in the set are tested against the general one to see if it subsumes them. Any classifiers that are subsumed are eliminated from the population.

DO ACTION SET SUBSUMPTION([A], [P]):

```
initialize cl
for each classifier c in [A]
   if(c COULD SUBSUME)
      if(cl is empty or number of # in C_c > number of # in C_cl or
         (number of # in C_c = number of # in C_cl and
                                        RandomNumber[0, 1) < 0.5))
         cl ← c
if(cl is not empty)
   for each classifier c in [A]
      if(cl IS MORE GENERAL than c)
         num_cl ← num_cl + num_c
         remove classifier c from set [A]
         remove classifier c from set [P]
```

Subsumption of a classifier For a classifier to subsume another classifier, it must first be sufficiently accurate and sufficiently experienced. This is tested by the *COULD SUBSUME* function. Then, if a classifier could be a subsumer, it must be tested to see if it has the same action and is really more general than the classifier that is to be subsumed. This is the case if the set of situations matched by the condition of the potentially subsumed classifier form a proper subset of the situations matched by the potential subsumer. The *IS MORE GENERAL* procedure accomplishes this. The *DOES SUBSUME* procedure combines all the requirements.

COULD SUBSUME(cl):

```
if(exp_cl > θ_sub)
    if(ε_cl < ε_0)
        return true
return false
```

IS MORE GENERAL(cl_{gen}, cl_{spec}):

```
if (the number of # in C_{cl_gen} ≤ the number of # in C_{cl_spec})
    return false
i ← 0
do{
    if(C_{cl_gen}[i] ≠ # and C_{cl_gen}[i] ≠ C_{cl_spec}[i])
        return false
    i++
}while(i < length of C_{cl_gen})
return true
```

DOES SUBSUME(cl_{sub}, cl_{tos}):

```
if(A_{cl_sub} = A_{cl_tos})
    if(cl_sub COULD SUBSUME)
        if(cl_sub IS MORE GENERAL than cl_tos)
            return true
return false
```

4 Summary

This paper has revealed the processes inside XCS as well as the problem interaction. We hope that presentation of the algorithm in pseudo code with explanations will lead to deeper understanding of XCS and simplify research. Moreover, the modular structure should enable the reader to program the system in any programming language quite easily. However, our description only includes the basic framework of XCS. Starting from this baseline, we encourage modification as well as enhancement of the system, and welcome feedback on progress and results.

Acknowledgments

The work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants F49620-97-1-0050 and F49620-00-0163. Research funding for this work was also provided by the National Science Foundation under grant DMI-9908252. Support was also provided by a grant from the U. S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0003. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The authors acknowledge support from the Automated Learning Group at the National Center for Supercomputer Applications in Urbana-Champaign.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, the U. S. Army, or the U. S. Government.

References

Dorigo, M., & Colombetti, M. (1998). *Robot shaping: An experiment in behavior engineering*. Intelligent Robotics and Autonomous Agents. Cambridge, MA: MIT Press.

Kovacs, T. (1997). XCS classifier system reliably evolves accurate, complete, and minimal representations for boolean functions. In Roy, Chawdhry, & Pant (Eds.), *Soft Computing in Engineering Design and Manufacturing* (pp. 59–68). Springer-Verlag. ftp://ftp.cs.bham.ac.uk/pub/authors/T.Kovacs/index.html.

Lanzi, P. L. (1999a). An analysis of generalization in the XCS classifier system. *Evolutionary Computation*, *7*(2), 125–149.

Lanzi, P. L. (1999b). Extending the representation of classifier conditions. Part I: From binary to messy coding. In Banzhaf, W. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)* pp. 337–344. San Francisco, CA: Morgan Kaufmann.

Lanzi, P. L. (1999c). Extending the representation of classifier conditions. Part II: From messy coding to S-expressions. In Banzhaf, W. (Ed.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)* pp. 345–352. San Francisco, CA: Morgan Kaufmann.

Lanzi, P. L., Stolzmann, W., & Wilson, S. W. (Eds.) (2000). *Learning classifier systems: From foundations to applications.* LNAI 1813. Berlin Heidelberg: Springer-Verlag.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* Cambridge, MA: MIT Press.

Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, *3*(2), 149–175.

Wilson, S. W. (1998). Generalization in the XCS classifier system. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., & Riolo, R. (Eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference* pp. 665–674. San Francisco: Morgan Kaufmann.

Wilson, S. W. (2000). Get real! XCS with continuous-valued inputs. In Lanzi, P. L., Stolzmann, W., & Wilson, S. W. (Eds.), *Learning Classifier Systems: From Foundations to Applications, LNAI 1813* pp. 209–220. Berlin Heidelberg: Springer-Verlag.