Evolving Optimal Populations with XCS Classifier Systems

Tim Kovacs

October 3, 1996

Abstract

This work investigates some uses of self-monitoring in classifier systems (CS) using Wilson's recent XCS system as a framework. XCS is a significant advance in classifier systems technology which shifts the basis of fitness evaluation for the Genetic Algorithm (GA) from the strength of payoff prediction to the accuracy of payoff prediction. Initial work consisted of implementing an XCS system in Pop-11 and replicating published XCS multiplexer experiments from (Wilson 1995, 1996a). In subsequent original work, the *XCS Optimality Hypothesis*, which suggests that under certain conditions XCS systems can reliably evolve optimal populations (solutions), is proposed. An optimal population is one which accurately maps inputs to actions to reward predictions using the smallest possible set of classifiers. An optimal XCS population forms a complete mapping of the payoff environment in the reinforcement learning tradition, in contrast to traditional classifier systems which only seek to maximise classifier payoff (reward). The more complete payoff map allows XCS to deal with payoff landscapes with more than 1 niche (i.e. those with more than 2 payoff levels) which traditional payoff-maximising CS find very difficult. This makes XCS much more suitable as the foundation of animat control systems than traditional CS. In support of the Optimality Hypothesis, techniques were developed which allow the system to highly reliably evolve optimal populations for logical multiplexer functions. A technique for auto-termination of learning was also developed to allow the system to recognise when an optimal population has been evolved. The self-monitoring mechanisms involved in this work are discussed in terms of the design space of adaptive systems.

Contents

1 Introduction

This work is based on that of Stewart Wilson in (Wilson 1995, 1996b), two papers which present a new form of classifier system called XCS, results of experiments with it and theoretical analysis. XCS is significant in that it allows better representations of problem environments than traditional classifier systems, and as a result has the potential to have more sophisticated *animat*[1] cognitive architectures based on it (e.g. those which incorporate planning.)

The current work endeavours to explore the potential of XCS further by presenting new mechanisms, new experimental findings and further analysis (in addition to replication of published experiments). The main theoretical contribution of this work is the proposal of the *Optimality Hypothesis* (see 5.7), which proposes that under certain conditions optimal classifier populations can reliably be evolved. (This is an extension of Wilson's *Generalization Hypothesis* (see 5.6)). Experimental results provide support for (but do not prove) the Optimality Hypothesis. In particular, it is shown that XCS can reliably evolve optimal solutions for 6 and 11 multiplexer problems (see section 6.5).

1.1 The Structure of this Document

This document is divided into the following major parts:

- Section 2 sketches the context of the current work as an analysis of the functional properties of certain classes of control system.
- Section 3 provides a brief overview of the architecture of the traditional classifier system, a well-known example of an adaptive system. It will be contrasted with its more sophisticated descendent, XCS.
- Section 4 introduces the reinforcement learning paradigm and the multiplexer problems used with the systems discussed here.
- Section 5 contains a high-level description of Wilson's recent XCS system, an analysis of the system and brief discussions of its relationships with two other widely studied adaptive systems, the traditional classifier system and the Q-Learning technique.
- Section 6 introduces work with a Pop-11 implementation of XCS. In this section replication of experiments from (Wilson 1995, 1996a) are reported. Following this the results of original work using the XCS system are presented.
- Following the conclusion, various topics of less general interest are addressed in the appendices.

2 The Design Space of Adaptive Systems

Adaptive systems include a vast range of living, natural and artificial systems. One way to compare these diverse systems is to conceptualise their *architectures* within a *design space* of functional systems.[2] *Architecture dominates mechanism* (Sloman 1993) sums up the view that architectures have a greater influence on the capacities of the system than the mechanisms it consists of. Designs map into a *niche space* of requirements, and understanding a design involves understanding how it maps into niche space.

The *design-based* approach involves taking the role of an engineer who is trying to design a system that meets certain requirements, and is inspired by software engineering and conceptual analysis in philosophy. It involves analysing alternative sets of requirements, designs and implementations in an attempt to establish the nature of their relationships. It allows a high level functional comparison of systems, both natural and artificial, despite differences in origin or implementation. This comparison seeks to identify which aspects of a system are essential for given functions and which are not. Note that this approach does not require a full understanding of the requirements or the available tools at the outset, nor does it assume that there is a single correct design to be found. For discussion of the design-based approach see (Sloman & Humphreys 1992; Sloman 1993, 1994, 1995, 1996).

[1] Animats are autonomous adaptive agents implemented in software or hardware.
[2] An architecture is some level of abstraction of a functional system.

Reinforcement learning systems (section 4) are one form of adaptive system. The current work will focus on a particular kind of reinforcement learning system, the classifier system (see section 3).

2.1 Feedback Loops

> "(A basic negative feedback loop) is the simplest example of a teleological machine, i.e. a machine that appears to function with a definite 'goal' in mind, a representation of a state-of-affairs that it attempts to bring about. A basic negative feedback loop has four elements: (a) a sensory signal measuring an environment variable, (b) a reference value, (c) a comparator, and (d) an effector signal that alters the environment variable. The comparator compares the sensory signal with the reference value and computes an effector signal. The effector signal is such that it moves the environment variables closer to the reference value. Such a negative feedback control loop will tend to maintain the variable around the reference value equilibrium. A thermostat is a well-known example of this kind of system (temperature = environment variable, dial position = reference value, bi-metallic strip = comparator, turn heating on/off = effector signal)." (Wright 1995)

Self-monitoring is a form of feedback in which the environmental variable is internal to the system, and includes cases where the variable is a statistic based on other variables. Introducing capacities for self-monitoring may be considered as effecting a movement in design space which introduces corresponding changes in the niche space of the system. Some of these movements may be involve continuous change in some dimension while others may cross discrete boundaries. This project can be seen as an examination of some of the uses of self-monitoring in a particular form of adaptive system, the classifier system.

3 Traditional Classifier Systems

3.1 A Brief Overview

Classifier systems are a form of adaptive system introduced by John Holland in the mid-1970's (Holland 1975, see also Holland 1986) as a form of domain-independent learning system. The term "traditional" will be used to refer to those systems closely related to Holland's original system, primarily in order to distinguish them from the more recent XCS classifier system.

A classifier is essentially a condition-action rule with some associated values or *parameters* (these are typically estimates of the classifier's utility to the system). The condition part of a classifier is represented by a ternary bitstring composed from the set {0, 1, #}, while the action part is composed from {0, 1}.[3] A classifier system generates, evaluates and makes use of classifiers for decision-making in interaction with some problem environment. These functions are carried out by three major subsystems: the *performance system*, the *learning* (or *credit assignment*) *system*, and the *rule discovery system*. Each classifier has a *strength* parameter which represents the system's evaluation of the utility of that classifier. In traditional classifier systems the strength parameter is used as a measure of utility in both performance and rule discovery.

The following subsections describe the main features of the traditional classifier system's version of these subsystems very briefly; those familiar with classifier systems may want to skip to the next section. In section 5 the versions of these subsystems used in XCS, a new form of classifier system, are described in more detail.

3.2 The Performance System

The performance system is composed of four parts:

Input Interface Generates messages in the form of binary bitstrings representing detected environmental features.

Message List Stores messages generated by the input interface and by classifiers.

[3] In the traditional CS, multiple bitstrings can be linked with logical ANDs in a classifier's condition. In addition, a negation prefix can be used to indicate that the classifier is matched when no input matches its condition.

Classifier List A list of classifiers which are applied to the message list and which may add messages to it.

Output Interface Translates action messages from the message list into actions in the environment.

The basic execution cycle of the performance system is called a *major cycle*. It consists of the following steps:

1. Add the messages from the input interface to the message list.

2. Compare each message on the message list to the condition part of each classifier and record which pairs of messages and classifiers have matched. The # symbol in a condition will match either 1 or 0 in a message and is sometimes called the "don't care" symbol.

3. Generate new messages from the matches in 2 using the action parts of the classifiers. Classifiers are required to compete against each other to be allowed to post their messages, so only a subset of the matching classifiers will post messages.

4. Replace the contents of the message list with the new messages.

5. Use the output interface to translate the new messages into the system's actions. An environmental *reward* may or may not be returned as a result of the actions taken.

6. Return to step 1.

The classifier system interacts with its environment by repeating the major cycle. One application of classifier systems is to drive animats in which case the input and output interfaces link the classifier system to the animat environment (which may be either simulated or real). Classifier systems are also used for more abstract problems, such as the boolean multiplexer problems which will be discussed later.

The classifier system program is simply a particular classifier list, i.e. a collection of condition-action rules. Note that classifiers are computationally complete; a list of classifiers can implement any computable function (although not necessarily very efficiently).

3.3 The Learning System

The performance system itself is not adaptive, so the classifier system must include some form of learning system. In the traditional classifier system the *bucket-brigade* algorithm is used to allow the system to perform *credit assignment* with the classifiers. The *credit assignment problem* is the problem of determining which components of a complex system are responsible for producing which outcomes.[4] The bucket-brigade operates by adjusting the strength parameter of classifiers in response to the *reward* (or *payoff*) the system receives as a result of its actions. A classifier's strength is thus a prediction of the reward the system will receive if it selects that classifier from the set of classifiers matching the current messages and allows it to post its message. (Another way at looking at strength is a measure of how useful the classifier has been in the past.)

To allow selective activation of some classifiers *satisfied* (i.e. matched) during a major cycle an element of competition is introduced: each classifier that has its condition part satisfied makes a *bid* to become active. Only the highest bidders are allowed to post their messages. The bid of a classifier depends on two factors:

- Its strength, which is a measure of the classifier's "usefulness" to the system. The strength of a classifier should be modified as a result of the system's experience with a particular task domain.
- Its *specificity*, the number of 1's and 0's in its condition part. Specificity can be thought of as a measure of a classifier's relevance to a particular set of messages.

[4]The *temporal credit-assignment problem* is the version where the outcome to which components contribute does not occur for some time.

A classifier's bid is:

$$B = k * strength * specificity$$

where 'k' is a universal constant for all classifiers. The higher a classifier's strength (usefulness), the higher its bids and the more likely it will win the competition and post messages. The same applies for specificity.

Consequently, the behaviour of a classifier system can be modified by changing the strengths associated with classifiers. If the strength of classifiers that tend to lead to "useful" behaviour can be increased, and the strength of classifiers that tend to lead to "useless" behaviour can be decreased, the system will learn to produce more useful behaviour. The bucket-brigade algorithm is designed to bring about these types of changes in strength.

The basis for the bucket-brigade is information from the environment about whether or not the classifier system as a whole is behaving correctly. This is achieved via rewards. The system receives positive reward from the environment when it produces the right behaviour and negative reward when it produces incorrect behaviour. In some situations neither positive or negative reward will be received, or it may be received some time in the future, and it may be contingent upon other factors (hence the credit assignment problem).[5]

The bucket-brigade acts in two ways:

1. When a reward is received the bucket-brigade adds the reward value to the strength of all classifiers active during the major cycle. In other words, the algorithm changes the strength of all classifiers that were directly associated (in time) with the receipt of the reward. The mapping from classifier system state to reward is called the *payoff function*.

2. When a classifier is activated it pays the amount it bid to the classifiers that made it possible to become active (i.e. those which posted the messages which matched the classifier's condition). Consequently, the strength of the active classifiers is decreased by the amount of its bid. In this way the bucket-brigade also acts to increase the strength of classifiers that indirectly lead to useful behaviour, i.e. rewards. The bucket-brigade allows rewards to "circulate back" to antecedent classifiers that produce rewarding behaviours.

The algorithm is called a "bucket-brigade" because strength flows from an ultimate source, the rewarding mechanism, to distal producers of useful behaviour.[6]

3.4 The Rule Discovery System

A complete classifier system needs some means of generating new rules for use in the performance and learning systems. Well-known *genetic algorithm* (GA) techniques have been used as the main source of rule discovery in CS. Genetic algorithms were inspired by natural selection and operate by evolving generations of individuals which are successively more fit according to some fitness evaluation function. In traditional classifier systems, a classifier's strength is taken as a measure of its *fitness* or utility in rule discovery (in addition to its use in the performance system).

The basic operation of a genetic algorithm is summarised as follows:

1. Select classifiers for reproduction. The probability of a classifier being selected as a parent is based on its strength.

2. Apply genetic operators to the new classifiers. Copies of the parent classifiers are generated and transformed using genetic operators. The most commonly used operators are *crossover*, which combines elements of the bitstrings of two parents as in sexual reproduction, and *mutation* which is a change effected probabilistically on some part of the bitstring.

3. Select classifiers for deletion. In order for the population of classifiers to remain within some reasonable size limit, existing classifiers must be deleted as new ones are introduced. There are various means of selecting classifiers for deletion, for example probabilistically based on an inverse function of the classifier's strength (i.e. weaker classifiers are more likely to be deleted).

[5]To be precise, this applies only for multi step problems. In single step problems, the reward (positive or negative) is always received immediately, and no cycle influences any other, except through the action of the learning system. Also, reward may be some scalar value rather than just a positive/negative signal. Single and multi step problems are explained in section 4. More complex forms of environmental feedback (e.g. supervised learning) are beyond the scope of this discussion.

[6]Much of this section was drawn from (Wright 1995) with permission. Wright's work was based on (Riolo 1988).

3.5 Some Forms of Self-Monitoring in Classifier Systems

There are many ways in which a classifier system may make use of internal state to improve its operation in some respect. One obvious example is the hierarchical classifier system where the input to one layer is based on another. Another example is the use of register memory, which has been shown to overcome simple perceptual aliasing problems for animats in non-Markov environments (Cliff & Ross 1994).[7] A more general form of internal memory is the message list of the traditional CS. This has the potential to form adaptive chains of behaviour, although useful coupling of classifiers appears to be difficult to achieve and maintain in the traditional system (Riolo 1989).

The current work revolves around an even more fundamental form of self-monitoring in the classifier system, that of credit-assignment to classifiers in response to environmental reward. Many credit-assignment schemes have been implemented to date (e.g. the bucket-brigade, ZCS's "implicit bucket-brigade", tabular Q-Learning) using a variety of mechanisms. However, only a small part of the design space of credit-assignment systems has yet been explored. For instance, many variations on existing schemes could be produced by adding additional self-monitoring capacities. Some such variations will be explored herein. In some cases, systems which do not make use of self-monitoring are missing opportunities to optimise learning with the help of additional information. In other cases, it may not be possible for learning to occur (or to advance as far) without the use of such additional information (examples being the use of register memory bits to overcome perceptual aliasing, or the use of accuracy based fitness to deal with payoff environments with more than one niche).

This work, then, can be seen as an exploration of some example uses of performance monitoring (e.g. reward received, ratio of correct responses to incorrect responses, the achievement of goals and subgoals) and database analysis (directing exploration, measuring representational efficiency) to improve performance.[8] In some cases experiments were actually carried out with an XCS system, in other cases it was used as a conceptual framework in which to base discussion.

4 Reinforcement Learning Problems

In the most basic form of reinforcement learning scenario the interaction between the environment (the teacher) and the learner is highly restricted. The following demonstrates a single trial (cycle) of this scenario (in which the learner is a classifier system):

- **Input** A random binary bitstring is generated and presented to the CS as input from its environment.
- **Action** The CS must respond with either a 1 or a 0. (The response is generally referred to as the system's *action*.)
- **Reward** The correctness of the response is determined by the environment (e.g. by computing some function on the input string to derive a value which is then compared to the response). Some reward is then returned to the CS by the environment, e.g. the CS may receive 100 reward units for correct answers and 0 reward units for incorrect answers. How much reward is returned is typically related to the "correctness" of the CS's actions (but can really use any function - see section 4.1).

This is a *single step* problem as the input is generated randomly; the choice of action by the learner does not affect future inputs at all.[9] This cycle is typically repeated thousands of times, giving the system the opportunity to learn which actions to associate with which inputs. To do so it must form some kind of internal representation of the problem in order to predict which answers will result in more reward being returned.

[7] This approach has the advantage of greater simplicity over the message lists of traditional classifier systems, but Cliff & Ross conclude that the technique is not likely to scale well.

[8] Optimising representational efficiency may be seen as a goal in its own right, but tends to have implications for performance as well.

[9] In *multi step* problems, the choice of action may affect possible future inputs. Animat environments are a form of multi step problem (e.g. the animat's choice of direction to walk in on cycle t affects what it will sense on cycle t_{+1}). In multi step problems the environmental reward may be due to series of actions and thus there may be a delay in receiving rewards (this is the temporal credit assignment problem from section 3.3).

4.1 Payoff Landscapes, Payoff Niches and $XxA \Rightarrow P$ Maps

In a reinforcement learning problem some *payoff landscape* is used to determine the reward value to return to the learning system for its response to a particular input.

Suppose a reinforcement learning experiment uses the following payoff landscape:

Input string	Reward Level if Correct	Reward Level if Incorrect
00	200	0
01	200	0
10	100	0
11	100	0

where the answer 1 is always correct and the answer 0 is always incorrect. Input/action pairs which pay off at the same rate (i.e. whose rewards for correct answers r_c are equal and whose rewards for incorrect answers r_i are equal) are said to belong to the same *payoff layer* or *payoff niche*. (The term payoff niche will be given preference to help distinguish layers from levels.) This landscape has 2 payoff niches. The first includes the inputs {00, 01} while the second includes {10, 11}. In this case, each niche has 2 *payoff levels*, one for correct answers and one for incorrect answers. (Condition/action pairs which have the same reward (i.e. XxA pairs with the same P) belong to the same payoff level.)

The internal model which a reinforcement learning system uses to predict reward levels will form some kind of map of the input X to action A to prediction P space (written $XxA \Rightarrow P$). Payoff niches form regions within this space, and the CS uses classifiers to describe the boundaries and payoff levels of these regions.

4.2 Boolean Multiplexer Problems

Multiplexer functions are used as reinforcement learning problems for classifier (and other reinforcement learning) systems, using the basic form of reinforcement learning scenario discussed in section 4. Neural networks, classifier systems and Q-Learning are all forms of reinforcement learning systems which are able to learn multiplexer problems (at least the shorter multiplexer problems). Classifier systems learn the multiplexer problems by generating, evaluating and basing decisions on classifiers as described in section 3.

Boolean multiplexer problems are highly nonlinear logical functions commonly used for testing machine learning systems. They provide a series of related problems of increasing difficulty which makes them suitable for evaluating the ability of a system to scale up. Further, they are useful as a common measure of performance for different systems as they have been used with a variety of systems including neural networks (Anderson 1986), simple GAs (Goldberg, 1989), messy GAs (Skurikhin & Surkan), niche GAs (Booker 1989), perceptrons (Wilson 1990) and the BOOLE system (Wilson, 1987). All the experiments reported in this work were on forms of multiplexer problem.

Multiplexer functions exist for strings of length $L = k + 2^k$ with $k > 0$, a series that begins {3, 6, 11, 20, 37 ...}. At present only the 6, 11 and 20 multiplexers are typically used due to the difficulty of the succeeding ones.

In disjunctive normal form, the six multiplexer is as follows (the primes indicate negation):

$$F_6 = x_0'x_1'x_2 + x_0'x_1x_3 + x_0x_1'x_4 + x_0x_1x_5$$

The first k bits may be considered as an address indexing the remaining 2^k bits. The value of the function is the value of the indexed bit. For example, the value of 010100 is 1. $k = 2$ for the 6 multiplexer, so the first 2 bits are used as an address (whose value is 1), which indexes bit 1 (the second bit) of the remainder of the string. Eight classifiers will match this input string, the most specific, 010100, has no #'s while the most general, 01#1## has $2^k - 1$ #'s (i.e. all remainder bits save the indexed one are #). 01#1## is *maximally general*; it cannot be made any more general (i.e. cannot have any more #'s added) without losing its accuracy.

Multiplexers are used as a form of single step *Markov* environment for the classifier system. The multiplexer is encoded as a binary bitstring by the classifier system's input interface, and the system's action is returned by the output interface. The environment then uses the input/action pair to locate the appropriate reward value in the payoff landscape. This problem environment is Markov as the value of the input string is sufficient to determine the correct response, and the value of the reward depends only on the response of the system and the state encoded by the input string.

The payoff landscape traditionally used with multiplexer problems simply associates one reward level with correct answers and another with incorrect answers (i.e. it has 2 levels). In this case, learning to predict the payoff corresponds to learning the logical value of the function. (Wilson 1996a) reports experiments with this form of payoff landscape (see section 6.1.3 for replication). However, (Wilson 1995) used a different payoff landscape, one with 8 niches (each with a unique value for correct and incorrect answers - see 6.1.1 for replication). The ability to learn payoff landscapes with more than 1 niche is one advantage of XCS systems over traditional CS discussed in section 5.1.

The Suitability of Multiplexer Problems

A 6 multiplexer problem is not a completely trivial problem. Imagine a human being attempting to work out a means of predicting the correct answer in place of the classifier system. The number of trials required to solve the problem (if, indeed, the subject was able to solve it at all) would depend heavily on the subject's prior knowledge. It would be far easier to do with pencil and paper than purely mentally, indicating that humans are not well suited to this type of problem, perhaps due to temporary memory requirements. It seems unlikely that a rat in a Skinner box[10] would be able to learn to predict the correct response no matter how many trials it was given. However, rats do possess some level of intelligence. These may be indications that multiplexer problems are not very suitable for evaluating intelligent systems in general. Certainly they seem to bear little resemblance to the problems faced by real creatures. Nonetheless, they do provide well defined benchmarks useful for evaluating current reinforcement learning systems.

Animat problems (e.g. the "woods" environments used by Wilson) may be more suitable ways of evaluating the kinds of intelligent behaviour we expect from real creatures (and from animats).

5 XCS

5.1 Introduction

XCS is a class of classifier systems introduced in (Wilson 1995) whose primary distinguishing features are the basing of classifier fitness on the accuracy of classifier reward prediction and the use of a *niche genetic algorithm* (i.e. a GA which only operates on a subset of the general classifier population [P] on each invocation). In addition, there are many more subtle differences between traditional classifier systems and XCS systems as the reader will discover later in this section. XCS has many features in common with Wilson's earlier ZCS work (Wilson 1994).[11]

Classifier systems have traditionally based fitness on classifier strength (i.e. a predictor of the payoff to be received if the classifier's action is taken). Wilson was motivated to change the basis of fitness calculation for a number of reasons.[12]

- Different environmental niches may have different payoff levels (see the example in 4.1). If fitness is simply based on payoff prediction, then classifiers in the higher-payoff niches will tend to take over the population. By basing fitness on classifier accuracy, XCS allows classifiers to evolve without regard for relative niche payoff level.

- Traditional classifiers systems only attempt to find the most rewarding classifiers, in contrast to reinforcement learning work which has focused on the construction of relatively complete maps of the payoff environment. Maintaining a more complete map is more expensive (at least in terms of space) but may benefit action selection as a range of alternative predictions are made. A more complete map

[10] I.e. the rat takes the place of the classifier system in the cycle above. Rather than some scalar value, the rat might receive a food pellet for correct answers and an unpleasant electric shock for incorrect answers. It could signal its answers by any number of means, e.g. by pressing a lever to indicate 1, and not pressing it to indicate 0.

[11] ZCS is intended as a minimalist classifier system whose mechanisms are more easily understood than those of the traditional CS. Wilson showed that learning in ZCS has strong similarities to the reinforcement learning technique *Q-Learning* (see 5.4).

[12] Please see (Wilson 1995) for a more detailed discussion.

may also make it easier to escape from local minima (Wilson 1995). Basing fitness on accuracy allows XCS to construct such payoff maps (see Appendix D).

- In traditional classifier systems there is a lack of pressure for accurate generalizations in the classifier conditions to evolve. However, XCS tends to evolve *maximally general* classifiers (classifiers which could not have any more #'s without becoming inaccurate) within the limits of an accuracy criterion (see section 5.6 on the Generalization Hypothesis). Evolving maximally general classifiers is desirable because it minimises the number of different concepts (represented as classifier conditions) the system needs to deal with and thus makes the working system simpler and more efficient. Accuracy-based fitness intrinsically promotes accurate generalizations.
- With fitness based on predicted payoff, the system does not distinguish between accurate classifiers and overgeneral classifiers with the same predicted payoff (see 5.2). At the same time, overgeneral classifiers may have an advantage since they are matched more often. (Although this should ultimately be offset by their inaccuracy, their early success could inhibit GA performance.) Basing fitness on accuracy allows the system to avoid giving undue preference in the GA to overgeneral classifiers.

In traditional CS, classifier strength plays a double role: it is used as a predictor of payoff in action-selection, and as a measure of fitness in the GA. In XCS, strength is replaced by three parameters: *payoff prediction, prediction error* and *fitness.* Prediction fills the role of strength in the action-selection component, and is also used to calculate prediction error, a measure of classifier accuracy. Fitness is an inverse function of the prediction error (put another way, it is a function of the accuracy of the prediction).

By relieving payoff prediction of its double role Wilson has significantly improved the classifier system architecture. In XCS, payoff prediction is only relevant in action selection which means XCS classifiers are free to map any region of the input/action/payoff space. As a results XCS tends to form "complete and accurate mappings X x A ⇒ P from inputs and actions to payoff predictions ... which can make payoff-maximising action-selection straightforward" (Wilson 1995). The attempt to construct a complete mapping of the payoff environment is in the spirit of much reinforcement learning work, and indeed the function of the learning subsystem in XCS is related to the reinforcement learning technique Q-learning (see section 5.4).

Following Booker's idea of conducting the GA in *niches* (Booker 1982), XCS's GA operates on subsets of the classifier list. Drawing classifiers for crossover from a subset of the population should be more effective than *panmictic* (i.e. global population) selection as the classifiers in a match or action set are related (see section 5.5.3 for definitions of these sets). Another interesting feature is that the probability of a classifier's deletion is proportional to its estimate of its match set size. This has the effect of allocating resources (GA invocations) approximately equally to the different match sets.

A number of other researchers have incorporated accuracy information in their classifier systems (for an overview see (Wilson 1995)). Indeed, Holland, in the original paper on classifier systems (Holland 1975), suggested that accuracy information be incorporated in calculating classifier fitness, but he later focused on payoff based fitness. Unlike other systems, XCS has completely shifted to accuracy based fitness.

Some of the problems with the traditional CS discussed above can be overcome by modifying its architecture (e.g. by sharing payoff among active classifiers, or by using a niche GA - See (Wilson 1995) for discussion). However, these modifications still leave the CS without evolutionary pressure towards accurate generalization, or a means of detecting overgeneral classifiers (see 5.2). In order to simplify matters, the discussion herein will be restricted to comparison of XCS and the traditional CS.

Wilson refined the XCS system in (Wilson 1996a) by introducing *subsumption deletion* (see section 5.5.7) and by moving the site of the niche GA from the match set to the action set. These changes were successful in producing smaller (but not yet optimal) final population sizes (see replication in section 6.1.2).

5.2 Overgeneral, Maximally General and Suboptimally General Classifiers

Classifiers express generalizations using the don't care symbol # in their conditions. For example, a classifier with condition 00# will match both 001 and 000 and treats these two inputs as equivalent. A classifier is either overgeneral, maximally (optimally) general, or suboptimally general in respect to the inputs it matches, as the following example illustrates.

Suppose an XCS system attempting to maximise payoff from the payoff landscape in section 4.1 has a population consisting of the following classifiers:

Classifier	Condition	Action	Predicted Payoff	Prediction Error	Accuracy
a	##	1	100	0.5	0.0
b	0 #	1	200	0.0	1.0
c	1 0	1	100	0.0	1.0
d	1 1	1	100	0.0	1.0

(The accuracy of a is 0.0 because its prediction error exceeds some threshold level called the *accuracy criterion*.) We can describe each classifier as one of the following:

- **Overgeneral** An overgeneral classifier matches too many inputs, as in the case of classifier a above. What makes the classifier overgeneral is that some of the condition/action pairs it refers to have different payoff levels (as shown in 4.1). Classifier a is overgeneral; its conceptualisation of the input space is inaccurate, and it should ideally be replaced by more specific classifiers whose conditions do not cross payoff level boundaries.

  If fitness is based on strength, overgeneral classifiers may survive as "guessers" which are sometimes correct and sometimes not. The prediction of a, 100, is an average of the payoffs it receives. To a system which bases fitness on strength (a prediction of payoff), a will look as valuable as c or d for use in rule discovery. However, to a system which bases fitness on accuracy of payoff prediction, it is clear that a is not as valuable as the other classifiers.

- **Maximally General** A classifier which matches all and only those inputs from the same payoff level is *maximally general*. It cannot become any more general (add any more #s) without becoming inaccurate (matching classifiers from more than one payoff level). Classifier b is maximally general as it matches all and only those classifiers in one payoff level (i.e. the one for inputs: {00, 01}).

- **Suboptimally General** Classifiers c and d are suboptimally general; each only matches inputs from the same payoff level, but neither covers its payoff level completely. Thus they could each be made more general without losing accuracy (notice that they are perfectly accurate, i.e. their accuracy is 1.0). Ideally they would both be replaced by a single more general classifier with condition 1#.

5.3 Limitations of the Descriptive Power of Single Classifier Conditions

There are two differences between the classifier conditions used in XCS work to date and those of traditional CS. In XCS, each classifier has been limited to a single condition bitstring, and no negation prefix is allowed. The following discussion refers to the single bitstring form of condition used with XCS.

Classifier conditions define regions within the input action space XxA. However, the ternary alphabet, as used in XCS classifier conditions, is not able to describe arbitrary regions within this space. For example, to match both 01 and 10, a classifier condition must be ##. However, this classifier will also match 00 and 11, which may belong to different payoff levels. I.e. no single condition can describe all three of the following $XxA \Rightarrow P$ mappings: i) 001 x 0 $\Rightarrow$ 100, ii) 010 x 0 $\Rightarrow$ 100 and iii) 000 x 0 $\Rightarrow$ 0, even though the first two belong to the same payoff level and thus could, in principle, be generalised over.

Another two locations that cannot both be matched by a single classifier are 001 x 0 $\Rightarrow$ 100 and 001 x 1 $\Rightarrow$ 100 as don't cares are not allowed in the action part of the classifier.

This may be viewed as an inability to express certain logical relationships between bit positions. E.g., a condition can specify that bits a AND b both be 0, but cannot specify that only bit a OR bit b be 0 (i.e. an exclusive OR).

Due to these limitations, inexpressible (for a single condition/action bitstring) generalizations may exist within the payoff landscape. This is the case with the multiplexer problems, and as a result XCS requires a minimum of 16 classifiers to completely map the payoff landscape of a 2 payoff level 6 multiplexer problem.

The use of conjunctions of classifier conditions and a negation prefix allow classifiers in the traditional CS to overcome the restrictions discussed above. If these two features were incorporated into XCS the optimal population size for a 2 level 6 multiplexer would be 2 rather than 16 (although the two classifiers involved would each be more complicated, which might tend to offset any advantage derived from there being less of them). In this case there would simply be one maximally general classifier for each payoff level, regardless of the number of payoff levels.

5.4 Relationship to Q-Learning

Like classifier systems, Q-Learning (Watkins 1989; Watkins & Dyan 1992) is a form of much-studied reinforcement learning system. A Q-Learning system maintains a reward prediction P for each input action XxA combination. (In Q-Learning work, P is referred to as the *quality* of the input action pair and is written $Q(x,a)$. Quality values are stored in a Q-Learning table.) Thus for a 6 multiplexer problem, a Q-Learning system needs to maintain XxA, or $2^6 x 2^1 = 128$ estimates of P (i.e. the system needs a table of size 128). The inability of the basic Q-Learning system to generalise is a serious weakness as the number of estimates it needs to maintain can easily become unmanageable.

Q-Learning and CS have traditionally been considered different approaches to reinforcement learning, possibly because of their different origins. However, the similarities between the two have recently been pointed out by a number of researchers (see Dorigo & Bersini 1994). It can be seen that the following definition of reinforcement learning can be applied to classifier systems.

> "Reinforcement Learning (RL) is a class of problems in which an autonomous agent acting in a given environment improves its behaviour by progressively maximising a function calculated just on the basis of a succession of scalar responses (rewards or punishments) received from the environment. No complementary guidance is provided for helping the exploration/exploitation of the problem space, and therefore the agent can rely only on a trial-and-error strategy." (Dorigo & Bersini 1994)

ZCS and XCS are forms of CS in which the traditional bucket-brigade has been replaced with a system resembling Q-Learning, drawing the fields of Q-Learning and CS research closer together. Each field can gain from insights provided by the other. As will be discussed, CS research can benefit from the Q-Learning approach of constructing complete mappings of the environment. In addition, it has been shown that in certain environments Q-Learning converges to an optimal policy (Watkins & Dayan 1992) and that a restricted form of CS does as well (Dorigo & Bersini 1994). In return, Q-Learning can benefit from existing CS work on the uses of internal state, structural change, and generalization (effected in CS by the use of the don't care symbol #). These are all areas of recent work in Q-Learning. A comparison of the ability of XCS, traditional CS, and Q-Learning to form complete and accurate maps of various multiplexer payoff environments is made in Appendix D.

5.5 Overview of XCS

This overview is drawn from Wilson's description of XCS in (Wilson 1995) and an attempt has been made to retain his terminology and notation in order to reduce confusion. The parameter list in section 10 reflects this. Some of Wilson's newer (post-(Wilson 1996a)) developments in the XCS architecture are included in this overview and hopefully some aspects of the system will be clarified.

5.5.1 The MAM Technique

The MAM technique ("moyenne adaptive modifiée") was introduced in (Venturini 1994) as a means of speeding up the estimation of various classifier parameters based on information obtained on successive cycles. Using this technique, a parameter is updated using one method early on and a second method later. The reasoning is that the first method can be used to quickly get a rough approximation of the true value of the variable, while the second method can make more conservative adjustments and refine the estimate.

In all experiments reported in this work the MAM technique was implemented as follows. Each classifier includes an *experience* parameter which is a count of the number of times it has occurred in the action set (and thus the number of times it has been updated). For the first few updates the parameter is simply an average of the samples to date. However, after $1/\beta$ updates have occurred, the standard Widroff-Hoff delta rule $p_j \leftarrow p_j + \beta(P - p_j)$ is used to update the parameter (see 5.5.4 for examples).

5.5.2 Macroclassifiers

XCS makes use of *macroclassifiers* in its operation as a means of increasing the run-time speed of the system. Statistics based on distribution of macroclassifier *numerosity* within the population also offer potentially interesting information.

Macroclassifiers are simply a type of classifier with a numerosity parameter. Instead of directly inserting newly generated classifiers into the population, XCS checks to see if there is an existing classifier with the same condition and action as the new one. If so, the new classifier is discarded and the existing classifier has its numerosity field incremented by one. Similarly, when a classifier is selected for deletion it is actually only deleted if it has a numerosity of one. Otherwise, its numerosity is decremented by one. The numerosity of a classifier indicates how many *microclassifiers* (regular classifiers) it represents. As a result, XCS needs to be implemented in such a way as to take a classifier's numerosity into account and treat it as an equivalent number of microclassifiers in all relevant cases. For example, in calculating the probability of selecting classifiers for reproduction, the system must treat a classifier with a numerosity of 5 as 5 times more likely to be selected than one of the equivalent microclassifiers.[13]

The following table shows a population of microclassifiers:

Classifier	Condition	Action	Predicted Payoff	Prediction Error	Accuracy
a	## 00 11	1	200.0	0.0	1.0
b	## 00 11	1	200.0	0.0	1.0
c	## 00 11	0	100.0	0.0	1.0
d	00 11 10	1	100.0	0.0	1.0
e	10 ## 01	0	501.1	0.0011	0.75
f	10 ## 01	0	500.8	0.0008	0.80
g	10 ## 01	0	500.3	0.0003	0.95

The next table shows the equivalent population implemented with macroclassifiers (note the additional numerosity parameter):

Classifier	Condition	Action	Predicted Payoff	Prediction Error	Accuracy	Numerosity
m	## 00 11	1	200.0	0.0	1.0	2
n	## 00 11	0	100.0	0.0	1.0	1
o	00 11 10	1	100.0	0.0	1.0	1
p	10 ## 01	0	500.3	0.0003	0.95	3

As a more fit population is evolved, the rule discovery component tends to produce more and more copies of the same highly fit condition/action pairs, so the proportion of the population consisting of classifiers with a numerosity greater than one tends to grow as the population evolves. As a result, there can be a significant reduction in the length of the classifier list, resulting in a improvement of processing speed.

In keeping with (Wilson 1995), the term "macroclassifier" will be reserved for those situations in which it makes explanation clearer. In an attempt to reduce confusion they will normally simply be referred to as "classifiers". However, it should be understood that all classifiers in XCS have a numerosity field and are thus macroclassifiers.

Wilson reports informally (Wilson 1995) that there is no apparent difference between a system which uses macroclassifiers and one which does not, other than the speed of operation. Appendix A reports the results of a comparison of POP-XCS running 6 multiplexer trials with and without the use of macroclassifiers.

5.5.3 The Performance System

The performance system in XCS is similar to that of a traditional classifier system, except that it has no message list and thus no internal memory. Strings are sent directly from the input interface to the module which generates the set of matching classifiers, and the action selected by the system is sent directly to the output interface.

Operation is as follows. A *match set* [M] is formed from those classifiers in the general population [P] which match the system's input. Next, a *system prediction* $P(a_i)$ is computed for each action a_i in [M] using a fitness-weighted average of the predictions of classifiers advocating a_i. The system prediction for each advocated action is placed in a *prediction array* in preparation for action selection. Some actions may not be advocated by any classifier in the current [M] and will have a void prediction, meaning they cannot be

[13]Technical note: Because the fitness calculation in XCS already takes numerosity into account, the fitness of a classifier is *not* multiplied by its numerosity in determining its probability of selection.

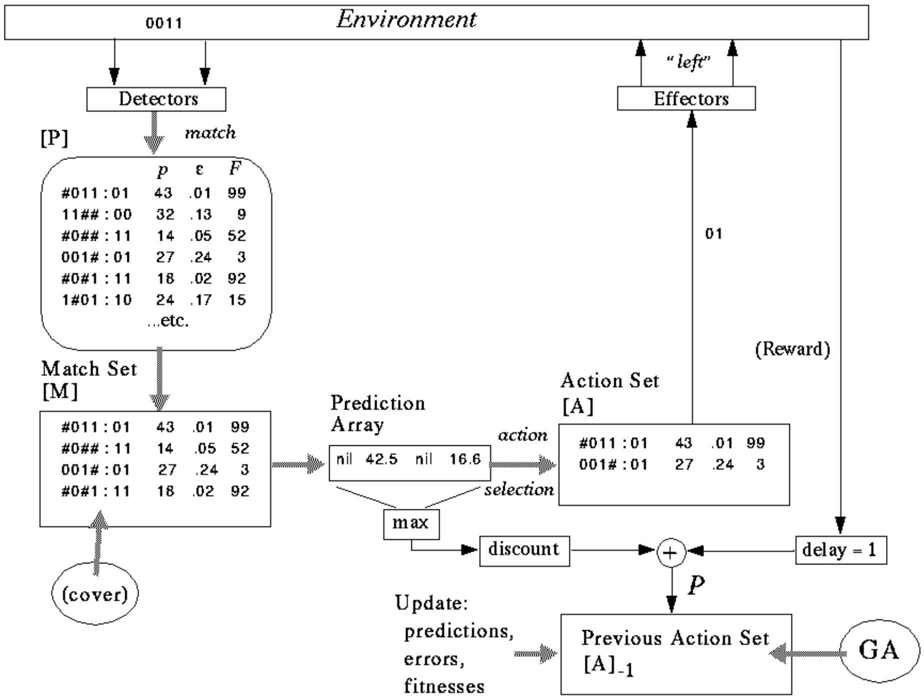

Figure 1: This schematic illustration of XCS is reproduced from (Wilson 1996a) by kind permission of Stewart Wilson.

selected. If no action is advocated (i.e. [M] is empty), then a random classifier is generated through *covering* (see section 5.5.5).

The system next selects an action from the prediction array and forms an *action set* [A] of classifiers in [M] advocating the selected action. Action selection may occur in any of a number of ways. The chosen action is sent to the output interface and an environmental reward may be returned as a result.

Wilson reports (Wilson 1996c) that he has begun to use a modified method of calculating the prediction array which is not mentioned in either of the XCS papers (Wilson 1995, 1996a). In this new version, if a classifier's experience is less than a threshold (e.g. 20), its fitness is considered as $1/16^{th}$ of its actual value for the purpose of calculating the system prediction. This has the effect of reducing the influence of relatively untested classifiers which may have unreasonably high fitness values. This technique has the advantage of having no effect if all classifiers involved are inexperienced since they are all then subjected to the same devaluation.

The basic execution cycle of the performance system is as follows (please refer to figure 1):

1. Obtain the single input string from the environment.

2. Form the match set [M] of classifiers whose condition matches the input string.

3. Select an action based on the advocacy of the classifiers in [M]. This may be accomplished by a variety of methods.

4. Form the action set [A] of classifiers in [M] which advocated the action selected in 3 above.

5. Translate the selected action into the output of the system.

The more complete $XxA \Rightarrow P$ mapping of XCS relative to traditional CS will allow the development of more sophisticated action selection techniques than previously possible in CS. XCS systems can predict the consequences of alternative (non payoff maximising) actions which in principle will allow them to deal more effectively with complex environments, e.g. those which require planning.

5.5.4 The Learning System (or Reinforcement Component)

In XCS the traditional bucket-brigade has been replaced with a Q-Learning-like system. The effect of the use of this system, in combination with accuracy-based fitness, is to construct more complete mappings of the problem space than traditional CS can.

Reinforcement in XCS works somewhat differently for single step and multi step problems. For single step problems the classifiers in [A] are updated using

$$P \leftarrow current\ cycle's\ reward$$

where P is the payoff returned by the environment. For multi step problems, the previous cycle's action set $[A]_{-1}$ is updated using the sum the of the previous cycle's reward and the discounted maximum of the prediction array.[14]

$$P \leftarrow previous\ cycle's\ reward + (\gamma \times \max(P(a_i)))$$

Wilson (Wilson 1996c) has experimented with the following parameter update order: 1. prediction error (with MAM) 2. prediction (with MAM) and 3. fitness (without MAM). (This is the order used in all work reported here.) This is a conservative order as the fitness begins at a low value and rises slowly. This is a result of updating the prediction error first (typically resulting in a large error on the first update as a default value is used for the as-yet unupdated prediction) and of not using the faster MAM technique for updating the fitness parameter. This order works well on the more difficult 20 multiplexer problem where fitness needs to established over the course of many input matches due to the size of the bitstrings involved. Other update orders may be more effective on other problems, but in any case the differences are not great.

Using the update order discussed in the previous paragraph, the reinforcement process for each classifier in the set (i.e. [A] or $[A]_{-1}$) is:

- **Update Prediction Error**

  if experience $< 1/\beta$ then

  $\varepsilon_j \leftarrow average(|P - p_j|)$

  else

  $\varepsilon_j \leftarrow \varepsilon_j + \beta(|P - p_j| - \varepsilon_j)$

- **Update Prediction**

  if experience $< 1/\beta$ then

  $p_j \leftarrow average(P)$

  else

  $p_j \leftarrow p_j + \beta(P - p_j)$

- **Update Fitness** This has three steps:

  - **Calculate Accuracy**

    if $\varepsilon_j > \varepsilon_o$ then

    $\kappa_j = \exp[(\ln \alpha)(\varepsilon_j - \varepsilon_o)/\varepsilon_o)] \times 0.1$

    else

    $\kappa_j = 1$

[14] If a multi step problem is completed on the first cycle the update occurs as for a single step problem.

- **Calculate Relative Accuracy** $\kappa'_j = \kappa_j / \sum \kappa_{[A]}$

- **Calculate Fitness** $F_j \leftarrow F_j + \beta(\kappa'_j - F_j)$

The accuracy calculation considers classifiers with error up to ε_o to be accurate (that is, they have an accuracy of 1.0). For error above ε_o, accuracy declines sharply. ε_o is the *accuracy criterion* which is used in section 5.6 to define what is meant by "equally accurate" classifiers.

5.5.5 The Rule Discovery System

The Niche Genetic Algorithm

The genetic algorithm in XCS acts upon niches rather than panmictically. Originally the GA operated upon [M] but was moved in (Wilson 1996a) to [A].

> "...the rate of incidence of the GA is controlled with the aim of allocating classifier resources approximately equally to the different match sets (such an allocation being consistent with the purpose of forming a relatively complete mapping). This cannot in general be achieved if the GA simply occurs with a certain probability in each match set. Depending on the environment, some match sets (niches) may occur much more often than others. Instead, the GA is run in a match set if the number of time-steps since the last GA in that match set exceeds a threshold. As a result, the rate of reproduction per match set per unit time is approximately constant - except in the most rarely occurring match sets." (Wilson 1995)

The genetic algorithm operates as follows:

1. **Select classifiers for reproduction.** The probability of a classifier being selected for reproduction is proportional to its fitness.

2. **Apply genetic operators to the new classifiers.** Copies of the parent classifiers are generated and transformed using genetic operators. Crossover occurs with probability χ per pair of chromosomes (i.e. per pair of bitstrings) and mutation occurs with probability μ per allele (i.e. per bit).

3. **Select classifiers for deletion.** If the population size M exceeds the limit N, classifiers are deleted to return M to N. In the following quotation from (Wilson 1995), Wilson discusses two methods of selecting classifiers for deletion with XCS.

    1. "Every classifier keeps an estimate of the size of the match sets in which it occurs. The estimate is updated every time the classifier takes part in an [M], using the MAM technique with rate β. A classifier's deletion probability is set proportional to the match set size estimate, which tends to make all match sets have about the same size, so that classifier resources are allocated more or less equally to all niches (match sets).[15]

    2. A classifier's deletion probability is as in (1), except if its fitness is less than a small fraction δ of the population mean fitness. Then the probability from (1) is multiplied by the mean fitness divided by the classifier's fitness. If for example δ is 0.1, the result is to delete such low-fitness classifiers with a probability 10 times that of the others." (Wilson 1995)

Covering

Another method of introducing classifiers into the population is *covering.* When a classifier is created through covering, its condition is made to match the current system input and it is given a randomly chosen action. Each allele in the condition is then mutated with probability $P_\#$ into a #. The covering classifier is then inserted into [P] and if necessary a classifier is deleted using the normal method.

Covering is used to supplement an existing [M] which has been found deficient for one of two reasons:

[15] An estimate of the size of [A] instead of [M] is kept if the GA operates in [A].

- [M] is empty. Covering ensures that some classifier always matches the input, and can be used in place of an initial population.
- The total prediction of [M] is less than ϕ times the mean prediction of [P]. This condition will be triggered in multi-step environments if the system becomes stuck in a loop as the discounting mechanism will cause the predictions of the classifiers involved to fall steadily. Inserting a new classifier via covering is usually sufficient to break the system out of the loop.

5.5.6 Classifier Parameter Initialisation

Each of the three means of introducing new classifiers into [P] has its own method of determining the initial settings of the classifier parameters. The aim in setting initial values is to make the best guess possible as to the true parameter value in order to improve system performance.

The Initial Population. If an initial population is used, the parameters are set to constant default values.

Covering. Classifiers created through covering are initialised as follows: initial prediction and error are set to the population means and initial fitness is set to 0.1 times the mean fitness of [P].

Genetic Algorithm. If crossover occurred, prediction is the mean of the parent's predictions. Otherwise, prediction is the same as the parent's prediction. Prediction error is always set to 0.25 times the mean error of [P] and fitness is set to 0.1 times the mean fitness of [P].

5.5.7 Subsumption Deletion

Subsumption deletion was introduced in (Wilson 1996a) as a way of biasing the genetic search towards more general (but still accurate) classifiers. Using subsumption deletion, when new classifiers are generated they are compared to existing classifiers rather than directly inserting them into [P]. This comparison is done to determine if the new classifier is *subsumed* by an existing one. New classifiers are compared to their parents and members of [A] in checking for subsumption.

New classifiers are subsumed by an existing classifier if the existing classifier i) has *experience* exceeding some subsumption threshold value (i.e. its parameters have been adjusted some minimum number of times), ii) if it is accurate (i.e. has an accuracy of 1.0) and iii) if it logically subsumes the new one (the inputs it matches are a superset of the inputs matched by the new classifier). If a new classifier is subsumed, it is discarded and the subsuming classifier's numerosity is incremented.

> "...(subsumption deletion) may be viewed genetically as a kind of directed mutation. In effect, for parents "known" to be accurate, the GA is constrained to generate and evaluate only offspring that are more general than the parents." (Wilson 1996b)

The use of subsumption deletion is effective in reducing the population size of the system (see 6.1.2 for evaluation).

5.5.8 Explore vs. Exploit Behaviour

Learning systems can engage in two forms of activity: exploration and exploitation. The problem of deciding how to allocate resources between these two activities is often called the "explore/exploit dilemma".[16] More resources should typically be allocated to exploratory behaviour in the early stages of problem-solving in order to avoid local minima. As the system learns about its environment, more resources can be shifted to exploiting it (maximising payoff).

The problem of selecting between exploration (maximising learning potential) and exploitation (maximising performance) was solved in XCS as follows. In all experiments the nature of each cycle (explore or exploit) was determined randomly at the outset with equal probability of either occurring. On explore cycles the system's actions were selected randomly from those advocated in the prediction array but the learning and rule discovery components operated normally. On exploit cycles, action selection was deterministic (the action with the highest advocacy was chosen), but learning and rule discovery were disabled. Results were only recorded for the exploit cycles (as indicated on the graphs).

[16](Wilson 1996b) discusses various means of addressing this dilemma.

5.6 Wilson's Generalization Hypothesis

It appears that the interaction of accuracy based fitness and a niche GA results in evolutionary pressure to generate classifiers that are both accurate and maximally general (as general as possible while remaining within some accuracy criterion). Wilson refers to this as the *Generalization Hypothesis* and explains the logic behind it thusly:

> "Consider two classifiers C1 and C2 having the same action, where C2's condition is a generalization of C1's. That is, C2's condition can be generated from C1's by changing one or more of C1's specified (1 or 0) alleles to don't cares (#). Suppose that C1 and C2 are equally accurate in that their values of ε are the same. Whenever C1 and C2 occur in the same action set, their fitness values will be updated by the same amounts. However, since C2 is a generalization of C1, it will tend to occur in more match sets than C1. Since the GA occurs in match sets, C2 would have more reproductive opportunities and thus its number of exemplars would tend to grow with respect to C1's (or, in macroclassifier terms, the ratio of C2's numerosity to C1's would increase). Consequently, when C1 and C2 next meet in the same action set, a larger fraction of the constant fitness update amount would be "steered" toward exemplars of C2, resulting through the GA in yet more exemplars of C2 relative to C1. Eventually, it was hypothesised, C2 would displace C1 from the population." (Wilson 1995)

The logical conclusion of this preference for the more general of two equally accurate classifiers is that the most general yet accurate, or *maximally general*, classifier for a payoff level will tend to be evolved and tend to displace others from the payoff level. This is the basis of the *Optimality Hypothesis* proposed in section 5.7.

5.7 Proposal of the Optimality Hypothesis

As an extension to Wilson's Generalization Hypothesis, I propose the *Optimality Hypothesis*. Given that, according to the Generalization Hypothesis, evolutionary pressure towards maximally general classifiers exists in XCS, an XCS system should eventually evolve a maximally general classifier for any payoff level which it is able to sample sufficiently. We will define the *criterion of sampling sufficiency* for a payoff level as the following requirements:

1. That the samples obtained are representative enough of the payoff level for the learning and rule discovery components to tend to locate the maximally general classifier for that payoff level.

2. That the samples are obtained sufficiently frequently to allow the system to evaluate the classifiers matching them before the GA deletes them.

Precisely what range of sample series is sufficient to meet these requirements for a payoff level is determined by the architecture and parameters of the system. One area of classifier design that may prove fruitful to explore is attempts to ensure this criterion is met by adjusting the GA search rate parameters for each payoff level.

Another requirement for the Optimality Hypothesis to apply may be that the problem be Markov (see 4.2). Ideally the criterion of sampling sufficiency would be formally defined and used in a proof of the Optimality Hypothesis. However, this is well beyond the scope of the current work.

Environments with Equiprobable Inputs

The first part of the criterion of sampling sufficiency essentially says that the actual sample inputs received must be well-distributed over the space of possible inputs. Problems where input strings occur equiprobably, as in the boolean multiplexer experiments, satisfy this in the long run.[17]

[17] "In the long run" is a reference to *Bernoulli's Theorem.* (Hayes 1988) offers this "more or less precise statement" of the theorem:

> "If the probability of occurrence of the event X is p(X), and if N trials are made, independently and under exactly the same conditions, then the probability that the relative frequency of occurrence of X differs from p(X)

If *all* input strings are equiprobable, *all* payoff levels should, in the long run, evolve maximally general classifiers. Furthermore, as a maximally general classifier will tend to occur in more match sets than any of the classifiers it subsumes and will thus have more reproductive opportunities, the ratio of its numerosity to any classifier it subsumes will increase.

In short, when an XCS system is run on a problem where all inputs occur equiprobably, the system will, in the long run, evolve a population containing a maximally general classifier for each payoff level, and that classifier will have a greater numerosity than any other in the payoff level.

Optimal Populations

We will refer to the set of maximally general classifiers for each payoff level in the payoff environment as [O]. On its own, [O] is the optimum population for a problem; it is the smallest possible population of classifiers to form an accurate covering of the input space.[18] Once the system has evolved a population which includes [O], the constant injection of new classifiers by the GA becomes problematic, i.e. once [P] contains [O], new classifiers are superfluous and render [P] non-optimal.

The Optimality Hypothesis states that:

> For each payoff level satisfying the criterion of sampling sufficiency, the GA will, in the long run, evolve a maximally general classifier which will have greater numerosity than any other classifier in the payoff level. When all payoff levels satisfy the criterion of sampling sufficiency, a complete set of maximally general classifiers, [O], will, in the long run, evolve within the population [P]. Further, elements of [O] will be distinguishable from the rest of [P] on the basis of their numerosity, so that, once [O] is complete, [P] may be reduced to [O] by removing all classifiers but those with the highest numerosity in their payoff level.

5.7.1 Obtaining Optimal Populations in Practice

Given that evolutionary pressure towards maximally general classifiers exists, the following problems remain in obtaining an optimal population:

- The point at which evolution of the complete set of maximally general classifiers [O] within [P] has occurred needs to be identified. One of the conclusions of this work is that in the problem-independent case it may only be possible to estimate this event (see section 6.4.1).
- Superfluous classifiers (elements of [P] not in [O]) need to be eliminated from the population. This can be achieved by the combined use of two techniques introduced by this work:
  - Dynamic Condensation (section 6.5), a technique for eliminating low-fitness classifiers from the population.
  - Auto-termination (section 6.6), a technique for identifying the point past which further condensation is counter-productive.

5.7.2 More Difficult Types of Problem

Problems in which all inputs occur equiprobably form a small subset of reinforcement learning problems which does not include many typical animat problems. Certainly real creatures encounter some situations far more commonly than others (e.g. eating breakfast vs. winning the lottery), and they may never encounter some possible situations. Nonetheless, we cannot expect a system to construct mappings of regions of the payoff environment it never encounters (not when it learns solely on the basis of a scalar reward for actions

---

by any amount, however small, approaches zero as the number of trials grows indefinitely large." (Hayes 1988)

For practical purposes it may be a matter of establishing confidence limits for the evolution of a maximally general classifier by a certain number of cycles.

[18] Saying that [O] is accurate means each classifier in [O] meets the accuracy criterion ε_o. E.g. a single classifier, ######, could be evolved as a complete solution to the 6 multiplexer problem but would obviously be very inaccurate (it would answer incorrectly half the time). To be the smallest possible population to cover the input space means that the classifier conditions will be non-overlapping; all possible inputs will be matched by one and only one classifier.

which are in response to input it cannot select). In other words the information available to a reinforcement learning system limits what it can learn.[19]

However, we can say something about environments in which inputs do not occur equiprobably. In environments where the input *tends* to satisfy the criterion of sampling sufficiency, XCS will *tend* to evolve a full set of maximally general classifiers. Similarly, XCS will, in the long run, evolve maximally general classifiers for those payoff levels in the environment which satisfy the criterion of sampling sufficiency, even if other payoff levels do not.

In multi step problems where the system is able to influence the input it will receive in the future, it might not be unreasonable for the system to actively pursue exploration of payoff level it feels could benefit from further exploration. For example, an animat might realise that it had infrequently (or never) eaten a certain type of plant, and that as a plant of this type was nearby and as the animat was in good health it might profit by sampling the plant to see if it was edible or poisonous. This particular scenario requires some sophistication on the part of the system, but should nonetheless illustrate the principle that directed exploration of payoff environments may be beneficial. Adding "local" (i.e. payoff level specific) mechanisms would allow active direction of exploration. This would help the system to allocate resources to exploration efficiently by targeting those regions of the payoff environment insufficiently explored.

5.8 Evaluating XCS

5.8.1 Boolean Multiplexer Problems

Wilson (Wilson 1995, 1996a) used boolean multiplexer problems to evaluate the ability of XCS to learn single step problems, using the basic reinforcement learning scenario described in section 4.2.

Due to the representational capacity of the ternary alphabet as used in XCS classifier conditions (see section 5.3), there are 8 maximally general classifiers conditions for the 2 payoff level 6 multiplexer. Each classifier advocates 1 of 2 actions so, since XCS attempts to construct a complete $XxA \Rightarrow P$ mapping, the 2 level 6 multiplexer has 16 maximally general classifiers. Thus the optimal population size (the smallest set of classifiers which completely covers the input space without any overlaps) is 16 classifiers.

Many payoff landscapes can be used with the 6 multiplexer, the simplest of which has 2 levels (one payoff level r_1 for correct answers and a different level r_2 for incorrect answers). The most complex payoff landscape is one in which there is a different payoff level P for each condition action XxA mapping. In this case no generalizations are possible as each permutation has its own payoff level.

It should be clear that the complexity of the payoff landscape affects the size of [O], and that increasing the complexity of the payoff landscape is another means of varying the difficulty of the multiplexer problem series. An interesting area of investigation would be the comparative difficulty of various payoff landscapes for the multiplexer problems, for both XCS and traditional strength based classifier systems. First, XCS systems should be much better able to learn payoff landscapes with more than 1 niche (see section 5.1) than traditional CS, setting a standard for other systems to meet. Second, the increasing complexity of the increasingly layered landscapes provides another series of benchmark problems for reinforcement learning systems. These benchmarks should complement the multiplexer series as their difficulty increases more rapidly in terms of completing [O] than in achieving correct performance (initial investigation supports this). Because the size of [O] should influence system error more than it does performance, this series may prove to be a more suitable means of evaluating system error. Although the difficulty of minimising system error increases in the multiplexer series, it is currently only practical to work with the first few multiplexers which limits the number of difficulty levels available for testing. An advantage of this new series of tests is that the run time resources required increase much more slowly than in the multiplexer series. In addition there is the issue of confounding effects between the increased performance difficulty and increased system error difficulty in the multiplexer series. Such confounding effects would be smaller in the niche increase series as performance difficulty does not rise as quickly. Densely layered payoff landscapes are discussed in more detail in Appendix D.

[19] The system could form hypotheses based on exploiting apparent patterns in the environment, although this would not work in patternless environments. The whole area of higher-level cognitive functions contributing top-down information is outside the scope of this discussion. However, both approaches are in practice open to exploitation.

5.8.2 Measures of Performance

Wilson collected three statistics when running XCS on the multiplexer problems, updating their values on each exploit cycle (see section 5.5.8). These three statistics are presented in graphs using the number of cycles as the x axis (see figure 3 in (Wilson 1995) and figure 2 on page 24 for examples). Moving averages are used in some cases where the statistic has a high variance as averaging tends to smooth out the curve and make it more comprehensible.

- **Performance** For each trial, the CS's response is either correct or incorrect. The performance statistic used on graphs is a moving average of the last 50 trials. The curve is scaled so that it reaches 1.0 when the last 50 trials have all been correct.
- **System Error** The sum of the predictions of the classifiers in [A] is referred to as the *system prediction*. The difference between the reward actually received from the environment and the system prediction is called the *system error*. A moving average of the system error for the last 50 trials is graphed. Perfect system error is 0.0, and suggests that the system will be able to exactly predict the reward it will receive.
- **Population Size** This is M, the population size in macroclassifiers. This statistic is not averaged as it tends to change by small amounts. For display purposes, the value is divided by 1000.

Because the multiplexer learning experiments were partly random (i.e. in the random generation of input strings and the operation of the GA), Wilson presented averages of these statistics over ten runs.

6 POP-XCS

POP-XCS is, to the best of my knowledge, the first replication of Wilson's XCS system.[20] It was written in Pop-11 by the author over the period May - August 1996 at the University of Birmingham. POP-XCS was implemented as the most up to date version of Wilson's XCS possible at the time. It can be configured to use the subsumption deletion and niche GA set switch from (Wilson 1996a) or to run as in (Wilson 1995). In addition, thanks to correspondence with Stewart Wilson, it includes several modifications not mentioned in either paper:

- The method of calculating system predictions was modified slightly to penalise classifiers with low experience (see 5.5.3).
- The accuracy formula differs slightly from that published, although the effect does not appear significant (see 5.5.4).
- The order in which parameters are updated has been revised and made more suitable for more difficult problems (see 5.5.4).

These modifications were used in all experiments reported in this work unless otherwise noted. In (Wilson 1995) two deletion methods are mentioned in section 3.3. Only the first has been used with POP-XCS.

The level of description of the XCS system given in section 5 is comparable to that given in (Wilson 1995), and should be sufficient for implementation. A brief overview of the implementation of POP-XCS is given in Appendix E, but more detailed description of the actual implementation would be too involved to present here.

6.1 Replication of XCS Experiments

6.1.1 Wilson (1995)

Replication work began with the 6 and 11 multiplexer experiments from (Wilson 1995). Figure 2 shows performance, system error and population size for the 6 multiplexer averaged over 10 runs of 10,000 cycles each. In this case the POP-XCS system was configured to run as the version of XCS described in (Wilson

[20] There is some indication that others have been working on XCS systems, but I have been unable to confirm this.

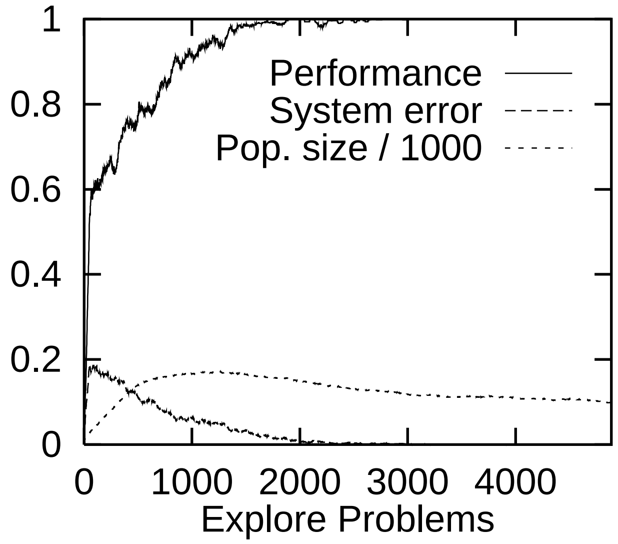

Figure 2: A 6 multiplexer experiment replicating that of (Wilson 1995). System settings are as in (Wilson 1995) figure 3, (i.e. as in section 14, except $N = 400$, the GA operated in the match set, subsumption deletion was not used and the payoff landscape had 16 levels). Curves are the average of 10 runs.

1995), and used the same parameter settings reported by Wilson for this experiment. As reported in (Wilson 1995), performance reaches essentially 100% within approximately 2,000 explore trials. System error reached zero at around the same point, indicating that an accurate payoff map had been constructed. Furthermore, it does appear that XCS tends to evolve a maximally general classifier for each payoff level (this is not shown on the graph), although other classifiers remain in the payoff level with it.

Figure 3 shows is an 11 multiplexer version of figure 2. This experiment again replicates Wilson's findings, with performance reaching 100% and system error reaching zero around 6,000 explore trials.

6.1.2 Evaluating the GA move and Subsumption Deletion (Wilson 1996a)

The next tests to be replicated were those evaluating the two changes to XCS introduced in (Wilson 1996a). Figure 4 shows population size for runs of the 6 multiplexer with POP-XCS in four different configurations produced by running the two GA conditions (in [M] or in [A]) with and without subsumption deletion. (Note that Wilson did not report curves for GA in [M] with subsumption.) Results show that population size is clearly smaller when subsumption deletion is used. The choice of GA niche appears to affect population size only early on, i.e. during the population peak, although this relative lack of effect may be due to symmetries in the multiplexer problem. (I.e. less symmetric problem spaces may show more effect of the GA niche move (see Wilson 1996a).)

Performance for the four configurations was roughly similar, but with a slight advantage for the GA in [M] without subsumption compared to the GA in [A] without subsumption. Performance appeared not to differ significantly according to GA niche used when subsumption was used (again possibly because of symmetries in the multiplexer problem - see (Wilson 1996a)).

6.1.3 Benchmark Experiments (Wilson 1996a)

Figures 5 and 6 show replications of the final 6 and 11 multiplexer experiments in (Wilson 1996a) respectively. System parameters were the same as in previous experiments, but the niche GA move and subsumption deletion were incorporated. Population sizes were $N = 400$ and 800 respectively. Additionally, the payoff landscape was simplified to the 2 level version commonly used in multiplexer problems, that is, the same positive reward is given for all correct answers and zero reward is given for all incorrect answers. Results again agreed with the original experiments, with 100% performance being reached at approximately 2,000

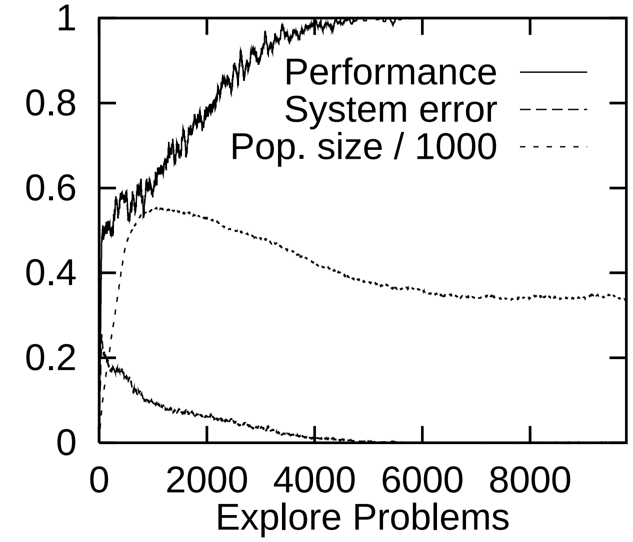

Figure 3: An 11 multiplexer experiment replicating that of (Wilson 1995). Settings are as in figure 2 except $N = 800$. Curves are the average of 10 runs.

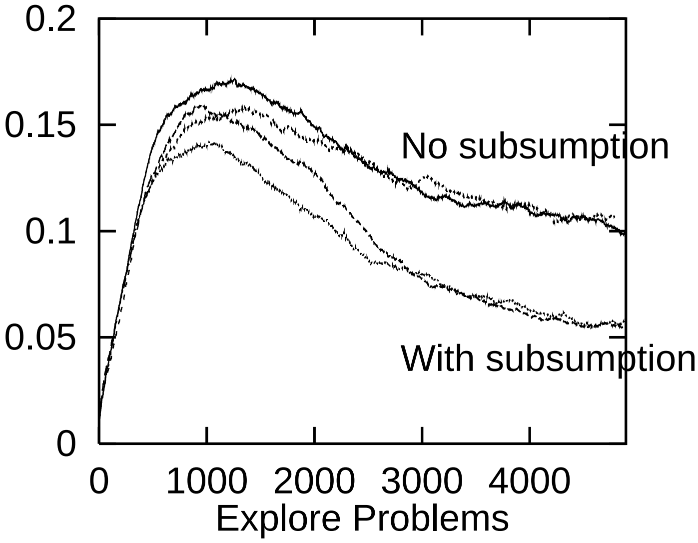

Figure 4: 6 multiplexer experiment showing population size for various system configurations. Reading from top to bottom at 2,000 cycles the curves are: i) GA in [M] w/out subsumption, ii) GA in [A] w/out subsumption, iii) GA in [M] w. subsumption, iv) GA in [A] w. subsumption. System settings are otherwise as in figure 2. Curves are the average of 10 runs.

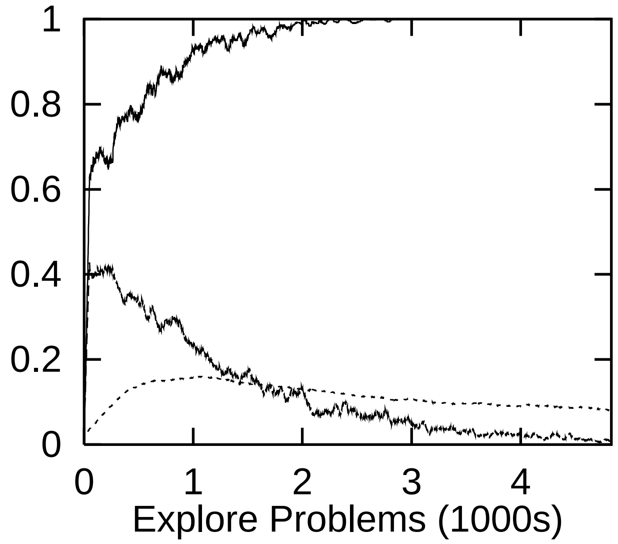

Figure 5: A 6 multiplexer experiment replicating that of (Wilson 1996a). Settings are as in section 14 except $N = 400$, payoff landscape has 2 levels, GA in [A] and subsumption deletion is on. Curves are the average of 10 runs.

and 10,000 trials respectively.[21]

6.2 Overview of Original Work

This section outlines the original work done with POP-XCS once replication work was finished. This overview is intended to clarify the course of the work and to emphasise the important points in the extended discussion which follows it.

The majority of the original experimental work was dedicated to exploring means of reliably evolving optimal populations. This involved:

- **Testing the Optimality Hypothesis.** This involved monitoring the evolution of [O] using a problem-dependent technique to see if a complete [O] really was generated (see section 6.4).
  - Investigating problem-independent means of measuring evolution of [O] (see 6.4.1).
- **Investigating condensation of the classifier population** (see 6.5).
  - Investigating dynamic condensation.
- **Investigating auto-termination of learning using $\overline{F}$** (see 6.6).
  - Investigating a better means of auto-termination by scanning for overlaps in the $XxA \Rightarrow P$ mapping.

In addition the following topics are addressed in the appendices:

- **Evaluation of the effects of the use of macroclassifiers.** See Appendix A.
- **Discussion of suitability of traditional CS, XCS and Q-Learning for learning payoff environments with more than one niche.** See Appendix D.

[21] Lack of time prevented formal replication of the 20 multiplexer. Initial work with 37 multiplexers indicates they are very much harder than the 20 multiplexer. System parameters, and perhaps the system architecture, will have to be carefully adjusted before the 37 multiplexer can be learnt.

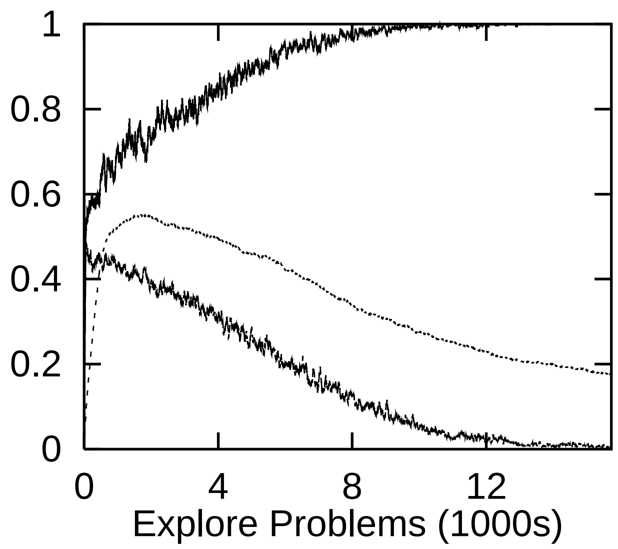

Figure 6: An 11 multiplexer experiment replicating that of (Wilson 1996a). Settings are as in section 14 except $N = 800$, payoff landscape has 2 levels, GA in [A] and subsumption deletion is on. Curves are the average of 10 runs.

6.3 Reviewing Measures of Performance

Performance and System Error as Measures of $XxA \Rightarrow P$ Mapping

In (Wilson 1995, 1996a), performance on the multiplexer problems was measured in terms of the number of cycles required for 100% correct performance. Wilson reports that system error reaches 0 at approximately the same point as 100% performance. However, work with POP-XCS suggests that two important qualifications need to be made to this statement. First, detailed analysis of system error suggests that it never quite converges on zero (unless the classifier population is optimal or very close to optimal). In other words, there is always some error due to additional classifiers in the population (those not in [O]), some of which will at any point have been newly created by the GA and will not have had time to learn accurate predictions.[22] Second, and perhaps more importantly, performance and system error do not reach asymptotic values concurrently for all problems. This is perhaps not surprising as it would seem easier to learn the payoff landscape well enough to choose actions correctly than to predict payoff levels correctly. The first only requires that the classifier with the correct answer for an input be assigned a higher prediction than the classifier with the incorrect answer. However, achieving an accurate payoff map requires that the predictions be on-target.

Figure 5 demonstrates a problem where the point at which system error becomes asymptotic differs from that for performance. This would seem to indicate that system error and performance are different measures of the system's progress in mapping the environment.

Table 1 shows a near-optimal population during an advanced stage of condensation. It has a full [O], but has an additional classifier (the first shown) which would eventually be eliminated by condensation (note its low fitness and numerosity). This classifier alone accounts for the non-zero system error as the others all have predictions exactly matching the actual rewards they receive. (Notice that performance would already be perfect for this population as the one overgeneral classifier is unable to change action selection when using the normal deterministic method.)

[22] Error due to the additional classifiers is very low, typically in the area of 0.01, and thus is not visible on figures using the standard scale (i.e. those shown here and in (Wilson 1995, 1996a)).

Condition	Action	Prediction	Error	Fitness	Numerosity
0# 1# ##	1	657.8	0.501	0.0	1
01 #0 ##	0	1000.0	0.0	1.0	19
01 #0 ##	1	0.0	0.0	1.0	27
11 ## #0	0	1000.0	0.0	1.0	22
00 1# ##	0	0.0	0.0	1.0	29
10 ## 1#	1	1000.0	0.0	1.0	24
10 ## 0#	1	0.0	0.0	1.0	30
11 ## #0	1	0.0	0.0	1.0	20
11 ## #1	1	1000.0	0.0	1.0	24
11 ## #1	0	0.0	0.0	1.0	28
00 0# ##	0	1000.0	0.0	1.0	27
00 0# ##	1	0.0	0.0	1.0	26
10 ## 1#	0	0.0	0.0	1.0	25
01 #1 ##	1	1000.0	0.0	1.0	25
00 1# ##	1	1000.0	0.0	1.0	21
10 ## 0#	0	1000.0	0.0	1.0	24
01 #1 ##	0	0.0	0.0	1.0	28

Table 1: A near-optimal population of classifiers from a 2 level 6 multiplexer experiment at an advanced point in condensation. Only the first classifier shown is not a member of [O], and it alone accounts for the non-zero system error.

New Measures of Performance: M′ and $\overline{F}$

In order to test the optimality hypothesis and work with the new dynamic condensation technique of section 6.5, two additional statistics were added to those Wilson monitored:

- **M′** The count of maximally general classifiers in [P], scaled so that the curve reaches 1.0 when all are present.
- $\overline{\mathbf{F}}$ The mean population fitness.

6.4 Testing the Optimality Hypothesis

The Optimality Hypothesis was tested by running a variety of experiments and monitoring the evolution of maximally general classifiers. Figure 7 demonstrates that a complete [O] can be evolved within [P]: the M′ curve shows the proportion of [O] which has been evolved, and reached 1.0 by 8,000 cycles on each run. The curves shown are averages of 10 runs and provide empirical support for the Optimality Hypothesis, although they do not prove its correctness. Further support for the Optimality Hypothesis is given by experiments shown in figures 8 and 9 in which 6 and 11 multiplexers invariably evolved complete [O]'s within [P] (non-[O] members of [P] were then removed using dynamic condensation yielding optimal populations - see section 6.5). Informally, experience with all multiplexer problems tested suggested that if sufficient resources were allowed (i.e. enough training cycles and enough classifiers), all would eventually evolve [O] within [P] as per the hypothesis.

6.4.1 Predicting the Evolution of the Set of Maximally General Classifiers [O] within the General Population [P]

It appears that progress towards evolving a complete [O] cannot be observed directly without a priori problem-depended knowledge (i.e. the optimal solution to the problem needs to be known in advance - this is how the M′ curve in figure 7 was generated). A number of system parameters were monitored in an attempt to find one that would reflect the evolution of [O] and that did not require problem-specific

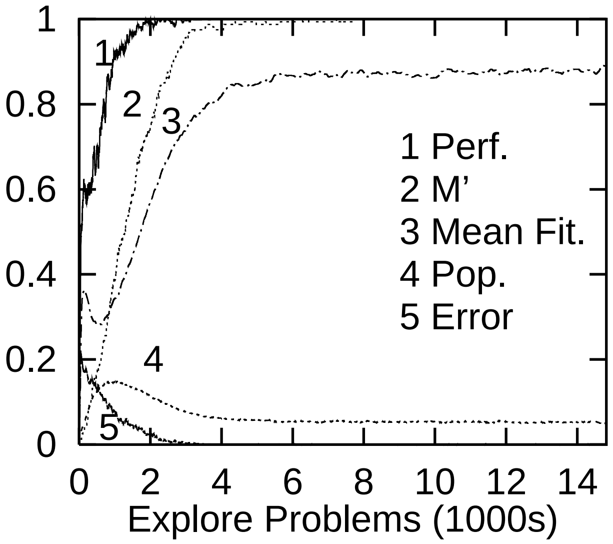

Figure 7: A 6 multiplexer experiment showing the evolution of maximally general classifiers M′ in support of the Optimality Hypothesis. Settings are as in figure 2, except GA is in [A] and subsumption deletion is on. Curves are the average of 10 runs. The M′ curve reaches 1.0 by 8,000 explore cycles for each run indicating that a complete [O] has been evolved within [P] as per the Optimality Hypothesis.

knowledge, but none was found. Reasonable heuristic estimates of the completion of [O] could be based on (see figure 7 for reference):[23]

System Error begins at approximately the random level of 0.5 but drops very rapidly. A very low but distinct plateau is reached when [O] is completed, but also sometimes when [O] lacks a few members (this is demonstrated nicely in the case of figure 7).

Population Size may be a useful indicator of progress in the evolutionary search. If the system has been implemented with macroclassifiers, the population size (in macroclassifier terms) will shrink as the system discovers useful rules. However, it does not appear to be of any more use than system error in predicting the evolution of [O].

Mean Population Fitness $\overline{F}$ is characterised by a fairly steep rise with a plateau which usually correlates with the completion of [O], but as with system error does not always.

These statistics all suffer from an inability to distinguish [P] with complete [O] from similar ones with near-complete [O] that evolve in the generations preceding the completion of [O]. A useful avenue of investigation might be monitoring GA search progress and resource allocation on a payoff level by payoff level basis to meet some statistical confidence limits for the evolution of a maximally general classifier.

As threshold levels for the plateaus seen in these statistics may vary from problem to problem, it would be better to make use of a rate of change statistic rather than using the value directly. For example, (Wilson 1996b) suggests using the rate of system error change to control allocation of explore/exploit cycles.[24]

The system performance curve is not considered as suitable as those listed above as it does not appear to be sensitive to the evolution of [O] and apparently invariably reaches 1.0 before [O] is complete. Nonetheless, it was found that in the specific multiplexer problems tested the simple heuristic of starting condensation by waiting for performance to reach optimum, then delaying for a fixed period, was sufficient to consistently evolve optimal populations. (In 6.5 the same technique is used, but with the system error statistic.)

[23]Graphs in which multiple runs have been averaged are generally not appropriate for demonstrating the predictive capacity of the other curves in terms of M′ as averaging tends to obscure subtle changes. Thus the figures shown here may be somewhat misleading in this respect.

[24]This form of statistic has the added advantage of being suitable for stochastic environments with fixed probabilities, as the rate of change will be influenced only by the rate of change will be influenced only by the learning process (see Wilson 1996b).

6.5 Dynamic Population Condensation

Once a population has evolved a complete set of maximally general classifiers it would be advantageous to remove all other classifiers from the system, leaving it with an optimal population. Such a population classifies its inputs using a minimal number of concepts and is therefore simpler, for both human and self-monitoring analysis, and more efficient.

In support of the Optimality Hypothesis, early experiments indicated that when given enough cycles on the 6 multiplexer experiment, XCS was able to reliably evolve populations which included all 16 maximally general classifiers. However, the classifier population invariably contained additional unnecessary classifiers.

The technique of subsumption deletion reduces the number of unnecessary classifiers, but invariably many still remain.[25] (Wilson 1995) reports informally that it is possible to reduce the population size of an already-trained system using a process called *condensation*. This consists of simply running the GA with mutation and crossover turned off. As a result, no new bitstrings are generated, but weaker (less fit) classifiers are gradually eliminated as a results of classifier selection/deletion dynamics in the GA. Informal experiments showed that an optimal population could be achieved using condensation, but apparently only if [O] had evolved within [P] before condensation started.

When condensation was initially experimented with, it was simply started once the learning process was completed (i.e. at a point chosen by the experimenter). I began experimenting with a technique which I will refer to as *dynamic condensation* in an attempt to explore more problem-independent means of evolving optimal populations. With dynamic condensation, the system uses some form of self-monitoring to trigger the condensation phase when a criterion is met.[26] During the condensation phase the explore/exploit nature of each cycle continues to be selected randomly, so the performance of the system (on the exploit cycles) continues to be monitored.

In the 6 multiplexer experiment shown in figure 8, condensation was triggered by the occurrence of 2,500 consecutive GA cycles during which system error was below 0.01. This experiment was successful in that all 100 out of 100 runs generated optimal populations, but there are several flaws with this approach. The number of condensation cycles allocated to the problem, the delay between error minimisation and condensation, and the error criterion level were chosen by the experimenter and would be inappropriate for other multiplexer problems, and most other problems generally. Further, although the system was able to evolve optimal populations each time, it was likely spending far more cycles doing so than it really required as the system error curve is not always a good indicator of when to start condensation (see 6.4.1).

Nonetheless, the results in figure 8 demonstrate that it is possible to reliably evolve optimal populations for multiplexer problems, at least under the conditions used here.

The dynamic condensation experiment was then run using 100 trials of an 11 multiplexer with the same condensation settings. In this case only 83% of the runs evolved optimal populations, although those that failed were quite close. It was hypothesised that this was due to violation of the criterion of sampling sufficiency, and that if a greater delay in terms of GA cycles was allowed that more populations would reach optimum. To test this, the delay in starting condensation was increased from 2,500 to 7,500 consecutive GA cycles and another 100 trials were run. In this case, 98% of populations evolved optimal populations. A final increase in the delay to 10,000 cycles yielded 100% optimal populations as shown in figure 9.

Analysis of Experiments to Evolve Optimal Populations

A requirement of 2,500 consecutive GA cycles with a system error below 0.01 appears to be about right for the 6 multiplexer as tested. If condensation occurred at a shorter delay the system occasionally did not have enough time to evolve a complete set of maximally general classifiers. If no delay was used, this was the most common result. If the system was left for less than 20,000 explore cycles in all, it occasionally did not have enough time to condense all populations into the optimal population.

If condensation occurred as soon as system error dropped below 0.01, the system failed to find the optimal population. Consideration of the evolution of [O] indicates that there is a lag between the point where system

[25] For the 16 level 6 multiplexer, in the long run the population size typically fluctuates around 50 when subsumption deletion is used, and about 100 without it. See figure 4.

[26] As implemented, if the monitored parameter subsequently falls out of the criterion range, condensation stops and normal GA operation resumes. Thus the system should in principle be able to respond to changes in the environment or unfortunate deletions from its population by resuming normal operation and creating new classifiers. However, in practice this does not occur as condensation is timed to start after the full [O] evolves and appears to eliminate only non-[O] classifiers.

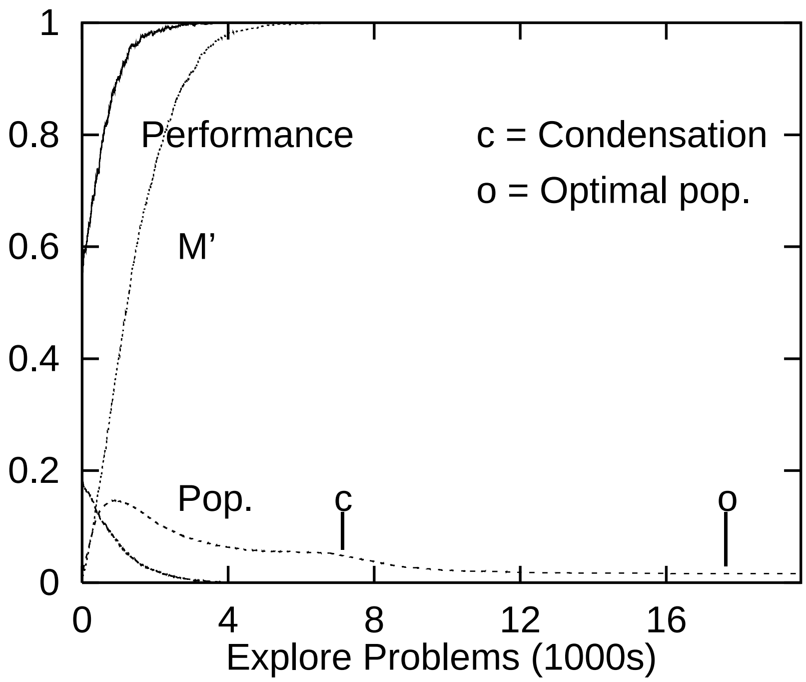

Figure 8: A 6 multiplexer experiment showing condensation (c) commencing at approximately 7,000 cycles and optimal populations (o) being achieved for all runs by approximately 17,500 cycles. M' is the number of maximally general classifiers, other curves are as described in preceding figures. Settings are as in section 14 except $N = 400$, payoff landscape has 16 levels, GA in [A] and subsumption deletion is on. Condensation was triggered by 2,500 consecutive GA cycles of system error below 0.01. Curves are the average of 100 runs.

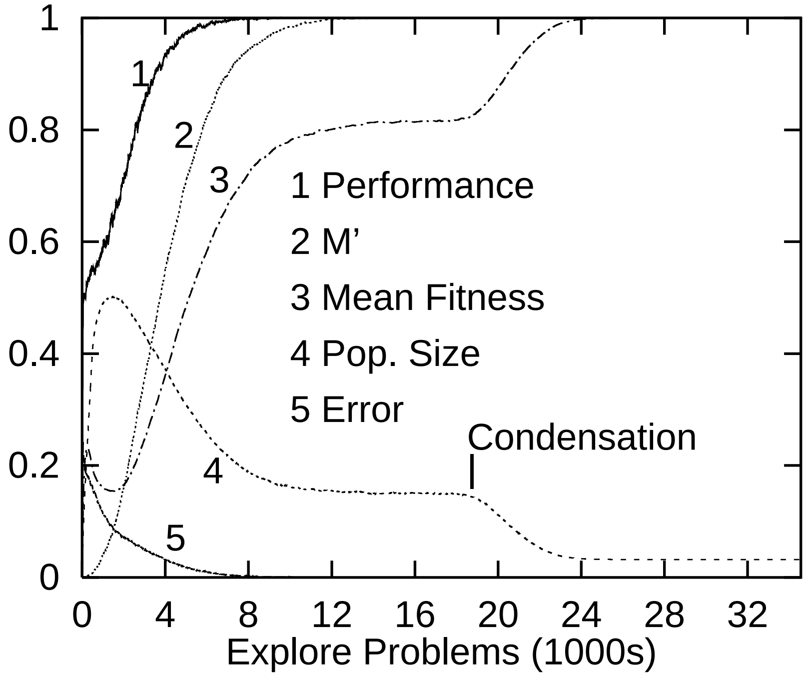

Figure 9: An 11 multiplexer experiment showing condensation commencing at approximately 19,000 cycles. Optimal populations were achieved for all runs by approximately 34,000 cycles. (This final convergence is not visible on the graph due to the scale.) Curves and settings are as in figure 8 except $N = 800$. Condensation was triggered by 10,000 consecutive GA cycles of system error below 0.01. Curves are the average of 100 runs.

error drops below 0.01 and the point where [O] is complete. If condensation begins during this period, it will prevent the system from generating the missing optimally general classifiers. Furthermore, condensation will not typically result in enough increase in system error for the GA to restart and generate the missing optimally general classifiers. It appears that the best course is to let the system evolve using a full population for some period in order to complete [O] before attempting to condense the population.

The number of cycles required to evolve a full [O] appears to vary considerably from run to run. As a results, although the constant values used in the experiments in figures 8 and 9 appear to work reliably for these cases, the delays used are considerably more than required on most runs.

Ideally the system would monitor the number of maximally general classifier found and commence condensation once the full set existed, but this presupposes a knowledge of the optimal population on the part of the system. A problem-independent solution must make use of the other parameters to estimate the sufficiency of evolutionary development and commence condensation. Other statistics not yet evaluated, such as mean fitness weighted generality, may be more reliable predictors of the completion of [O].

6.6 Auto-termination of Learning

By monitoring the action sets the system can determine when the population contains a single classifier for each input category (payoff level). This is detectable because at this point the action set will only ever contain a single classifier. This indicates the presence of an optimal population, and thus the point at which further learning (as long as the environment does not change) is fruitless. Using such a technique, the system can be left to run for as many cycles as necessary, rather than some preset limit. The system ceases running trials when it has achieved an optimal population and can then report this. This could be very useful for problems for which the solution is not known to the experimenter, for example, an XCS system could perhaps learn the optimal categorisation of a feature space derived from a training set of digitised handwritten character samples. Giving XCS the ability to handle stochastic environments would significantly increase the usefulness of such systems for practical applications.

Monitoring $\overline{F}$

One statistic which could be used to monitor the convergence of [P] on [O] is $\overline{F}$, the mean population fitness. $\overline{F}$ converges towards 1.0 as the number of macroclassifiers in [A] drops to 1. However, because the delta rule used to update the fitness parameter minimises the difference between the current estimate and the next sample, $F_j \leftarrow F_j + \beta(\kappa'_j - F_j)$, an $\overline{F}$ of 1.0 is never quite achieved. Rather than attempt to determine when $\overline{F}$ is close enough, a different approach involving the analysis of the $XxA \Rightarrow P$ mappings to detect overlapping classifier conditions was adopted. It has the added advantage of scanning the entire population at once, rather than relying on the environment to provide adequate samples for each payoff level during dynamic condensation.

Scanning [P] for Overlapping $XxA \Rightarrow P$ Mappings

By comparing the conditions of each pair of classifiers which can be formed from [P], the system can detect overlapping mappings. Two classifiers overlap if their conditions match the same input string and they have the same action. An overlap indicates that there is some input for which an [A] of size > 1 will be formed and thus that an optimal population cannot have been achieved. The lack of overlaps does not guarantee that an optimal population has been achieved, except (as proposed under the Optimality Hypothesis) under conditions which satisfy the criterion of sampling sufficiency. In other words, if insufficient trials have been run and condensation has begun too soon, a non-overlapping yet non-optimal population may form. No problem-independent means of detecting this state has been discovered. If this state was detectable, rule discovery might be restarted in an attempt to rectify the problem (see section 6.8).

6.7 Detecting Incomplete $XxA \Rightarrow P$ Mappings

There are some environments where some input categories are rarely if ever generated. As an example, an animat world might include a state lottery which rewarded the animat once on average for every 100,000 tickets it bought. At this rate, if the animat only bought about 100 tickets per simulation, it would rarely encounter the input string indicating it had won the lottery. Its map of the input space would therefore be incomplete. However, this deficiency is detectable if the system enters a phase where it systematically generates fantasy input strings to ensure that all input possibilities are covered by a classifier, or builds some form of input region contiguity map. For abstract single step problems this may well be pointless, but it might be useful for animats actively attempting to explore their environment (see section 5.7.2).

6.8 Restarting Rule Discovery

In a changing environment the system might find that its optimal population was suddenly no longer optimal. In this case it could restart the GA search (e.g. in response to changes in some measure of performance), evolve classifiers until it estimated [O] was again complete and then restart condensation.

When the GA needs to be restarted to find missing maximally general classifiers it may be possible to restrict the search to those parts of the environment not already optimally covered (those whose input strings produce action sets with size other than 1). This would greatly restrict the search area and should thus speed up the process considerably.

An interesting question is whether having an optimal population may be an advantage to the system if the environment changes and it must start generating new classifiers again. One scenario where this might occur is in an animat problem where a new element is introduced after the system has been allowed to adapt to the environment. For example, a new type of food or predator might be introduced, changing the $XxA \Rightarrow P$ mapping - this sort of change confronts real creatures continuously. Even at a population level environmental niches move continuously as species co-evolve. If an animat has an optimal classifier population there may be some respects in which it finds it easier to discover new rules to deal with a changed environment. It may also find it easier to monitor and analyse its own reasoning. However, this has yet to be established.

The more complete mapping of $XxA \Rightarrow P$ generated by XCS would surely give it an advantage over traditional classifier systems in situations where the payoff level structure does not change, but relative values do. For example, a particular type of food might suddenly become poisonous to the animat and change from giving a reward of 1000 to a reward of -1000. In other words the P values in $XxA \Rightarrow P$ would suddenly change. A traditional CS would have to discard the classifiers dealing with the now-poisonous food and discover new ones to maximise environmental payoff. In contrast, XCS would retain its more complete conceptual mapping of the input space (i.e. its classifier population) and adjust the predictions of the classifiers involved.

7 Conclusion

(Wilson 1995) introduced XCS and demonstrated its superiority in several respects to the traditional CS (see section 5). The most important respect for the current work is that XCS tends to evolve classifiers which are maximally general within the limits of an accuracy criterion, in contrast to traditional CS which lack adaptive pressure towards accurate generalization. The present work continued Wilson's by demonstrating the ability of XCS (with the additional mechanisms of dynamic condensation and auto-termination) to reliably evolve optimal populations for the 6 and 11 multiplexer problems, and by presenting analysis of these results.

Contributions of the present work include:

- Proposal of the XCS Optimality Hypothesis.
- Investigation of means of estimating the evolution of [O].
- Investigation of techniques to dynamically control condensation and terminate testing with the aim of reliably evolving optimal solutions in as few cycles as possible.
- The suggestion that performance and system error are indicators of different degrees of learning in the system (section 6.3).
- Replication of multiplexer results published in (Wilson 1995, 1996a).

- The implementation of a configurable general-purpose XCS system in Pop-11 which will be made available at the University of Birmingham.
- Investigation of the effect of the use of macroclassifiers. (See Appendix A)
- Investigation and analysis of payoff landscapes with more than one niche. (See Appendix D for more on this.)

In section 6.4.1 it was proposed that the point at which a complete [O] had evolved within [P] could only be estimated in the problem-independent case. This hypothesis is subject to refutation and, given the utility of determining this point in evolving optimal populations, may be the subject of future research. Several related means of estimating the evolution of [O] were suggested (6.4.1).

Building a Better Animat

Self monitoring is a form of feedback with many applications in adaptive systems. XCS uses self monitoring of the accuracy of its concepts (i.e. its classifiers) as a means of evaluating their utility, which gives it fundamental advantages over the traditional CS (see section 5). As a classifier system XCS improves on existing designs and advances classifier systems technology. The broader theoretical implications of the development of XCS are twofold:

- CS are now more suitable as animat control systems (e.g. they can now support more sophisticated planning systems thanks to the more complete world model).
- It has been demonstrated that a reinforcement learning system can evaluate concepts on the basis of their accuracy and that this is a useful way of building models of the world.

Accuracy information is useful in high level cognitive processes (e.g. planning) and yet (in XCS at least) can be derived from local computations on scalar values. This suggests that it may be useful as form of currency in an economy of mind (see Minsky 1987; Wright 1995, 1996).

Examples of uses of self-monitoring in classifier systems given in this work illustrate principles which are hopefully broadly applicable, not only to machine learning systems but to natural ones as well. Neural networks (both artificial and natural) may be thought of as classes of statistical engines and may well be suited to calculating the more abstract sort of statistic discussed herein, such as the rate of change of system error.

Much work remains to be done investigating the design space of adaptive self-monitoring mechanisms in classifiers systems and adaptive systems more generally.

8 Acknowledgements

Many thanks to Ian Wright, Aaron Sloman and Stewart Wilson for their encouragement, comments and suggestions during this work.

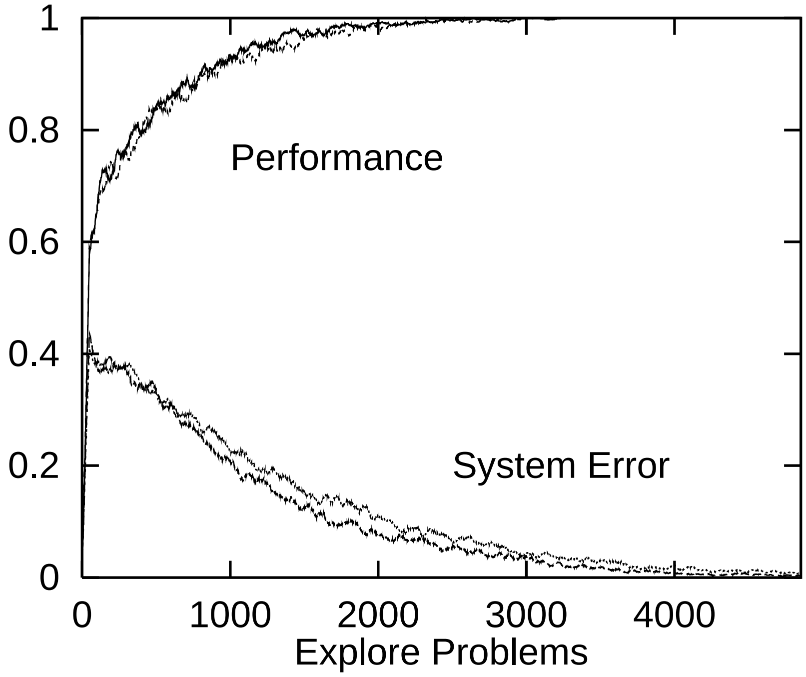

Figure 10: Comparison of XCS with and without macroclassifiers on a 2 level 6 multiplexer problem. Curves are the average of 30 runs. System error is slightly lower when macroclassifiers are used on 2 level problems, but unchanged when used on more densely layered problems. Performance is very similar for the two cases.

9 Appendix A - Evaluation of Macroclassifiers

In order to evaluate the effect of the use of macroclassifiers on a typical problem, two series of tests were run using the 6 multiplexer problem. In the first series of tests, macroclassifiers were used as normal, but in the second they were disabled by omitting the test for matching classifiers upon insertion of the new classifier into [P]. Thus in the second case all classifiers had a numerosity of 1, and classifiers with identical conditions and actions were permitted to coexist. In neither case was subsumption deletion used, as this only makes sense when macroclassifiers are available.

Figure 10 shows the results of these runs. System parameters were set as in section 14, except $N = 400$. A 2 level payoff landscape was used. Although performance is highly similar, there is a difference in the system error for the two cases: the system with macroclassifiers generally had a slightly lower system error. This experiment was repeated using payoff landscapes with 16 and 32 levels and in neither case was there any observed difference between the use of macroclassifiers and their lack.

When macroclassifiers are used, new duplicate classifiers are dropped and instead the numerosity of an existing macroclassifier is increased. As a result, an existing better-evaluated classifier becomes stronger, and the new one, which would have some form of initial value for prediction and prediction error, is not introduced into the population. This may account for the observed difference in system error. In the more heavily layered landscapes the actual reward may tend to be closer to the new-classifier-influenced system prediction because the reward levels have less space between them. This could account for the lack of effect outside the 2 level landscape. In any case reducing the level of system error is a beneficial if slight side-effect of the use of macroclassifiers.

The use of macroclassifiers has important implications for subsumption deletion and condensation. Recall that subsumption deletion involves incrementing the numerosity of a macroclassifier rather than inserting a new duplicate microclassifier, so is not possible to implement without macroclassifiers.

With macroclassifiers, condensation can actually reduce the population to the optimal size. Without macroclassifiers, the population would tend to fill up with duplicate microclassifiers which would have to be eliminated using some other technique. Trials run with condensation but without macroclassifiers indicate that this does actually happen. In addition these trials suggest that condensation is not as effective at eliminating superfluous classifiers (it seems to take longer) when macroclassifiers are absent. However, it is difficult to compare systems in this respect due to the lack of subsumption deletion in the microclassifier-only version.

In summary, experience with POP-XCS strongly supports the use of macroclassifiers; all comparisons indicate that, if they have any effect, they improve the system. They appear to perform essentially as equivalent microclassifier populations, allow the use of subsumption deletion and the elegant form of condensation reported here, and increase run time speed.

10 Appendix B - Notation

Miscellaneous Notation

p A classifier's prediction parameter.

ε A classifier's prediction error parameter.

F A classifier's fitness parameter.

P The payoff (reward) derived from the environment. The calculation differs for single and multi step problems (see 5.5.4).

M The population size in macroclassifiers.

$[M]$ The match set.

$[A]$ The action set.

$[A]_{-1}$ The action set from the previous cycle.

$[P]$ The general classifier population (the classifier list).

$[O]$ The set of classifiers forming an optimal population (this is the set of classifiers which are maximally general for their payoff level). [O] is sometimes referred to as a subset of some [P].

$\overline{F}$ Mean population fitness. This can in principle be used to estimate convergence on the optimal population during condensation.

M' The count of maximally general classifiers in [P].

$XxA \Rightarrow P$ The mapping of inputs X to actions A to predictions P constructed by a reinforcement learning system.

$\leftarrow$ The update or reassignment operator.

System Parameters

N Population size limit in microclassifiers (i.e. the sum of the numerosities of all classifiers on the classifier list cannot exceed this value).

β Learning rate used with the Delta rule.

γ Discount factor applied to reward from the previous time step in calculating P (payoff) for multi step problems.

θ Do a GA in this [A] if the average number of time steps since the last GA is greater than θ.

ε_o Accuracy criterion. Classifiers with error $\varepsilon_j > \varepsilon_o$ have sharply lower fitness.

α Accuracy falloff rate. This controls the slope of the falloff in the accuracy calculation function.

χ Probability of crossover per invocation of the GA.

μ Probability of mutation per allele in an offspring classifier in the GA.

ϕ If the total prediction of [M] is less than ϕ times the mean prediction of [P], covering occurs.

$P_{\#}$ Probability of a # at each allele position in the condition of a classifier created through covering or in the initial population.

p_I, ε_I, F_I The prediction, prediction error and fitness assigned to each classifier in the initial population.[27]

[27] No initial population was used in any of the experiments reported here or in (Wilson 1995, 1996a).

11 Appendix C - Potential Sources of Confusion

The reader should be aware of several particular potential sources of confusion in understanding the material presented in this document:

- One measure of the system's performance is the fraction of the last 50 inputs to which it has responded correctly. This is simply referred to as the system's performance, despite the fact that other measures, such as system error, are also measures of the system's performance.
- The reader should bear in mind that macroclassifiers are invariably used with XCS (except as noted in the macroclassifier evaluation in Appendix A), but that system parameters are always treated in terms of microclassifiers (i.e. the equivalent number of microclassifiers obtained by summing the numerosities of the macroclassifiers in question). However, population size curves *do* reflect the number of macroclassifiers. (The number of microclassifiers quickly reaches the allowed limit and remains there, so it is not very interesting to graph.)
- Traditional classifier systems use a parameter called *strength* as a measure of the classifier's value for both action selection and rule discovery. In XCS, strength is replaced by *prediction*, which takes the place of strength in action selection, and *accuracy*, which takes the place of strength in rule discovery. Strength and prediction are both measures of predicted reward (payoff) to be received if the classifier is used (although the calculation differs between traditional CS and XCS).

12 Appendix D - Comparison of Ability to Form $XxA \Rightarrow P$ Mappings

Traditional classifier system are able to form "implicitly complete" mappings of 2 level payoff environments: the payoff maximising GA tends to locate classifiers describing the payoff level of "correct" answers (i.e. those with the higher of the two payoff levels), and alternative actions for these classifier are considered implicitly incorrect (and are not represented within the system). Due to the representational capacity of the ternary alphabet used in classifier conditions (see 5.3), a minimum of 8 classifier conditions are required to completely map the 2 level 6 multiplexer. (As noted earlier (in 5.3), a traditional CS can use multiple conditions in a single classifier and could in principle represent the 6 multiplexer with a single classifier. However, there is no evolutionary pressure towards this unlikely occurrence.)

In contrast, XCS explicitly maps each payoff level because classifier fitness is based on the accuracy of the prediction and not the quantity predicted. Thus, XCS requires a minimum of 16 classifiers to completely map the 2 level 6 multiplexer.

Q-Learning, as it does not use any means of generalization, needs 128 estimates of P, one for each input/action pair.

In terms of the completeness of the world model, all three systems are thus essentially equivalent for this problem (although they differ in terms of the size of their minimal representation). However, when one increases the number of payoff levels involved in the problem, the traditional CS, while still representationally able to map the payoff environment, does not in practice learn to do so. This is because the payoff maximising GA will prefer classifiers which map payoff levels with higher reward at the expense of those with lower reward. (In the extreme case this would lead to a mapping of only the highest payoff level.) Because all unrepresented regions of $XxA \Rightarrow P$ are implicitly grouped together as the "wrong answer", the system cannot be said to form an accurate and complete map of the environment. Further, "correct" classifiers (i.e. those with higher payoff predictions than their counterparts in the niche) with low payoff predictions will tend to be overlooked by the GA and not be maintained in the population. This leads to an inability to select payoff maximising actions in low payoff niches as the relevant classifiers have not been maintained (even if they are evolved).

However, XCS can deal with more heavily layered problems; it simply maps each payoff level as in the 2 level problems. A minimum of 32 classifiers are required for XCS to form a complete map of a 32 level 6 multiplexer.[28] Q-Learning is also able to form a complete map of this problem, and, as the size of $XxA \Rightarrow P$ has not changed, does not need any additional estimates of P. The advantage of generalization in XCS over Q-Learning becomes apparent when one moves to the 11 multiplexer problem where Q-Learning requires a table of 4,096 entries (regardless of the number of payoff levels) while XCS requires only 32 classifiers for the 2 level case.[29]

The number of classifiers required by XCS to describe the payoff environment is thus dependent on the number of levels it contains. The fewer levels, the more XCS is able to make generalizations about the landscape and the fewer classifiers it needs to describe it. For the 6 multiplexer, the minimum number of classifiers required to describe the environment ranges between 16 and 128 depending on the amount of regularity present. A minimum of 16 classifiers is required due to limitations on the representational capacity of the ternary alphabet used in classifier conditions. This is more a limitation of the representational scheme currently used in XCS than of XCS itself; alternative forms of representation (e.g. lisp S-expressions) could be used instead.

To summarise, traditional CS, XCS and Q-Learning are all able to form complete and accurate maps $XxA \Rightarrow P$ of payoff landscapes with only 1 niche (i.e. 2 levels). Traditional CS are not suitable for more complex payoff landscapes[30] but XCS and Q-Learning are. In terms of the number of concepts required to map a payoff landscape, in the worst case XCS is equivalent to Q-Learning. However, XCS *can* take advantage of many (but not all, because of the limitations of the ternary alphabet) potential generalizations in the

[28] We assume for the sake of simplicity in this discussion that the more heavily layered payoff environments are constructed by simply subdividing the regions defined by each maximally general classifier into equal halves as many times as desired. Otherwise, the limitations on the descriptive power of single conditions discussed in 5.3 may complicate matters.

[29] It should be noted that various means of introducing generalizations into Q-Learning based systems have been investigated (see the survey of Kaelbling, Littman & Moore 1995).

[30] Although modified versions, as mentioned in 5.1, may be.

payoff landscape and reduce the number of concepts (classifiers) it employs. According to the Optimality Hypothesis of section 5.7, XCS *will* take advantage of these generalizations.

13 Appendix E - Implementation of POP-XCS

POP-XCS is conceptually implemented in three layers, each of which consists of multiple source code files.

Layer 1 - Startup Files

The only way to run POP-XCS is by compiling a startup file. Many startup files are currently used, and more can be created as needed. These files define various macros, constants and variables which control compilation and parameters of the rest of the system. (Conditional compilation is used extensively to optimise run time performance.) To configure the system for a particular type of test, an existing startup file should be modified or a new one created. There is a standard format for startup files which is exemplified by start.pop.xcs.example.p (A complete set of configuration options needs to be included in the startup file or the system will refuse to compile.)

Layer 2 - Problem Files

There are two forms of problem which can be used with POP-XCS: single-step and multi-step. Each requires the inclusion (by the startup file) of a different problem file which defines code to run the problem environment the classifier works in. (Note: at this point the multi-step configuration has not been completed. Results have only been reported with the single step system.) Because only one type of single step problem (the multiplexer problems) and one type of multi step problem (the “woods” environments (see (Wilson 1995)) have been implemented for POP-XCS, the problem layer is split across two files. (One for multiplexers and one for woods problems. Ideally these files would be slightly generalised into generic single and multi step files, and the truly problem-specific code put into a new sublayer.)

Layer 3 - POP-XCS Core

This layer implements XCS itself. The core is not problem-independent at runtime, but provides alternative compilation and run-time configurations which are controlled by the other two layers (e.g. the system can be compiled to run in either normal or debug mode depending on the settings in the startup file). The core resembles a library of classifier system functions. Most functions are written so as to pass an XCS object, or its components (e.g. lists), to each other and could in principle interleave the execution of multiple XCS systems. (The point here is that the dependence on global variables has been minimised as most of the information needed to run a particular system is encapsulated by an XCS object. However, this encapsulation could be taken further. It was limited by experimentation with various additional mechanisms which have not yet been encapsulated by the XCS object type.)

Running POP-XCS

POP-XCS writes statistics out to various files when it has completed the preset number of trials. These datafiles are in ASCII format and also contain information about system settings for the experiment. They are in gnuplot-readable format. (gnuplot is a widely available interactive plotting program.)

POP-XCS will be made available for use in the school of computer science at the University of Birmingham.

For an example of system execution, compile this startup file:

~tyk/project/start.pop.xcs.example.p

the results files (which show the same curves as figure 9) can be viewed with the following command:

gnuplot plot.example

This example startup file is in the standard format of all POP-XCS startup files. It runs a typical 6 multiplexer experiment and writes the results to various files (all beginning with the prefix ‘example’). If a datafile already exists when the system tries to write it out, it appends as many ‘-’ to the end of the filename as necessary to produce an unused name.

Additional information is also written to the standard output during the run, including system settings and a milestone for every 100 cycles completed.

14 Appendix F - Standard System Settings

Unless otherwise noted, the following system parameter settings were used in all experiments reported in this work. These settings are largely the same as those in (Wilson 1995) Figure 3: $\beta = 0.2$, $\gamma = 0.71$, $\theta = 25$, $\varepsilon_o = 0.01$, $\alpha = 0.1$, $\chi = 0.8$, $\mu = 0.04$, $\phi = 0.5$, $P_{\#} = 0.33$. In all experiments reported in this work, the initial population size was 0 and initial classifiers were generated as needed by covering. (Wilson 1995) discussed two deletion techniques, but only the first has been used with POP-XCS. The second employs a parameter, δ, which is not used in POP-XCS. Subsumption deletion was used unless otherwise noted.

Additional standard system parameters were as follows (please refer to system documentation for discussion):

- PREDICTION COVERING LEVEL = 0.5
- ADVOCACY THRESHOLD = 20
- GA THRESHOLD = 25
- SUBSUMPTION THRESHOLD = 20

The conservative update order was used, and classifier initialisation was optimised for rapid learning.

15 Bibliography

Anderson, C. W., (1986). *Learning and Problem Solving with Multilayer Connectionist Systems.* Ph.D. Dissertation, Computer and Information Science, Univ. of Massachusetts.

Booker, L. B., (1989). Triggered rule discovery in classifier systems, in: *Proc. Third International Conference on Genetic Algorithms*, J. D. Schaffer, (ed.), Morgan Kaufmann, San Mateo, CA.

Cliff, D., & Ross, S., (1995). Adding Temporary Memory to ZCS. *Adaptive Behavior*, 3(2), 101-150. Also University of Sussex School of Cognitive and Computing Sciences Technical Report CSRP 347, February 1995.

Dorigo, M., & Bersini, H., (1994). A comparison of Q-learning and classifier systems. In D. Cliff, P. Husbands, J.-A. Meyer, and S. W. Wilson (eds.), *From Animals to Animats 3: Proceedings of the Third Conference on Simulation of Adaptive Behavior* (pp. 248-255). Cambridge, MA: MIT Press/Bradford Books.

Goldberg, D. E., (1989). *Genetic algorithms in search optimisation and machine learning.* Reading, MA. Addison-Wesley.

Hayes, W., L., (1988). *Statistics.* 4^{th} Edition. Forth Worth: Harcourt Brace Jovanovich.

Holland, J. J., (1975). *Adaptation in natural and artificial systems.* Ann Arbor, University of Michigan Press.

Holland, J. H., (1976). Adaptation. In R. Rosen & F. M. Snell (Eds.), *Progress in theoretical biology*, 4. New York: Plenum.

Holland, J. H., (1986). Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), *Machine learning, an artificial intelligence approach. Volume II.* Los Altos, California: Morgan Kaufmann.

Kaelbling, L., Littman, M., & Moore, A., (1995). Reinforcement Learning: A Survey. *Practice and Future of Autonomous Agents Vol 1.* Centro Stefano Franscini, Monte Veita, Ticino, Switzerland.

Minsky, M., (1987). *The Society of Mind.* Picador.

Riolo, R. L., (1988). CFS-C: A package of domain independent subroutines for implementing classifier systems in arbitrary, user-defined environments. Logic of Computers Group, Division of Computer Science and Engineering. University of Michigan.

Riolo, R. L., (1989). The emergence of coupled sequences of classifiers. In J. D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, (pp. 256-264). San Mateo, CA: Morgan Kaufmann.

Skurikhin, A. N., & Surkan, A. J. *Messy Genetic Algorithm Learns Classifier for Multiplexer Design.* Email: alexei@tarzan.cr.cyco.com, surkan@cse.unl.edu

Sloman, A., & Humphreys, G. (1992). The Attention and Affect Project. Appendix to JCI proposal.

Sloman, A. (1993). The Mind as a Control System, in *Philosophy and the Cognitive Sciences*, (eds) C. Hookway and D. Peterson, Cambridge University Press, pp 69-110, 1993.

Sloman, A. (1994). Explorations in Design Space, in *Proceedings ECAI*, August 1994.

Sloman, A. (1995). Exploring design space and niche space. Invited talk for 5th Scandinavian Conference on AI, Trondheim, May 1995. In *Proceedings 5th Scandinavian Conference on AI*, IOS Press, Amsterdam.

Sloman, A. (1996). What sort of control system is able to have a personality? To appear in *Proceedings Workshop on Designing personalities for synthetic actors.* Vienna, June 1995. Robert Trappl (Ed).

Venturini, G., (1994). *Apprentissage Adaptatif et Apprentissage Supervisé par Algorithme Génétique.* Thèse de Docteur en Science (Informatique), Université de Paris-Sud.

Watkins, C. (1989). *Learning from Delayed Rewards.* Ph.D. Dissertation, Cambridge University.

Watkins, C., & Dayan, P., (1992). Technical note: Q-Learning. *Machine Learning*, 8, 279-292.

Wilson, S., (1987). Classifier systems and the animat problem. *Machine Learning* 2, 199-228.

Wilson, S., (1990). Perceptron Redux: Emergence of Structure. *Physica D*, 42(1-3), 249-256. Republished in *Emergent Computation*, S. Forrest (ed.), MIT Press/Bradford Books.

Wilson, S., (1994). ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1), 1-18.

Wilson, S., (1995). Classifier Fitness Based on Accuracy. *Evolutionary Computation.* Vol. 3, No. 2. 1995

Wilson, S., (1996a). Generalization in XCS. Submitted to *ICML '96 Workshop on Evolutionary Computing and Machine Learning.*

Wilson, S., (1996b). Explore/Exploit Strategies in Autonomy. To be published in *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior.* P. Maes, M. Mataric, J. Pollack, J.-A. Meyer, and S. Wilson (eds.), Cambridge, MA: The MIT Press/Bradford Books.

Wilson, S., (1996c). *Personal Communication.*

Wright, I., (1995). Draft of: *Cognition and Currency Flow. Notes Toward a Circulation of Value Theory of Emotions.* August 1995.

Wright, I. P., (1996). Reinforcement Learning and Animat Emotions. To appear in *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior.* P. Maes, M. Mataric, J. Pollack, J.-A. Meyer, and S. Wilson (eds.), Cambridge, MA: The MIT Press/Bradford Books.

A selection of Stewart Wilson's papers is also available at: http://netq.rowland.org/sw/swhp.html

Useful questions and answers on XCS are available at: http://netq.rowland.ord/wilson/xcs/q.html