CFS-C: A Package of Domain Independent Subroutines for Implementing Classifier Systems in Arbitrary, User-Defined Environments.†

Rick L. Riolo

January 1986
Updated November 1988

Logic of Computers Group
Division of Computer Science and Engineering
University of Michigan

†This work was supported in part by National Science Foundation Grants DCR 83-05830 and IRI 86-10225, and by a University of Michigan Presidential Initiaive Fund Grant to the BACH research group.

Contents

1 Introduction **1**

2 Classifier Systems **3**
2.1 The Basic Classifier System 3
2.2 Learning Classifier Systems 7
2.2.1 Competition to Post Messages 7
2.2.2 The Bucket-Brigade Algorithm 8
2.2.3 Discovery Algorithms 9

3 The CFS-C Classifier System **11**
3.1 Messages and the Message List 12
3.2 Classifiers and the Classifier List 13
3.2.1 The Condition-Part 13
3.2.2 The Action-Part 14
3.2.3 Bidratios and Specificities 16
3.2.4 Classifier Strength 17
3.3 Matches 17
3.4 Support 18
3.5 Bidding 19
3.6 The Competition to Generate New Messages 20
3.7 The CFS-C Output Interface 22
3.8 The Flow of Strength in CFS-C 23
3.8.1 The CFS-C Bucket Brigade Algorithm 24
3.8.2 Taxes 28
3.8.3 Classifiers Created by Discovery Algorithms 29
3.9 The CFS-C Discovery Learning Mechanisms 29
3.9.1 The Genetic Algorithm 32
3.9.2 Cover Detector Messages. 35
3.9.3 The Cover Effectors Operator 36
3.9.4 Triggered Chaining Operators 37
3.9.5 The Low-Bidders Operator 39

4 How to Define an Environment for the CFS-C System **41**
4.1 Defining the Environment State-Space 42
4.2 Defining the Detectors 43
4.3 Defining Effectors 44
4.4 DoBkgBeh(): Do Background Behavior 46
4.5 Defining the Payoff Function 46
4.6 Displaying and Saving the Environment 46
4.7 Loading the Environment 48
4.8 DoEnvCmd(): Do Environment Specific Commands 52

5 Future Research **53**

A Bibliography 57

B The CFS-C System Commands 61

C CFS-C System Constants and Variables 85

C.1 System Constants . . . 85
C.2 The Basic CFS-C System Variables . . . 85
C.3 I/O Control Variables . . . 94
C.4 Variables That Control the Discovery Algorithms . . . 97

D How to Change the CFS-C System 103

D.1 How to Install a New User-Command . . . 103
D.2 How to Define a New System Variable . . . 104
D.3 How to Change the Basic String Length, the Classifier List Size, and Other Basic Characteristics of the CFS-C Classifier System . . . 105
D.4 How to Add or Remove Variables Associated with Message or Classifier Nodes 106
D.5 How to Define a New Classifier Action-Type . . . 107

E Technical Notes 111

E.1 CFS-C Data Structures . . . 111
E.1.1 The Message Lists . . . 111
E.1.2 The Classifier List . . . 112
E.1.3 Matches . . . 112
E.1.4 Effectors . . . 113
E.2 The CFS-C Major-Cycle . . . 113
E.3 Representation of Message, Condition, and Action Strings. . . . 115
E.3.1 Storage . . . 115

1 Introduction

This document describes the CFS-C system, a package of subroutines (and data structures) that can be used to implement learning classifier systems for arbitrary, user-defined task-domains/environments. The CFS-C subroutines implement the core, domain-independent parts of a classifier system, including routines to implement the following steps of the "major-cycle" of a classifier system:

- Compare messages to classifiers and record all matches.
- Calculate bids, run a competition, generate new messages by activating the winning classifiers.
- Apply the "bucket-brigade" algorithm, a learning mechanism that redistributes strength (i.e., allocates credit) to classifiers that tend to lead to useful system behavior.

The CFS-C package also includes subroutines that implement learning algorithms that "discover" (generate) new, potentially useful classifiers. In particular, the CFS-C system includes subroutines to implement "genetic algorithms" like those described by [Holland, 1976] and other "triggered" algorithms like those described in [Holland, 1986c].

The CFS-C system provides a simple, well-defined interface between the core classifier system and the domain-dependent parts of the system, i.e., the classifier system's "detectors," "effectors," and "payoff function." This makes it very easy to implement the CFS-C classifier system in a variety of differnt task-domains, without having to modify the domain-independent parts of the classifier system defined by the CFS-C subroutines.

The CFS-C package also includes subroutines that implement a command-processor that acts as an interface between a user and the classifier system. The processor provides ways to:

- Run the major-cycle of the classifier for n steps before reprompting for a command (where n is specified by the user).
- Load messages and classifiers from files into the system's message and classifier lists.
- Display the contents of the message and classifier lists, either in response to a command or automatically every i major-cycle steps (where the user specifies i).
- Display and set the values of various system parameters, e.g., the message list size, the bid proportionality constant, tax rates, effector and detector rates, and so on.
- Produce a log of a session, including commands entered by the user and responses and displays produced by the system, in a file or on a printer.
- Save the complete state of the system in a file, either in response to a command or automatically, every i major-cycle steps.
- Restart the system from a file that contains a "saved" state of the system.

The command-processor is table-driven, so that it is easy to add new commands, system variables, and so on. "Macro's" are used extensively in the source code so that it is easy to make changes to fundemental characteristics of the system (e.g., the length of message strings, the maximum size of the message list, and so on).

The CFS-C package is written in the C language, primarily to make the system portable to a wide variety of machines and operating systems. The CFS-C system has been used to implement classifier systems in 80x86-type machines (using Microsoft, Lattice and CI-86 compilers), on on the Apple MacIntosh (using MPW C, as a "tool" in the MPW shell environment, not as yet as a stand-alone "application" with a standard Mac-type interface), Apollo computers (using the Apollo Domain C compiler), on SUN and Vax computers, and on IBM mainframes using a version of Bell Labs C compiler for those machines. A second reason for choosing a high-level language like C was to make it easy to modify all parts of the CFS-C classifier system, including the subroutines that calculate bids, pick winning classifiers, generate new messages, implement the bucket-brigade and rule discovery learning algorithms, and so on.

Note that the CFS-C system is *not* a stand-alone program that a user can run. It is a package of subroutines that must be linked to the subroutines and data structures that implement a specific task-domain/environment and the environment-dependent parts of a classifier system (i.e., an input interface, output interface, and payoff function). Section 4, "How to Define an Environment for the CFS-C System," describes in detail how to implement the environment-dependent subroutines that must be linked to the CFS-C subroutines to form a runable program.

For a description of a "runable" CFS-C classifier system, see *LETSEQ1/CFS-C: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences* (Riolo, 1986, revised 1988). or *FSW1/CFS-C: An Implementation of the CFS-C Classifier System in a Task–Domain that Involves Learning to Traverse a Finite State World* (Riolo, 1988). Note that these systems employ only simple line-by-line (non-graphical or window based) interfaces, to make them as portable as possible.

The rest of this document describes the CFS-C classifier system and how to use it. Section 2 gives a brief introduction to classifier systems in general. Section 3 then describes the characteristics of the CFS-C classifier system, e.g., how classifier bids are calculated, how winning classifiers are chosen, and so on. Section 4 describes how to use the CFS-C package to implement classifier systems in an arbitrary task-domain/environment. Section 5 briefly describes some research issues I think should be addressed in the near future. Appendices provide detailed descriptions of various features of CFS-C system. In particular, Appendix B contains descriptions of all CFS-C commands, Appendix C describes the CFS-C system constants and variables that affect the behavior of the classifier system, and Appendix D includes some notes about how to modify basic characteristics of CFS-C system, e.g., the length of message strings, the size of the message list, and so on. Finally, Appendix E describes the files that contain the source code for the CFS-C package.

NB: I have tried to keep this document up-to-date as I change the system, but if there is some question, be sure to see the code or contact me, as these descriptions may be out of date.

2 Classifier Systems

Classifier systems are special types of message-processing, rule-based (production) systems first introduced by Holland and Reitman (1978). Many rules can be active at the same time in a classifier system, so information is processed in a highly parallel way. The rules and messages in classifier systems are very simple, so that it is very easy to determine whether or not a rule's condition-part is satisfied. Classifier system rules are also designed to be modifiable by general-purpose learning algorithms.

Several different classifier systems have been implemented (see Appendix A for references). Each system was designed to tackle a different problem, and each implements a somewhat different version of a classifier system. (See Forrest, 1985, Goldberg, 1983, Booker, 1982, and Wilson (1988) for four quite different classifier systems used for very different tasks.) This section gives a brief description of the features of the "standard" classifier system as defined by Holland (1986). First the basic components and operation of all non-learning classifier systems will be defined. Then the features proposed by Holland (1986) to implement learning in classifier systems will be introduced.

2.1 The Basic Classifier System

As defined by Holland (1986), a (non-learning) classifier system consists of four basic parts:

1. A *classifier list*, which consists of a set of condition-action rules called *classifiers*;
2. A *message list*, which acts as a "blackboard" for communications and short term memory;
3. An *input interface*, which provides the classifier system with information about its environment; and
4. An *output interface*, which provides a way for the classifier system to communicate with or change its environment.

At any given time, the *message list* in a classifier system contains zero or more *messages*. Each message has the same simple structure, i.e., its is a string of fixed length L built from the binary alphabet 0, 1 . Note that this constraint on the structure of messages does not restrict the computational power of classsfier systems, since more complex messages can be mapped into the binary messages by selecting a sufficiently large L. On the other hand, using a simple structure for messages (and in turn, for conditions) makes it very easy to compare messages to conditions to determine if a classifier (i.e., rule) should be activated. Also note that the order of messages on the list is not important: the message list may be thought of as an unstructured "blackboard" which can be read from and written to by the other parts of the classifier system.

The *classifier list* consists of one or more *classifiers* (condition-action rules) of the form:

$$cond_1, cond_2, ..., cond_n/action$$

The $cond_i$'s constitute the condition-part of the classifier, and *action* is the action-part. For $n = 1$, a classifier is called a *single condition* classifier; for $n > 1$, a classifier is called a *multiple condition* classifier. Note that the order of the classifiers in the list is not important: thus the list may be thought of as a set of classifiers.

Each $cond_i$, called a *condition string* or just a *condition*, consists of a string of fixed length L (i.e., the same length as message strings) built from the tertiary alphabet $\{0, 1, \#\}$, optionally prefixed by a "∼" (a "not-sign" character). A condition string $cond_i$ that has no "∼" prefix is *matched* by any message on the message list that has 0's and 1's in exactly the same positions as the 0's and 1's in the condition string. That is, a logical comparison is made between the condition string and a message string, but the # in the condition string is considered a "wildcard" symbol that can match a 0 or a 1. For example, for $L = 4$, the following table shows the result of comparing some messages to a condition string:

condition string	message	Match?
10##	1010	Yes
10##	1001	Yes
10##	1110	No

A condition string prefixed by a "∼" (not-sign) character is matched just when *no* message on the message list matches the condition string itself. A condition string prefixed by a "∼" is sometimes called a *not-match* condition.

Note that each condition string partitions the space of possible messages into two subsets, one containing messages that match the condition and the other consisting of messages that don't match. Condition strings with more wildcard's (i.e., more # symbols, sometimes called "don't care" symbols) match larger subsets of messages than do condition strings with fewer # symbols. In particular, a condition string of all #'s matches *all* messages, while each condition string that contains no #'s matches exactly one message.

A classifier's condition-part is *satisfied* when *each* of its conditions $cond_i$, $i = 1 \ldots n$, is matched by some message on the message list (or, for $cond_i$'s that are not-match conditions, when no message matches them). For example, assume the message list contains the following messages:

0011
1010
1111

The following table shows some classifier condition-parts (each with two condition strings), and for each condition-part, whether or not it is satisfied by the three messages shown above:

condition-part	Satisfied?
00##, 0#11	Yes
1##1, #111	Yes
0##0, #111	No
0##1, ∼ 0000	Yes
0##1, ∼ 1111	No

Note that a classifier's condition-part is satisfied even when the same message matches two (or more) of its condition strings. For example, both condition strings of the first condition-part in the above table are matched by the message "0011" and by no others.

The condition-part of a classifier also specifies a subset of all possible messages, namely the intersection of the subsets specified by the "n" $cond_i$ parts of that classifier. The number of "don't care" symbols in a classifier's condition-part is a measure of the classifier's *specificity*: the more #'s, the less specific the requirements are to satisfy the classifier's condition-part. Classifiers that have many # symbols in their condition-parts may be thought of *generalists* (their condition parts are often satisfied) while classifiers that have few #'s in their condition-parts are *specialists*.

The *action-part* of a classifier also consists of a string of fixed length L (again, the same length as message strings) built from the tertiary alphabet $\{0, 1, \#\}$, When the condition-part of a classifier is satisfied, the classifier becomes *active*, i.e., its action-part produces one or more new messages that will be placed on the message list at the next time-step. In particular, for every message on the current message list that matches the left-most condition $cond_i$ of an active classifier, the action-part generates one new message. The messages that match the left-most condition may be thought of as *input* messages to the classifier, and the new messages may be thought of as *output* messages produced by the action-part. (Note that input and output in this context refer to a particular classifier, not to the classifier system's input and output interfaces.) Each output message produced by a classifier has 0's and 1's in just those loci where the classifier's action string has 0's and 1's. The value of an output message in those loci corresponding to the #'s in the action string is determined by the input message being processed, i.e., the value in the output message is the same as the value in the input message.

For example, consider the following two-condition classifier (with $L = 4$):

$$10\#\#, \#11\#/\#1\#1$$

If the current message list contains the following messages:

1111
1000
1011

the classifier shown above will produce the following new messages:

1101
1111

Note that the # characters in a classifier's action-string may be thought of as a *pass-through* operator, since the values found in the corresponding loci of input messages are "passed-through" to the new, output message.

The *input interface* for a classifier system can be any device or subroutine that generates messages of the proper form, i.e., consisting of binary strings of length L. The input interface provides the classifier system with information about the classifier system's environment. For example, if the classifier system is used to control a robot, the input interface could be a set of *detectors*, i.e., devices that map information about the state of the environment into binary messages of the proper length. For instance, the detectors could generate messages that provide information about the location and properties of objects in the robot's immediate vicinity.

The *output interface* for a classifier system can be any device or subroutine that takes messages (i.e., binary strings of length L) as input and produces some kind of output or other effect in the classifier system's environment. For example, if the classifier system is used to control a robot, the output interface can be thought of as message-activated *effectors* that the classifier system can use to control the robot's behavior (turn it left or right, control its speed, and so on.)

A classifier system is run by repeatedly executing the following steps:

1. Add messages generated by the input interface to the current message list.

2. Compare all messages on the message list to the condition-parts of all classifiers and record all matches for classifiers that have their condition-parts satisfied.

3. Generate new messages by activating each classifier that has its condition-part satisfied, i.e., by applying the action-part to each message that matches the left-most condition of the classifier.

4. Process the new messages through the output interface.

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

6. Go to Step 1.

These steps are called the *major-cycle* of the classifier system. Executing the major-cycle once is the basic unit of time in a classifier system.

Note that a classifier system can be thought of as a particular kind of parallel computer. Classifiers correspond to instructions that take two arguments, a message string (i.e., a message that matches the left-most condition-string of the classifier) and an action-part string. A classifier system program is just a particular classifier list. The program is run by repeatedly executing the major-cycle. Input to the computer is provided through the input interface and output is produced by the output interface.

Also note that the classifier list is the only permanent memory the classifier system has. Since the contents of the message list are replaced at the end of every major-cycle, it acts mainly as a short-term memory which indicates the current state of the system. (However, classifiers can be used to refresh specific messages indefinitely, providing a kind of volatile long-term memory.

As with other computers, higher level structures and sequences of coordinated actions are built from the classifier system's primitives (i.e., classifiers and messages) using various addressing schemes. In classifier systems, the condition-parts of classifiers act as addresses of classifiers. For example, consider the following single condition classifier (for $L = 8$):

$$0000\#\#\#\#/0001\#\#\#\#$$

Any messages that begins the string "0000" is in effect "sent to" to the classifier shown above. Those messages can pass arbitrary information to this classifier, since messages that match it can have any sequence of 1's and 0's in the their rightmost 4 loci. The messages that begin with "0000" also tranfer control to this classifier, since they satisfy its condition-part and thus activate it. (Of course in a classifier system, control is shared by all the classifiers active during a given major-cycle.)

Sequences of actions can be carried out by *coupled* classifiers, i.e., classifier's that produce messages addressed to other specific classifiers in the sequence. For example, consider the following classifiers:

0000####/0001####
0001####/0010####
0010####/0011####
0010####/1011####

If the message list contains the message 00001111 at step t, the classifiers shown above will be executed in a specific sequence, namely, the first in the list at t, the next in the list at t+1, and the last two both active at t+2.

By using appropriate tags and addressing schemes, classifier systems can be programmed to carry out arbitrarily complex operations. Coupled classifiers cam also be used to represent information in high-level data structures like linked-lists, networks, and default hierarchies. For a further discussion of tags, coupling, and the representation of complex data structures and operations in classifier systems, see Holland (1986), Forrest (1985) and Robertson and Riolo (1988).

2.2 Learning Classifier Systems

The basic classifier system defined in the previous section has no way, in itself, to modify its behavior as a result of its experience–it has no way to learn. In that simple classifier system, all classifiers that have their condition-parts satisfied by messages on the current message list are activated, i.e., they generate new messages that are placed on the message list for the next major-cycle. Thus given the same classifier list and history of messages generated by its input interface, it will produce the same behavior and output.

There are two ways to introduce learning into a classifier system:

1. Given a fixed list of classifiers, change the major-cycle so that the activation of each classifier depends on some additional parameter that can be modified as a result of experience.

2. Actually change the contents of the classifier list, by removing classifiers, adding new classifiers, or changing the condition or action-parts of existing classifiers.

The following sections briefly describe changes in the operation of the basic classifier system that implement the learning mechanisms described above.

2.2.1 Competition to Post Messages

First, in order to selectively activate only some of the classifiers satisfied during a major-cycle, Holland (1986) proposes having classifiers *compete* to become active and post messages. That is, each classifier that has its condition-part satisfied makes a *bid* to become active. A competition is then carried out and only the highest bidders are allowed to become active and post messages.

Next, Holland suggests making a classifier's bid depend on two factors:

1. Its *strength*, a value that is meant to be a measure the classifier's "usefulness" to the system. Thus the strength of a classifier is a parameter that should be modified as a result of the system's experience in a particular task domain.

2. Its *specificity*, i.e., the number of 1's and 0's in its condition part. Specificity can be thought of as a measure of relevance to a particular situation (i.e., set of messages): high specificity classifiers, if satisfied by the messages on the list, are more likely to be relevant to that *specific* set of messages than are low specificity classifiers, which can be matched by many different messages.

In particular, a classifier's bid is just:

$$B = k * strength * specificity$$

where "k" is a proportionality constant that is the same for all classifiers (typically 1/8 or so). The higher a classifier's strength (usefulness), the more it bids and so the more likely it will win the competition and post messages. Similary, the greater the classifier's specificity (relevance), the more it will bid and more likely if will be activated.

Thus given a set of classifiers with fixed condition- and action-parts, the behavior of the classifier system can be changed simply by changing the strengths associated with those classifiers. And in particular, if the strength of classifiers that tend to lead to "useful" system behavior (however usefulness is defined) can be increased, and the strength of classifiers that tend to lead "bad" system behavior can be decreased, the system will learn to produce more useful behavior. The next section describes a learning algorithm designed to bring about just these types of changes in classifier strengths.

2.2.2 The Bucket-Brigade Algorithm

The *bucket-brigade* algorithm is a learning mechanism designed to allocate strength (credit) to classifiers that tend to produce useful system behavior. This algorithm has several nice properties:

1. No high-level, sophisticated critic is required to evaluate classifiers.

2. The algorithm is domain-independent, in that the only feedback the system needs is information about when it has produced useful behavior or achieved desired goals.

3. The strengths of classifiers are adjusted in parallel, in ways that fit well into the paralllel nature of a classifier system's activity.

This section will briefly describe the bucket-brigade algorithm. For a further discussion, see Holland (1986) or Riolo (1987a, 1987b).

The basis for the bucket-brigade algorithm is information from the environment about whether or not the classifier system as a whole is behaving correctly. One simple yet useful way to provide this kind of information is in the form of *rewards*. That is, whenever the classifier system (i.e., its output interface) behaves in ways that bring about states of the environment that are considered desirable, the classifier system receives a positive reward value from the environment. Whenever the system behaves in ways that bring about undesirable states, the system receives a negative reward. Most often, however, the system receives no reward (i.e., a zero reward value), since most states of the environment are in and of themselves neither desired nor undesired. The association of rewards with states of the classifier system's environment is sometimes called a *payoff function.*

For example, if the classifier system is controlling a robot that is supposed to carry out certain tasks, when it successfully completes a task the system receives a reward. Similarly, if the robot is to avoid doing some things, if it does one it receives a negative reward.

Given the reward mechanism described above, the bucket-brigade algorithm acts in two ways:

1. When a reward is received from the environment, the bucket-brigade algorithm adds that reward value to the strength of *all* classifiers active during that major-cycle. That is, the algorithm acts to change the strength of all classifiers that were *directly* associated (in time) with receiving the reward.

2. When a classifier is activated, it pays the amount it bid to the classifiers that made it possible for it to become active. That is, the strength of the activated classifier is decreased by the amount it bid, and that strength is transfered to the classifiers that produced the messages that matched the activated classifier's condition-part. Thus the bucket-brigade also acts to increase the strength of classifiers that *indirectly* lead to useful behavior (i.e., lead to rewards).

The algorithm is called a "bucket-brigade" because strength flows from the ultimate source, the rewards from the environment, through classifiers that are associated directly with those rewards, to classsifiers that made it possible for those directly-rewarded classifiers to be active, and so on.

2.2.3 Discovery Algorithms

While the bucket-brigade learning algorithm can lead to an improvement in a classifier system's behavior, it is limited by the fact that the classifier conditions and actions are never changed. If the original classifier list is missing some key classifiers, the bucket-brigade can never overcome that problem. Therefore classifier systems need an additional type of learning algorithm, one that *creates* or *discovers* new classifiers. (The term "discovery" is appropriate in the sense that the learning algorithm must search the space of all possible classifier lists and "discover" which lists lead to the most successful behavior of the classifier system.)

Holland ([Holland, 1976]) defined a class of rule discovery algorithms, called *genetic algorithms*, that are well suited for use with classifier systems. The basic genetic algorithm works as follows:

1. Pick some classifiers to act as parents, i.e., classifiers that will be copied. Typically the probability a classifier is picked is proportional to its strength, i.e., the system's estimate of the classifier's usefulness.

2. Make copies of the selected parent classifiers and then apply *genetic operators* to those copies to modify them. The most important genetic operator is the *crossover*

operator, which acts to exchange a portion of one classifier with a portion of another. (For an example, see Section 3.10, "The CFS-C Discovery Learning Mechanisms.")

3. Pick classifiers to be replaced, remove them from the classifier list, and replace them with the classifiers produced by step 2. Typically the the lower a classifier's strength, the more likely it will be replaced.

Both analysis (Holland, 1975) and simulations ([Goldberg, 1983], [Booker, 1982], [Wilson, 1986b]) have shown that genetic algorithms, when used with algorithms that apportion credit (strength) to useful members of a population (as the bucket-brigade algorithm does for classifiers), are a very powerful way to search for and discover useful populations.

While a full discussion of genetic algorithms as they apply to learning in classifier systems is beyond the scope of this document, a few points are worth noting:

1. New classifiers produced by the genetic algorithm do not usually replace their parents, since parents tend to be high-strength classifiers and the classifiers that are replaced tend to have low strengths. Since many classifiers can be active at once in a classifier system, this means new classifiers can be tried without losing the expertise accumulated in their parents.

2. New classifiers are tested and evaluated like all other classifiers, i.e., when the classifier wins the bidding competition, posts its messages, and has its strength adjusted by the bucket-brigade algorithm. This means it is usually a good idea to run the genetic algorithm at at slower pace than the bucket-brigade is run (i.e., every major-cycle). This gives the bucket-brigade time to adjust the strength of classifiers before those strengths are used by the genetic algorithm to create and replace classifiers.

3. The recombination of portions of condition-strings from pairs of parent classifiers leads to the creation of new tags (addressses) and new groups of coupled classifiers. Thus the genetic algorithm is in effect constantly creating new high-level structures, using structures already in the system as *building-blocks.*

For a further discussion of these and other issues, see [Holland, 1976], [Holland, 1986c] or [Goldberg, 1988].

Besides using a genetic algorithm to discover new rules, CFS-C also uses some other *triggered* algorithms [Holland, 1986c] to create new rules. For example, a *triggered chainging operator* [Robertson and Riolo, 1988] is available in CFS-C, to make it possible for the system to discover useful chains of coupled classifiers. Other triggered algorithms provide ways for the system to create rules that match novel detector messages, generate new actions, and so on. These are discussed in more detail later in this document.

3 The CFS-C Classifier System

This section describes the characteristics of the classifier system implementated by the CFS-C subroutines. The emphasis in this section will be on giving a functional view of operation of the CFS-C classifier system, not on the way the system is implemented. Note that all parameters that are mentioned are described in detail in Appendix C.

In some respects the CFS-C system is an enhanced version of a classifier system (e.g., CFS-C classifier action-parts can perform arbitrary operations on messages strings). In other respects the CFS-C system is more constrained than the generic classifier system described in the previous section (e.g., all CFS-C classifiers have two conditions). In either case, since the CFS-C system is written in a high-level language (C), it should be reasonably easy to change various aspects of its operation. (Appendix D, "How to Change the CFS-C System" describes how to make changes to much of CFS-C code.) Also, many parameters that affect the behavior of the CFS-C classifier system can be changed at run-time: see Appendix C, "CFS-C System Constants and Variables" for a list of these modifiable parameters.

Note that the CFS-C classifier system is a learning classifier system like of the type described in Section 2.2. Therefore the CFS-C major-cycle is somewhat different from that given in Section 2.1 for non-learning classifier systems. In particular, the CFS-C major-cycle consists of the following steps:

1. Add messages generated by the input interface to the current message list.

2. Compare all messages on the message list to the condition-parts of all classifiers and record all matches.

3. Calculate bids for those classifiers that have their condition-parts satisfied, run a competition between the bidders, and pick the winning classifiers.

4. Generate new messages by activating each classifier that won the competition (i.e., by applying the classifier's action-part to the messages that match it).

5. Process the new messages through the output interface to produce the system's behavior. Get any reward from the payoff function defined for the environment.

6. Apply the bucket brigade algorithm to reallocate strength.

7. Apply the system's discovery algorithms to produce new classifiers while replacing other, low-strength classifiers.

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

9. Go to Step 1.

Also note that the CFS-C system is not, by itself, a *runable* classifier system. In particular, the CFS-C package does *not* provide the *environment-dependent* components of a classifier system, i.e., the input interface, the output interface, and the payoff function. The interactions between these three components and the CFS-C subroutines that call them are briefly described in this section. For a complete description of the environment-dependent parts the CFS-C system, see Section 4, "How to Define An Environment for the CFS-C System."

Finally, a few comments about the particular implementation of CFS-C. As I have said, there is no one standard, definitive description of what a classifier system or its algorithms should be. Thus there are many ways to implement these algorithms, with many "philosophical" and technical decisions to make all along the way. (A few examples: How many conditions in rules, how to produce new messages from multiple matches, what taxes to use (if any), how to calculate bids, how to allocate rewards.) The implementations in CFS-C are in some ways much like a natural system: things have been added, modified and removed over time, all working on top of a moving target (i.e., what experience has indicates to be good or bad ideas *and* what various people think *might be* good ideas), trying to meet moving constraints and goals (e.g., what are good domains, how should the domains be represented in terms of detectors, effectors and rewards). Also, over time the definition of the C language has changed (e.g., from the K. & R. "standards" like 8 characters max for variable names, to the ANSI standards of today), compilers have changed and grown closer to the standards, compiler support tools have become widely available (like editors that recognize and use braces), and C has become popular and well known (so braces aren't scary to non-programmers any more).

All this has led to a CFS-C package I consider to be somewhat of a mess! In some ways it has grown to contain lots of alternative ways to do things, rather than just one "standard" set of algorithms. Perhaps in CFS-C 2 (or 3 or ...) some things will be "known" so the code can be simplified and cleaned up. At the least, I continue to (slowly) work through the code to make it a bit more uniform looking, to use longer variable names, to change to braces, and to use real tabs instead of blanks. At any rate, I hope CFS-C subroutines can be of some use to people just starting using classifier systems, by at least showing one way to do things, and perhaps by serving as a starting point from which you can make your own modifications and discoveries.

NB: I have tried to keep this document up-to-date as I change the system, but if there is some question, be sure to see the code or contact me, as these descriptions may be out of date.

3.1 Messages and the Message List

In the CFS-C system, the length of message strings is set when the CFS-C subroutines are compiled. In particular, the value assigned to a macro, *STRNGSZ*, defines the length of message, condition, and action strings. In the distributed version of CFS-C, STRNGSZ is set to 16. Note that message strings may contain blanks, to make it easier for humans to compare message and condition strings; the CFS-C system ignores the blanks.

The size of the CFS-C message list is also set when the CFS-C subroutines are compiled. The value assigned to the *MSGLSTSZ* macro defines the maximum number of messages that the message list can hold. In the distributed version of CFS-C, MSGLSTSZ is set to 32.

While the message list in CFS-C has a fixed maximum size set at compile-time, not all of that list is available for new messages produced by classifiers. A fraction of the message list is reserved for messages produced by the classifier system's input interface (i.e., its detectors). Thus during any given major-cycle step, classifiers are limited to producing only as many messages as will fit in the part of the list not reserved for detector messages.

The fraction of the list reserved for detector messages is specified by the system variable *NmDMsgMx* ("number of detector messages, maximum"). For example, if NmDMsgMx is set to 20, classifiers will be limited to producing 12 new messages during each major-cycle (assuming a MSGLSTSZ of 32). This value should initially be set by the domain dependent subroutines linked to CFS-C to some value appropriate for the domain. The value may also be changed at any time during a run.

A portion of the message list also can be reserved for different "types" of messages, i.e., effector-activating messages and internal messages. See the parameters *NmIIMMx* and *NmIEMMx* for more information. Other parameters that affect message production include *DelHall* (to delete "hallucination" messages), *OneMPerC* (limit each classifier to producing one message per step), and *MxDupMsg* (limit the number of duplicate messages on the list.

Associated with each message is a real value called the message's *intensity*. The intensity of a message produced by a classifier is just the bid that classifier made when it produced that message. The intensity of a message added to the message list by the classifier system's "detectors" (i.e., the input interface) is a value supplied by the detector that produced the message. In either case, the intensity of a message may be thought of as a measure of the importance of that message, as judged by the producer of the message. In CFS-C, message intensities are one factor that influences the bids made by classifiers–see section 3.4, "Support," and section 3.5, "Bidding" for more details.

Also associated with each message is a pointer to the classifier that *produced* it. (If the message was produced by the input interface, the pointer is null.) This pointer is used by the CFS-C bucket brigade mechanism to determine, when the message is used as "input" by an active classifier, where to send the payment. (See section 3.8 for details.)

3.2 Classifiers and the Classifier List

The maximum size of the CFS-C classifier list is set when the CFS-C subroutines are compiled. The value assigned to the *CFLSTSZ* macro defines the maximum number of classifiers that the list can hold. In the distributed version of CFS-C, CFLSTSZ is set to 200.

The basic form of a CFS-C classifier is as follows:

$$condition - part/action - part, bidratio, strength$$

The following sections describe these four parts of a CFS-C classifier in more detail.

3.2.1 The Condition-Part

All CFS-C classifiers are two-condition classifiers, i.e., each classifier's condition-part has two condition strings. The length of each condition string is defined by the system macro STRNGSZ, which defines the length of message strings. In CFS-C, the right condition may be a not-match condition, but the left condition may *not* be a not-match condition. Not-match conditions are indicated by the "$\sim$" prefix; match conditions typically have no prefix, or they may be prefixed by an "m" (for "match"). CFS-C condition strings may contain blanks to help humans read them–the blanks are ignored by the system.

For example, the following shows two CFS-C condition-parts (assuming the basic string size is 16):

$$11\#0\ 1100\ 0\#\#\#\ 1\#01, \sim 1111\ 0000\ 1111\ 0000$$
$$m\#\#01\ \#\#\#\#\ 1010\ 0110, \sim 00\#\#\ \#\#\#\#\ \#\#\#\#\ \#\#\#\#$$

Note the right condition of the second condition-part is a not-match condition.

3.2.2 The Action-Part

The action-parts of classifiers in the CFS-C system are more complicated than the action-parts of the traditional classifiers described in section 2.1. In particular, the form of classifier action-parts in CFS-C is:

action-type / action-string

where:

action-type specifies how the classifier will act on input (matching) messages to produce new messages. (See below for details.)

action-string is a string of length STRNGSZ (the basic size of messages in CFS-C) consisting of 0's, 1's, or #'s, as in traditional classifier action strings.

If the $action-type$ is PASS, then the classifier's action-type is the traditional pass-through action as described in section 2.1. That is, wherever the action-string has a 0, the new message has a 0, wherever the action string has a 1, the new message has a 1, and whereever the action-string has a # (the pass-through character), the new message has the same value as is found in the corresponding position in the input message that matches the classifier's left condition.

For example, consider this classifier (based on length 8 messages):

1###0###,m10######/PASS/0110####

The following table shows messages that match its left condition along with the new messages produced when the action-part is applied to those messages:

input message	output message
11110101	01100101
10001100	01101100

Note that a classifier's action-type is analogous to a standard computer operation, e.g., add the contents of registers, shift the contents of a register left or right 1 or more bits, and so on. For instance, the operands for the classifier instruction could be defined to be the message string that matches the classifier's left condition and a constant operand, the action-string itself. When the classifier instruction is executed it produces a result string, namely the new message. Thus one way to generalize the basic definition of classifier systems to allow different *action-types*, just as standard computers provide different instructions.

Since a classifier system that uses just the pass-through action-type (along with non-match conditions) is computationally complete, one might ask "Why introduce more action-types?" The main reason is that some simple operations that are very frequently required for many different tasks are inconvenient to implement using just the pass-through operator. For example, simple counting requires a number of pass-through classifiers (i.e., where portions of some messages are interpreted as binary numbers to be incremented or decremented). Similarly, producing a message that is equal to the result of combining two arbitrary messages using the logical operators AND and OR can require many classifiers.

For this reason the CFS-C system was designed to allow the definition of arbitrary action-types for classifiers. The distributed CFS-C system includes six action-types, but it is easy to define others. The following list gives the names of action-types implemented in the CFS-C system and describes how they produce new messages:

PASS Produce a new message using the traditional pass-through action-type.

AND Produce a string that is the logical AND of the messages that match the classifier's first and second conditions, and then perform the traditional pass-through action on that string to produce the new message.

OR Produce a string that is the logical OR of the messages that match the classifier's first and second conditions, and then perform the traditional pass-through action on that string to produce the new message.

XOR Produce a string that is the logical EXCLUSIVE-OR of the input messages that match the classifier's first and second conditions, and then perform the traditional pass-through action on that string to produce the new message.

ADD1 Produce a string by interpreting the four rightmost loci of the message that matches the left condition of the classifier as a binary number and adding 1 to it, and then perform the traditional pass-through action on that string to produce the new message. Overflow just returns "0000" as the value of the rightmost four loci of the string that is used as input to the pass-through action.

SUB1 Produce a string by interpreting the four rightmost loci of the message that match the left condition of the classifier as a binary number and subtracting 1 from it. Then perform the traditional pass-through action on that string to produce the new message. Underflow just returns "0000" as the value of the rightmost four loci of the string that is used as input to the pass-through action.

Note that in CFS-C, each special action-type (AND, OR, etc.) uses a two-step process to generate new messages: (1) use the special action-type to generate an intermediate string, and (2) use that string as input to the traditional pass-through action to produce the final product message of the classifier. This procedure was selected so that *tags* (portions of message strings that have a particular string of 1's and 0's) could easily be maintained even though messages are processed by the special action-types.

For example, suppose the message list contains the following two messages (of length 8):

11000011
00000001

The following table shows the messages produced by classifiers that use the AND and ADD1 action-types:

classifier	string	message
11######,0000####/AND/##11####	11000011	11110011
11######,0000####/ADD1/##11####	11000100	11110100

For instructions on how to define other action-types in the CFS-C system, see Appendix D, "How to Change the CFS-C System." That appendix also contains a brief discussion of some of the issues that should be addressed when defining new action-types for classifiers.

3.2.3 Bidratios and Specificities

The *bidratio* of a classifier is another factor that is included in the calculation of a CFS-C classifier's bid. Other things being equal, the higher a classifier's bidbratio, the more it bids (see section 3.6 for details). A classifier's bidratio, which can be a number between 0 and 1, can be thought of as a measure of a classifier's willingness to risk strength during a single major-cycle. That is, if the bidratio is close to 1, the classifier will make a bid that is a large proportion of its total strength. In making such a large bid, the classifier risks losing a lot, since if it wins the bid competition, it must pay the full bid to post messages, but it may not get any return on those messages (i.e., if no classifier matches those messages during the next major-cycle).

In the CFS-C system, a classifier's bidratio can be any real number; however, in general only bidratio's between 0.0 and 1.0 should be used. Bidratio's can be assigned to classifiers by the system's user (e.g., when classifiers are loaded into the system from some file). Bidratio's also can be assigned when a classifier is created by a discovery learning algorithm.

However, if neither the CFS-C user nor a CFS-C discovery mechanism assigns a special bidratio to a classifier, that classifier's bidratio is just its *specificity*. In the CFS-C system, the specificity of a classifier is defined as follows:

$$specificity = (S(cond_1) + S(cond_2))/(STRNGSZ * 2)$$

where:

STRNGSZ is the length of message (and condition) strings.

$cond_i$ is condition string i of the classifier.

$S(cond_i)$ is the number of 1 or 0 symbols in $cond_i$ if it is a match-type condition,
or
is the number of # symbols in $cond_i$ if it is a not-match condition.

For example, the following table shows some classifier condition-parts (for $STRNGSZ = 8$) along with the specificities calculated using the formula defined above. Recall that the "$\sim$" symbol before a condition string means the condition is a not-match condition.

$cond_1$	$cond_2$	$specificity$
11110000	$\sim$ 10101010	1.00
1010####	$\sim$ #1#0#0#1	0.50
########	$\sim$ ########	0.00
10100101	$\sim$ 1010####	0.75

Specificity values are always in the range 0.0 to 1.0. Note that classifiers with specificities near 1 are specialists of sorts, since those classifiers' conditions are satisfied only by a small subset of messages. On the other hand classifiers with low specificity values are generalists, since their condition-parts are satisfied by messages from a large subset of all the possible messages.

Also note that interactions between the set of messages that satisfy a match condition and those that satisfy a not-match condition can lead to specificities that don't reflect the actual size of the set of messages that satisfy the classifier's condition-part as a whole. For example, a classifier with the following condition-part:

$$\#\#\#\#\#\#\#\#, \sim \#\#\#\#\#\#\#\#$$

can never be satisfied, i.e., the subset of messages that satisfy its conditions is empty, since the right condition is satisfied only when there are no messages. However, this classifier's specificity (as defined above) is 0.50!

3.2.4 Classifier Strength

Associated with each CFS-C classifier is the classifier's *strength*, stored as a real number (i.e., a C float-type value). Strength may be positive or negative in the CFS-C system, but keeping strengths greater than 0 is probably a good idea. (This can be done by setting the CfStrMin variable to 0.0001 or some similar postive value.)

Each CFS-C classifier also has a "change-in-strength" value associated with it. This value is amount the classifier's strength changed from the previous major-cycle step to the current one. This value is useful for debugging classifiers, to see how much strength classifiers are paying out or receiving during a particular step. The change-in-strength value could also be used by discovery leaning algorithms, e.g., to identify classifiers that make a lot of profit in one time step as potential building-blocks for creating new classifiers.

3.3 Matches

The second step of the major-cycle of a classifier system involves comparing all messages to all classifiers and recording matches for all satisfied classifiers. In the CFS-C system, matches are used to produce new messages; matches also are used by the bucket brigade algorithm to determine which classifiers should be paid for the use of the messages they have produced. The may also be used to generate new classifiers.

In the CFS-C system, all classifiers have two-conditions. Thus matches are generated at the beginning of each major-cycle step by comparing all messages on the current message list to the the first condition of each classifier, and for each classifier that has its first condition matched, comparing all messages to its second condition. For each classifier that has its condition-parts satisfied, a list of the *matches* it is involved in recorded. Each match may be thought of as a tuple of the form:

(classifier, message-1, message-2)

where *message-1* is a message that satisfies the left condition string of the classifier, and *message-2* is a message that satisfies the right condition of the classifier. (If the right condition is a not-match condition, *message-2* is null.)

For example, if the message list contains the following messages:

0000
0001
1111

the following table shows the matches generated for the two classifiers shown:

classifier	Matches
$000\#, \sim \#\#\#1$	$(0000, 0001)(0000, 1111)$
	$(0001, 0001)(0001, 1111)$
$1\#\#\#, \sim 1100$	$(1111, null)$

Note that in classifier systems that use only the standard pass-through action-type, only the message that matches the leftmost condition of a classifier is required to generate a new message. However, in systems like the CFS-C system, in which classifier action-parts can carry out arbitrary operations on one or all of the messages that match a classifier's condition-part, all the messages matching the conditions are important.

Matches are actually recorded by marking bits in two bit-arrays stored in each classifier (one for the first condition, one for the second). The number of messages matching each condition is counted, and messages that are matched by some classifier are marked as "used" (for use by the Cover Detectors discovery algorithm).

3.4 Support

As mentioned in Section 3.1, each CFS-C message has an associated *intensity*, a value equal to the bid made by the classifier that produced it, or, for messages from the input interface, a value supplied by that interface. Thus the intensity of a message can be thought of as a measure of the importance of that message as rated by the message's producer.

In CFS-C the *support* for a classifier is defined as the sum of the intensities for all messages on the current message list that match any of the conditions of that classifier. Thus the support for classifier i at step t is:

$$V_i(t) = \sum_{m'} I_{m'}(t)$$

where m' ranges over all messages that match classifier i at step t, and $I_{m'}(t)$ is the *intensity* of message m'.

The support for a classifier can be thought of as the extent to which the classifiers that were active on the previous time step and messages from the input interface on the current time step "vote" for activation of that classifier on the current step. Support is one factor used in CFS-C to calculate classifier bids (see section 3.6): other things being equal, the higher the support accumulated for a classifier, the more it will bid, and so the more likely the classifier win the competition to post messages.

For example, suppose the message list contains the following length 8 messages with the indicated intensities:

Msg Id	Message	Intensity
1	00000000	75
2	00000001	75
3	00001000	50
4	11110001	50
5	11110010	50
6	11110111	50
7	11111111	50

The following table shows the condition-parts of two classifiers along with the support values accumulated for them, given the messages shown above:

condition-part	Support	Matched Message Id's
0000####, 00001###	200	1, 2, 3
1111####, 00001000	250	3, 4, 5, 6, 7

Note that a single message can match both the first and second conditions of a classifier (e.g., message #3 matches both conditions of the first classifier shown). Thus a question arises: "Should the intensity of a message be added to the support of a classifier only once during a major-cycle step?" There doesn't seem to be an obvious answer, so some experience with different approaches will probably be necessary to settle this issue. In the CFS-C classifier system, a message's intensity is added *only once* to the support value of any classifier it matches. However, the CFS-C source code could easily be changed to try different approaches.

Also note that classifiers with a not-match right condition are satisfied just when some one (or more) messages match its left condition and *no* messages match its right condition. Using the definition of support given above, the support for such a classifier will be just the sum of the intensities of the messages that match the left-most condition. However, the fact that no messsge on the list matches a classsifier's not-match condition also could be taken as a vote to activate the classifier, i.e., a vote by *not* voting to *not* activate it. The CFS-C system attempts to reflect the "silent" support for satisfied not-match conditions by adding a fixed amount to the support of classifiers with not-match conditions. The amount added is specified by the system variable *CfSuppDf* ("classifier support default"), which is set to 100 in the distributed version of the CFS-C system.

Another way to treat support with not-match conditions is to allow the classifier to be satisfied (i.e., eligible to bid), but *decrement* its total support by the intensity of each message that matches the negated condition. Thus the negated condition would be changed from its current "logical" all-or-none form to more of an anolog form (similar to negative weights in connectionist networks). To my knowledge no one has tried this, but I think it might be a more useful type of not-condition when support is factored into bids.

3.5 Bidding

As in other classifier systems, all CFS-C classifiers that have their condition-parts satisfied **bid** to become active and post messages. The bid made by a classifier is the amount it pays if it becomes active and posts new messages—see section 4.5, below. The bid is also used to determine which classifiers win the competition to post messages, as described in the next section.

In the CFS-C system, the *bid* of classifier i at step t, $B_i(t)$, is calculated as follows:

$$B_i(t) = k * S_i(t) * BidRatio^{BRPow} * RelSupport_i(t)$$

k is a small constant (the *Bid_k* parameter in CFS-C, default value 1.0) which acts as a "risk factor", i.e., it determines what proportion of a classifier's strength it will bid and so perhaps lose on a single step. $S_i(t)$ is the classifier's strength at step t. $BidRatio$ is a number between one and zero that is a measure of the classifier's specificity, i.e., how many different messages it can match. $BRPow$ is a parameter that can be changed from its default, 1, to control the importance of the *BidRatio* is determining a classifier's bid. $RelSupport_i(t)$ is the *support* for classifier i at step t divided by the average support for all other bidding classifiers.

If a classifier's bid as calculated above is less than *BidMin*, then the classifier's bid is set to BidMin. MinBid should in general be greater than or equal to zero, especially when classifiers may have negative strengths. Negative bids would create problems for the CFS-C competition mechanism (see the next section) and for the bucket brigade algorithm (should a classifier with a negative bid manage to become active).

Note that support can be excluded from the bid calculation by setting the *SuppBid* parameter to 0; by default, SuppBid is 1, i.e., support is used in the bid calculation.

Also note that as bids are defined here, both the BidRatio and support will alter the fixed-point strengths of classifiers (see the section on bidding for a discussion of fixed-point strengths). I am not sure this is a good idea (see [Riolo, 1987b]). It may be that these factors should both appear only in the effective bid, described in the next section.

3.6 The Competition to Generate New Messages

Once bids have been calculated for all candidate classifiers (i.e, classifiers that have had their condition-parts satisfied), the CFS-C system must decide which classifiers will be allowed to post new messages.

Recall that in the CFS-C system, if a classifier is activated, it will produce one message for each match it is involved in (see Section 3.4). Also recall that the CFS-C system limits the number of messages classifiers can post in one major-cycle (see section 3.1). In particular, the maximum number of messages classifiers can produce is the maximum message list size (defined by the MSGLSTSZ macro) less the number reserved for messages from the input interface (equal to the value of the NmDMsgMx system variable).[1]

If the number of messages that will be produced by activating all the candidate classifiers is less than or equal to the maximum allowed, then the CFS-C system just activates all those classifiers. No competition is run, since there is room on the message list for all the messages they will produce.

On the other hand, if the number of messages that will be produced by activating all the candidate classifiers is greater than the limit, a *competition* is run to select classifiers

[1] It is also possible to limit the number of messages of specific types—see the NmIIMMx and NmIEMMx parameters.

that will be activated. In particular, winners are chosen by making (pseudo) random draws (without replacement) from the set of candidate classifiers. The probability that a given (satisfied) classifier i will succeed is:

$$Prob(\ i\ wins\ ) = \beta_i(t) / \sum_j \beta_j(t)$$

where j ranges over all classifiers bidding at step t. The *effective bid*[2], $\beta_i(t)$, of a classifier i at step t, is:

$$\beta_i(t) = B_i(t)^{BidPow} * BidRatio_i^{EBRPow}$$

$BidPow$ is a parameter that can be set to alter the shape of probability distribution used to choose classifiers to produce messages. For example, if $BidPow = 1$ then $\beta_i(t) = B_i(t)$, i.e., a classifier's probability of producing messages is just its bid divided by the sum of bids made by all satisfied classifiers. $BidPow = 1$ is the standard value that has been used in most classifier system studies. Setting $BidPow$ to 2, 3, and so on, makes it more likely that classifiers with the highest bids will win the competition.

$EBRPow$ ("effective bidratio power") is a parameter that can be used to bias the probability of winning the competition toward (or away from) specialists (high bidratio) or generalists (low bidratio). For example, if $EBRPow = 3$, a specialist rule will tend to win a competition with a generalist rule, other things (i.e., strength) being equal. For a discussion of the use of this parameter to make *default hierarchies* perform as expected, see [Riolo, 1987b].

Why use an "effective bid" in the competition? The main reason is to make it possible to bias the probability of winning the competition to post messages *without* altering the redistribution of strength. See [Goldberg, 1988], [Riolo, 1987b] and [Riolo, 1988d] for a further discussion of the issues involved.

As each winning classifier is chosen, it is activated–that is, it generates new messages by applying its action-part to all the matches it is involved in. The competition is continued and more winners are selected and activated until the number of messages produced is equal to the maximum number allowed. Note that if a classifier is chosen and only some of the messages it wants to produce will fit on the message list, some of that classifier's messages will be posted and some will not (i.e., it will generate messages until the limit is reached).

A number of parameter can be used to influence the production of new messages: *DelHall* can be used to prevent the posting of "hallucinatory" messages, i.e., messages that look just like detector messages; *OneMPerC* can be used to limit each classifier to producing one message per step; *MxDupMsg* can put a limit on the number of duplicate messages posted during a step; and *NmIEMMx* and *NmIIMMx* can be used to limit the production of effector-activating and internal messages, respectively.

There are also two parameters that can further bias the effective bid, *BtCSB* and *BtCMB*. I consider these to be "kludges" that have (so far) proved to be an easy way to get long chains of coupled classifiers to form using just the Triggered Chaining algorithm—see section 3.9.4 for more details.

[2]To my knowledge Goldberg [Goldberg, 1983] first used the idea of an "effective bid," as a way to add random noise to the bidding competition. Holland (personel communication) has also suggested using effective bids in a number of contexts.

3.7 The CFS-C Output Interface

As with other classifier systems described in section 2.1, the CFS-C system processes new messages through an output interface. In the CFS-C system, the output interface is a set of one or more message-activated *effectors*. Each effector may have to resolve conflicts, e.g., when one classifier produces a message that says "set effector A_1 to value x" and another produces a message that says "set effector A_1 to value y". Since the effector can only be set to one value at a time (e.g., a robot can either lift its arm or lower it, but not both at once), an *effector conflict resolution mechanism* must be used. Once an effector value is chosen, all messages that are inconsistent with that setting are *deleted* from the new message list.

Thus each effector in CFS-C consists of four parts:

- A *condition-part* that defines which messages match the effector. Effector condition-parts are just like classifier condition-parts, except that effector condition-parts consist of just one condition string. (The condition may even be a not-match condition.) By convention all effector conditions start with "10", to distinguish them from detector messages, which start with "00" and other internal messages from classifier to classifier, which start "11" or "01".

- A *support–collection subroutine,* which collects "support" for each particular action an effector can take. When an effector's condition part is satisfied by a particular message, the associated support–collection subroutine is called determine what action (or actions) that message advocates (as determined by some of its bits); the intensity associated with that message is then added to the "support" for that action.

- A *action–selection subroutine,* which selects a specific action for the effector to take, given the support collected. This is called after all new messages have been processed by the effector-support subroutine. Basically, when there are conflicts, the effector value r is chosen probabilistically, with the probability that $r = r'$ equal to:

  $$\sum_{m'} \beta_{m'}(t) / \sum_{m} \beta_m(t)$$

  where m' ranges over the effector messages that say "set r to r'", $\beta_{m'}(t)$ is the effective bid of the classifier that posted message m' (i.e., it is the intensity associated with that message), m ranges over all messages matching the effector, and $\beta_m(t)$ is the effective bid of the classifier that posted message m. The winning r value is used to determine what the system's effector will do.

- A *consistency-check subroutine,* which determines for each message that matches an effector whether or not that message supported the action actually selected. This subroutine takes as a parameter a message that matched the effector, and returns TRUE if the action(s) advocated by that message is consistent with the action actually selected, or FALSE otherwise. The CFS-C system then deletes each message from the new message list that is not-consistent with the action taken. This is done so that the contents of the message list available to classifiers during the next major-cycle indicate unamiguously which effectors, if any, were activated, and what messages were used to activate them.

Effector subroutines are *not* included in the CFS-C package: effector subroutines must be defined in the domain-dependent code that must be defined, compiled, and linked to the CFS-C subroutines. Up to 10 effectors may be defined (more can be defined by changing the value of the EFFLSTSZ macro and recompiling the system). Note that effector condition-parts are also not defined in the CFS-C system. In fact, effector-conditions are loaded at run-time. See Section 4, "How to Define an Environment for the CFS-C System" for more details on effector subroutines and conditions.

In CFS-C effectors are not necessarily activated every major-cycle, even if messages match their condition-parts. That is, the system variable "EffectRt" ("effector rate") controls how often effectors are activated. If EffectRt is 1 (the default), effectors are activated every step (if their condition-parts are matched, of course); if EffectRt is 2, they are activated every other step; and so on. If EffectRt is 0, efffectors are *never* activated. Note that message are compared to effector-conditions every step, no matter what the value of EffectRt, and if more than one message matches a particular effector condition, the conflict resolution mechanism is still invoked every major-cycle, no matter what the value of EffectRt.

In the CFS-C system, every major-cycle, after all effectors have been tested and (perhaps) activated, another *domain-dependent* subroutine DoBkgBeh() ("Do Background Behavior") is called. This subroutine, which must be defined and linked to the CFS-C subroutines, can be used to simulate transitions in the classifier system's environment that are not controlled directly by the classifier system's effectors. (For some domains, the DoBkgBeh() subroutine does nothing.)

3.8 The Flow of Strength in CFS-C

The distribution of strength in a classifier system is important for two reasons:

1. It guides the system's short-term behavior, i.e., which classifiers win the bidding competition, post messages, and thereby activate the system's effectors and classifiers (in the next time-step).

2. It biases the search for and generation of new classifiers by the system's discovery algorithms.

This section describes how strength is re-distributed in the CFS-C classifier system as model-time passes, i.e., how strength is transfered from one part of the system to another as the major-cycle is executed.

First note that in the CFS-C system, the total strength is kept constant as model-time passes. To accomplish this, strength is stored in two locations in the CFS-C system:

1. In the classifier list, i.e., each classifier has its own strength.

2. In a global "system treasury" variable, *SysTreas*.

Thus, the total strength in the CFS-C system is the sum of all the classifiers' strengths plus the system treasury value. To keep the total strength constant, then, when one component (e.g., a classifier) gains strength another component (another classifier or the system treasury) must lose an equal amount of strength.[3]

There are three mechanisms in the CFS-C system that act to transfer strength:

1. The Bucket Brigade algorithm.
2. Taxes.
3. The systems discovery algorithms, which create new classifiers and remove classifiers from the classifier list.

These mechanisms are described in detail in the following sections.

3.8.1 The CFS-C Bucket Brigade Algorithm

The CFS-C system implements the "standard" bucket brigade algorithm as described in section 2.2. That is:

1. *Rewards* are transferred from the environment to active classifiers; and
2. *Bids* are transferred from active classifiers to previously active classifiers.

First note that the *payoff function* that specifies which states of the environment are associated with non-zero rewards is *not* part of the CFS-C package. In the CFS-C system, rewards are provided by a *domain-dependent* subroutine GtSysRew(), Get System Reward, which must be compiled and linked to the CFS-C subroutines. The GtSysRew() subroutine is called once every major-cycle step, just after messages have been processed through the system's output interface (the effectors) and after the "background behavior" subroutine has been evoked (see the previous section). The GtSysRew() subroutine simply returns a real (float) value that represents the reward associated with the current state of the environment.

When the GtSysRew() subroutine does return a non-zero reward to the CFS-C system, *every* classifier active at that time step has the *full* reward added to its strength (where an active classifier is one that has produced a message during that major-cycle step).

One obvious alternative to this distribution mechanism would be to divide the reward amoung the classifiers active when the reward is received [Booker, 1982] [Wilson, 1986b]. This can be done by setting the *ShareRew* parameter to 1. Note that this would penalize classifiers that act in concert with others, since the larger the cluster of cooperating classifiers, the less reward each would receive. On the other hand, sharing reward acts as a natural limit on the number of resources available in a particular niche, thus acting to limit the number of phenotypically similar classifiers (i.e., classifiers responding to the same situation and suggesting the same action). I have found sharing reward to help with this "premature convergence" problem, although it doesn't work as cleanly in systems with limited message lists sizes (like CFS-C) as it does in those with unlimited message lists.

Another important CFS-C parameter that influences the distribution of rewards is *FrPayNEC*, which controls the fraction of a reward (or reward-share) paid to non-effector

[3]**NB:** I do not use SysTreas, so it is no longer kept up to date; it should probably just be removed from the system.

activating classifiers (i.e., rules that don't produce messages that start "10"). In the traditional system, *all* rules active when a reward is received have it (or a share) added to their strengths, (i.e., FrPayNEC = 1.0). One problem this leads to is *parasitic* classifiers, i.e., classifiers that are active when rewards are received but that produce nonsense messages not used to control effectors or by other classifiers. However, if FrPayNEC is set to 0, then *only* classifiers that produce effector-activating message receive rewards from the environment. All other rules must make profits by being paid by other classifiers that use their messages. This eliminates the many parasites of the type described above, since they no gain strength and so are eventually eliminated from the system.

Many other alternative reward mechanisms could and probably should be tried. The CFS-C system is designed to make trying alternative mechanisms relatively easy.

Note that in some domains/environments, rewards may be negative values. In this case the flow of strength is from the classifiers to the system treasury.

Besides passing rewards to active classifiers, the bucket brigade algorithm also transfers strength from currently active classifiers to those classifiers that made it possible for them to be active. To see how this transfer is carried out in the CFS-C system, first recall that classifiers can be thought of as simple message processors. That is, when a classifier is activated it takes as input all the messages that matched its condition-part, and produces as output new messages created by its action-part. Let a "BidShare" be defined $BidShare = Bid/n$, where Bid Is the bid made by the classifier during the current major-cycle, and n Is the number of messages matching the classifier when it produces new messages. When a classifier i wins the competition and posts a new message, its strength is reduced by the amount it bid. Further, a BidShare is added to the strength of each classifier that produced (on the previous time step) a message that matched classifier i and enabled it to become active.

In summary, the bucket brigade algorithm acts in two ways:

1. It adjusts the strength of those classifiers that are active when a payoff is received from the environment. Each classifier's strength is changed incrementally until in the long run, it is proportional to the average of the payoffs the system receives when that classifier is active.

2. It redistributes strength from each active classifier to classifiers that posted messages that activated it. Each classifier's strength is modified incrementally until it is proportional to the strength of the classifier(s) it activates.

Over time the bucket brigade algorithm re-allocates strength from classifiers that *directly* lead to payoffs from the environment to those classifiers that *indirectly* lead to payoff, i.e., to classifiers that post messages that "set the stage" for those classifiers directly reponsible for receiving payoffs.

In particular, the strength $S_i(t+1)$ of a classifier i that wins the bidding competition and posts messages at t is[4]:

$$S_i(t+1) = S_i(t) + R(t) + P_i(t) - B_i(t)$$

where $S_i(t)$ is the strength of classifier i at t, $R(t)$ is the reward from the environment during step t, $P_i(t)$ is the sum of all payments to classifier i from classifiers that posted messages at step t by matching messages produced by i at step $t-1$, and $B_i(t)$ is the classifier's bid during step t. (For classifiers that did not win the bidding competition at step t, the equation is $S_i(t+1) = S_i(t) + P_i(t)$.) Clearly a classifier's strength reaches a fixed-point when the amount of strength it receives is equal to the amount it pays. Thus in the long run a classifier's fixed-point strength, S_{fp}, approaches:

$$S_{fp} = (R + P)/(k * BidRatio)$$

where R and P are the average amounts the classifier receives *per activation* as rewards from the environment and payments from other classifiers, respectively.

The bucket brigade algorithm has several characteristics that make it ideal for use with highly parallel systems like classifier systems. First, the bucket brigade algorithm uses only *local* information: when adjusting the strength of a classifier, it only needs to know which classifiers *directly* activated it and which classifiers it *directly* activates. There is no need for complicated book-keeping or for high-level critics to analyze sequences of actions and assign credit accordingly. This is in contrast to systems like LEX [Mitchell *et al*, 1983], in which full traces of system behavior are used to allocate credit to rules that are active early in a multi-step process. Second, the bucket brigade algorithm works in a highly parallel way, changing the strength of many (or all) rules at the same time.[5] Third, the bucket brigade algorithm acts *incrementally*, changing the strength of classifiers gradually. By changing the strength of classifiers only a small amount at a time, the classifier system tends to learn *gracefully*, without the precipitous changes in performance that may result from making a large change in response to a single, possibly anomalous, case. In this respect the bucket brigade algorithm is similar to the "temporal differences methods" described by Sutton [Sutton, 1988], a class of incremental learning procedures which he has shown to be superior (in terms of *both* learning speed and computational costs) to traditional supervised learning procedures for problems that can be cast as multi-step and single step prediction tasks. Finally, if the bucket brigade algorithm performs as expected, a classifier's fixed-point strength will be proportional to the average payoff the system has received when it was activated, other things being equal. Thus when the bid competition is run to see which classifier will become active and which will not, those classifiers that on average lead to higher payoffs will tend to win, post messages, and control the system's behavior.

One well known problem with the bucket brigade is the exponential fall-off of strength up chains in which one condition of each classifier matches a detector message while the other matches a message from its predecessor in the chain (see [Riolo, 1987a]).

[4]See [Goldberg, 1983] for a derivation of a fixed-point strength for classifier strength. Actually the CFS-C system also provides for several types of "taxes", e.g., a tax on every classifier every step, a tax for bidding, and so on, which are described later in this chapter. I have ignored those in this derivation of the fixed-point strength of a classifier.

[5]Robertson [Robertson, 1987] has shown that almost all aspects of learning classifier systems are parallelizable by implementing a classifier system on a Connection Machine.

In order to alleviate that problem, I have introduced a new variable, *DMShare* (detector message share) into the CFS-C system. The basic idea is to bias payments away from messages from detectors to messages produced by other classifiers, thus reducing the falloff in in strength up a non-reflex chaing of coupled classifiers. The way I have implemented this is to bias the BidShare paid for each matching message used to produce new messages so that messages from detectors are paid DMShare times share paid for non-detector messages. Thus if DMShare is 1 (the default), the system acts as originally defined, i.e., pay the same for all messages used. If DMShare is 0.1, then the amount paid for each detector message matched is 10the bid is paid "up the chain."

This is implemented roughly as follows: Let B = bid, b = bid share to non-detector messages, f = DMShare, m = total matched messages, d = number of detector messages matched, and g = FrPayDet, the existing Fraction Pay Detector Messages variable (see below).

All the bid shares must equal the bid, B, so we want:

$$(f * d * b) + (m - d)b = B$$

that is, pay-detectors + pay-classifiers = Bid, so

$$b = B/m + d(f - 1)$$

Call (f - 1) the "detector messages discount", DMDscnt. There are three cases for a given classifier that bid B to post messages:

- All messages used (matched) are from detectors: In this case just decrease the classifiers strength by g * B. DMShare doesn't have an effect here, since there are no source classifiers to be paid.
- All messages used (matched) are from other classifiers. In this case b is paid for each matched message used to post a new message. Again DMShare has no effect.
- Some matched messages from detectors and some from classifiers. In this case first decrement S by d (f g b) for the d detector messages matched. Then pay b for each matched message produced by classifiers.

[Implementation note: Each classifier contains a counter, *NmDetMM* (number of detector messages matched). It is initialized in PostCfMs(), incremented in MakeNMsg(), and (possibly) decremented during the effector conflict resolution mechanism in GenBehav(). (All these subroutines are in the file CORE.C)]

A second way to alter the amount paid for detector messages is the *FrPayDet* ("fraction pay detector") parameter. In particular, for each input message that comes from a detector,

$$BidShare * FrPayDet$$

is paid out, rather than a full BidShare, where $0 \leq FrPayDet \leq 1.0$. Thus since the amount paid for messages from detectors may be less than the BidShare value, the total paid by a classifier may be less than the amount it bid. By default, FrPayDet is 1, i.e., pay the same for detector messages as for other messages. FrPayDet may be usefull to bias the system to classifiers that match detector messages by actually paying less for those messages (and not transfering that payment to other classifiers, as DMShare does).

3.8.2 Taxes

There are several kinds of taxes that classifiers may have to pay. While the different types of taxes have different *trigger-conditions* and possibly have different tax-rates, each tax is applied in a similar way (with one exception, described below). In particular, if a classifier is taxed at rate R at step t, its strength at the next step (ignoring other changes) is:

strength(t+1) = strength(t) - (strength(t) * R)

Each tax rate can be set at run-time to values from 0 to 1. Thus a particular tax can be turned off by setting its rate to 0.

There are three basic types of taxes implemented in CFS-C:

1. The Head Taxes (parameters *HeadTax* and *AHeadTax*).

   The head taxes are applied to every classifier once every model time-step (i.e., once every major-cycle step). The HeadTax parameter is used as a tax rate, i.e. strength is decremented by strength times HeadTax. The AHeadTax parameter is used as a fixed head tax, i.e., strength is reduced by AHeadTax itself.

   The main reason for these taxes is to put a pressure on classifiers that never bid. That is, if the head tax is 0, classifiers that never bid would never lose strength. When the head tax is set to some small, non-zero rate, classifiers that never bid gradually lose strength, so that eventually they become more likely to be replaced by new classifiers generated by the CFS-C discovery mechanisms.

2. The Bid Taxes (parameters *BidTax* and *FBidTax*).

   The bid taxes are applied selectively, to classifiers that bid. The *BidTax* is applied to every classifier that makes a bid at a given cycle step, whether or not that classifier actually wins the bidding and posts messages. The FBidTax, if used, is applied only to those classifiers that make a bid but that do *not* win the competition.

   One reason for a bid tax is to penalize overly general classifiers, i.e., classifiers that bid on every step but perhaps seldom win because they have a low bidratio (specificity) that leads to low bids. A second reason for a bid tax is that to properly model some task-domains, it should be possible to assign a cost to bidding, e.g., to model the fact that bidding does use system resources. A third reason (related to the first) is to control the spread of certain classifiers, until there are many classifiers that all do the same thing, to the detriment of others.

   The FBidTax was introduced to help control convergence without hurting the flow of strength down chains. To see how this works, note first that every tax causes an exponential decrease in the strengths of rules as they are furthre and further removed from rewards from the environment, since every classifier in a chain is taxed. Thus it is best to minimize taxes. However, bid taxes do serve to control the spread of phenotypically identical classifiers, since if there are more copies of a classifier type than there is room on the message list, some of the copies will bid (and be taxed) but not gain any income (since they will lose the competition to their cohorts). The FBidTax acts in this way to limit the spread of classifiers, too, but it also does *not* hurt chains since it is not applied to rules that win the competition.

3. The Producer Tax (system variables *PrdTaxMx* and *PrdTaxPw*).

   The producer tax is applied only to those classifiers that actually produce new messages during a particular major-cycle.

   Like the bid tax, one reason for a producer tax is to put additional pressure on classifiers that produce lots of messages but seldom get rewarded. For example, a classifier that produced a message that matched its own conditions would, once it was started, continue to produce messages indefinitely, paying only itself if it was the sole source of its input messages! While there are ways to make use of such a classifier, e.g., as a kind of permanent memory, in general such repetitive activity is not useful. A non-zero producer tax ensures that such a classifier would lose strength every step and thus become a candidate for replacement by a new classifier.

   The tax on a classifier i during step t is calculated as follows:

   $$r_i(t) = PrdTaxMx * (m_i(t)/M)^{PrdTaxPw}$$

   where $PrdTaxMx$ is the maximum producer tax, $m_i(t)$ is the number of messages produced by classifier i during step t, M is the maximum message list size, and $PrdTaxPw$ is typically some small value larger than 1, e.g., 1.5 or 2. Thus if a classifier produces just one message, the tax is quite low, but if the classifier produces enough messages to fill the message list by itself, it is taxed quite heavily.

   Note that classifiers that win the bidding process but have all their messages removed from the new message list by the conflict resolution mechanism at the effector interface are **not** charged the producer tax. (See section 3.7 for a description of the conflict resolution mechanism used by the CFS-C effectors.)

3.8.3 Classifiers Created by Discovery Algorithms

The CFS-C discovery learning algorithms (see Section 3.9) create new classifiers that are variations of existing, typically successful, classifiers. When a new classifier is created, it must have a strength value assigned to it. Different discovery algorithms may use different rules to decide what strength to assign to each new classifier, e.g., equal to the average of the parents, equal to the average of all classifiers, etc.

Typically the strength of new classifiers is the current average classifier strength. Since the classifiers being replaced tend to be low strength classifiers, this mechanism tends to increase the average classifier strength over time. This inflation could be avoided by assigning the strength of the classifiers being replaced to the new classifiers replacing them. However, then new classifiers would also tend to have very low strengths, which means they would tend to be replaced rather quickly, and so not get a chance to be used before being replaced!

3.9 The CFS-C Discovery Learning Mechanisms

The CFS-C system has several learning algorithms that generate (discover) new classifiers. Each algorithm can be thought of as selecting some "parent" classifiers (typically one or two), applying an *operator* to them, producing one or two new "offspring" classifiers, and then replacing some existing rules (typically not the parent rules) with the offspring.

The CFS-C system currently contains the following discovery algorithms:

1. A *Genetic Algorithm (GA),* which creates new classifiers by modifying offspring (copies) of existing, high-strength (i.e., useful) classifiers in ways that lead to a rapid exploration of portions of the space of all possible classifiers that are likely to contain other useful classifiers. The GA uses *Crossover* and *Mutation* operators.

2. A *Cover Detector Messages (CDM)* algorithm, which makes it possible for the CFS-C system to rapidly create classifiers that match all detector messages the system has seen.

3. A *Cover Effectors (CE)* algorithm, which helps the system discover classifiers that at least activate the systems effectors, though perhaps not always in the correct situation or way.

4. Two mechanisms to promote "coupling" of classifiers, so that rules can activate other rules and receive payment (via the bucket brigade algorithm) for doing so. The two mechanisms are called the *Cover Stage-Setters (CSS)* and *Couple Profitable Classifiers (ACPC)* algorithms; in fact, the CSS can be thought of as a generalization of ACPC, so the ACPC will not be described here (and I will probably remove it soon).

5. The *Low-Bid (TLB)* algorithm, to generate new rules with new actions when the all the rules responding to a situation are low bidding (i.e., low strength) rules, i.e., ones that have not led to useful system behavior.

A more detailed description of the operation of each of these is given below. For complete details, see the source code in the files DSCLEARN.C (main entry point for discovery algorithms), DSCOPS.C and DSCOPS2.C.

All the discovery algorithms make use of classifier strengths (or bids) as an indicator of the usefulness of classifiers. That is, each of these mechanisms generates new classifiers that typically are modified copies of existing, high strength classifiers. Further, the new classifiers generated by these mechanisms tend to replace low strength classifiers. Thus these learning mechanisms depend on the action of the CFS-C bucket-brigade algorithm (see section 3.9), which allocates strength to those classifiers that have (in general) contributed directly or indirectly to meeting the goals or solving the tasks defined for the CFS-C classifier system in a particular environment.

Only a certain fraction of the classifier list can be replaced per time step (i.e., their is maximum number of new rules per step). This maximum is controlled by the *FrNewCfs* parameter. Typically the maximim number of new classifiers per step is $2 - 4\%$ of the classifier list.

All the discovery mechanisms *except* the genetic algorithm are *triggered* algorithms—they are candidates to be activated only when the system detects some particular criteria being met. Each triggered operator can be applied probabilistically, i.e., even when its trigger is satisfied it will be applied with a probability set at run-time. If the operator is activated, and if there is still room for more new classifiers in the current step, it will typically produce one or two new classifiers, which are temporarily saved. Once all the triggered operators have had a chance to fire, if there is still room for more new classifiers, the GA may be activated to produce new classisifers up to the maximum allowed per step. When all the new classifiers have been created, classifiers in the current list are picked to be replaced by the new ones.

Some important parameters that control the discovery algorithms include:

- *NwCfStrV.* This controls the strength of new classifiers. If it is 0 (the default), the strength is just the average of the entire list multiplied by *NwCfStrF*, another parameter. If NwCfStrV is 2, the new strength is half way between the strength of the parents and the average for the entire list. If it is 3, the average of the parents is used.

- *PckPrnWOR.* This controls whether or not the GA picks parents without replacement, or if a rule can be selected as a parent more than once in one step.

- *BidFit.* If this is 0 (the default), strength is used as a "fitness" measure when selecting parents and deleting rules. If it is 1, the *bid_k* times strength is used as a fitness measure.

For more details, see the rest of this section and Appendix C.4.

No matter what mechanisms are used to generate new classifiers, the same algorithms are used to choose which rules are to be replaced. By default, rules are replaced with probability inversely proportional to strength (or bid-estimate if BidFit is 1). However, there are a number of parameters that affect the replacement algorithms:

1. *RplcBPow.* The probability that a classifier is replaced is proportional to the inverse of its strength raised to this power. By default this is 1. Raising it obviously biases things against weak rules.

2. *RplCfUBd.* This can be used to protect higher-strength classifiers from replacement. In particular, all rules with strength greater than the average rule strenght times RplCfUBd are protected. Thus a good value for RplCfUBd is 1.0, i.e., protect above average strength rules. Setting it to 100 effectively turns protection off.

3. *RplACUBd* and *RplACSBd.* These can be used to protect recently active rules. A classifier is considered "recently active" if active in the last RlpACSBd steps. If it is recently active, and its strength is greater than RplACUBd times the average rule strength, it will be protected.

4. *RandRplc.* If this is 0, classifiers to be replaced are selected probabilistically, proporional to inverse fitness (strength or bid). If it is 1, classifiers are picked using a uniform random selection procedure, with no bias based on stength or bid. If it is 2, classifiers are picked using a uniform random selection procedure, but only from below average strength rules.

5. *CrowdFac.* This can be used to implement a *crowding factor* into the replacement algorithm, as described by [DeJong, 1975]. The default for CrowdFac is 1. If CrowdFac is set to say 4, then when new rule is to replace another one, 4 rules are chosen

as candidates for replacement, and the one that is syntacically most like the new one is actually replaced. This mechanism, when used with RandRplc of 1, is intended to control the "premature convergence" problem: however, no experiments have been done to show that it does.

6. *MxCfCopy.* This can be used to put an absolute limit on the number of genotypically identical rules. The default value is 0, which means to limit the number. If this is set to, say, 2, then every time a new classifier is to be added to the list and replace an old one, if there are are already 2 identical rules it will just replace the lowest strength one of those. Thus MxCfCopy is a heavy-handed way to control premature convergence.

The rest of this section describes the various discovery algorithms and operators in more detail.

[**Note well:** The discovery algorithms included in CFS-C are meant to serve as examples of the kinds of algorithms that can be (and have been) used to promote learning in classifier systems. This list is not meant to be complete—other useful algorithms could be (and probably should be) added to classifier systems to make them closer to "general" learning systems. See [Holland, 1986c] for some suggestions of other useful discovery algorithms.

Also note that the particular implementations in CFS-C are not meant to be definitive descriptions of the discovery algorithms, either. To repeat: There are many ways to implement these algorithms, with many "philosophical" and technical decisions to make all along the way. The implementations in CFS-C are in some ways much like a natural system: things have been added, modified and removed over time, all working on top of a moving target (i.e., the non-learning parts of CFS-C) trying to meet moving constraints and goals (new ideas of what are good operators, what fitness should be, etc.), all leading to what I consider to be a bit of a mess! Perhaps in CFS-C 2 (or 3 or ...) some things will be "known" so the code can be simplified and cleaned up.]

3.9.1 The Genetic Algorithm

The genetic algorithm (GA) used by CFS-C is basically a very standard type like that defined by [Holland, 1976] and [Holland, 1986c]. Basically, new classifiers are generated by:

1. Generating offspring (copies) of existing, high-strength classifiers; and
2. Crossing some of the offspring to produce new classifiers different from but similar to the parent classifiers.
3. Applying a mutation operator to some of the offspring.

Actually, the mutation operator is applied to **all** new rules, whether produced by the GA or by any of the other rule discover algorithms.

I call the GA the *background* discovery mechanism in CFS-C, since it is activated at a specified rate (i.e., with a given probability every step) independent of any specific triggering conditions, and only produces new rules if the other triggered algorithms have not already produced the maximum number of new rules allowed per step. When the GA is activated, the following is repeated until the maximum number of new rules has been produced:

a. Two parent classifiers are chosen and a copy of each is made. In general, high-strength classifiers are chosen to be reproduced, but the selection is done probablistically so that all classifiers have some chance of being reproduced. The BidFit and BkgGABP parameters also control how classifiers are chosen, but the default is to use strength as a "fitness" measure for reproducing classifiers.

b. The *crossover* operator is then applied to some of the offspring classifiers to produce new variants of each. (Some offspring may not crossed, i.e., they are just copied.)

The crossover can be done in two ways:

- The string to be crossed can be either the first condition, the second condition, or the action (chosen with equal probability). For example, suppose the two new classifiers are these:

  $aaaaaaaaaaaaaaaa, bbbbbbbbbbbbbbbb/PASS/cccccccccccccccc$
  $dddddddddddddddd, eeeeeeeeeeeeeeee/PASS/ffffffffffffffff$

  If the part to be crossed in the second condition string, and the location of the cross is the boundry between the eighth and nineth loci (counting the left-most loci as the first), the resulting classifiers will be these:[6]

  $aaaaaaaaaaaaaaaa, bbbbbbbbeeeeeeee/PASS/cccccccccccccccc$
  $dddddddddddddddd, eeeeeeeebbbbbbbb/PASS/ffffffffffffffff$

  Note that the condition-types and action-types are not crossed.

- The full classifiers can be treated as a strings to be crossed. For example, suppose the two new classifiers are these:

  $aaaaaaaaaaaaaaaa, bbbbbbbbbbbbbbbb/PASS/cccccccccccccccc$
  $dddddddddddddddd, eeeeeeeeeeeeeeee/ADD1/ffffffffffffffff$

  If the cross occurs between the eigth and ninth loci of the second condition, in this case the offspring wil be: For example, suppose the two new classifiers are these:

  $aaaaaaaaaaaaaaaa, bbbbbbbbeeeeeeee/ADD1/ffffffffffffffff$
  $dddddddddddddddd, eeeeeeeebbbbbbbb/PASS/cccccccccccccccc$

  Note that the action is exchanged in this case.

c. In any case, the new offspring is subject to mutation, as described below.

The value of the system variable *BkgGARt* specifies the probability that the background genetic algorithm is applied on each step. The probability that crossover is done within a classifier-part is *BgaSCPr*, the probability that the full classifiers are treated as strings to cross is *BgaFCPr* - *BgaSCPr*, and the probability that no crossover is done is 1.0 - *BgaSCPr*

[6] If this is printed in proportional fonts, the characters may not line up—you will have to count letters to find the crossover points.

- *BgaSCPr.* (That is, these three parameters define a cummulative probability distribution if values are assigned properly.)

Although the examples given above describe *single* crossovers, the default crossover in CFS-C is currently to do *double* crossovers. This involves picking two crossover points and exchanging all loci between them. There is some controversy in the literature as to which is better, but double crossover seems to be favored of late. At any rate, in the file UTILITY.H there are compiler switches which can be changed to have the system use single crossovers if you wish.

The Mutation Operator

All new classifiers are subject to *mutation,* no matter which algorithm produces them. Each new classifier will be mutated at zero, one, two, etc., loci, the exact number controlled by a poisson distribution with a (default) mean of 0.05. This mutation rate is controlled by the *MuPrTot* variable, which can be set to any value in the range 0.0 to 1.0 at run-time.

If a classifier is to be mutated, the probability that the mutation will occur in a particular component is specified by the following cummulative distribution function:

Component	Cum. Prob.	System-Variable
Condition-1	0.30	MuPrCnd1
Condition-2	0.60	MuPrCnd2
Action	0.90	MuPrAct
Condition-2-Type	0.95	MuPrCT2
Action-Type	1.00	MuPrAT

Note that the variables on the right can be changed by the SET command. However, the user must be sure the values always define a cummulative probability distribution over the five components.

If a condition or action string is to be mutated, a locus to mutate is chosen at random. Then the value at that loci is set to 0, 1, or # with the following probabilities:

Value	Prob.
0	0.40
1	0.40
#	0.20

This distribution is governed by one system variable, *MuFrNSP* ("mutation fraction of non-specific loci", i.e., # symbols). The default value for MuFrNSP is 0.20, as shown in the above table. The probability of the loci mutating to a 1 or 0 (specific loci values) is then equal to the rest (0.80), divided equally between 0 and 1.

If the second condition-type is mutated, the new condition-type is a match condition-type with probability 0.95, and a not-match condition-type with probability 0.05. The system variable *MuFrMC* ("mutation fraction match condition") can be changed from the the default value of 0.95 to any value between 0 and 1.

If the classifier's action-type is mutated, the result is determined by a random draw from the following distribution:

Action-Type	Cum. Prob.
PASS	0.90
AND	0.02
OR	0.02
XOR	0.02
ADD1	0.02
SUB1	0.02

This distribution cannot be changed during a run. Of course the mutation of action-types can be disabled by changing the value of the PrMuAT variable described earlier to 0.

3.9.2 Cover Detector Messages.

When a classifier system is started with random classifiers, or whenever the system encounters novel situations, there will be detector messages not matched by *any* classifiers. Using just the genetic algorithm it may be some time before a rule is discovered that responds to the novel messages at all, and then it must be refined to be a useful rule. Since it is imperative that the system attend to information about its environment (i.e., detector messages), a number of schemes have been proposed for handling unmatched detector messages [Booker, 1982], [Wilson, 1986b], [Davis and Young, 1988], including schemes like the Cover Detector Messages operator used in CFS-C.

The trigger condition for the Cover Detector Messages operator is the existence of a detector message that is not matched by any classifier during a given step. The rate of application is controlled by the *CDMsgRt* parameter, which specifies the probability the triggering condition will be *tested* on each cycle.

If the parameter *CDMsgs* is 0, the operator is turned off. If CDMsgs is 1, it generates a new rule is constructed by:

a. If there are classifiers that bid during the current step (but of course didn't match the detector message), pick one of those (with probability proportional to bid) and copy it. If no classifier bid that step, pick a parent classifier that is the *closest match* to the unmatched message, and copy it. The match score runs from 0 to STRNGSZ * 2, where 1 is added for each matched, specific locus.

b. Use the unmatched detector message to minimally generalize one or both of the conditions of the offspring rule so that it matches the message and will bid next time the same situation occurs.

If CDMsgs is 2, it generates a new rule is constructed by:

a. Selecting a parent classifier, with probability proportional to strength, and copy it.

b. In each condition, change the specific loci (i.e., those with 0 or 1) to match the value of the message.

No matter how the offspring was produced, if the value of *CDMsgAct* is 0 (the default), the new classifier with the modified conditions is added to the list. If CDMsgAct is 1, then the action part of the new classifier is changed to be a random action-string.

I leave it as an exercise for the reader to determine the pros and cons of these (and other) ways of generating classifiers to cover detector messages. I will say that unless some method is used, the system will not get very far if started with random rules or when faced with novel detector messages, as has been pointed out by [Booker, 1982], [Wilson, 1986b], [Davis and Young, 1988] and others.

3.9.3 The Cover Effectors Operator

Just as the Cover Detector Messages operator serves to keep the system responsive to information from the environment, the Cover Effectors operator acts to keep the system able to respond. That is, when started with random rules or with a large repetoire of possible actions, the system will not in general have rules that fire all effectors or even any effectors! Of course if the system doesn't activate effectors, it cannot affect the reception of payoffs (or lack thereof), which in turn means there is no "credit" to allocate (via the bucket brigade) to rules, and so there is no information for the genetic algorithm to use about which rules are better or worse.

Further, when the system first discovers a rule that does recommend a *useful* action, its strength grows rapidly and it becomes the source of all new offspring, leading to a "premature convergence" problem in which the population has lost most of its variance, especially in effector strings. In such cases crossover is no longer an effective way to generate new rules with novel actions.

The Cover Effectors (CEO) operator is a triggered operator intended to maintain a diversity of action-strings in the population. Basically it creates a new rule that has some random string as its action-part, and has conditions that are satisfied by messages on the list when the operator is activated.

The CEO is turned "on" by setting the parameter *CEffs* to a value of 1 or 2 (the differences are described below) and by setting the *CEffsRt* parameter to some value greater than 0 and less than or equal to 1. CEffsRt specifies the probability the operator is activated when the trigger conditions are met.

The trigger condition for the CEO is set by the domain-specific subroutines, by setting the *MadeMtsk* variable to TRUE. Thus the trigger can be a general condition, like "no effector message at all was posted recently," or it can be more specific, like "the wrong effector action was selected." (Of course in many domains there is no obvious way to define "wrong" action, thus the need for more general rules.) For example, in LETSEQ1 and in FSW1, the trigger is set whenever the reward received was less than 0, indicating explicitly that a "mistake" was made and some other action should be tried.

When the CEO is applied, it creates either one or two new classifiers, as specified by the value of the *CEffs* parameter. The parents of the new classifiers are selected as follows: if there are classifiers that activated the effector, select those (picking them at random if there are many to choose from); if no classifiers activated the effector, but some did bid, select parents from the bidders, with probability proportional to their bids; if there were no bidders, don't select any, and don't create new classifiers. Copy the parent(s) selected, and then modify the offspring by generating a new random, action-string for them.

Thus the CEO is like the mutation operator, in that it implements a kind of "random walk" search of the space of possible actions, with its main purpose being to maintain diversity in the population. Unlike mutation, the Cover Effectors operator is a triggered operator, so that it only is used when it is needed.

Some other variables that control the behavior of the Cover Effectors operator include:

1. *CEffSplz*. When set to 1, one of the offspring has one of its conditions "specialized" by changing a randomly selected don't-care symbol to a one or zero. The idea is that if a mistake is made, there might be two causes: (a) the conditions are correct, but the action was wrong, or (b) the action is correct, but it is applied too generally.

2. *NmEAct*. This is a counter in each effector structure which indicates the number of messages that match the effector in the current step. Thus this can be used to determine if the Cover Effectors operator should be applied.

3. *CEffFlg*. This is a flag in each effector structure. If set to TRUE (when the domain is initialized), the effector will be checked for "mistakes" and new classifiers to activiate it will be generated. If it is FALSE, the Cover Effectors operator will not apply to it. Note that for this to be really useful, it would probably be a good idea to introduce separate triggers for each effector.

Again, the exact implementation of the CEO currently in CFS-C is by no means known to be the best, or even good! However, it does seem to be necessary, at least in some domains, to get the system started learning and to keep the population diverse enough to allow efficient use of the Crossover operator of the GA. See [Robertson and Riolo, 1988] for a further discussion of the CEO.

3.9.4 Triggered Chaining Operators

Chains of coupled classifiers are required if classifier systems are to get beyond simple stimulus-response paradigms, to maintain short-term memory of messages, to carry out arbitrary computations, and so on. (A classifier C_1 is said to be *coupled* to classifier C_2 when a message produced by C_1 satisfies some condition of C_2.) Using just the GA, coupled chains rarely if ever form. Thus the CFS-C system includes special *Triggered Chaining operators* designed to promote the formation of coupled rules. The implementation is similar to that proposed in [Holland, 1986c] and used in experiments described in [Robertson and Riolo, 1988] and [Riolo, 1988d].

There are actually two subroutines in CFS-C that form coupled rules: DscACPC() and DscCSS(). The latter implements what I call the *CSS* (Couple Stage Setter) operator, which is a generalization of the former ACPC operator. Thus I will describe only the CSS here (see DSCOPS2.C for the ACPC description).

The basic purpose of the CSS operator is to create a pair of *coupled* rules that reflect an initially *accidental* activation of one rule after another. Like all classifiers, the members of the coupled pairs produced by the CSS operator are evaluated and assigned credit (or blame) by the bucket brigade algorithm. If the coupled rules improve performance (and so gain strength), they serve as building blocks for longer chains and other more complicated structures. If the coupled rules do not improve performance, they lose strength and are replaced.

In order to bias the production of coupled classifiers toward plausibly useful chains, the CSS operator is activated only when two conditions are met:

1. A classifier C_2 makes a *profit* at step $t-1$, and

2. There was a classifier C_1 active just prior to classifier C_2, at step $t-2$.

where the *profit* of classifier i at step $t-1$, $P_i(t-1)$, is defined as the reward, $R_i(t-1)$, classifier i received at $t-1$ less the bid, $B_i(t-1)$, classifier i paid at $t-1$, *plus* the amount paid to it by other classifiers active at step t. (Thus the profit at step $t-1$ can only be calculated at step t.)

For example, consider the following two classifiers, A and B, active at steps $t-2$ and $t-1$, respectively:

$$\begin{array}{ll} (t-2): & A = aaaa, aaaa/xxxx \\ (t-1): & B = bbbb, bbbb/yyyy \end{array}$$

If $P_B(t-1) > 0$ and if A and B are not already coupled, then the Triggered Chaining operator would produce the following new classifiers:

$$\begin{array}{l} A' = aaaa, aaaa/mmmm \\ B' = bbbb, mmmm/yyyy \end{array}$$

where $mmmm$ is a string that can be constructed in a variety of ways (as is discussed later in this section). The result is a pair of coupled classifiers constructed so that if the conditions that caused A to fire are again encountered, A' also fires and produces a message $mmmm$, which may cause classifier B' to fire on the next step, which in turn produces a message $yyyy$ which was associated with making a profit. If $yyyy$ does indeed consistently lead to profit, the bucket brigade algorithm will allocate strength to the coupled rules and they will survive and act as building blocks for longer chains.

As mentioned above, the string $mmmm$ that is used to couple two classifiers can be constructed in a variety of ways [Holland, 1986c]. For example, one way is to just use the string $xxxx$ from action-part of the parent classifier A. This is possible because in classifier systems like CFS-C both the condition-parts and the action-parts of classifiers are just strings build from the $\{0, 1, \#\}$ alphabet. One possible problem with this approach is that the new classifier B' will be coupled to both the new classifier A' and to the parent classifier, A. This will result in the bid of classifier B' being *shared* by classifiers A and A', which can lead to stability problems for sequences of classifiers.

Another way to construct the string $mmmm$ is to generate it randomly. One problem with this approach is that if the parent rule A was an effector activating classifier that happened to move the system into a state where B could be used, the new rule $A\prime$ probably would not produce that same action. Thus the the usefulness of the coupled pair $A' \rightarrow B'$ will depend on the actions of another classifier, A, to which they are not coupled.

In the CFS-C system, a third approach is taken which is a combination of the above methods. The new message $mmmm$ is constructed by replacing a designated portion of the string $xxxx$ with a randomly generated string over $\{0, 1\}$. The portion replaced is a fixed portion that is not used by the effector interface, so that the new action *mmmm* of classifier A' will still produce the same effector actions (if it is indeed an effector activating message) as its parent, A. However, since the string *mmmm* also contains a random sub-section it is unlikely that the message produced by the parent A will match that condition of B'.

There are a number of parameters that control the CSS operator:

1. *CSSRt.* The CSS trigger condition is tested with this probability every cycle.

2. *CSSFrM.* This specifies the fraction of coupled pairs with linkage message forced to have a "source tag" (left-most 2 loci) of "11" (i.e., an "internal memory" type message is used). Otherwise the action "source tag" is just copied from "stage setter" parent's action.

3. *CSSFrRf.* This specifies the fraction of new times the "successor" has both conditions satisfied (equal to) the action of its "stage setter". Otherwise one condition is copied from the successor parent and one is equal to the action of the stage setter.

4. *TagLMx* and *TagRMx.* The random "tag" is inserted into linking string between TagLMx and TagRMx. Typically TagLMx is 2, right after the "source tag" region (00 for detector, 10 effector, etc.), and TagRMx is some value that ensures the tag region does not overlap the loci used by effectors to specify actions.

5. *NwCCStrF.* This controls the strength assigned to new rules produced by the CSS operator. If NwCCStrF is -1.0, the average strength of the parents is used. If it is 0.0 (the default), the strength is assigned just as it is for the GA (i.e., as controlled by NwCfStrV and NmCfStrF). For other values of NwCCStrF, the new strength is just the average classifier strength times NwCCStrF itself.

6. *ACPCt.* This controls the "profit" threshold that is used to trigger the CSS operator. In particular, the operator is triggered when the "successor" classifier makes a "profit" (bid + tax + SysRew + payments to it for messages) that is greater than the classifiers strength times the value of ACPCt. For example, if ACPCt is 0, the classifier just has to make a profit greater than 0.

3.9.5 The Low-Bidders Operator

CFS-C has Yet Another Kludge which may be useful for solving one type of "convergence" problem. This problem arises when some rule is doing relatively poorly in a given situation, and when there are lots of rules doing well in lots of other situations. In this case the poor rule has relatively low strength, so it doesn't get manipulated by the GA, so no competitors for it are generated and evaluated, so the poor rule continues to control the behavior. If overall convergence is controlled, the rule will survive indefinitely.

So...a new rule discovery operator is triggered when the average bid during a step falls below a threshold, which is a product of an estimate of the overall average bid (i.e., Average-Strength * bid_k * Average-BidRatio) times a threshold parameter, *TLBt* ("Trigger on Low Bid threshold").

When triggered, this operator takes as a parent some classifier that made a low bid, and then copies it and changes the action part to some random string. The idea is that low bids may be a result of conditions that are attending to the right messages, but actions that are wrong. This is not the only reason a rule could have a low strength (and so low bid), however—e.g., it could have overally general conditions. So this operator is not really all it should be.

At any rate, there are three parameters that control it:

- *TLBt.* The threshold, as described above.
- *TLBRt.* The trigger condition is tested with this probability every step.
- *TLBFrE.* The fraction of the random actions that should be forced to be effector activating message producers (i.e., start 10 in the action part's left loci).

Also, the variable *TLBTot* counts the number of applications of this operator.

I have only tried this in a few prelimary tests, so the jury is still out. I do think this operator or one like it will prove itself a useful addition.

4 How to Define an Environment for the CFS-C System

This section describes how to implement a specific task-domain or environment for the CFS-C classifier system to operate in. Since the CFS-C system is intended to be run as part of a self-contained model of a classifier system and its environment, implementing an environment for the CFS-C system means defining:

1. The environment state-space itself, and

2. Several subroutines that act as interfaces between the CFS-C classifier system and the environment.

The subroutines that must be defined are all called by subroutines in the CFS-C package, so they must be linked with the subroutines from the CFS-C system to form a program that can actually be run. (A standard C library of subroutines must also be linked to the system, to define the basic C subroutines like fopen(), getc(), and so on.)

In particular, the following subroutines must be implemented:

1. GetDMsgs(): Get Detector Messages.

   This subroutine acts as the input-interface from the environment to the classifier system, i.e., the set of detectors that generate messages that provide the classifier system with information about about the current state of the environment.

2. Effector Functions.

   A CFS-C domain must provide one or more effectors, to provide the system with a way to affect its environment. Each effector must respond to certain messages on the message list by performing some action that changes the state of the environment. There must three subroutines defined for every effector, one to map messages that match the effector into "support" for various actions, one to choose an action based on the support collected from all messages, and one to determine which messages are consistent with the action actually selected and which are not.

3. DoBkgBeh(): Do Background Behavior.

   This subroutine provides a way to implement dynamic environments, i.e., environments in which the state changes independently of actions taken by the system's effectors.

4. GtSysRew(): Get System Reward.

   This subroutine implements a payoff function for the classifier system, i.e., a function that provides the classifier system with rewards when certain environment-states are attained. The payoff function basically defines the goals for the system.

   This subroutine should also want to set the value of the unsigned int variable *MadeMstk* to 0 or 1, to indicate when the system is judged to have made a "mistake," if such a judgement is possible (0 means no mistake, and should be the default). This variable is used to trigger the Cover Effectors Operator (see below). So at the least, it is wise to set this to 1 when no effector has been activated during a given step (or recently).

5. LoadEnv() and ReadEnv(): Load/Read the Environment.

   These two subroutines are called to load the initial state of an environment from a file or device. Initial values for variables that control the payoff function and the condition-parts of the effectors are also loaded.

6. DsplEnv() and WrtEnv(): Display/Write the Environment.

   These two subroutines are called to write the current state of the environment to a file or device. Current values for variables that control the payoff function and the effector condition-parts are also displayed.

7. DisCfIE(): Display classifiers in an "interpreted" form.

   This subroutine is called when the user displays the classifier list with a format code of 21 to 29. Given a classifier, the subroutine must return in a buffer a display of what that classifier does as "interpreted" for the given domain. While the subroutine can be defined to do nothing for your domains, I would suggest using it: it soon gets tedious looking at long strings of the $\{0, 1, \#\}$ alphabet and trying to determine what they signify.

8. DoEnvCmd(): Do Environment Command.

   This subroutine provides an easy way to invoke environment-specific utility commands, e.g., commands that can be used to debug the environment-dependent subroutines.

The rest of this section describes the implementation requirements for these subroutines and for the environment state-space itself in more detail. The LETSEQ system, described in [Riolo, 1988b] was implemented using the CFS-C package. Thus the source-code files for the LETSEQ system contain examples of all the subroutines described in this section. Although that task-domain is very simple, it does show how to implement a specific environment for the CFS-C system. The FSW1 domain [Riolo, 1988c], also distributed with CFS-C, is another example of how to implement the domain-specific subroutines that must be linked to CFS-C.

4.1 Defining the Environment State-Space

The environment state-space may be defined in whatever way seems best for the particular environment or task-domain being implemented.

For example, in the LETSEQ system, the task for the classifier system is to learn to predict what letter it will encounter next, based on the sequence of letters it has already seen. Because the LETSEQ task domain is so simple, the code that defines the LETSEQ environment is very simple: its just a circular list of letters, a pointer that indicates what letter was just read (and therefore what letter will be next), and a variable to store the system's prediction about what letter will come next.

4.2 Defining the Detectors

The GetDMsgs() subroutine ("Get Detector Messages") acts as the input interface from the environment to the CFS-C classifier system. The specification for the GetDMsgs() subroutine is:

> **int GetDMsgs (MStrings, MIntens)**
>
> > **char MStrings[]; float MIntens[];**

where **MStrings[]** is a character array long enough to hold DMSGSSZ message strings, each STRNGSZ long, stored one after the other; and **MIntens[]** is a float array with DMSGSZ entries, i.e., large enough for one entry for each message string in the MStrings[] array. This subroutine returns the number of messages placed in MStrings.

GetDMsgs() should examine the current environment-state and decide which, if any, detector messages should be added to the message list. Those messages should of course consist of all 0's and 1's, and should be STRNGSZ long. The messages to be added should be placed one after another in the MStrings[] array, with no NULL characters between each message.

For each detector message placed in the MStrings[] array, an intensity value should be placed in the array MIntens[]. In particular, the intensity of the first detector message should be in MIntens[0], the intensity for the next in MIntens[1], and so on. On entry to the GetDMsgs() function the value of the system variable MsgIntDf ("message intensity default") is stored in each entry of the MIntens[] array; therefore, if no values are assigned to the MIntens[] entries, detector messages will all have intensities equal to the value of the MsgIntDf variable.

GetDMsgs() can use any rules that seem appropriate to map environment states onto detector message strings. However, usually it is a good idea to have some portion of each message string act as a *tag* field that indicates which messages are from detectors and which are not. For example, each detector message could have "00" as the value of its two left-most loci.

GetDMsgs() also can return any values it chooses as intensity values for the detector messages. However, it is a good idea not to use negative intensities: intensities are summed to give support values for classifiers that match messages, so that negative intensities could lead to negative support values!

GetDMsgs() is called from the CFS-C major-cycle by the RdDetect() subroutine (see the file CORE.C). RdDetect() actually adds the messages returned by GetDMsgs() to the system's current message list. RdDetect() also sets the system variable NmDetMsg ("number of detector messages") to be equal to the number of detector messages actually added to the list.

GetDMsgs() is called only every DetectRt steps, where DetectRt ("detector rate") is a variable that can be modified at run time. The default for DetectRt is 1, i.e., call GetDMsgs() every step. However, in domains where it is not appropriate to read detector messages every major-cycle step, DetectRt can be set to a value greater than 1.

While the compiled maximum number of detector messages is DMSGSSZ, there is a system variable, NmDMsgMx, which can be set during a run and which is the actual maximum number of detector messages that will be added to the message list. That is, if GetDMsgs() returns more than NmDMsgMx messages in MStrings[], a warning message will be printed and only the first NmDMsgMx messages will be added to the message list.

Note that the variable DetMsgMn is a lower bound for NmDMsgMx; that is, if a user tries to set NmDMsgMx directly (or indirectly, by setting NmIMsgMx) to a value less than DetMsgMn, a warning message is printed. For DetMsgMn to be a useful lower bound, it should be set once, when an environment is initialized, to be the maximum number of detector messages the GetDMsgs() subroutine will generate in one major-cycle step.

4.3 Defining Effectors

A CFS-C classifier system implemented for a particular task-domain may have one or many different effectors. Each effector consists of four basic parts:

1. A *condition-part* that defines which messages match the effector. Effector condition-parts are just like classifier condition-parts, (i.e., it consists of an "m" or "$\sim$" followed by STRNGSZ symbols from the $\{0, 1, \#\}$ alphabet) except that effector condition-parts consist of just one condition string. By convention all effector conditions start with "10", to distinguish them from detector messages, which start with "00" and other internal messages from classifier to classifier, which start "11" or "01".

2. A *support–collection subroutine,* **Eff_Set**, which collects "support" for each particular action an effector can take. When an effector's condition part is satisfied by a particular message, the associated support–collection subroutine is called determine what action (or actions) that message advocates (as determined by some of its bits); the intensity associated with that message is then added to the "support" for that action.

3. A *action–selection subroutine,* **Eff_Fn**, which selects a specific action for the effector to take, given the support collected. This is called after all new messages have been processed by the effector-support subroutine. Basically, when there are conflicts, the effector value r is chosen probabilistically, with the probability that $r = r'$ equal to:

$$\sum_{m'} \beta_{m'}(t) / \sum_{m} \beta_m(t)$$

   where m' ranges over the effector messages that say "set r to r'", $\beta_{m'}(t)$ is the effective bid of the classifier that posted message m' (i.e., it is the intensity associated with that message), m ranges over all effector messages, and $\beta_m(t)$ is the effective bid of the classifier that posted message m. The winning r value is used to determine what the system's effector will do.

4. A *consistency-check subroutine,* **EqEfAct**, which determines for each message that matches an effector whether or not that message supported the action actually selected. This subroutine takes as a parameter a message that matched the effector, and returns TRUE if the action(s) advocated by that message is consistent with the action actually selected, or FALSE otherwise. The CFS-C system then deletes each message from the new message list that is not-consistent with the action taken. This

is done so that the contents of the message list available to classifiers during the next major-cycle indicate unamiguously which effectors, if any, were activated, and what messages were used to activate them.

Effector conditions are loaded and associated with the appropriate effector subroutines when the system is run (see Section 4.7, below).

The specification for the first subroutine is:

int Eff_Set (MsgStrng, Support)

char MsgStrng[]; float Support;

where **MsgStrng[]** is character buffer of length STRNGSZ that contains the message string (0's and 1's) that matched and activated the effector and suggest an action or actions; **Support** is the support for the specified action, typically the effective bid made by the classifier that posted the message. This subroutine must examine the message to determine what actions are being recommended, and then add the support specified to the total support for those actions.

The specification for the section subroutine is:

int Eff_ Fn ( )

This subroutine should select an action from the supported actions, typically chosing probabilistically proportional to support. One an action is selected, it should change the environment in the appropriate way.

For example, suppose the classifier system is meant to represent an animal trying to live in a two-dimensional space. One effector may be defined to change the speed at which the animal is moving to a new speed that depends on the content of the message passed to it. For example, the the last 4 bits of the message passed to the effector could be interpreted as the speed the effector is to establish as the animals new speed.

In general, effectors can also be used to set system variables, not just the environment-specific variables. For example, an effector could be defined that increases or decreases the proportion of the message list reserved for detector messages, allowing the system to control how much attention it is paying to information from the environment. Of course care must be used when implementing effectors that change CFS-C system variables.

The specification for the third subroutine is:

int EqEfAct (MsgStrng)

char MsgStrng[];

where **MsgStrng[]** is character buffer of length STRNGSZ that contains the message string that matched and the effector. The subroutine returns TRUE if the action specified by the message is consistent with the actual action selected by Eff_ Fn, otherwise FALSE.

Note that effectors are activated only every EffectRt steps, where EffectRt ("effector rate") is a system variable that can be modified at run time. The default for EffectRt is 1, i.e., activate matched effectors every step. However, in domains where it is not appropriate to activate effectors every major-cycle step, EffectRt can be set to a value greater than 1.

4.4 DoBkgBeh(): Do Background Behavior

In some domains the state of the environment is changed only when the classifier system activates an effector to make the change. In other domains the state of the environment should be able to change even when the classifier system does not activate an effector. For example, if the environment is a two-dimensional space filled with objects, some of those objects may be moving on their own, e.g., if they are predator or prey objects.

The DoBkgBeh() subroutine can be defined to implement environmental change that is independent of effector actions. DoBkgBeh() is called once every major-cycle step by the GenBehav() subroutine after effectors are checked and (possibly) activated (see the file CORE.C). Note that DoBkgBeh() is called every step no matter what value is assigned to EffectRt (the effector rate).

The specification for DoBkgBeh() is:

int DoBkgBeh()

The integer value returned has no significance. DoBkgBeh() may be defined to change the state of the environment in any way that seems appropriate to a particular environment, or it may be defined to do nothing. However, DoBkgBeh() must be defined as part of the environment definition code for all task-domains linked to the CFS-C classifier system, whether it does anything or not.

4.5 Defining the Payoff Function

The GtSysRew() subroutine ("Get System Reward") acts as the "payoff function" for the CFS-C classifier system. Once every major-cycle step, after all changes to the environment-state have been made for that step (i.e., after effectors have been activated and after the DoBkgBeh() subroutine has been executed), GetSysRew() is called to get the reward (the payoff) from the environment.

The specification for GtSysRew() is:

int GtSysRew ()

where the integer returned has no significance. GtSysRew() must set the system variable *SysRew* ("system reward") every time GtSysRew() is called. The value assigned to SysRew should depend on the state of the the environment, so that GtSysRew() will implement the desired payoff function for the environment or task-domain being defined. A value of 0.0 should be assigned to SysRew for non-rewarding states.

GtSysRew() should also have a **side effect** of setting the unsigned int variable *MadeMstk* to 0 (the default) or 1. Setting it to 1 acts as a triggered for the Cover Effector Operator, so it should only be set to one when it is appropriate to generate a new classifier with a random action, e.g., when no effector activating classifier has been activated recently or when a "mistake" has been made. (Of course for some domains there is no natural definition of what a mistake is, while for some there is.)

4.6 Displaying and Saving the Environment

The CFS-C system calls two subroutines, DsplEnv() and WrtEnv(), to display the state of the classifier system environment.

The specification for the DsplEnv() subroutine is:

int DsplEnv (FDName, Format)

char *FDName; int Format;

where **FDName** is a null-terminated character string that specifies where the environment state is to be displayed; and **Format** is an integer that indicates how the environment-state is to be displayed.

DsplEnv() is called when the user enters a DISPLAY ENV command. DsplEnv() should do the following:

1. Open the specified file or device, FDName;

2. Call WrtEnv() to display the environment in the specified format; and

3. Close the file or device.

If FDName points to a null-string, then DsplEnv() should use the default file/device used by all DISPLAY processors. (The name of the default device is stored in the system variable DisOutFN[] and a FILE pointer to it is stored in the system variable DisOFILE.) Note that the default file/device pointed to by DisOFILE is always open, so DsplEnv() should not try to open or close that file/device.

The specification for the WrtEnv() subroutine is:

int WrtEnv (FilePtr, Format)

FILE *FilePtr; int Format;

where **FilePtr** is a pointer to a FILE structure for an already opened file or device on which the environment-state is to be displayed; and **Format** is an integer that indicates how the environment-state is to be displayed.

As mentioned above, WrtEnv() is the subroutine that actually displays the state of the environment. For example, WrtEnv() is called when a user enters a DISPLAY ENV command. Typically WrtEnv() is used to display the state of the environment in some "line-by-line" format similar to the default formats provided by the CFS-C system to display classifiers and messages.

While the WrtEnv() subroutine can display the state of the environment in any format that seems appropriate, there are some conventions that should be respected in WrtEnv():

1. Typical Format parameter values are 0, 1, 2, and so on. Higher Format values mean more information should be displayed. A Format of 0 means don't display anything.

2. The file UTILITY.H defines some macro-names associated with some selected format numbers (e.g., SAVEFMT is defined to be 99). You might want to use these macro-names in WrtEnv() to keep the environment code consistent with the CFS-C code.

3. If the user does not specify a Format number on a DISPLAY ENV command, the Format value passed to DsplEnv() is 1.

4. If the Format number is SAVEFMT (i.e., 99), the state of the environment and effector conditions should be written in a format that can be read by the LoadEnv() subroutine described in Section 4.7.

5. When the Format is SAVEFMT, WrtEnv() should use the standard C fprintf() subroutine. The fprintf() subroutine should be used so that if logging is being done and the system then is saved, the saved information will not be written into the log!

6. When the Format parameter value is not SAVEFMT, WrtEnv() should call the CFS-C system subroutine WriteS() to actually write values to the specified file/device. WriteS(), which is defined in the file CFSIO.C, should be used so that the log facility provided as part of the CFS-C system will work.

Note well that the CFS-C subroutine SaveSys() also calls WrtEnv() when the classifier system state, including the state of the environment, is to be saved to a file. In particular, SaveSys() calls WrtEnv() with a Format value of SAVEFMT. (SaveSys() is defined in the file RUNCFS.C).

The CFS-C classifier system includes one subroutine, WrtEffs(), that usually should be used to write the effectors when the Format is SAVEFMT. WrtEffs() may be called as follows:

WrtEffs(FilePtr);

where FilePtr is the FilePtr parameter passed to WrtEnv(), i.e., it is a pointer to a FILE structure for the (open) file/device to which the information is to be written. The reason to use WrtEffs() is that it writes out all the effectors in a format that can be read by the ReadEffs() subroutine, another CFS-C subroutine that is described in Section 4.7, below.

4.7 Loading the Environment

The CFS-C system calls two subroutines, LoadEnv() and ReadEnv(), to read information from a specified file or device and use that information to initialize both (a) the state of the environment linked to the system and (b) the effectors defined for that environment.

The specification for the LoadEnv() subroutine is as follows:

int LoadEnv (FDName)

char *FDName;

where **FDName** is a pointer to a null-terminated character string buffer that indicates where the environment-state is to be loaded from. LoadEnv() is called when the user enters a LOADENV ("Load Environment") command. LoadEnv() is also called from the Init_CFS() subroutine when the classifier system if run using separate files for initializing the message list, the classifier list, and the state of the environment. (See the file RUNCFS.C for the definition of the Init_CFS() subroutine.)

LoadEnv() should do the following:

1. Open the specified file or device, FDName;

2. Call ReadEnv() to actually read in the information that defines the initial state of the environment; and

3. Close the file or device.

If FDName points to a null-string, then LoadEnv() should use the default file/device which is specified by the system variable "EnvInFN".

The specification for the ReadEnv() subroutine is as follows:

int ReadEnv (FilePtr)

FILE *FilePtr;

where **FilePtr** is a pointer to a FILE structure for an already open file or device from which the environment state is to be loaded. As mentioned above, ReadEnv() is called from the LoadEnv() subroutine. ReadEnv() is also called from the LoadSys() subroutine, when the the system is run using a single initialization file for loading system variables, messages, classifiers, and the environment. (LoadSys() is defined and called in the file RUNCFS.C.)

ReadEnv() must do three general tasks:

1. Initialize any data structures, state variables, etc., that are part of the definition of an environment itself. This step is completely environment-dependent, so not much will be said about it here. Note that besides reading initial values for the environment state-space, it is usually a good idea to read values for any variables that define the payoff function for the environment.

2. Assign a value to the DetMsgMn variable. This value should be equal to the maximum number of detector messages the GetDMsgs() subroutine can generate in one major-cycle step. This value is used as a lower bound on the NmDMsgMx value (see Section 4.2).

3. Initialize the effector list data structure that is defined as part of the CFS-C system. This step is relatively environment-independent, so it will be described here in some detail.

While the ReadEnv() subroutine can be defined to read in information that is layed out in almost any format, the following conventions usually should be observed:

1. The ReadS() subroutine defined in the file CFSIO.C should be called to read lines from the specified file.

2. Lines that begin with a semicolon (";") in column 1 should be ignored, i.e., they should be treated as comment lines.

3. The values that define the environment state should be read first.

4. The values that define the payoff function should be read next.

5. The values that define the effector-list should be read last.

The CFS-C effectors are stored in the EffLst ("effector-list"), which is actually an array of structures, one structure for each effector. (The EffLst[] array is defined in the file CORE.DEF). Each effector structure contains (amoung other things):

1. The effector's condition-string and condition-type;

2. The effector's name;

3. A variable to count the total number of messages that have matched the effector's condition during a run;

4. A variable to count the total number of times the effector has been activated during a run; and

5. The address of the subroutine that implements the effector.

The easiest way to initialize the effector-list is to:

1. Store the addresses of the subroutines that implement the effectors in structures in the EffLst[] array; and

2. Call the ReadEffs() subroutine to read in the effector conditions, condition types, and so on, and store those values in the appropriate structures in the EffLst[] array.

In particular, the following code-fragment shows how to store the addresses of two subroutines, named Eff1() and Eff2(), in the first two structures in the effector list:

```
extern int Eff1(), Eff2(); /* Eff1, Eff2 addressable */
extern struct EffNode EffLst[]; /* The effector 'list' */

EffLst[0].Eff_ Fn = Eff1(); /* Store the address of Eff1 */
EffLst[1].Eff_ Fn = Eff2(); /* Store the address of Eff2 */
```

The ReadEffs() subroutine, which is defined in the file RUNCFS.C, may be called as follows:

ReadEffs(FilePtr);

where FilePtr is a FILE pointer to the file or device from which the environment is being read, i.e., it should be the FilePtr value passed to the ReadEnv() subroutine.

The ReadEffs() subroutine expects to read lines layed out as follows:

$effector - condition, effector - name, TotEMtch, TotEAct$

.

. and so on, one line for each effector.

.

END-EFFS

where

$effector - condition$ is the effector's condition-part, which is just like a classifier condition, i.e., a string of length STRNGSZ from the $1, 0, \#$ alphabet. It also may be prefixed by an "m" or an "~", for match and not-match conditions, respectively.

$effector - name$ is an optional name for the effector (up to 16 characters).

$TotEMtch$ is an integer number that is the number of times the effector has been matched so far during a run. The default for this optional value is 0.

$TotEAct$ is an integer number that is the number of times the effector has been activated so far during a run. The default for this optional value is 0.

The $effector - name$, $TotEMtch$, and $TotEAct$ values are optional; these values included here because they are written by the WrtEffs() subroutine, so the ReadEffs() subroutine must be able to read them.

The ReadEffs() subroutine reads each effector condition, name, etc., from the specified file and stores them in the next available effector structure. The ReadEffs() subroutine stops reading lines when it encounters the line with "END-EFFS" on it. When the ReadEffs() subroutine is done, it returns the number of effectors actually read and stored. This value should be stored in the system variable NmEffs by the subroutine that called ReadEffs().

For example, suppose the file pointed to by FilePtr contained:

$$m10\#\#\#\#\#\#\#\#\#\#\#\#\#\#, Eff1, 0, 0$$
$$\sim 10\#\#\#\#\#\#\#\#\#\#\#\#\#\#, Eff2, 0, 0$$

```
END-EFFS
```

and the addresses of the effector subroutines Eff1() and Eff2() have been stored in the first two structures of the EffLst[] array (as shown earlier). After the call to ReadEffs(FilePtr), the condition associated with the Eff1() subroutine would be the "$m10\#\#\#\#\#\#\#\#\#\#\#\#\#\#$", and the condition associated with Eff2() would be "$\sim 10\#\#\#\#\#\#\#\#\#\#\#\#\#\#$".

The ReadEffs() subroutine is designed to read information that has been written by the WrtEffs() subroutine described in Section 4.6. Thus if the WrtEffs() subroutine is used to write the effectors when the system is saved (i.e., the Format value passed to DsplEnv() is SAVEFMT), the ReadEnv() subroutine can use the ReadEffs() subroutine to load the effectors from saved files.

While any string that is a valid classifier condition can be an effector condition, it is usually a good idea to include a special tag in all effector conditions. For example, all effector conditions could start (in their left-most two loci) with the value "10". Then only messages that begin with "10" will activate effectors.

It is a good idea to make effector conditions mutually exclusive, i.e., satisfied by mutually exclusive subsets of messages. If effector conditions are not mutually exclusive, one message may match two effectors. In many cases this will not cause a problem. However, there will be a problem if one message matches two effectors and if one of those effectors is also matched by some other message(s). In that case, the message that matches two effectors may be removed from the message list as a result of the conflict-resolution that occurs when an effector is matched by more that one message. The other effector matched by that message will then be activated by a message that isn't on the message list anymore!

4.8 DoEnvCmd(): Do Environment Specific Commands

The code for every environment linked to the CFS-C system must define a special utility subroutine, DoEnvCmd().

The specification for this subroutine is:

int DoEnvCmd (Param)

char *Param;

where **Param** points to a null-terminated character string that contains any parameters entered by the user.

The DoEnvCmd() subroutine is called from the command-processing loop when a user enters the ECMD command (see the file RUNCFS.C). The subroutine is passed the address of a character string buffer that contains anything entered on the line with the ECMD command (not including the "ECMD" itself). Note that any alphabetic characters in the parameter are converted to lowercase.

The DoEnvCmd() subroutine is intended to be a way to execute a variety of domain-dependent commands, all with just one entry point from the CFS-C system. That is, typically the parameter on an ECMD command is itself some domain-dependent command, e.g., a command to display some environment-specific variable, to set a environment-specific variable to a new value, and so on. The DoEnvCmd() subroutine can also be used to execute domain-dependent debugging features, e.g., to provide a way for the user to execute an effector directly, to alter the environment state "by hand," and so on.

Of course the DoEnvCmd() can be defined to do nothing. However, if it is used, one of the first features you might want to implement is a environment-specific HELP command, i.e., a way for a user to get information about environment-specific variables, comamnds, and so on.

5 Future Research

While classifier systems have been used successfully in a variety of settings (*cf.* [Wilson, 1986c], [Robertson and Riolo, 1988], [Sannier and Goodman, 1988]), a number of problems have surfaced over the past couple of years. Some of the most important problem areas are:

- Premature Convergence. The problem of the "premature convergence" of a population manipulated by a genetic algorthm (i.e., the loss of variation in many or most loci before a satisfactory solution is found) has been known for some time [DeJong, 1975]. This phenomena has also shown up in classifier systems that use genetic algorithms ([Robertson and Riolo, 1988], [Wilson, 1988]). One cause of convergence is the high fixed-point strengths of "generalist" rules, i.e., rules with low bid ratios (specifity). Convergence also results from a kind of "founder effect," i.e., when early in a run a rule is discovered that is much better than all the rest. In both cases, rules that have relatively high strength proliferate, causing the loss of all other rules. This happens even when the other rules are useful in situtations not covered by the high strength rules, e.g., when there are many subproblems to be solved, each by different groups of rules.

- Stability of Chains. While chains can emerge and serve to increase performance on simple problems ([Riolo, 1987a], [Robertson and Riolo, 1988], [Riolo, 1988d]), chains longer than 4 or 5 rules are not very stable. Several causes of the instability have been identified, including: (a) the effects of taxes on the fixed-point strengths of rules at the beginning of chains; (b) rules that act as "parasites," i.e., are active and draw strength from useful rules, but that do not themselves contribute to the system's performance; and (c) the failure to complete execution of chains when there are many competing chains. Since chains of couple classifiers are required for carrying out arbitrary computations and for implementing short term memory, ways to maintain chains must be found.

- Multiple Roles of Strength. Classifiers have one value, strength, that really serves (at least) three roles: (1) it acts as a kind of *capital* that allows rules to make mistakes; (2) it acts to control the *immediate behavior* of the system, through the bid competition; and (3) it acts to control the *long term learning* (and behavior) through its use as the fitness measure for the genetic algorithm. Also note that in some domains strength serves as a measure of the system's ability to reach specified goals, while in other it serves as a measure of predictive power [Holland, 1986c]. The problem is that these roles are sometimes in conflict, leading to lower than expected performance ([Riolo, 1987b], [Riolo, 1988d]) or to difficulties when working in domains where it would be useful for a system to build and utilize models that are both *predictive* and *prescriptive* [Booker, 1988] [Burks and Holland, 1988].

- Emergence of Co-Adapted Sets of Classifiers. CFS-C is based on the *Michigan approach* to classifier systems using the genetic algorithm, i.e., each rule is treated as an individual in the population being manipulated by the GA, and the full population is treated as a performance system. A second approach is the *Pitt approach*, in which each "individual" consists of a a *set* of rules, each of which is evaluated as a complete performance system. One advantage of the Pitt approach is that the the "schema theorem" [Holland, 1976] rather naturally leads to the emergence of co-adapted sets of rules in the better members of the population of performance systems [DeJong, 1988].

Its not so clear that co-adapted sets of rules will emerge as readily in in systems (like CFS-C) based on the Michigan approach. It may be necessary to modify the basic genetic algorithm(and/or the bucket brigade algorithm) to faciliate the formation of co-adapted sets of rules that implement "subroutines", portions of "semantic nets", and so on. Further, it is not clear whether using the bucket brigade algorithm to allocate strength, along with the appliation of the genetic algorithm and other special "triggered" rule discovery mechanisms, results in an overall system that satisfies the schema theorem. Until such a formal result is obtained, there is no guarentee that the genetic algorithm is being used to its full potential.

- Limited Message List Size. The competition to post messages to a limited message list is designed to allow higher strength rules to control the system more often than lower strength ones; since high strength rules are presumed to lead to higher payoffs, this mechanism in effect allows the system to focus activity in those rules that tend to lead to higher payoff.

  One problem with limiting the message list occurs when some classifiers are posting effector activating messages while other are posting messages to be used by other classifiers in chains [Riolo, 1988d]. If there are many classifiers of one type or another, or if rules of one type has generally higher strengths than rules of the other type, then the message list may be filled by messages of one type. In such a case the system may fail to take an appropriate action or it may disrupt messages passing down coupled sequences of rules and so lead to poor performance a few steps in the future. This problem can also be exacerbated when some overly general classifiers try to post many message during a given step, or when there are many classifiers that are "genotypically" different by "phenotypically" the same (i.e., they produce the same messages in response to the same situations).

  Another problem with limiting the message list is that it limits the number of rules that are evaluated on a given step [Wilson, 1988]. And in more general terms, limiting the message list limits the overall parallelism of classifier systems, both in the performance part of the system and in implicitly parallel genetic algorithm. It may be useful to think about the effects of using unlimited (or very large) message lists, both on the expected properties of such a system and on the implementation of such a system using both serial machines (using software like CFS-C) and parallel hardware (like a Connection Machine).

- "Representational" Power of Individual Classifiers Standard classifiers are very simple productions, with a fixed "syntax" and limited processing capabilities. Some of advanatges of using simple classifiers is that (a) the rules can be implemented in parallel hardware, (b) the rules are well suited for use with the genetic algorithm, and (c) the system can be studied formally. The problem with simple rules is that it takes a lot of rules, sometimes coupled in very complex and brittle structures, to do higher-level computations ([Forrest, 1985], [Sannier and Goodman, 1988], [Davis and Young, 1988], [Schuurmans and Schaeffer, 1988]).

Two approaches that could be taken to this dilemma are: (1) to use classifier systems with many, many rules, with the goal being to have higher order processing emerge (an approach similar to that take by people using Connectionist networks); or (2) increase the processing power of individual classifiers, trying to maintain the advantages of simpler rules (e.g., still be able to implement them in parallel).

A related issue is that of "distributed representation" in classifier systems. It seems to me that one of the arguments for using classifier systems has always been that knowledge would be distributed in ways that avoid "brittleness," e.g., so that removing or contradicting one or a few rules would not lead to disastorous drops in performance. I think attaining this end will require careful thought about how the interface (i.e., the detectors and effectors) to a classifier system is implemented. For example, I think the LETSEQ1 representation is a poor one for attaining distributed knowledge representation: the rules are just too coarse grained—each is on the level of a "concept" or "symbol" or stand-alone rule for behavior, rather than having many rules participate in making each prediction.

I plan to work on some of these problems in the next couple of years. One approach will be to make relatively minor changes to the CFS-C version 1.0 system. For example, it may be possible to alter (or eliminate) some of the tax mechanisms now used and to implement a new rule replacement algorithm, to make chains more stable. Similarly, it may be possible to make minor alterations to the rule discovery mechanisms to control the premature convergence problems.

A second approach I plan to take is to consider more fundemental changes of the basic classifier system architecture. For example, it may be useful to have several different "strengths" associated with each rule, for the different roles that the single strength value now serves. Similarly, it may be useful to alter the bid competition mechanism and the bucket brigade algorithm to facilitate the emergence and performance of default hierarchies. These changes would result in CFS-C 2.0 (or 3.0 or whatever).

A Bibliography

References

[Booker, 1982] Booker, Lashon B. "Intelligent Behavior as an Adaption to the Task Environment." Ph.D. Dissertation. University of Michigan (1982).

[Booker, 1988] Booker, Lashon B. "Classifier Systems that Learn Empirical World Models." Submitted to *Machine Learning*, 1988.

[Booker *et al.*, 1987] Booker, Lashon B., David E. Goldberg and John H. Holland. *Classifiers and Genetic Algorithms.* Technical Report No. 8, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, Ann Arbor, MI (1987).

[Burks, 1986a] Burks, Arthur W. "The Logic of Evolution, and the Reduction of Holistic-Coherent Systems to Hierarchical-Feedback Systems." In *Causation in Decision, Belief Change and Statistics*, William Harper and Bryan Skyrms (Eds). Kluwer Academic Publishers, Dordrecht, Holland (1986).

[Burks, 1986b] Burks, Arthur W. "A Radically Non-Von Neumann Architecture For Learning and Discovery." In *CONPAR 86: Proceedings of a Conference on Algorithms for Parallel Processing,* pp 1–17, Handler, Wolfgang (Ed). Aachen, F.R. Germany, 17-19 September 1986. Springer-Verlag, Berlin.

[Burks and Holland, 1988] Burks, Arthur and Holland John. "Classifier Systems: Parallel Machine Learning with Higher-Level Building Blocks." Draft of proposal to NSF (1988).

[Davis and Young, 1988] "Classifier Systems with Hamming Weights." Davis, Lawarence, and Young, David K. In *Proceedings of the Fifth International Conference on Machine Learning,* pp 162–173. San Mateo, CA: Morgan Kaufman (1988).

[DeJong, 1975] DeJong, Kenneth. A. "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." Ph.D. Dissertation, University of Michigan (1975).

[DeJong, 1988] DeJong, Kenneth. A. "Using Genetic Algorithms to Learn Task Programs: the Pitt Approach." *Machine Learning, 3*, 2-3 (1988).

[Forrest, 1985] Forrest, Stephanie. "A Study of Parallelism in the Classifier System and Its Application to Classification in KL-ONE Sematic Networks." Ph.D. Dissertation, University of Michigan (1985).

[Goldberg, 1983] "Computer-Aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning." Ph.D. Dissertation, University of Michigan (1983).

[Goldberg, 1988] Goldberg, David E. "Genetic Algorithms in Search, Optimization, and Machine Learning." Addison Wesley, Reading MA (1988).

[Goldberg and Richardson, 1987] Goldberg, David E. and Richardson, Jon. "Genetic Algorithms with Sharing for Multimodal Function Optimization." In *Proceedings of the Second International Conference on Genetic Algorithms and their Applications,* 41-49. John J. Grefenstette (Ed.). Cambridge, Massachesetts, July 28-31, 1987.

[Grefenstette, 1986] Grefenstette, John J. "Optimization of Control Parameters for Genetic Algorithms." *IEEE Transactions of Systems, Man and Cybernetics,* Vol. SMC-16, number 1 (1986).

[Grefenstette, 1987] Grefenstette, John J. "Multilevel Credit Assignment in a Genetic Learning System." In *Proceedings of the Second International Conference on Genetic Algorithms and their Applications,* 202-209. John J. Grefenstette (Ed.). Cambridge, Massachesetts, July 28-31, 1987.

[Holland, 1976] Holland, John H. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, MI (1976).

[Holland,1984] Holland, John H. "Genetic Algorithms and Adaptation." In *Adaptive Control of Ill-Defined Systems*, p317-333. Selfridge, O. G., E. L. Rissland, and M. A. Arbib (Eds.), Plenum Press, New York, 1984.

[Holland,1985] Holland, J. H. "Properties of the Bucket Brigade." *Proceedings of an International Conference on Genetic Algorithms and their Applications,* 1-7. John J. Grefenstette (Ed.). Carnegie-Mellon University, Pittsburg, 1985.

[Holland, 1986a] Holland, John H. "Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems." In *Machine Learning: An Artificial Intelligence Approach, Volume II*, Michalski, Ryszard S., Carbonell, Jamie G., and Mitchell, Tom M. (Eds). Morgan Kaufman Publishers, Inc., Los Altos, CA (1986).

[Holland, 1986b] Holland, John H. "A Mathematical Framework for Studying Learning in Classifier Systems." *Physica 22D,* 307-317 (1986).

[Holland, 1986c] Holland, John H., Holyoak, Keith J., Nisbett, Richard E. and Paul A. Thagard. *Induction. Processes of Inference, Learning, and Discovery.* The MIT Press, Cambridge, MA, 1986.

[Holland and Reitman, 1978] Holland, John, and Reitman, J. S. "Cognitive Systems Based on Adaptive Algorithms." In *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F, (Eds.), Academic Press, 1978.

[Holland and Burks, 1987] Holland, John H. and Burks, Arthur W. "Adaptive Computing Sustem Capable of Learning and Discovery." United States Patent 4,697,242. September 29, 1987.

[Mitchell *et al*, 1983] Tom M. Mitchell, Paul E. Utgoff, and Ranan Banerji. "Learning by Experimentation: Acquiring and Refining Problem–Solving Heuristics." In *Machine*

Learning: An Artificial Intelligence Approach, Volume I, Michalski, Ryszard S., Carbonell, Jamie G., and Mitchell, Tom M. (Eds). Tioga Publishing Company, Palo Alto, CA (1983).

[Riolo, 1987a] Riolo, Rick L. "Bucket Brigade Performance: I. Long Sequences of Classifiers." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms,* 184-195, John J. Grefenstette (Ed.). Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1987.

[Riolo, 1987b] Riolo, Rick L. "Bucket Brigade Performance: II. Simple Default Hierarchies." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms,* 196-201, John J. Grefenstette (Ed.). Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1987.

[Riolo, 1988a] Riolo, Rick L. "CFS-C: A Package if 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, Ann Arbor, 1988.

[Riolo, 1988b] Riolo, Rick L. "LETSEQ1: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences." Logic of Computers Group, Division of Computer Science and Engineering, University of Michigan, Ann Arbor, 1988.

[Riolo, 1988c] Riolo, Rick L. "CFS-C/FSW1: An Implementation of the CFS-C Classifier System in a Domain that Involves Learning to Control a Markov Process." Logic of Computers Group, Division of Computer Science and Engineering, University of Michigan, Ann Arbor, 1988.

[Riolo, 1988d] Riolo, Rick L. "Empirical Studies of Default Hierarchies and Sequences of Rules in Learning Classifier Systems." Ph.D. Dissertation, University of Michigan (1988).

[Robertson, 1987] Robertson, George G. "Parallel Implementation of Genetic Algorithms in a Classifier System." In *Proceedings of the Second International Conference on Genetic Algorithms and their Applications,* 140-147. John J. Grefenstette (Ed.). Cambridge, Massachesetts, July 28-31, 1987.

[Robertson and Riolo, 1988] Robertson, George G. and Riolo, Rick L. "A Tale of Two Classifier Systems." *Machine Learning,3,* 2-3 (1988)

[Samuel, 1959] Samuel, A. L. "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development, 3,* 211-232 (1959).

[Sannier and Goodman, 1988] Sannier, Adrian V., III, and Goodman, Erik D. "Midgard: A Genetic Approach to Adaptive Load Balancing for Distributed Systems." In *Proceedings of the Fifth International Conference on Machine Learning,* pp 174–180. San Mateo, CA: Morgan Kaufman (1988).

[Schuurmans and Schaeffer, 1988] Schuurmans, Dale, and Schaeffer, Jonathan. "Representational Difficulties with Classifier Systems." Unpublished manuscipt, Department of Computing Science, University of Alberta, Edmonton, Canada (1988).

[Sutton, 1988] Sutton, Richard S. "Learning to Predict by the Methods of Temporal Differences." *Machine Learning, 3:* 9-44 (1988).

[Wilson, 1985] Wilson, Stewart W. "Knowledge Growth in an Artifical Animal." *Proceedings of an International Conference on Genetic Algorithms and their Applications,* 16-23. John J. Grefenstette (Ed.). Carnegie-Mellon University, Pittsburg, 1985.

[Wilson, 1986a] Wilson, Stewart W. "Hierarchical Credit Allocation in a Classifier System." *Research Memo RIS No. 37r.* The Rowland Institute of Science, Cambridge MA, 1986.

[Wilson, 1986b] Wilson, Stewart W. "Classifier System Learning of a Boolean Function." *Research Memo RIS No. 27r.* The Rowland Institute of Science, Cambridge MA, 1986.

[Wilson, 1986c] Wilson, Stewart W. "Classifier Systems and the Animat Problem." *Machine Learning, 2,* pp. 199-228 (1987).

[Wilson, 1987] Wilson, Stewart W. "Hierarchical Credit Allocation in a Classifier System." In *Genetic Algorithms and Simulated Annealing,* Lawrence, David (Ed). Morgan Kaufman Publishers, Inc., Los Altos, CA (1987).

[Wilson, 1988] Wilson, Stewart W. "Bid Competition and Specificity Reconsidered." *Research Memo RIS No. 54r.* The Rowland Institute of Science, Cambridge MA, 1988.

B The CFS-C System Commands

Summary of the CFS-C commands:

APPENDCF [fdname]

APPENDMSG [fdname]

CLASSIFY steps

DISPLAY { what [, format [, fdname]] } ...

ENVCMD command

GENRANDCFS [number [, meanstr [, meanbr [, RA]]]]

HELP [SET | DISPLAY]

LOADCFS [fdname]

LOADENV [fdname]

LOADMSGS [fdname]

SAVESYS [fdname]

SET variable=value ...

STEP

STOP

Notes on Notation:

UPPERCASE items are literal strings, to be entered as shown.

lowercase items are parameters for which values are to be given.

{ **p1** | **p2** } indicates a choice that must be made ('p1' or 'p2' must be supplied).

[**p1**] indicates an optional item 'p1'.

[**p1** | **p2**] indicates a choice of optional items ('p1' or 'p2' or nothing is allowed).

. . . indicates the preceding item or group may be repeated (separated by one or more blanks).

Command: APPENDCFS

Purpose: Append (add) classifiers to the classifier list.

Prototype: { APPENDCFS | APPC } [fdname]

Parameters **fdname** The name of a file or device (e.g., the keyboard) from which the classifiers are to be read. If fdname is not specified, classifiers will be loaded from the default classifier-input file (usually INIT.CF).

Examples: APPC TEST1.CF
APPENDCFS

Description: Classifiers are read, one per line, from a file or device. Each classifier that is successfully read is added to the classifier list, i.e., the classifier list is not emptied before adding new classifiers. See the LOADCFS command for a description of the input format.

Notes: See the notes for the LOADCFS command.

Command: APPENDMSGS

Purpose: Append (add) messages to the message list.

Prototype: { APPENDMSGS | APPM } [fdname]

Parameters: **fdname** The name of a file or device (e.g., the keyboard) from which the messages are to be read. If fdname is not specified, messages will be loaded from the default message-input file (usually the file INIT.MSG).

Examples: APPM TEST1.MSG
APPENDMSGS

Description: Messages are read, one per line, from a file or device. Each message that is successfully read is added to the current message list, i.e., the current message list is not emptied before adding new messages. See the LOADMSGS command for a description of the input format.

Notes: See the notes for the LOADMSGS command.

Command: CLASSIFY

Purpose: Execute the major-cycle a specified number of steps.

Prototype: { CLASSIFY | CL } steps

Paramters: **steps** An integer number of major-cycle steps to be executed.

Examples: CLASSIFY 1000
CL 20

Description: The CLASSIFY command is used to actually run the classifier system, i.e., execute a specified number of major-cycle steps. The user is not prompted for another command until the specified number of major-cycles are executed. However, the "auto-display" facility may be used to display selected items (e.g., the classifier list) periodically while CLASSIFY command is executing. Also, the "auto-save" facility may be used to write periodically the entire state of the classifier system into files. See "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences." for a description of the auto-save and the auto-display facilities.

Notes:

1. The steps parameter is stored as an unsigned int variable (as is the major-cycle counter, CycleStp). Therefore is cannot be larger than the largest unsigned int value allowed for the particular machine/compiler being used.
2. The Classify() subroutine in the file RUNCFS.C is the entry-point for the CLASSIFY command.

Command: DISPLAY

Purpose: To display various items, e.g., the classifier list or environment state.

Prototype: { DISPLAY | DI } { what [, format [, fdname]] } ...

Parameters: **what** What item is to be displayed:

- **ML** current message list
- **CL** classifier list
- **VARS** selected system variables (e.g., system treasury, tax rates, effector rate, etc.)
- **ENV** environment-specific variables.
- **SETTINGS** selected system control variables (e.g., the names of the default files for loading messages and classifiers.)
- **varname** may be the name of a system variable.

format An integer number ≥ 0 that controls the format of the display. The relationship between integers and the format depends on what item is being displayed—see below for more details. The default format number is 1.

fdname Where the item is to be displayed, i.e., the name of a file or device. By default, items are displayed on the standard output device (e.g., the computer's screen).

Examples:

```
DI CL,2
DISPLAY ML CL,2 VARS,3,OUTFILE1.VAR
DI ENV CL CL,999,SAVECF VARS
```

Description: The DISPLAY command can be used to display one or more items on the computer's screen, on a printer, or in a file.

If the fdname parameter is included with an item, the item will be displayed on the specified file or device. By default items are display on the "stdout" device (usually the computer's screen).

If the format parameter is not included, the item is displayed in a default format (equivalent to format=1). While the exact effect of a particular format integer depends on which item is being displayed, in general a larger format number means more information will be displayed. The currently supported format numbers for the items are as follows:

1. For the message list (item ML), format 1 will display. for each message, its Id number, its message string, its intensity, and the Id of the classifier that produced it.
2. For the classifier list (item CL):

   Format 1 For each classifier, its Id number, the classifier itself (i.e., its conditions, condition types, etc.), the classifier's strength, and its bidratio.

Format 2 For each classifier, a series of numbers that summarize the classifiers history are displayed in columns. The following shows the titles displayed for each column and what the numbers in the columns mean:

Id The classifier Id number

Strngth Its strength.

ChgStr Change of strength in the previous step.

BidR BidRatio (the specificity, usually).

TBid Total number of times it bid.

TMat Total number of times its conditions were satisfied.

TPro Total messages produced.

TPst Total messages posted (after effector conflict-resolution).

TEff Total number of times it activated an effector.

TPRw Total number of times it received a positive reward.

TNRw Total number of times it received a negative reward.

Crtd The major-cycle step it was created.

TOfs Total number of offspring it has had.

3. For system variables (item VARS), formats 1 and 2, the values of a selected set of system variables are displayed. Each value is preceded by a short description–for details, try the command.

4. For special I/O control (item SETTINGS), format 1, the values of a selected set of system variables that control I/O are displayed. Each value is preceded by a short description. For details, try the command.

5. For environment variables (item ENV), format 1 and other values will probably be implemented. The display depends on the particular environment implemented. Other format numbers are usually supported as well.

A format value of 99, which can be used for all items, displays the item in a "save-format," i.e., the entire state of the item is displayed in a format that can be used to initialize the CFS-C classifier system.

Individual CFS-C system variables also can be included on a DISPLAY command. The system variables displayed in this way are always displayed in a format like this:

$$headtax = 0.0001$$

where "headtax" is the name of the system variable and 0.0001 is the current value assigned to that variable.

Notes:

1. The auto-display facility may be used to automatically display selected items (e.g., classifiers, variables, etc.) every N steps of the major-cycle (where N is set by the user). For more information, see "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."

2. The SAVESYS command may be used to display the state of the entire system (the variables, messages, classifiers, and environment) in a format that can be used to re-initialize the system. See the description of that command for more details.
3. The subroutine Display() in the file USERCMD.C is the main entry-point for code that implements the DISPLAY command.

Command: ENVCMD

Purpose: Execute a special 'environment' command.

Prototype: { ENVCMD | ECMD } command

Parameters: **command** The name of the command to be executed.

Examples: ENVCMD HELP

Description: The list of commands depends on the particular environment being used. See the documentation specific to an environment for more details. (The command HELP should be implemented for all environments).

Notes: The subroutine DoEnvCmd() is the entry-point for the ENVCMD command. Since it is an environment-specific command, it must be implemented as part of the enviroment/domain source-code.

Command: GENRANDCFS

Purpose: Generate random classifiers.

Prototype: { GENRANDCFS | GRCFS } [n [,meanstr [,meanbr [,ra [, i]]]]]

Parameters: **n** Number of classifiers to generate ($1 \leq n \leq CFLSTSZ$). (Default = 16.)

meanstr A target mean strength for the classifiers produced. (Default = the current value of the CfStrDf variable). The classifiers generated are given strengths uniformly distributed in the range ($meanstr - (meanstr/20)$) to ($meanstr + (meanstr/20)$).

meanbr A target mean bidratio for the classifiers produced. If a meanbr value is supplied, the classifiers are given bidratios uniformly distributed in the range ($meanbr - (meanbr/20)$) to ($meanbr + (meanbr/20)$). If a meanbr value is not supplied, then the bidratio for each classifier is set equal to the specificity of the classifier's conditions.

ra If present, this literal string causes the ActType (action-type) to be generated at random. If not present (the default), all classifiers will have the PASS (pass-through) action-type.

i If present, this means initialize (empty) the classifier list before producing the random classifiers. Otherwise just add the random classifiers to the existing list.

Examples: GENRANDCFS 100
GRCFS 200,5000,,ra

Description: The classifiers generated by the GENRANDCFS command are added classifiers already in the classifier list. (However, the final parameter i can be included to have the list first emptied.) The StpCrtd ("step created") value is set to 0 for all the new classifiers (just as it is after a LOADCFS command).

The condition and actions strings are assigned 0, 1, or # at each loci at random under a distribution controlled by the system variable GRCFrNSL ("fraction non-specific loci"). That is, the probability that a given loci has a "don't care" (#) value is given by GRCFrNSL; if a loci is not assigned a #, then it is assigned a 1 or 0 with equal probability. The default value for GRCFrNSL is 0.60. The SET command may be used to set GRCFrNSL to other values.

The first condition of every classifier is a 'MATCH' type condition. The second condition is either MATCH or NOTMATCH, assigned at random using a distribution controlled by the system variable GRCFrMC ("fraction match conditions"). The default value for GRCFrMC is 0.95. The SET command may be used to set GRCFrMC to other values.

If the 'RA' parameter is not given on the GENRANDCFS command, all classifiers are assigned the PASS (pass-through) action-type. If the 'RA' parameter is given, action-types are assigned at random to the classifiers, under this cummulative probability distribution:

Action-Type	Cum. Prob.
$PASS$	0.90
AND	0.92
OR	0.94
XOR	0.96
$ADD1$	0.98
$SUB1$	1.00

This distribution function is not modifiable at run-time.

Notes:

1. The cummulative probability distribution that controls generation of action-types is defined in the CfOpNode (classifier operator nodes) that define the available action types. See the file CFOPS.DEF.
2. The subroutine GnRndCfs() in the file GRANDCFS.C is the entry point for the GENRANDCFS command.

Command: HELP

Purpose: Display command syntax, display paramters, and some SET parameters.

Prototype: HELP [DISPLAY | SET]

Parameters: **DISPLAY** A brief description of the parameters for the DISPLAY command are displayed.

SET A brief description of the parameters for the SET command are displayed.

Examples: HELP
HELP DISPLAY

Description: The HELP command provides some online aid for CFS-C users. If HELP is entered without parameters, a list of the CFS-C commands are displayed, including abbreviations and parameters. If HELP is entered with the DISPLAY or SET parameter, information about those commands is displayed.

Notes: The subroutine Help() defined in the file USERCMD.C is the entry-point for the HELP comamnd.

NB—the Help command may be commented out because it is not up to date.

Command: LOADCFS

Purpose: To load classifiers from a specified file/device.

Prototype: { LOADCFS | LC } [fdname]

Parameters: **fdname** The name of a file or device from which classifiers are to be read. If no file is specified, classifiers are read from the current default file for classifier input, (DISPLAY SETTINGS to see what the default file is.) Normally this file is INIT.CF, but the SET command can be used to make another file the default file for loading classifiers.

Examples: LOADCFS
LC TEST1.CF

Description: The LOADCFS command is used to load classifiers from a specified file into the current classifier list. Any classifiers on the list before the LOADCFS command is executed are replaced. (To load classifiers without replacing ones there, see the APPENDCFS command.) From 0 to NmCfsMx can be loaded (where NmCfsMx is a SET-able system variable that specifies the maximum number of classifiers on the list).

The input file must look like this:

```
; lines that have a semicolon in the
; first (leftmost) column are
; ignored–they may be used for comments
;
cond1,cond2/acttype/action,strength,bidratio, ...
.
. and so on, one line for each classifier.
.
ENDCFS
```

Each non-comment line must contain at least the values for the following required components:

cond1 The classifier's first condition, of the form:

$$m01\#01...0\#$$

where 'm' is the condition-type ('m' for match) and the rest of the condition consists of the sequence of 0's, 1's, or #'s that define the first condition string.

cond2 The classifier's second condition, of the form:

$$m01\#01...0\#$$

where 'm' is the condition-type (may be 'm' for match or '~' for NOT-MATCH type), and the rest of the condition consists of the sequence of 0's, 1's, or #'s that define the second condition string.

acttype The classifier's action-type. The action-types currently supported are:

- **PASS** traditional pass-through
- **AND** logical AND
- **OR** logical OR
- **XOR** logical EXCLUSIVE OR
- **ADD1** Add 1
- **SUB1** Subtract 1

See Section 3.2.2 for a description of these various action-types.

action The classifier's action string, of the form:

10#01...10#

that is, a sequence of symbols from the alphabet $\{0, 1, \#\}$.

The rest of the values are optional. If specified, these values must be included in the following order, each value separated from its neighbors by a comma (all on the same line):

Strength The classifier's strength. The default value is the value of the CfStrDft system variable. (Actually, the default value is the CfStrDft value with some random noise added to it.)

BidRatio The classifier's bidratio. The default value is the classifier's specificity.

Cf_Id An id number for the classifier. By default Id's are just assigned in order from 1 as classifiers are added or created.

ChngStr The change in strength during the previous step.

TotNmBid The total number of times the classifier has bid. The default is 0.

TotMtch The total number of times the classifier has had its conditions matched. (Note that a classifier may be matched many times during one major cycle step.) The default is 0.

TotProd The total number of messages the classifier has produced. The default is 0.

TotPost The total number of messages **posted**, i.e., the number produced less any that were removed by the effector conflict resolution mechanism.

TotEMtch The total number of messages produced by the classifier that have matched effector conditions. The default is 0.

TotEAct The total number of messages produced by the classifier that have activated effectors. The default is 0.

TotPosRw The total number of positive rewards the classifier has received (i.e., rewards¿0). The default is 0.

TotNegRw The total number of negative rewards the classifier has received (i.e., rewards < 0). The default is 0.

TotNmOfs — The total number of offspring produced from the classifier. The default is 0.

StpCrtd — The major-cycle step during which the classifier was created. The default is 0 (created at the beginning of the run).

StpLPrd — The major-cycle step during which the classifier last won the bidding competition and produced some messages. The default is 0 (not won yet).

StpLPst — The major-cycle step during which the classifier last **posted** messages, i.e., produced messages that were *not* eliminated by the effector conflict resolution mechanism. The default is 0 (not won yet).

NoBB — If 1, the bucket brigade algorithm is not applied to this classifier. The default is 0, i.e., apply bucket brigade algorithm to the rule. (**NB:** This may not be implemented yet—check the source.)

NoRpl — If 1, the classifier will *not* be replaced, no matter what its strength. The default is 0, i.e., the rule is replaceable.

NoPrn — If 1, don't use the rule as a "parent" rule in the genetic algorithm. (**NB:** This may not be implemented yet—check the source.)

When classifiers are displayed using a format number of 99, (the "save format"), the values for each classifier are displayed in just the order shown above, so that classifiers written in that format can be re-loaded using the LOADCFS command.

Notes:

1. When the CFS-C is run, classifiers are loaded from: (a) a single initialization file, if one is specified on the run command; or (b) the file INIT.CF (if it exists). For details on how to run the CFS-C in a particular domain, see "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."
2. The subroutine LoadCfs() defined in the file CFMSGIO.C is the entry point for the code that implements the LOADCFS command.

Command: LOADENV

Purpose: Load the state of the environment from a file or device.

Prototype: { LOADENV | LE } [fdname]

Parameters: **fdname** The name of a file or device from which the state of the environment is to be read. If no file is specified, the values are read from the current default file for environment input (DISPLAY SETTINGS to see what the default file is.) Normally this file is INIT.ENV, but the SET command can be used to make another file the default file for loading the state of the environment.

Examples: LOADENV
LE TEST1.ENV

Description: The LOADENV command is used to load all the values that define a state of the environment, and to load any values that define the payoff function and effectors for the task-domain. The previous state of the environemnt is replaced.

Of course the contents of the input file/device depends on the particular environment linked to the CFS-C code. For details, see the documentation for the domain/environment.

Notes:

1. When the CFS-C is run, the state of the environment is loaded from: (a) an initialization file, if one is specified on the run comamnd; or (b) the file INIT.ENV (if it exists). For details on how to run the CFS-C in a particular domain, see "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."

2. While for exact format of the input file is dependent on the environment, two rules should be observed across all environments: (a.) Lines that have a semi-colon in the first (leftmost) column should be ignored–they can be used for comments. (b.) The file should end with one or more lines that define the condition-parts of the effectors for the domain, followed by a line that has the literal string:
`END-EFFS`
starting in the first column.

   For more details, see Section 4, "How to Define an Environment for the CFS-C Classifier System."

3. The subroutine LoadEnv() is the entry point for the code that implements the LOADENV command. Since this code is environment/domain dependent, it must be included in the code that defines each particular environment.

4. For an example, see "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."

Command: LOADMSGS

Purpose: To load messages from a specified file/device into the current list.

Prototype: { LOADMSGS | LM } [fdname]

Parameters: **fdname** The name of a file or device from which messages are to be read. If no file is specified, messages are read from the current default file for message input (DISPLAY SETTINGS to see what the default file is.) Normally this file is INIT.MSG, but the SET command can be used to make another file the default file for loading messages.

Examples: LOADMSGS
LM TEST1.MSG

Description: The LOADMSGS command is used to load messages from a specified file onto the current message list. Any messages on the list before the LOADMSGS command is executed are replaced. (To load messages without replacing ones there, see the APPENDMSGS command.) From 0 to NmIMsgMx can be loaded (where NmIMsgMx is a SET-able system variable that specifies the maximum number of internally generated messages allowed on the list).

The input file must look like this:

```
; lines that have a semicolon in the first (leftmost)
; column are ignored–they may be used for comments.
msg-string, id, intens, prod-id, inputmsg1, inputmsg2
.
. and so on, one line for each message.
.
ENDMSGS
```

Each non-comment line must contain all the values for one message. The only required value is the "msg-string", i.e., the string of symbols from $\{0, 1\}$ that is the message string itself. The rest of the values, all optional, are:

- **id** An Id for the message. In general, the default Id numbers that are supplied by the system should be used. These are unique within each cycle step, but the Id's are re-used every third major-cycle step.
- **intens** The message's intensity value, i.e., the bid made by the classifier that produced the message. The value of the MsgIntDf system variable is used as the default (after some random noise is added).
- **prod-id** The Id of the classifier that produced the message. The default is 0 (no producer).
- **inputmsg1** The Id of the message that matched the left condition of the classifier that produced this message. The default is 0 (no producer, so no input-message).

inputmsg2 The Id of the message that matched the right condition of the classifier that produced this message. The default is 0 (no producer, so no input-message).

When the DISPLAY command is used to display messages in format 99 (the "save format"), the messages are displayed in the format described above. Thus messages written in that format can be be read in by the LOADMSG command.

Notes:

1. When the CFS-C is run, messages are loaded from (a) a single initialization file, if one is specified on the run comamnd; or (b) the file INIT.MSG (if it exists). For details on how to run the CFS-C in a particular domain, see "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."
2. The subroutine LoadMsgs() defined in the file CFMSGIO.C is the entry point for the code that implements the LOADMSGS command.

Command: SAVESYS

Purpose: To save the complete state of the classifier system, including the environment state, so that the system can be re-started in the same state.

Prototype: SAVESYS [fdname]

Parameters: **fdname** The name of the file or device to which the complete state of the system is to be written. If no fdname is specified, a fdname will be constructed by concatenating the values stored in the following system variables:

SaveSyPa Usually a path or drive specification. The default value is "" (no path).

SaveSyFN The name of a file, without an extension. The default value is constructed from the date and time a run was started and is of the form "Smddhhmm", where:

S is a constant

m is the month (in hex)

dd is the day

hh is the hour (24 hr clock)

mm is the minute

SaveFNum Is an integer number of the form "001", "002", and so on, which is used as a filename extension. The number stored in SaveSFNum is incremented each time the system is saved.

Examples: SAVESYS exp-1.sav

The state of the system will be saved to the file named EXP-1.SAV.

SAVESYS

Assuming the classifier system was run on 10 November starting at 14:01 (2:01 pm), and that this is the second time the system was saved during the run, the state of the system will be written to the file named "SB101401.002".

Description: At any time during a classifier system run, the SAVEVSYS command can be used to save (i.e., write) the entire state of the system, including the message list, the classifier list, the state of the environment, and so on, to a user-specified file. This file can then be used to re-start a classifier system in exactly the same state it was in when the system was saved to that file.

The files created by the SAVESYS command contain the following information, in the order described below:

1. System Constants.

Several system constants are written, including:

STRNGSZ The length of message, condition, and action strings.

MSGLSTSZ The maximum message list size.

CFLSTSZ The maximum classifier list size.

The values assigned to these and other system-wide macro-names are written on comment lines.

2. The Old Message List.

   The contents of the old message list (the list used to produce the current messages) are written out. The messages are written using format 99, the "save format," which is the format messages must be in to be loaded. (See the LOADMSGS command for a description of this format.)

3. The Current Message List.

   The contents of the current message list are written out. The messages are written using format 99, the "save format," which is the format messages must be in to be loaded. (See the LOADMSGS command for a description of this format.)

4. The Classifier List.

   The contents of the current classifier list are written out. The classifiers are written using format 99, the "save format," which is the format classifiers must be in to be loaded. (See the LOADCFS command for a description of this format.)

5. The State of the Environment.

   Values that define the state of the environment are written out. The exact format depends on what environment is linked to the CFS-C system. However, the environment-specific values should be terminated by one of more lines, each containing the condition-part of an effector. The last line of the environment-specific values should have the literal string "END-EFFS" beginning is column one.

6. The CFS-C System Variables.

   The CFS-C system variables are written out, one to a line, in a format like the following:

   $$headtax = 0.001$$

   where "headtax" is the name of the variable and 0.001 is the value of that variable. The variables are followed by a line that contains the literal string "END-VARIABLES" starting in column one.

   For a complete description of the format of the information in the file created by the SAVESYS command, run the program, issue the command, and list the file created.

Notes:

1. Besides saving the state of the system by entering the SAVESYS command, a user can have the state of the system periodically saved to a series of files, each file containing the state of the system at a particular major-cycle step. For a description of this auto-save faciltity, see "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."

2. For a description of how to start the CFS-C system using a file created by the SAVESYS command, see "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."
3. The subroutine SaveSys() defined in the file RUNCFS.C is the entry point for the code that implements the SAVESYS command.

Command: SET

Purpose: To set system variables to user-specified values.

Prototype: SET { variable=value } ...

Parameters: **variable** The name of a CFS-C variable that can be set.

value A value that is valid for the specified variable.

Description: The SET command can be used to change the value of many of the CFS-C system variables. These system variables control the bidding process, the bucket-brigade, and so on. These variables also control some of the system's I/O facilities, e.g., the auto-save and auto-display features provided as part of the CFS-C system.

For a description of each the system variables that can be included in a SET command, see "Appendix C: CFS-C Constants and Variables." For a description of the auto-save and auto-display facilities provided by the CFS-C system, see the document "LETSEQ: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Predict Letter Sequences."

Note that all variables that can be changed by the SET command can have their current values displayed by the DISPLAY command; see the description of the DISPLAY command for more details.

Examples:

SET $bid\_k = 0.05$

SET headtax=0.01 bidtax=0.04 prodtax=0.06

SET log=on

Notes:

1. The complete list of variables that can be set is defined by the INIT.CFS file that is supplied when the system is run (see the document *How to Use CFS-C 1.0*), and in the tables in the USERCMD.DEF file.
2. The Set() subroutine defined in the file USERCMD.C is the entry point for the code that implements the SET comamnd.

Command: STEP

Purpose: To execute one major-cycle step.

Prototype: { ST | STEP }

Parameters: None.

Description: The STEP command is equivalent to the command:

```
CLASSIFY 1
```

See the description of the CLASSIFY command for more information.

Command: STOP

Purpose: To stop a classifier system run.

Prototype: STOP

Parameters: None.

Description: The STOP command is used to stop a run and return to the host operating system.

C CFS-C System Constants and Variables

C.1 System Constants

All important constants in the CFS-C source-code are defined by assigning values to macro-names. The macro-names always consist of upper-case letters. Most of these macro-names are defined in the file CORE.H or in the file UTILITY.H. However, the values for some constants are defined in the file COMPILER.H, which must be edited to select the particular machine/compiler you wish to use. See that file for instructions on how to do that.

Note that there are several macro-names that end in the letters "SZ". These macros usually define the size of some array. Since C arrays always start with the 0th entry, the largest legal entry is always one less than the size of the array. Therefore, for each macro that ends in the letters SZ, there is another macro that ends in "MX" that is defined as one less than the value of the SZ variable.

The following constants are used in many files, especially the first one, which probably is used in all files. These are all defined in CORE.H.

Macro	Meaning
STRNGSZ	The basic size of all messages, conditions, and action-strings in the classifier system. The default value is 16.
MSGLSTSZ	The maximum size of the message list, including detector messages and messages generated by classifiers. The default is 32.
CFLSTSZ	The maximum number of classifiers. The default is 256.
EFFLSTSZ	The maximum number of effectors. The default is 8.

C.2 The Basic CFS-C System Variables

These system variables control or record various mechanisms associated with the CFS-C major-cycle. All of these variables can be included as an item on the DISPLAY command. Most of these variables can be assigned new values by using the SET command–those variables that should not or cannot be modified by the SET command are noted below.

NmCfs Number of Classifiers.

The number of classifiers currently in the classifier list.

NmCMsgs Number of Current Messages.

The number of messages in the current message list.

NmCfsMx Number of Classifiers, Maximum.

This may be set at run time to a value form 1 to CFLSTSZ. The default for NmCfsMx is equal to CFLSTSZ.

NmIMsgMx Number of Internally Generated Messages, Maximum.
NmDMsgMx Number of Detector Generated Messages, Maximum.

Messages on the message list may come from two sources:

1. From other classifiers, i.e., internally generated, and

2. From the detectors, i.e., from the input interface.

These two variables control the share of the message list that is reserved for messages from these two sources. During a single major-cycle, classifiers may post up to NmIMsgMx messages; if the number of candidate messages is greater than NmIMsgMx, then a competition is run to select which ones actually get posted.

Note that these two variables always add up to MSGLSTSZ, the maximum size of the message list. Therefore if the SET command is used to change one of these two variables, the other variable automatically will be changed to maintain a constant sum. Also note that NmDMsgMx must be less than DMSGSZ, the maximum number of detector messages allowed at one step.

DetMsgMn Detector Messages, Minimum.

A minimum value for NmDMsgMx. DetMsgMn should be set when an environment is initialized, to the largest number of detector messages that could be generated at one step. Thus DetMsgMn specifies a lower bound for the NmDMsgMx value, i.e., the portion of the list reserved for detector messages, and an upper bound for NmIMsgMx, the portion reserved for messages generated internally, i.e., by classifiers. If the user tries to set NmDMsgMx ¡ DetMsgMn, a warning message is printed.

DelHall Delete Hallucination Messages.

To make if possible for a classifier system to distinguish messages from the environment from other messages, messages from detectors have certain "tag-bits" set to indicate that the message is from the detectors (in CFS-C the leftmost bits of detector messages are "00"). However, there is nothing to prevent a classifier from producing messages that start with that tag! Such a message can be thought of as a "hallucination" since it is an internally generated signal that looks like it came from the environment.

Hallucinations cause terrible performance stability problems for classifier systems [Robertson and Riolo, 1988] [Riolo, 1988d], just as they do for humans. In CFS-C hallucinations can be eliminated by setting DelHall=1. When DelHall=1, if a classifier wins the bidding competition and produces a hallucination message, that message is just not *posted*, i.e., it is not added to the new list. However, since the classifier *produced* a message, it pays its bid and is taxed appropriately. Thus rules that only produce hallucinations rapidly lose strength, since they pay to produce them but have not income.

Note that the NmProd member of a CfNode is incremented even when the classifier tried to produce hallucinations that were not really added to the new message list.

The default for DelHall is 0 (off). Probably it should be 1; in fact, probably it should be done unconditionally.

Of course there are other ways to prevent the production of hallucinatory messages: for example, new classifiers could be examined, and if they have any chance of producing hallucinations, they could be modified to prevent that. This would move the test for hallucinations out of the inner-loop, and so would speed the system up. On the other hand, because of the pass-throughs and wildcards, its not always obvious which rules will produce hallucinations.

OneMPerC One Message per Classifier.

Setting OneMPerC to 1 limits each classifier to producing one message per step. If more than one message matches a condition, only one is chosen two produce a new message; the matching messages used to produce it are chosen probabilistically using their intensities.

The default for OneMPerC is 0, i.e., rules can produce many messages. I suspect (and John Holland has told me he concurs) that classifiers should always be limited to producing one message per step. Allowing classifier to produce multiple messages per step causes problems for chains (when there are mutiple copies of each rule in a chain), and when there are generalist classifiers that match lots of messages per step. But maybe there are cases where producing more than one per step would be useful, so I leave the optional available.

NmIEMMx Number of Internally Generated Effector Messages, Maximum.
NmIIMMx Number of Internally Generated Internal Messages, Maximum.

Classifiers can produce three kinds of messages:

1. Effector activating messages, i.e., messages that start with the tag-bits that mean "message to effectors" ("10" is the tag used in CFS-C).
2. Detector messages, i.e., messages that start with the tag-bits that mean "message from the detectors" ("00" is the tag used in CFS-C). See the *DelHall* parameter for more about these messages.
3. Internal messages, i.e., messages that are not detector messages or effector messags.

NmIEMMX and NmIIMMx can be used to reserve portions of the message list for effector and internal messages. For example, if NmIMsgMx (maximum number of internally produced messages) is 8, and NmIEMMx and NmIIMMx are each 4, then half the message list is reserved for effector messages and half for internal messages. If a classifier tries to produce an effector message when there are already 4 effector messages on the new list, it is treated as if it lost the competition to post messages, i.e., no message is produced and its NmProd and NmPost counters are not incremented, so that it does not pay its bid or any producer tax. Similar actions are taken if the number of internal messages on the new list is equal to NmIIMMx and a classifier tries to post another one.

These limits are by default off, i.e., the system acts as originally defined and there can be any number of any type of message posted. NmIEMMx = 0 turns the limits off, and setting NmIEMMx > 0 turns the limits on.

These parameters make it possible to have a segmented message list, at least in terms of posting new messages. That is, there is one part of the list reserved for effector messages, and another part reserved for "internal memory" messages. The idea is to prevent the message list from being filled by effector messages produced by copies of a successful classifier (including variants that have different genotypes but that still produce the same effector messages). When the list is so filled, it is impossible (or at least difficult) for a classifier to post a "memory" message.

MxDupMsg Maximum Duplicate Messages per Step.

One issue that has never been resolved is whether the message list should contain duplicate messages. One obvious reason to allow duplicates is to allow the bucket brigade to pass strength back to multple similarly functioning rules, so that no one rule is critical for performance. However, allowing duplicate messages can also lead to a loss of effective message list bandwidth. Two examples of the problem with allowing lots of duplicates are:

1. Classifiers that are general and match messages produced by themselves can cause an explosion of messages, swamping the message list. (This is the reason for the OneMPerC parameter, and for a progressive producer tax.)
2. Multiple copies of the same (or functionally equivalent) classifiers produce all the same messages, swamping the message list.

In any case, filling the message list with duplicate messages means there is no room for other messages. This makes it difficult to, for example, have messages posting lots of internal "memory" messages for use in the future while at the same time having the system generate effector-activating messages. With no control of duplicates, the classifiers that produce effector-activating messages proliferate (they get rewards from the environment and so have high strengths), and all the copies produce effector activating messages with bids that are greater than those classifiers trying to produce "memory" messages.

If MxDupMsg is > 1, when a message is generated it is checked against all existing new messages, and if there are already MxDupMsg copies of that message, the new message is not produced. For the classifier that tried to make that message, its as there just wasn't room on the message list for that message. Thus it doesn't pay the suppliers of that message, and if it didn't produce any other messages, its as if it just didn't win the bidding competition.

MxDupMsg is a heavy-handed, inefficient way to deal with some problems caused by duplicate messages. Other ways include using the MxIIMMx and

MxIEMMx parameters, or the OneMPerC parameter. But the essential issues involved have yet to be addressed in a systematic way, I think.

MsgIntDf — Message Intensity Default.

When messages are loaded at the beginning of a run, or when the system's detectors add messages to the message list, if no value is supplied for a message's intensity, the value of MsgIntDf is used. The default value for MsgIntDf is 50.

CfStrDf — Classifier Strength Default.

When classifiers are loaded at the beginning of a run, if no value is supplied for a classifier's strength, the value of CfStrDf is used. (Actually, the value used is generated by adding some random noise to the CfStrDf value.) The default value for CfStrDf is 1000.

CfStrMx — Classifier Strength Maximum.

The maximum strength of a classifier is limited by the value assigned to this variable. If the strength of a classifier is greater than this limit, the excess strength is transferred from the classifier to the system treasury. The default value for CfStrMax is 99999.

CfSuppDf — Classifier Support Default.

When a classifier has a not-match condition, that condition is satisfied only when no message on the message list is in the subset of messages specified by that condition. When a not-match condition is satisfied, the value of CfSuppDf is added to that classifier's total support. (Support is usually the sum of the intensities of all the messages that match a classifier's conditions.) The default for CfSuppDf is 100.

CycleStp — Cycle Step.

This is the the system's internal clock. It is set to 1 at the beginning of a run, and is incremented by 1 at the end of each major-cycle of the classifier system.

EffectRt — Effector Rate.

EffectRt controls how often the system's effectors are activated. That is, an effector is activated only when both:

1. Its condition-part is satisfied by messages on the new message list, and
2. CycleStp % EffectRt == 0

(where % is the symbol for the modulus operator in C). For example if the EffectRt is 1, then effectors are activated whenever their condition-parts are satisfied. The default value for EffectRt is 1.

Note that the new message list is examined every major-cycle step for messages that satisfy effector conditions, no matter what the EffectRt value is.

Further, if more than one message satisfies the condition-part of one effector, a competition is held and all those messages but one are removed from the new message list. This competition and weeding occurs every major-cycle step, no matter what the EffectRt is. The EffectRt only affects the rate at which effectors are activated (i.e., execute some subroutine and change the the state of the environment or of the system in some way).

I think there needs to be a more flexible approach to the basic issue here, i.e., how should the system decide when to *do* something, after "thinking" about it for a while. Some solution that uses some system-generated rating of when it is ready to act should be implemented, e.g., even some crude threshold of support for action notion.

DetectRt — Detector Rate.

DetectRt controls how often the system's detectors are sampled. That is, a detector can add a message to the current message list only when

    CycleStp % DetectRt == 0

(where % is the symbol for the modulus operator in C). For example if the DetectRt is 2, the systems detectors add messages to the message list every other major-cycle step. The default value for DetectRt is 1.

Detector sampling is another area that needs lots of thought and work. Should they be sampled every step, should detector messages be repeated several steps (to allow the system to use them for processing several steps before the inputs change)?

SysTreas — System Treasury.

This variable acts as a pool of strength for the system that is used to maintain a constant total strength in the system. That is, the total strength in the system is the sum of the strengths of all the classifiers plus the SysTreas value. Whenever strength is removed from classifiers (e.g., when taxes are paid), that strength is added to the SysTreas variable. Conversely, when strength is added to classifiers (e.g., when a reward is added to all classifiers currently active), that reward is removed from the SysTreas.

NOTE WELL: SysTreas is not currently kept in balance, probably because strength is not accounted for when rules are deleted and created. I have not used SysTreas, so it doesn't bother me that it is not balanced; if you want to use it, you should fix it!

TotCfStr — Total Classifier Strength.

This variable contains the sum of the strengths of all classifiers currently in the classifier list. It is also used to calculate the average classifier strength. This variable should not be set by the user.

TotCfBR — Total Classifier BidRatio.

This variable contains the sum of the bidratios of all classifiers currently in the classifier list. It is also used to calculate the average bidratio. This variable should not be set by the user.

Bid_k Bid Constant.

This value is used to calculate a classifiers bid. This value can be thought of as a risk factor, i.e., a control on what proportion of its strength a classifier will bid at one time. The default value for *Bid_k* is 0.10.

SuppBid Support factored into Bidding.

In the section on bidding in CFS-C, the bid a classifier makes depends in part on the *support* for that classifier, i.e., a value proporational to the number and intensity of all the messages it matches.

SuppBid can be used to control the use of support in bidding. By default, SuppBid is 1, i.e., support is used in making bid calculatios in CFS-C. To not use support, set SuppBid = 0.

Almost all my work has been done with SuppBid=0: I just didn't want one more confounding factor to worry about. Also note that I think support should probably be a factor in calculating the expected bid, not the actual bid. The reason is that anything in the bid calculation affects the fixed point strength of rules.

BidPow Bid Power.
EBRPow Expected-Bid Bidratio Power.

These two parameters alter the probability that a rule with a certain bid will win the competition; they do *not* alter the bid itself. BidPow can be used to increase the probability that high bidding rules will win the competition EBRPow can be used to increase the probability that specialists will win the bid over generalists. See the section on bidding in CFS-C for complete details.

CfBidMin Classifier Bid, Minimum.

This is the minimum bid a classifier can make. If the bid calculated based on the classifier's strength, bidratio, and so on, is less than CfBidMin, the value of CfBidMin is used as the classifier's bid. This makes is possible to keep all bids greater than zero, even when some classifiers have negative strengths. The default value for CfBidMin is 1.0.

HeadTax Head Tax.
AHeadTax A constant-value Head Tax.

These taxes are applied to every classifier once every major-cycle. That is, if HeadTax is not 0, every classifier's stength is decreased by (Strength * HeadTax) every step, and if AHeadTax is not 0, the strength is reduced by AHeadTax.

Head taxes greater than 0 seem to be necessary to prevent the spread of "deadwood" classifiers. However, note that in a chain of coupled classifiers, taxes have exponential effects, so the tax should be kept as low as possible.

BidTax Bid Tax.
FBidTax Failed Bid Tax.

The BidTax tax is applied to every classifier that bids during a major-cycle step. That is, every classifier has its conditions satisfied during step T has its strength decreased by (Strength * BidTax) at the end of step T.

The FBidTax is applied to every classifier that bids but does *not* produce messages during a major-cycle step. Thus it is a tax applied to failed bidders.

I have found the FBidTax causes less problems for chains, where taxes accumulate exponentially.

ProdTaxMx Producer Tax Maximum.
PrdTaxPw Producer Tax Power.

The producer tax is applied to every classifier that produces messages during a major-cycle step. The tax is calculated as follows:

$$tax = PrdTaxMx * (m/M)^{PrdTaxPw}$$

where m is the number of messages produced by the classifier during the current step (NmProd in the CfNode structure) and M is the maximum number of messages that can be produces, i.e., NmIMsgMx. Thus for $PrdTaxPw >$ 1, as a classifier uses more of the message list capacity, the tax gets rapidly steeper, to a maximum, i.e. to PrdTaxMx.

GRCFrNSL Generate Random Classifiers, Fraction Non-Specific Loci generated.

When a classifier is generated by the GENRANDCFS ("generate random classifiers") command, the fraction of loci that are set to be 0's, 1's, and #'s is controlled by the GRCFrNSL variable. In particular, the value for each loci is determined by making a random draw from a distribution defined by the GRCFrNSL variable: that is, the value of GRCFrNSL is the probability that the new value will be a # symbol; if the new value is not a #, it is changed to a 1 or a 0 with equal probability.

The default value for GRCFrNSL is 0.6, i.e., 60 percent of the loci generated will be # symbols, 20 percent 1's, and 20 percent 0's.

ShareRew Share Reward from Environment.

By default, with ShareRew set to 0, when a reward is received from the environment, the full reward is paid to all active classifiers (or to all effector-activating rules—see FrPayNEC).

If ShareRew is set to 1, then the reward is divided equally amoung all those eligible to receive the reward, rather than giving the full reward to each.

Sharing reward seems to help control the spread of high-strength classifiers, and so help control what has been called *premature convergence* or *fixation.*

ShareRew can also be set to 3 or 4. If it is 3, rewards are not shared but they are biased by bidratio. If it is 4, rewards are shared and biased by bidratio. See BBLEARN.C for more details on these two little used options.

FrPayNEC Fraction Pay Non-Effector-Activating Classifiers.

In the original descriptions of classifier systems (e.g., [Holland, 1986a] when a reward is received from the environment, that amount is added to the strength of *all* active rules. In CFS-C, this occurs by default, when FrPayNEC is 1.

However, a number of studies have shown that paying all rules active leads to the spread of various "parasite" rules, i.e., rules that are active when rewards are received but that do not produce usefule messages. (For example, consider a rule that produces a message that satisfies its own conditions...) To overcome this problem, CFS-C allows one to bias the payment of rewards so that NECs (non-effector activating classifiers) receive no (FrPayNEC = 0) or only a faction of the reward ($0 < FrPayDet < 0$) paid to effector activators.

I use FrPayNEC = 0, though some small fraction may be useful, e.g., for promoting complex structures with bridging classifiers.

DMShare Detector Message Share.

By default DMShare is 1.0, i.e., pay the same for messages from detectors as is paid for messages from other rules. The problem with this is that if a rule has one condition that matches a detector message and a second that matches a message from another rule, then that other rule only gets paid one-half the bid made to use its rule. In what I have called "non-reflex" chains [Riolo, 1987a] this leads to an exponential fall-off of strength as one gets farther from the rewarded rules.

The basic idea of DMShare is to bias payments away from messages from detectors to messages produced by other classifiers, thus reducing the falloff in in strength up a non-reflex chain of coupled classifiers. The way I have implemented this is to bias the BidShare paid for each matching message used to produce new messages so that messages from detectors are paid DMShare times share paid for non-detector messages. Thus if DMShare is 1 (the default), the system acts as originally defined, i.e., pay the same for all messages used. If DMShare is 0.1, then the amount paid for each detector message matched is 10the bid is paid "up the chain."

GRCFrMC Generate Random Classifer, Fraction Not-Match Conditions.

When a classifier is generated by the GENRANDCFS ("generate random classifiers") command, the probability that its second condition will be a match or a not-match condition is controlled by the GRCFrNSL variable. In particular, the probability that a particular second condition with be a match condition is equal to the value of the GRCFrMC variable.

The default value for GRCFrMC is 0.95, i.e., 95 percent of second conditions are match conditions.

URndSd — Uniform Random Distribution Seed.

This variable contains the initial seed and the continuing value used to generate random integers. The other random distributions (e.g., uniform in 0 to 1) are generated using this number as well (directly or indirectly). See the files UTILITY.C and UTILITY.H for details.

C.3 I/O Control Variables

SaveInt — Save Interval.
SaveSyFN — Save System File Name.
SaveSyPa — Save System Path Name.
SaveFNum — Save File Number.

The entire state of the system is written to a file whenever

CycleStp % SaveInt == 0

(where % is the symbol for the modulus operator in C). If SaveInt is 0, then the system-state is not save automatically. The default value for SaveInt is 0, i.e., the system is not automatically saved.

This variable can be used to have the system-state written to a series of files or devices periodically during a run. The files are named:

SMDDHHMM.nnn

where "S" is a constant, and

M is the month of the run (in hex),

DD is the day of the run,

HH is the hour the run started,

MM is the minute the run started, and

nnn is 001, 002, and so on, one for each file saved during a run.

The file name up to the period is stored in the SaveSyFN variable. The nnn extension numbers are counted in the system variable SaveFNum. The files are written to the drive or directory specified in the SaveSyPa variable. The default for SaveSyPa is a null string, i.e., write the files to the active drive or directory.

Each "save" file contains values for almost all the system constants and variables, the current and old message lists, the classifier list, and the state of the environment at the end of the major-cycle during which the file was written.

The contents of these files can be analyzed and/or summarized to determine and describe what happened during a particular experimenmt.

The system can also be re-started using the contents of any one of these saved files: that is, the classifier system can be run and the contents of the file can be used to initialize the message list, classifier list, environment state, and so on, so that the run will be started in the same state as it was in when the save file was created.

AdCfInt Auto-Display Classifiers, Interval.
AdCfFmt Auto-Display Classifiers, Format.

The classifier list is automatically displayed whenever

CycleStp % SaveInt == 0

(where % is the symbol for the modulus operator in C). If AdCfInt is 0, then the classifiers are not displayed automatically.

The classifiers are displayed in the format specified by the AdCfFmt variable. For example, if AdCfFmt is 1, each classifier's conditions, action, strength and bidratio are displayed, and if AdCfFmt is 2, then each classifier's strength, bidratio, change-of-strength, number of times it bid, number of offspring, and so on are displayed.

The default value for AdCfInt is 0, i.e., auto-display is off. The default value for AdCfFmt is 2.

AdCMInt Auto-Display the Current Message List, Interval.
AdCMFmt Auto-Display the Current Message List, Format.
AdDMInt Auto-Display the Detector Messages, Interval.
AdDMFmt Auto-Display the Detector Messages, Format.
AdVarInt Auto-Display selected system Variables, Interval.
AdVarFmt Auto-Display selected system Variables, Format.
AdCycInt Auto-Display the CycleStp Variable, Interval.
AdCycFmt Auto-Display the CycleStp Variabel, Format.
AdEnvFmt Auto-Display the Environment state, Interval.
AdEnvFmt Auto-Display the Environment state, Format.

See the description of AdCfInt and AdCfFmt.

StdOutFN Standard Output File Name.
StdOFILE Standard Output FILE Pointer.

All user-prompts, error messages, etc., are written to the file/device specified by this (character string) variable. The variable StdOFILE is a FILE pointer for this device, which is always open.

The default value for StdOutFN is "std", i.e., the standard C output stream (normally the computer's screen).

DisOutFN Display Output File Name.

DisOFILE — Display Output FILE Pointer.

Whenever the message list, the classifier list, system variables, or the state of the enviroment is displayed by the standard CFS-C display mechanisms, the output is written to the file/device specified by this (character string) variable. That is, whenever a user enters a DISPLAY command, or whenever one of the items listed above is displayed by the auto-display facilities described above (see the AdCfInt variable), the output is written to the DisOutFN file/device.

The variable DisOFILE is a FILE pointer for this device, which is always open.

The default value for DisOutFN is "std", i.e., the standard C output stream (normally the computer's screen).

LogFlg — Log Flag.
LogFName — Log File.

All output written to the standard output file/device or to the display output file/device can also be written to a log file. This facility is most useful when the the standard and display output is being written on the computer's screen, but the user also wants to make a copy of the output from a run for later examination.

The default value for LogFlg is 0, i.e., no logging is done. To begin logging, enter

    SET LOG=lfile

where "lfile" is the name of the file/device to which the log is to be written. LogFlg will then be set to 1 (logging on) and LogFName will be set to "lfile".

CfInFN — Classifier Input File Name.

This variable contains the name of the file used as a default file for loading classifiers, i.e., it is the file used when the user doesn't include a filename on a LOADCFS command.

MsgInFN — Message Input File Name.

This variable contains the name of the file used as a default file for loading messages, i.e., it is the file used when the user doesn't include a filename on a LOADMSGS command.

DsCfLst — Display Classifier List control.

The parameter can be used to control what rules are displayed when the classifier list is displayed. The default is DsCfLst = 0, i.e., display them all. Other options are:

a — display only cfs with Strength ¿ Average.

b — display only bidders.

w display only bidders that won competition.

p display only winners that posted messages (after effector interface conflict resolution).

idlist display only those with specified id numbers (either one number or a list separated by commas).

C.4 Variables That Control the Discovery Algorithms

DscDemo Discovery, demonstration level.

This variable controls what demonstration information, if any, is displayed when the discovery mechanisms are activated. If the value is 0 (the default), nothing is displayed. If DscDemo is set to 1, some information is displayed, and if it's set to 2, even more details are displayed. (The subroutines must be compiled with the CTEST switch #define-d to 1 for DscDemo to be used.)

FrNewCfs Fraction of New Classifiers.

The maximum number of new classifiers allowed during one step is equal to (NmCfs * FrNewCfs).

NwCfStrV New Classifier Strength Version.
NwCfStrF New Classifier Strength Fraction.

These two parameters control the strength of new classifiers. If NwCfStrV is 0 (the default), the strength is just the average of the entire list times NwCfStrF (which has a default value of 1.0). If it is 2, the new strength is half way between the strength of the parents and the average for the entire list. If it is 3, the average of the parents is used.

CDMsg Cover Detector Messages Operator Control.
CDMsgRt Cover Detector Messages Operator Rate.
CDMsgAct Cover Detector Messages Operator Action Creation Control.

These variables control how the Cover Detector Messages Operator actts—see section 3.9.2 for details.

CEffs Cover Effectors Operator Control.
CEffsRt Cover Effectors Operator Rate.
CEffSplz Cover Effectors Operator Specialization Control.

These variables control how the Cover Effectors Operator works—see section 3.9.3 for details.

BkgGARt Background Genetic Algorithm Rate.

This variable controls how often the "background" discovery mechanism is activated. It specifies the probability that the GA will be applied on each step. The default value for BkgGARt is 0, i.e., the mechanism is disabled; BkgGARt of 1 means apply the GA every step.

PrntBPow Parent Fitness Bias Power.

BidFit Use Bids as Fitness.
PkPrnWOR Pick Parents without Replacement.
BgaBPPr Pick bidding parents only probability.

These parameters affect how parent rules are selected.

By default, the fitness of a classifier is based on its strength. In particular, it is its strength raised to the power PrntBPow. Thus setting PrntBPow > 1 biases the selection of parents toward high strength rules, favoring exploitation over exploration (and perhaps leading to fixation or premature convergence problems).

If BidFit is 0 (the default), strength is used as a "fitness" measure when selecting parents and deleting rules. If BidFit is set to 1, then an estimate of a classifier's bid, its strength times *bid_k*, is used as fitness when selecting parents (and when deleting rules).

PckPrnWOR controls whether or not the GA picks parents without replacement. If PkPrnWOR is 0, a rule can be selected as a parent more than once in one step; if it is 1, then a rule can be selected as a parent only once during a step.

BgaBPPr is the probability that parents are selected from *bidding* parents (rather than from the entire list), using *actual* bids as the fitness measure. The default for BgaBPPr is 0, i.e., parents are chosen from the full list as described earlier. If BgaBPPr is say 0.5, one half the time the GA is applied, parents will first be chosen from classifier that bid during the current step; if not enough are available, then parents are chosen as usual from the entire list.

Choosing only from bidding classifiers introduces a kind of *restricted mating* scheme [Booker, 1982]. If fact, it is in some ways a very natural restriction, since presumable rules bidding at the same step are all working on the same "subproblem."

RandRplc Random Replacement Control.
RplcBPow Replacement Biasing Power.
RplCfUBd Replacable Classifiers, Upper Bound on Strength.
RplACUBd Replacable Active Classifiers, Upper Bound on Strength.
RplACSBd Replacable Active Classifiers, Active Step Bound.

These variables (and CrowdFac) control how rules are selected to be replaced.

If RandRplc is 0, classifiers are chosen for replacement with probability proportional to the inverse of strength (or the inverse of their bid if BidFit is 1). In particular, the probability that classifier i is chosen for replacement is:

$$Prob_i = (1/Strength_i)^{RplcBPow} / \sum (1/Strength_j)^{RplcBPow}$$

where the sum is over all classifiers *eligible* for replacement (more on eligibility below). However, classifiers with strength ≤ 0 are always chosen first (unless they are ineligible). The default for RplcBPow is 1; larger values bias replacement to weaker rules.

Some rules are not eligible for replacement:

1. Rules that have the NoRplFlg set TRUE in their CfNode are never chosen for replacement.
2. Rules created during the current step are never replaced.
3. Rules that have a strength greater than the average strength (AveCfStr) times RplcUBd. RplcUBd default is 100, so than in effect no rules are protected. If RplcUBd is set to 1, then above average strength rules are protected.
4. Recently *active* rules can also be protected. Recently active is defined as any rule that has *produced* a message with the last RplACSBd steps; thus if RplACSBd is 0, no rules are protected because they were active. If RplACSBd is 10, the rules with $StpLPrd > CycleStp - RplACSBd$ are protected if they have strength greater than AveCfStr times RplACUBd (where $StpLPrd$ is the CycleStp in which the rule last produced a message).

If RandRplc is 1, then rules are chosen with equal probability. Again rules with strength ≤ 0 are chosen first, and rules with NoRplFlg set TRUE are never chosen. However, rules with strength above values specified by RplCfUBd or active rules with strength above RplACUBd are *not* protected.

If RandRplc is 2, then rules are chosen as with RandRplc 1, except rules *are* protected from replacement by the RplCfUBd, RplACSBd and RplACUBd variables. Care must be taken not to try to protect too many rules—this will cause the system to thrash in its search for rules to replace.

CrowdFac Crowding Factor.

By default CrowdFac is 1, in which case one rule is chosen to be replaced for each new rule to be injected into the population. If CrowdFac > 1, then CrowdFac rules are chosen for each new rule to be injected, and the rule actually replaced is selected from this group. The one selected is the one genotypically most like the new rule to replace it.

The idea here (first described by [DeJong, 1975]) is to control the spread of high fitness rules. If there are many copies (or close variants) or some high fitness rule and if another copy (or new variant) of this rule is the new rule to be injected, there is a good chance it will just replace a rule much like it, since one of them is likely to be in the group selected when CrowdFac > 1. That is the idea, at least; no one has really shown it works yet, though.

BgaSCPr Background Genetic Algorithm, String Crossover Probability.
BgaFCPr Background Genetic Algorithm, Full Crossover Probability.

BgaNCPr Background Genetic Algorithm, No Crossover Probability.

These three variables control how crossovers are done. The probability that crossover is done within a classifier-part is *BgaSCPr*, the probability that the full classifiers are treated as strings to cross is *BgaFCPr* - BgaSCPr, and the probability that no crossover is done is 1.0 - BgaSCPr - BgaSCPr. (That is, these three parameters define a cummulative probability distribution if values are assigned properly.)

MuPrTot Mutation Probability, Total.

This variable is the mutation rate used to control the mutation discovery mechanism. Each new classifier will be mutated at zero, one, two, etc., loci, the exact number controlled by a poisson distribution with a (default) mean of 0.05.

MuPrCnd1 Mutation, Probability in left condition string.
MuPrCnd2 Mutation, Probability in right condition string.
MuPrAct Mutation, Probability in the action string.
MuPrCT2 Mutation, Probability in right condition-type.
MuPrAT Mutation, Probability in action-type.

These five variables define a cummulative probability distribution over the five parts of a classifier that can be mutated. When a classifier is chosen to be mutated, the part to be mutated is determined by making a random draw from this distribution.

The default distribution is:

MuPrCnd1	0.30
MuPrCnd2	0.60
MuPrAct	0.90
MuPrCT2	0.95
MuPrAT	1.00

Note that if one of these variables is changed, some other one (or more) must be changed to maintain a valid distribution.

MuFrNSL Mutation, Fraction Non-Specific Loci generated.

When a condition or action string is chosen to be mutated, one loci is chosen to be mutated at random with each loci having an equal probability of being mutated. Once a loci is chosen, the new value (a 0, 1, or #) at that location is determined by making a random draw from a distribution defined by the MuFrNSL variable. In particular, the value of MuFrNSL is the probability that the new value will be a # symbol; if the new value is not a #, it is changed to a 1 or a 0 with equal probability.

The default value for MuFrNSL is 0.2, i.e., 20 percent of the mutations are to # symbols, 40 percent are to 1's, and 40 percent are to 0's.

CSSRt	Couple Stage-Setters Rate.
ACPCt	Couple Profitable Classifiers, threshold.
TagLMx	Tag-region Left Maximum locus.
TagRMx	Tag-region Right Maximum locus.
CSSFrM	Couple Stage-Setters, Fraction Memory Rules.
CSSFrRf	Couple Stage-Setters, Fraction Reflex Rules.
CSSNwStr	Couple Stage-Setters, New Classifier Strength.

These variables control the Couple Stage-Setter (CSS) Operator, one of the "triggered chaining" operators in CFS-C. mechanism to create linked rules. See section 3.9.4 for more details.

D How to Change the CFS-C System

This appendix describes how to make changes to several basic parts of the CFS-C package, including how to add user-commands, how to add system variables, how to change basic attributes of the CFS-C classifier system (e.g., the length of message strings), and so on. Note that this appendix assumes you have read the preceding parts of this document, including appendices A through C.

D.1 How to Install a New User-Command

The subroutine GUserCmd(), which is defined in the file USERCMD.C, prompts the user for a command, determines if that command is legal, and if it is, returns a command code for it (an 'unsigned int' unique to each command). GUserCmd() also places any parameters entered with the command in a local buffer. The command code is used to determine which subroutine to call to actually execute the specified command. The address of the local buffer that contains any command parameters is passed to the subroutines that execute user commands.

It is quite easy to add new commands to the system. The association between command names and codes is defined in the files USERCMD.H and USERCMD.DEF. The association between command codes and the subroutines that execute the commands is made by a "switch" statement in the main() subroutine in the file CFSC.C.

To add a new user command, do the following:

1. Define a command code for the command.

   Use a #define statement to define a macro-name for the code in the file USERCMD.H. By convention, the macro-names for command codes end in "_CC" (e.g., STOP_CC, SET_CC, etc.).

2. Define a name for the command and associate it with the command's code.

   Each occurence of the structure CmdNode [see USERCMD.DEF] in the table (array) Commands[] associates a command name to a code. Note that:

   (a) There may be multiple names for each command.

   (b) There may be a minimum abbreviation defined for each name.

   Be sure the names (and their abbreviations and synonyns) are unique.

3. Write a subroutine to actually execute the command.

   Note that this subroutine should have one formal parameter of type "pointer to char". The value passed to the subroutine will be the address of a "char" string buffer. This buffer will contain any command parameters entered by the user. That is, the bufffer will contain any characters entered by the user on the same line as the command (NOT including the command-name itself).

4. Insert a call to the subroutine in the command-processor loop.

This involves inserting a case in a "switch" statement in the main() subroutine (see the file CFSC.C). The switch is conditional on the command code returned by GUser-Cmd(), and each "case" just calls the appropriate function.

5. Insert a brief description of the command in the Help() subroutine (see the file USER-CMD.C).

That's all there is to it (although you might test it a little, too)!

D.2 How to Define a New System Variable

All user-settable variables for the CFS-C system are listed in a table implemented as the RTVAdd (Run-Time Variable Addresses) table. The same variables must also appear in the file INIT.CFS, which is read when the CFS-C system is started. The variables in that file and table:

1. Are all saved when the system state is saved (e.g., by the SAVESYS command).

2. Can have their default values automatically overridden when the system is started.

3. Can have their values displayed by the DISPLAY command.

4. Can have new values assigned to them at run time by using the SET command.

This table-driven approach should make it easy to add, change, or remove system variables, since you won't (usually) have to change code here and there to make each new variable DISPLAY-able, SET-able, automatically saved and reloaded, and so on.

Note that there are some values that define the CFS-C classifier system that are not variables (e.g., the size of message strings, STRNGSZ). These constants are defined in the files CORE.H, CFOPS.H, and CFSIO.H. Also note that variables that define the state of a particular enviroment are not listed in the RTVAddNd Table.

To add a new system variable involves the following steps:

1. Define the variable in one of the files that end with the extension ".DEF".

   You will notice that I have tried to keep variable names to 8 or fewer characters: I know this is archaic, but to make the code as portable as possible before the ANSI standards, this was necessary. I am slowly trying to change to longer names.

2. Assign a unique code to the variable and add an entry to the RTVAdd table in USERCMD.DEF.

   Each entry includes the code for the variable and its address. Thus you will also have to make the variable addressable in USERCMD.DEF using an extern.

3. Add a line to the INIT.CFS file for the variable.

   Each line contains the code for the variable, a name for it (and maybe an alias) to be used on SET and DISPLAY commands, an indication of its type (short int, int, unsigned int, float, char), an flag to mark it as one to load from a save file, for numeric variables, a minimum and maximum value, and an initial value. See that file for examples of existing variables.

This file is read by InitVT() in CFSC.C. If you make a mistake in the table or this file, you will get and error message at start-up time from this subroutine.

Why do I load from the INIT.CFS file? The reason is that I want to allocate space for the full variable table from memory via malloc, for those compilers that limit the amount of global storage compiled (e.g., MPW C 2.0 or less).

That's all there is to do define a new system variable. The Set(), Display(), and related subroutines will do all the rest. That is, that's all there is to do UNLESS the variable is a "special type" variable that requires special processing when new values are assigned to it.

For example, the variable LogFName, which is a string variable that contains the name of the file to use for logging output, requires special processing when a new value is assigned to it, e.g., if there already is another log file it must be closed, and then the new file must be opened if possible, etc. For more details about this and other "special-type" variables, see the Set() subroutine (and others that it calls) defined in the file USERCMD.C.

D.3 How to Change the Basic String Length, the Classifier List Size, and Other Basic Characteristics of the CFS-C Classifier System

The file CORE.H defines a number of basic system constants. These constants are used throughout the CFS-C code, and in general will be used in some of the code that defines any particular enviroment. Thus the file CORE.H should be included (by an #include statement) in all source files.

The following table shows some of the marcos defined in the file CORE.H:

Name	Default	Meaning
STRNGSZ	16	Size of message strings, classifier conditions, etc.
CFLSTSZ	200	Maximum number of classifiers on the classifier list.
MSGLSTSZ	32	Maximum number of messages on the message list.
DMSGSSZ	8	Maximum number of detector messages at one step.
EFFLSTSZ	10	Maximum number of effectors.

To change the values for these constants, just change the value assigned to the appropriate macro-names defined in CORE.H, and then re-compile and re-link the entire system. All the CFS-C subroutines should still work properly. The only problems you may have is that the default formats provided for displaying classifiers, messages, and so on, may not look right (e.g., because classifiers will not fit on one line).

Note that any domain-dependent code you implement should make use of these macros if you want to be able to change these values and have your code continue to function! In particular the code to load and display effectors, the code to construct detector messages, and the code that interprets messages to activate effectors should be written using these macros.

Also note that you should be sure the macro-name INTSZ (defined in the file COMPILER.H) has as its value the size (in bits) of the "int" data type on the machine you are using. This value is used, along with the STRNGSZ value define in the file CORE.H to assign a value to INTPRSTR ("int per string"), i.e., the number of "int" array entries used to store one message string, one classifier condition, or one action string.

D.4 How to Add or Remove Variables Associated with Message or Classifier Nodes

Each classifier and each message is stored in a CfNode and a MsgNode, respectively (these structures are defined in the file CORE.DEF). Besides the message string, each MsgNode contains other information about the message, .e.g., the message intensity (how much was bid to produce it), which classifier produced it, and so on. Similarly, each CfNode contains a lot of information besides the classifier conditions and action strings, the strength, etc. For example, there are a number of members of CfNode's that are used to keep track of the classifier's history, e.g., how many times it has bid, how many time it produced new messages, how many offspring have been produced from it and so on.

While the structure of the MsgNode is relatively stable, it frequently is useful to add new members to the CfNode structure to keep more information about each classifier's history of activity. This section describes the steps that should be taken when a new member is added to the CfNode structure. Note that similar steps must be taken when changes are made to the MsgNode structure.

First, the basic definition of the CfNode must be changed. This definition is in the file CORE.DEF. At the same time it is a good idea to change the "extern" declaration of the CfNode in the file CORE.EXT. After changing these two files, recompile ALL the CFS-C source code files. However, before doing that, there are some changes that must be made to some subroutines:

1. The subroutines Init_ CFS() and InitCfLs() initialize the classifier list and all the (initially empty) CfNodes in the list (see the file CFSC.C). Sometimes changes will have to be made to these subroutines. For some new CfNode members is may also be necessary to make changes to the subroutines StartStp() and EndStp(), which are executed at the start and the end of each major-cycle step, respectively (these subroutines also are defined in the file CFSC.C).

2. Classifiers are loaded by the StoreCf() and AddCf() subroutines, and classifiers are displayed by the PutCNd() subroutine, all defined in the file CFMSGIO.C. Therefore, for almost all new CfNode members, the PutCNd() subroutine must be changed to display the value of a new CfNode member when classifiers are saved (i.e., the display format value is SAVEFMT), and the StoreCf() and AddCf() subroutines must be changed to read in and store the value saved. Note the changes must be coordinated so that PutCNd() writes the value in the same place that StoreCf() tries to read it from!

   Of course if the new CfNode member is to be displayed in other formats, other changes will have to be made to the PutCNd() subroutine.

3. The subroutine GRndCfs(), which generates a list of random classifiers when the user enters the GENRANDCFS command, will also have to be changed, e.g., to initialize the new CfNode member.

Finally, ou will of course want to make changes to actually USE the new CfNode member, e.g., to increment the member value if it is a simple counter of some event.

D.5 How to Define a New Classifier Action-Type

Every classifier has an *action-type* (sometimes called a *classifier operator*). A classifier's action-type, along with its action-string, defines what message the classifier will generate, given one or two "input" messages, i.e., messages that match the classifier's conditions.

The most common action-type is the traditional pass-through (PASS) action. Other action-types defined in the CFS-C system are:

AND logical AND two messages.

OR logical OR two messages

XOR logical exclusive OR two messages.

ADD1 add 1 to the message matching the left condition.

SUB1 subtract 1 from the message matching the left condition.

The above action-types are NOT meant to be a definitive list; rather, they are meant to be suggestive of the types of action-types one might define. (They also happen to be operators that are useful in almost any domain.)

In the CFS-C system, an action-type is defined by:

1. An action-type name (e.g., PASS).

2. An action-type code number (an unsigned integer).

3. A subroutine that implements the action-type.

The associations between action-type names, codes, and subroutines are defined in the files CFOPS.H, CFOPS.DEF, and CFOPS.C.

To add a new classifier action-type, do the following:

1. Define an action-type code.

   Use a "#define" statement to define a macro-name for the code in the file CFOPS.H. By convention, the macro-names for action-type codes begin "OC_" (e.g., "OC_PASS").

2. Write a subroutine to actually execute the action-type.

   Subroutines for the action-types supplied with the CFS-C system are all defined in the file CFOPS.C, so new subroutines should probably be defined in that file as well.

   A classifier action-type may be thought of as an operator on:

   (a) one classifier (the action-string part of it), and

   (b) one (or sometimes two) message-strings, to produce a new message string.

Thus, an action-type subroutine must have four formal parameters:

CfPtr: pointer to the CfNode (classifier node) that contains the classifier action-string.

Msg1Ptr: pointer to a MsgNode (message node) that contains the message string that matched the left-condition of the classifier pointed to by CfPtr.

Msg2Ptr: pointer to a MsgNode (message node) that contains the message string that matched the right-condition of the classifier pointed to by CfPtr. (This is NULL if there is no second message string, i.e., if the second condition is a not-match type.)

NewMPtr: pointer to a MsgNode into which the new message string is to be placed.

Each action-type subroutine must extract the appropriate strings (i.e., the input message strings and the classifier action string) from the nodes pointed to by the parameters. The subroutine must also store the new message string in the appropriate part of the MsgNode pointed to by NewMPtr. (See the file CFMSGIO.C for subroutines that can be used to extract and store message and action strings from MsgNode's and CfNode's.)

3. Define a name for the action-type, and associate it with the action-type's code number and subroutine.

   Each occurrence of the structure "CfOpNode" (see the file CFOPS.DEF) in the table (array) "CfOps[]" associates an action-type code, name, and function. Each CfOPNode includes the following members:

   CfOpName: The name of the action-type. (Four-character names, padded with blanks if necessary, are suggested–to make the displays nicer.)

   CfOpCode: the action-type code defined in CFOPS.H. Be sure this and the name are unique, of course!

   CfOpfn: pointer to the subroutine that implements the action-type.

   CfOpCPrb: A value that, along with the CfOPCPrb values stored in all the other CfOps[] nodes, defines a cummulative probability function over the action-types. This function is used to generate random samples of classifiers. (See the GRndCfs() subroutine in the file LEARN.C.)

   Note that the CfOpCPrb values must increase (or at least not decrease) from the beginning nodes in the CfOps[] array to the end, and the value in the last node must be 1.00.

   Also note that there must be a statement of the form:

   ```
   extern int COpXXXX();
   ```

   where "COpXXXX" is the name of the subroutine that implements the action-type being added (so that the pointer to the subroutine stored in the CfOps table can be resolved.)

That's all there is to it. Once an action-type is defined in this way, it can be used by:

1. Including its name in the appropriate place when describing classifiers to be loaded from a file,
2. Allowing the action-type to be generated when a list of random classifiers are generated (see the GENRANDCFS command), or
3. Implementing rule discovery subroutines that can genreate the action-type.

Note that by convention the names of the subroutines that implement action-types begin with "COp" (e.g., COpPASS, COpAND, etc.)

Note that a number of issues must be addressed when defining new action-types. A few of these issues are:

1. Should the pass-through action be applied after other actions?
2. How many and what action-types should be defined?
3. How to handle exception conditions, e.g., arithmetic overflows, action-types that require two messages having only one input message to work with (i.e., when the second condition is a not-match condition), and so on?
4. How should the systems discovery mechanisms create and manipulate action-types?

More experience with classifier systems that support arbitrary action-types will undoubtedly raise more issues and questions.

E Technical Notes

This appendix contains technical notes on the CFS-C system, including descriptions of the basic data structures used to implement the message lists, classifier list, and so on, and a brief description of the subroutines that implement the major-cycle of the CFS-C classifier system. A brief description of the method used to store messages, conditions, and actions, and to compare messages to conditions is also included.

NB: I have tried to keep this up-to-date as I change the system, but if there is some question, be sure to see the code or contact me, as these descriptions may be out of date.

E.1 CFS-C Data Structures

This section briefly the main data structures used to implement the CFS-C message lists, classifier list, match list, and effectors. These data structures are all defined in the CORE.DEF file.

E.1.1 The Message Lists

There are three message lists maintained by the CFS-C classfier system:

1. The current message list, which contains messages posted by classifiers active during the previous major-cycle step (along with any messages added by the system's detectors).

2. The new message list, which contains messages posted by the classifiers active during the current major-cycle step.

3. The old message list, which contains the messages that were on the current list during the previous major-cycle.

Each list is implemented as an array of MsgNode structures, one structure for each message. See the CORE.DEF file for a description of these structures.

Three pointer variables are used to keep track of the three message lists: CurMsgs, NewMsgs, and OldMsgs. Thus at the end of each major-cycle, when the current message list becomes the old list, and the new message list becomes the current list, the pointer variables are just changed to point to the appropriate arrays. The variables NmCMsgs, NmNMsgs, and NmOMsgs indiciate the number of messages in the current, new, and old messages lists, respectively.

When new messages are posted, they are stored in the array pointed to by NewMsgs, in order as they are generated. The int variable LastNMsg indicates the end of the new message list. (Similary, the variables LastCMsg and LastOMsg indicate the end of the current and old message lists.) That is, before any new messages are posted, NmNMsgs is -1, to indicate an empty list. As each new message is generated, NmNMsgs is incremented by 1. Thus all new messages are stored in entries 0 to LastNMsg of the array pointed to by the NewMsgs variable.

However, note that some messages in the current messages list are marked as deleted messages (i.e., the IsMsg member of the MsgNode is FALSE). These are messages that were were posted but then deleted by the conflict-resolution mechanism used when more than one message matches the same effector. Thus LastCMsg does not necessarily equal NmCMsgs - 1.

E.1.2 The Classifier List

The classifier list is implemented as an array of CfNode structures, one structure for each classifier. See the file CORE.DEF for a description of these structures.

The NmCfs variable contains the number of classifiers in the list. The variable CurCfs points to the first CfNode that contains a classifier, and the rest of the classifiers are stored in the next NmCfs - 1 CfNodes in the array. Thus when the messages are compared to classifiers to determine which classifiers are matched, the CfNode's can be processed simply by incrementing a pointer through memory.

The variable NxtFrCf points to the next free CfNode, i.e., the next one that doesn't contain a classifier. Thus when a new classifier is added to the list, it should be stored in the CfNode pointed to by NxtFrCf, and that pointer should be incremented. The variable NxtCfId is assigned as the id of a classifier when it is added to the list, and the NxtCfId variable is incremented. (The subroutine StoreCf() defined in the file CFMSGIO.C should be called to add new classifiers.)

As mentioned above, the CfNode's are also accessable through a 2-way linked list in which the classifiers are sorted by strength. In particular, the variable StrSrtCf (strength sorted classifiers) points to the lowest strength classifier, and the variable HiStrCf points to the highest strength classifier. In each CfNode, the member NxtStrCf points to the classifier with the next-highest strength, and the member PrvStrCf points to the classifier with the next lowest strength. This sorted classifier list is used to display classifiers, so it is resorted when classifiers are displayed.

During each major-cycle step, each classifier that has its condition part satisfied is added to a list of candidate classifiers, i.e., classifiers that may post messages during this cycle step. The variable CandCfs points to the CfNode for the first candidate classifier. The member NxtCand in each CfNode points to the CfNode containing the next candidate (NxtCand is NULL in the CfNode for the last candidate). The variable NmCandCf contains the total number of candidate classifiers in the list.

E.1.3 Matches

Each classifier (CfNode structure) has two bit-strings, each with MSGLSTSZ bits (one per possible message). When a message matches the first condition of a classifier, the corresponding bit in the first bit-string is set to 1. Similarly, when a message matches the second condition, a bit is set in the second bit-string. The number of matches matching the first and second conditions are also recorded, as is the support for the classifier (as it was before).

A classifier is a candidate for producing new messages just when one or more bits in the first bit-string are set to 1, and for second conditions that are 'match' type, one or more bits in that bit-string are set on (or for a not-match condition, none are set on).

When a classifier is chosen to produce messages, the bit-strings are examined to see which messages to use as input to the classifier's 'action'.

NOTE:I'm not happy with the implementation of support accumulation: the difficulty comes because a message adds its intensity only once to a classifier, even if it matches both conditions.

I'm also not happy with the way I am re-initializing support and the NmMMC1 and NmMMC2 counters, i.e., its re-initialized for all classifiers every step. But since some classifiers that have matches for one condition but not the other will not be on the candidate classifier list, I can't think of any easy way to know which to re-initialize and which not to.

E.1.4 Effectors

In the CFS-C system, effectors consist of two basic parts:

1. A condition, which is just like a single condition for a classifier (i.e., it defines a set of messages). The effector is activated only when its condition is satisfied, i.e., when there is a message on the new message list that matches the condition string (or, if the condition is a not-match condition and if there are no messages that matches the condition string).

2. A action, which is defined by a subroutine that implements the the effector's action. The effector subroutine is called with a single parameter that is the message that satisfies the effector's condition.

Effectors are stored as an array of EffNode structures, one structure for each effector. The NmEffs variable indicates the number of effectors defined. The variable EffLst points to the EffNode for the first effector, and the others effectors are stored in the next NmEffs-1 entries of the array.

Each EffNode contains members to store the effector's condition and to store a pointer to the subroutine that implements the effector's action. Each EffNode contains members that serve as counters to keep track of the activity of the effector during a run, e.g., how many times the efffector has been matched and activated.

Each EffNode also contains a member EffMtch, which points to a list of messages (i.e., MsgNodes) on the new message list that match the effector's condition during a given major-cycle step. Each MsgNode contains a member NxtEMsg that points to the next message that matches the effector, and the last message that matches the effector's condition has a value of NULL stored in its NxtEMsg member. The EffNode member NmEMtch indicates how many messages are on this list. If NmEMtch is greater than 1, then a conflict-resolution mechanism is invoked to select one message on the list to be the message that activates the effector during the current step. (The other messages that match the effector are deleted from the new message list–for details, see the description of the GenBehav() subroutine later in the appendix.)

E.2 The CFS-C Major-Cycle

The CFS-C major-cycle is implemented in the Classify() subroutine, which is defined in the file CFS.C. The Classify subroutine in called with one parameter, an integer number that indicates the number of major-cycle steps that should be executed before returning control to the user.

The following is a brief description of each subroutine called during one major-cycle step. For a complete description of these subroutines, see the comments associated with the subroutines in the source-code files.

StartStp(): Start Step Initialize variables used during a major-cycle step (e.g., NmCandCf, NmMtchs, etc.). Also re-initialize classifiers that were candidate (matched) classifiers during the previous major-cycle, and set flags in each MsgNode on the current message list to indicate that the messages have not yet been matched or used to produce messages during the current major-cycle.

RdDetect(): Read Detectors This is the interface between the CFC-C system and the domain-dependent subroutines that act as detectors. Add any messages generated by the GetDMsgs() (get detector messages) subroutine to the current message list.

GenCands(): Generate Candidate Classifiers and Matches Compare all classifiers to the current message list and generate a list of matched classifiers (the candidate classifier list). Mark messages matched by each classifier, accumulate support for each candidate classifier, and accumulate the total support for all candidates.

GenNMsgs(): Generate New Messages Calculate bids for all candidate classifiers. If the number of messages those classifiers will generate is less than or equal to the maximum allowed (indicated by the NmIMsgMx variable), just generate those messages and add them to the new messages list. If the number of messages the candidate classifiers will generate is greater than NmIMsgMx, then run a competition to select winning classifiers, i.e., candidate classifiers that are selected to post messages this step. Select winning classifiers until the messages they generate fill the new messages list to the allowed limit. The winning classifiers are selected probablistically; the probability that a classifier will win is proportional to the square of the bid it made.

GenBehav(): Generate Behavior Compare new messages to each effector's conditions, and link those messages that match an effector to the EffNode that contains that effector. If an effector's condition is satisfied, first call a subroutine to decide which action to take, and then call the subroutine that implements that effector's action. After all effectors have been checked and perhaps activated, call the subroutine GtSysRew() (get system reward), which implements the environment-specific payoff function.

Note that messages that are deleted by the conflict resolution mechanism (i.e., messages that suggest actions not consistent with the action actually chosen) are deleted from the new messages list by setting the IsMsg member to FALSE in the MsgNode's that contain those messages.

UpdCfStr(): Update Classifier Strength Update the strength of all classifiers. Apply taxes and the bucket-brigade, transfering strength from any reward received and from classifiers to other classifiers.

Discover(): Discover New Classifiers Test the trigger conditions that control how often the discovery mechanisms are to be activated, and execute any mechanisms that should be activated during this major-cycle step.

EndStp(): End Step Discard the old message list, make the current message list the old list, make the new message list the current list, and empty the new message list.

E.3 Representation of Message, Condition, and Action Strings.

This section briefly describes how message, condition, and action strings are implemented in the CFS-C system.

E.3.1 Storage

The basic problem to solve when storing classifier condition and action strings is that each locus in these strings may have one of three values, namely a 1, 0, or #, which means a condition of length k can't be stored as a k-bit binary value. A related problem, given a storage strategy, is how to rapidly compare condition strings to message strings to determine whether or not the condition is matched by the message.

In the CFS-C system, messages are just stored as k-bit binary strings, one bit per string locus. Condition and action strings are stored in two parallel k-bit binary strings. Basically, one of the parallel arrays indicates the specific loci (the loci with 0's and 1's) and the other array acts as a mask that in effect marks the non-specific loci (i.e., those with #'s). As shown below, this storage mechanism makes it possible to compare a message to a condition using two fast logical operators; further, the comparison can be done b loci at a time (where b is the size of the arithmetic registers in the machine being used).

In particular, a k-loci message is stored as an array of int (or long int) variables. For example, if each int is 16 bits long, and message strings are 32 loci long, each message is stored as an int array with two entries.

A k-loci condition is stored as two int arrays, for example one called CondBits[] and the other CondMask[]. Each locus of a condition string is stored as two parallel bit values, one bit in the CondBits[] array and one in the CondMask[] array. The condition locus value is mapped into binary values as shown by this table:

condition value	CondBits value	CondMask value
0	0	1
1	1	1
#	0	0

Similarly, a k-loci action is mapped into two int arrays, for example one called ActBits[] and the other called ActMask[]. Each loci of a action string is stored as two parallel bit values, one bit in the ActBits[] array and one in the ActMask[] array. The action locus value is mapped into binary values as shown by this table:

action value	ActBits value	ActMask value
0	0	0
1	1	0
#	0	1

For example, condider the following macro's (which are actually defined in the file CORE.H):

Macro	Meaning	File
INTSZ	16	Size of int variable
STRNGSZ	32	Num. of loci in msgs, conditions, action
STRNGMX	(STRNGSZ-1)	Remember, C arrays are counted from 0!
INTPRSTR	(STRNGSZ/INTSZ)	Number of int's needed to represent one string.
LOCIPRI	INTSZ	Number of loci represented by an int variable.
LSB	01	'least significant bit'

Note that for the values defined above, INTPRSTR is 2–it takes two ints to store a message. Thus a array to store a message could be defined:

```
int Message[INTPRSTR];
```

i.e., an array consisting of 2 int's. A condition could be stored as:

```
int CndBits[INTPRSTR];
int CndMask[INTPRSTR];
```

and an action could be stored in these arrays:

```
int ActBits[INTPRSTR];
int ActMask[INTPRSTR];
```

The file CFSUTIL.C contains several subroutines that convert the ASCII-form of message, condition, and action strings into binary form. I leave it as an exercise for the reader to convince him/herself that the above code transforms condition and action strings into parallel binary arrays (and vice versa) as described above. I also leave it as an exercide for the reader to figure out how the matching is done in CORE.C, and how new messages are created. Note that since messages, conditions and actions are perhaps stored as multiple entry arrays, each match and generate algorithm must loop over all the entries of the parallel arrays.