1992

An application of classifier systems to the reduction of finite state machines

Bijan Marjan
Lehigh University

AUTHOR: Marjan, Bijan

TITLE:

An Application of Classifier Systems to the Reduction of Finite State Machines

DATE: May 31, 1992

AN APPLICATION OF CLASSIFIER SYSTEMS TO THE REDUCTION OF FINITE STATE MACHINES

by

Bijan Marjan

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1992

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Date: 5/12/92

[illegible]

Advisor in charge,

S. David Wu

Electrical Engineering and Computer

Science Department Chairperson,

Kenneth K. Tzeng

ACKNOWLEDGMENTS

I would like to thank all those who inspired and aided me in completing this paper. First of all, I am grateful to Dr. S. David Wu, my advisor, for having spent much time with me, giving me numerous suggestions and hints on how to proceed. I would also like to thank Rick L. Riolo for having generously provided his software for my usage and allowed me to consult him when I had difficulties. My thanks are also extended to Minho Jo and Mona A. El–Gayar for all the ideas and support they gave me on possible applications. I also owe many thanks to the Lehigh University Interlibrary Loan Service for all the patience they had with me in obtaining materials that I needed. Special thanks to my parents and brother Alessandro for all the support, both financial and moral, they have given me, allowing me to pursue my education. Finally, I would like to thank all those whose names I have not mentioned, but who have greatly helped in writing my thesis.

TABLE OF CONTENTS

ABSTRACT **1**

CHAPTER 1 INTRODUCTION **2**
1.0 Adaptive Systems 3
1.1 Classifier Systems Applications 7
1.3 Classifier Systems and other Artificial Intelligence Systems 9
1.4 Classifier Systems and Finite State Machines 11
1.5 Classifier Systems, Finite State Machines, and Communication Protocols 11

CHAPTER 2 CLASSIFIER SYSTEMS **14**
2.0 Classifier Systems and Induction 14
2.1 Elements of Classifier Systems 16
2.1.1 Types of Classifier Systems 17
2.1.2 The Rule–Base 20
2.3 Schematas and Mental Models 21
2.3.1 Schemata and Building Blocks 22
2.3.2 Mental Models and Q–Morphisms 24
2.4 The Classifier System Major Cycle 28
2.5 Modifying the Classifier List 32
2.5.1 Credit Assignment: the Bucket Brigade Algorithm 32
2.5.2 Taxes 35
2.6 Genetic Algorithms: The Discovery Method 36
2.6.1 Genetic Algorithms and their Operation 37
2.6.2 Other Operators that Complement the Genetic Algorithm 41
2.6.3 Genetic Algorithms and Appropriate Usage 43
2.7 Parallelism and Emergence 44
2.7.1 Emergence 44
2.7.2 Parallelism 46
2.8 Trade–Off Between Knowledge and Search 47

CHAPTER 3 FINITE STATE MACHINE REDUCTION AND COMMUNICATIONS PROTOCOL APPLICATION **50**
3.0 The Finite State Machine World 51
3.0.1 The Markov Process 51
3.0.2 The Learning Process 54
3.1 The Finite Space Reduction Task 56
3.1.1 The Reduction Task Approach 56
3.2 The Protocol Environment 57
3.2.1 The Protocol Specifications 58

3.2.2 Reduction of the Protocol 59

CHAPTER 4 EMPIRICAL RESULTS FOR THE FINITE STATE MACHINE REDUCTION TASK 62

4.0 Testing Procedures for the Finite State Machine Task 62

4.0.1 Preliminary Tests 63

4.0.2 Parameter Settings 65

4.0.3 Results with Modified Parameter Settings 67

4.0.4 Selecting the best Parameter Settings 71

4.0.5 Using Different Heuristics 75

4.1 Using the Classifier System Appropriately 77

4.2 Protocol Analysis 79

4.2.1 Protocol Reduction 79

4.2.2 Protocol State Convergence 83

4.2.3 Perturbation and its Effects 84

4.2.4 Rewards 86

CHAPTER 5 CONCLUSION 90

5.0 Results 90

5.1 Communication Protocols Used as the Environment 92

5.2 Classifier Systems Revisited 93

REFERENCES 96

APPENDIX A PROTOCOL ENVIRONMENT SPECIFICATION 104

APPENDIX B THE CLASSIFIER LIST 116

VITA 125

LIST OF TABLES

Table 4.1 Parameter Settings for the Classifier System 63

Table 4.2 Test Set for FSM 66

Table 4.3 Payoff Values for FSM States at Major Cycle 12800 Using Different Values for mutprop, bid_k, and crowdfac ... 68

Table 4.4 Deviations from the Mean of Payoff Values for the FSM at Major Cycle 12800 70

Table 4.5 Best Parameter Settings 71

Table 4.6 Parameter Settings for FSM 1 Problem 1, Including Deviations from the Mean 73

Table 4.7 Results of Discriminatory Analysis for Different Parameter Settings and State Sets from FSM 1 Problem 1 ... 74

Table 4.8 Results of Discriminatory Analysis for all Tests 75

Table 4.9 Gap Sizes for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 327] for Payoffs and [0, 24] for Transitions 76

Table 4.10 Gap Sizes for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 250] for Payoffs and [0, 18] for Transitions 76

Table 4.11 Gap Sizes for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 150] for Payoffs and [0, 12] for Transitions 77

Table 5.1 Gap Sizes as a Percentage of the Range of Deviation of Payoff Values for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 327] for Payoffs and [0, 24] for Transitions 91

Table 5.2 Gap Sizes as a Percentage of the Range of Deviation of Payoff Values for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 250] for Payoffs and [0, 18] for Transitions 91

Table 5.3 Gap Sizes as a Percentage of the Range of Deviation of Payoff Values for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 150] for Payoffs and [0, 12] for Transitions 92

Table A.1 Data–Link Layer State Descriptions 105

LIST OF FIGURES

Figure 1.1.: An Adaptive System 3
Figure 1.2: Amount of epitasis for different search and optimization techniques 6
Figure 1.3 A Finite State Machine Representation of a

Figure 2.1: A Classifier System 18
Figure 2.2: A Classifier System with Long–Term Memory 19
Figure 2.3: A transition function which detects the input from the environment and outputs some behavior that affects the environment. 25
Figure 2.4: A transition function which detects the input from the environment and and produces the approximate internal category 26
Figure 2.5: A Q–morphism showing the two levels of abstraction that are used to model the environment, using function P ... 27
Figure 2.6: The Hertzian Model 28
Figure 2.7 The Genetic Algorithm Behavior Mechanism 38
Figure 2.8 Genetic Algorithms in Classifier Systems 39
Figure 2.9: Trade–off between Knowledge (Rules) and Search (Tasks) 49
Figure 3.1 A Simple Finite State Machine Defining a Markov Process 53
Figure 3.2 The Open Systems Interconnection 58
Figure 3.3 Reduction of a Communications Protocol 60
Figure 4.1 Final Payoff Values for a Finite State Machine Using Five Different Initial Payoff Values 64
Figure 4.2 Final Payoff Values for a Finite State Machine Using Five Different Initial Payoff Values 65
Figure 4.3 Payoff Values at Major Cycle 12,800 Using Eight Different Parameter Setting Combinations 67
Figure 4.4 Deviations from the Mean of Payoff Values for FSM State 12 69
Figure 4.5 Deviations from the Mean of Payoff Values for FSM State 15 69
Figure 4.6 Payoff Values at Major Cycle 12,800 for FSM 1 with Payoff Set 1 72
Figure 4.7 Payoff Values at Major Cycle 12,800 for FSM 1 with Payoff Set 1 72

Figure 4.8 Convergence of State Payoff Values Using a Threshold of 1000 78
Figure 4.9 Payoff Values at Major Cycle 12,800 using the Communications Protocol 80
Figure 4.10 Number of Classifier Schemata in the Classifier List Supporting Each State at Cycle Step 1000 81
Figure 4.11 Number of Classifier Schemata in the Classifier List Supporting Each State at Cycle Step 12,000 Superimposed with the Payoff for those States at the Same Time 82
Figure 4.12 Flow of Strength in Classifier List 82
Figure 4.13 Convergence of Payoff Values for Protocol Using Three Different Heuristics 83
Figure 4.14 Increase in Payoff Values for Protocol 84
Figure 4.15 Comparison of Fitness of two Classifiers Supporting States 2 and 6.1 with no Perturbation 85
Figure 4.16 Comparison of Fitness of Classifiers Supporting States 2 and 6.1 in Presence of Perturbation 86
Figure 4.17 Flow of Strength in Classifier List with no Rewards Received 87
Figure 4.18 Difference in Fitness Between a Strategy which Shared Rewards Among all Active Classifiers and One that Does Not 88

ABSTRACT

Classifier systems are massively parallel, rule–based adaptive systems that use genetic algorithms for the generation of new rules and the production of dynamic behavior. Classifier systems, which are now emerging as new computational models of learning, use intermittent feedback from the environment as a guide to future behavior generation. One way in which this environment can be modeled is via Finite State Machines (FSMs). FSMs can properly represent an environment since they are characterized by a number of states and transitions among those states which designate the condition of the environment at any given time. FSMs are an ideal tool for representing a multitude of synthetic environments such as communication protocols. One problem an FSM often encounters is its size, since the computation involved in evaluating an FSM is directly affected by its size. In this thesis, we proposed a method which uses classifier systems to reduce and refine a given FSM. The classifier learns the relevance of each state and gives rewards to the states based on the frequency of visitation. At the end of the computation, the classifier system identifies a specific subset of states that demonstrates the highest level of relevancy. In this research, a number of FSMs tested using various parameter settings in the classifier system. A specific FSM of a communications protocol was used to demonstrate the efficacy of this approach.

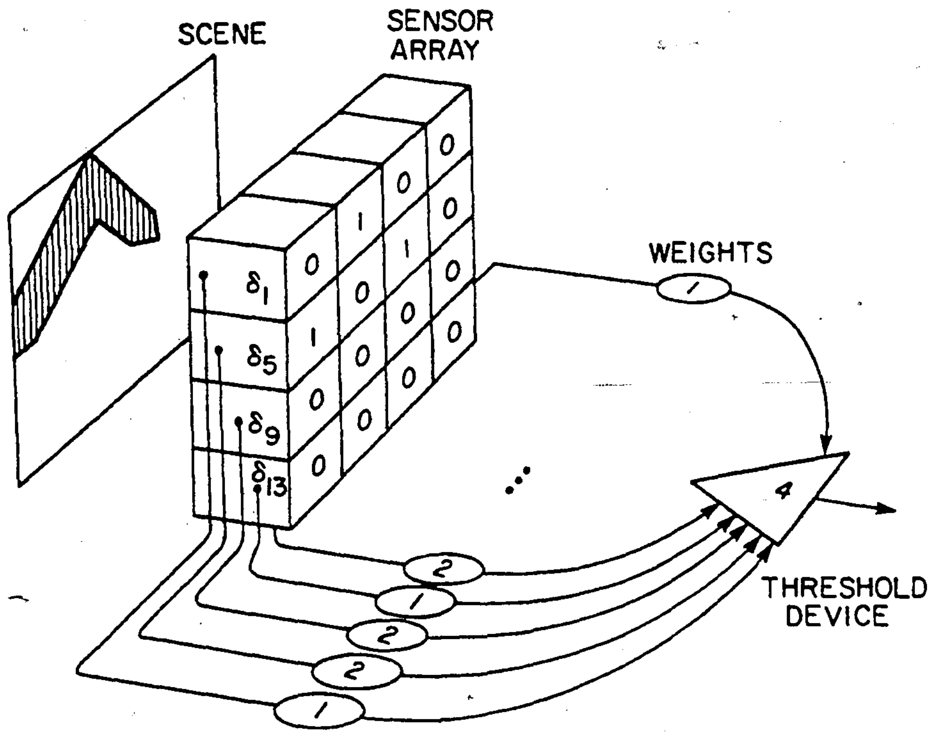
CHAPTER 1

INTRODUCTION

In the recent history of artificial intelligence (AI), new methods have emerged that allow machines to operate on the basis of cognitive models. Much effort has been placed in creating intelligent systems that can learn based on the environment with which they interact. Such systems have to be able to analyze large knowledge bases and evaluate the significance of the residing information therein. Connectionist systems have been very popular in this area for a number of years. However, they are now flanked with a new computational model that is also an isomorphic paradigm to connectionist systems. Classifier systems, which like connectionist systems are used to classify real–world data in some fashion, are rising as a new powerful tool in the cognitive arena. Classifier systems are rules–based, massively parallel, message–passing systems that learn via intermittent environmental feedback. They constitute a bridge that links not only researchers in artificial intelligence, but also those in parallel distributed processing, cognitive science, machine learning, and artificial life (AL). This last category is really a combination of all the previous ones since it introduces the concept of *adaptive nonlinear networks* (ANNs). John Holland introduced this category which incorporates classifier systems, connectionist systems (cognitive psychology), immune system, economic systems, ecologies, genetic systems, and so on. In addition, a number of other AL research programs have recently emerged that attempt to produce the natural living system in artificial forms. These have extensive simulation capabilities and include *computer viruses*, *evolving computer processes*, *biomorphs* and *ontogenetically realistic processes*, *robotics*, *autocatalytic networks*, *cellular automata*, *artificial nucleotides*, *cultural evolution*, and *evolutionary reinforcement learning (ERL)* [1]. A good description of these is given by Taylor [72].

1.0 Adaptive Systems

In order for computer systems to be able to produce intelligent behavior, they must be capable of interacting with the environment and learning from it. This is a continuous knowledge acquisition task that we as human beings always indulge in. Adaptive systems gain knowledge via environmental feedback. Figure 1.1 shows how an adaptive system might look like [33]. Note that the system gathers input from the environment, processes



The scene shown is classified as C^+ because $\sum_{i=1}^{16} w_i\delta_i(t) = \delta_1(t) + 2\delta_2(t) + 2\delta_3(t) + \delta_4(t) + 2\delta_5(t) + 4\delta_6(t) + \cdots > 4$. where δ is a weight given to each cell in the sensor array.

Figure 1.1 : An Adaptive System
From: Holland, John H. Adaption in Natural and Artificial Systems. Ann Arbor, MI: The University of Michigan Press (1975)

the input (in this case represented by a summation function), and then generates output. This adaptive system resembles a classifier system. The sensor array would consist of a number of *classifiers* that have weights assigned to them called *fitnesses* or *strength.* These values designate the importance of a classifier in the environment being probed. This mechanism provides a learning mechanism since the classifiers are constantly modified given the input that is received from the environment. Any change in the environment should be reflected by a change in the behavior of the classifier system.

Classifier systems are ANN systems. To fit this category, certain criteria have to be met, as Holland requires [37]. First of all, an ANN system exhibits hierarchical organization. This implies that building blocks exist that combine intelligent units at lower levels to produce intelligent units at higher levels. Such behavior can be found in all living systems (as seen with *meiosis* in biological systems). In classifier systems, classifiers are built via a process that begins with the establishment of certain building blocks called *schemata* and the successive refinement of them in time. A second characteristic of ANN systems is that a certain level of competition exists that drives the formation of schemata and the eventual formation of system behavior. The fitness value is the commanding element in such competition. Third, the environment with which the system interacts provides incentives for the classifier system to produce desirable outputs. This is done via the assignment of *payoff* values or system *rewards*. A fourth criteria is that of the exploitation of environmental regularities. In other words, the environment produces behavior that is often used by a system as a basis for its own behavior. A classifier system that represents a robot probing some environment would, hence, use light as its tool in its examination. This restraint causes the system to function at non–optimal levels since the robot could well be using other, more efficient means of driving its behavior. It is therefore difficult to exploit all possible environmental regularities for the generating system behavior. Fifth, there is a tradeoff between exploitation and exploration. To explore an entire search space, such as the robot trying to find which direction to move in, is very expensive since it requires much time. Exploitation, however, may not consider all desirable movements but focus on a single one and would, therefore, produce less than optimal behavior. Hence, a trade–off exists between long searches but higher levels of confidence and brief searches and lower levels of confidence.

A sixth factor is denoted by a tradeoff between tracking and averaging. The environment may generate behavior that changes too fast in accordance to the systems behavior, therefore the environmental data has to be averaged in time to gather qualitative information. But if the behavior is synchronous with the system behavior, then the data can be tracked. A seventh momentous characteristic of ANN systems resides in the nonlinear feature. In a natural system, such as a genetic system, which resembles classifier systems, genes depend on other genes during the building block process [15]. In genetic terms, a *phenotype*, which would be the resulting organism from the interaction of genes, emerges. But genes tend to inhibit other genes from acting in given ways or activate them at other times. This creates the condition called *epitasis*, where one gene affects the behavior of another one. The genes, interact together to produce some higher level organization (a new organism). But due to epitasis, it is not possible to determine from the genes alone the constituency of the higher level organism (via the schemata building block). This factor introduces nonlinearity in the system. In classifier system, this effect is produces via the search technique called *genetic algorithms*. The amount of nonlinearity present in the functioning of genetic algorithms is important to control since too much epitasis can lead to very complex schemata that are hard to understand while too little epitasis does not provide an adequate building block mechanism. Davidor [15] compares genetic algorithms in comparison to other search techniques and identifies how much epitasis each method should have, as displayed in Figure 1.2. Note that *chaotic* behavior of the environment is also an endogenous factor, beyond linear estimation. The functioning of the classifier system depends on the environment and how the environment interacts with the system. If the environment displays nonlinear behavior, where it acts in bursts rather than in a deterministic fashion, then the classifier system's behavior will also be additionally unpredictable.

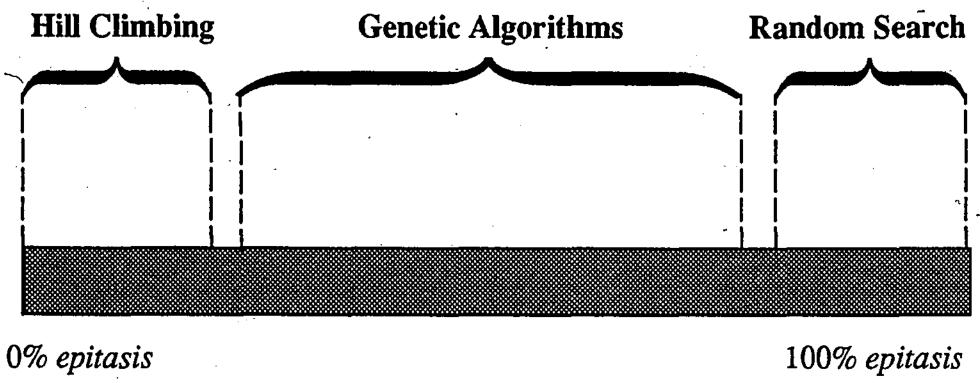


Figure 1.2: Amount of *epitasis* for different search and optimization techniques
From: Davidor, Yuval. Genetic Algorithms and Robotics: A Heuristic Strategy for Optimization. Teaneck, NJ: World Scientific (1991)

An eight facet to ANNs is coupling. This is somewhat related to hierarchical organization since the combination of classifier constituents gives rise to different classifiers. Two classifiers can be coupled when the action of one satisfies the condition of another one, resulting in the organization or coupling of classifiers. Ninth, there can be certain building blocks of a general nature (*generalists*), those upon which many classifiers can be built, and there can be *specialists*, restricting themselves to a small pool of classifiers. Tenth, the cooperation of different building blocks in producing system behavior within a certain environment can lead to more flexible and complete system behavior. The eleventh point concerns the construction of internal models. This puts ANNs in the perspective of cognitive models, where systems learn from the environment. Internal models provide a better understanding of how the system adapts and develops. They define what the system is getting from the environment, what is it doing to the input, and what output does it produce in any given time step. Such topic has also been examined in the neural network area [20]. Finally, ANN systems exhibit a growth in entropy [15]. As building blocks are created within a system, that system has to use more resources in order to continue its building process. In this fashion, the amount of (finite) resources increase as the system complexity increases. Entropy puts a ceiling on how much a system can evolve due to limited resources. Note that this is recurring problem in the design of information systems, where sometimes the amount of information to be processed is onerous. In cases where the amount of data becomes unmanageable, the *haystack syndrome* results, where it becomes difficult to determine what data is really useful.

Once the above requirements are met, the adaptive system can be created. Genetic algorithms provide a useful means to manipulating and adapting classifier systems.. The mechanisms employed by genetic algorithms focus on *chromosomes*, which are the structures that encode the traits of living beings. These beings take shape given the decoded chromosomes. Genetic algorithms have four key features. First of all, the chromosomes are the means by which we evolve and therefore any adaption that takes place directly affects these structures. Second, evolution via natural selection is based on the strength of the chromosomes (their ability to survive in time). This happens because certain chromosome "blue–prints" (or schemata) are better fit for the environment being tested. Third, during reproduction, new offsprings evolve that are hopefully more fit than their parents (in the given environment). Fourth, since biological evolution has no memory, the certain chromosome blue–prints will be forgotten and surpassed in time [16].

1.1 Classifier Systems Applications

Classifier systems represent a computational and cognitive model that can be easily applied to a great number of domains. This is done by changing the environmental interface of the classifier system to support new applications. The application used in this thesis used a package that provides a facilitated construction of a specific domain [56].

Classifier systems have been employed in a number of domains. These include strategy acquisition [30], learning models of consumer behavior [27], discovering scheduling heuristics [32], learning sequential decision rules [29], detecting events in synthetic images [47], recognizing letters [22], negotiating [43], natural language processing [3] and various others, some of which are explained very well by Goldberg [23].

A number of different systems have evolved for developing classifier system applications. They are based on two programmatic approaches to classifier systems [17]. One of them encapsulates all the rules in the classifier system within one data structure which is the "Pitt" approach, after the University of Pittsburgh, indoctrinated by DeJong, while the other one takes a less holistic approach where each rule is a separate data structure. This latter approach is characteristic of classifier systems of the "Michigan" type, after the University of Michigan, designed by Holland.

Among the different systems used in implementing classifier systems we find the LS–1 (Learning System–1) family of programs [64] [65] the ARGOT system [63], the GOFER family of systems [8], the classifier system with long–term memory (CSM) [76], the KL–ONE family of classifier systems [21] which use LISP as the programming language, the CS–1 (Cognitive System One) program [34] which was the first classifier system, the ANIMAT system [73], the BOOLE system 4 [74], and CFS–C [56], which is used in this thesis and is written in C. CFS–C also uses the "Michigan" approach to classifier systems.

Although various programs have been developed for classifier systems, there still rests much work to be done in the area. Currently provided are the basic classifier system computational mechanisms, which are not yet capable of handling commercial–scale problems. A number of refinement processes for classifier architectures and genetic algorithms are still necessary. In addition, user interfaces are primitive and have to be enhanced.

Programming in classifier systems has been done in a number of languages. The ideal language is one that does not necessarily provide symbolic processing (as in the case of Prolog or OPS5), but one that allows good numerical computations. CFS–C has been designed in C, which provides sufficient computational power.

Classifier systems are well suited for implementations on parallel machines.

One such implementation was created with the CFS–C package on a Connection Machine [60]. A number of other systems have been designed for parallel implementations [14] [50] [61] [39] [71] [24]. Most of these incorporate the parallelism inherent in genetic algorithms.

1.3 Classifier Systems and other Artificial Intelligence Systems

Classifier systems would seem to have a number of disadvantages at first glance. First of all, the rule–base seems to consist of a great number of rules encoded in binary format (0's and 1's), giving classifier systems very little descriptive power. In addition, classifier systems support parallelism where many rules can be active simultaneously. There is a problem in the management of rule activations in such a case since somehow the rules that should be activated must be found and selected in some qualitative fashion [10]. However, the classifier system language allows one to build representations using the binary encodings. In fact, schemata (or building blocks or blue–prints) permit the creation of hierarchical associations among rules. A good example of this is given by Holland and Booker[1] where networks are build via associations among different rules. Another description of high–level hierarchical associations is given by Antonisse [2]. Rule clustering is also becoming of great relevance in classifier systems. Such clustering is useful especially when the system attempts to solve problems via associations or analogy [76]. In such case, rules that are similar in some ways are recalled in attempting to solve the problem at hand. Another concept is that of rule *corporations* [74]. This idea involves having rule clusters cooperate together in solving problems (just like corporate collusion).

Rule clustering and combination can also be represented via the disjunction of complementary rules. Booker [11] also argues that using *complementary feature manifolds*, where a chromosome can be used to represent one value, such as the color blue or red

1. This is done via the tags assigned to rules (which will be discussed later). Holland and Booker give a good description in [10] on page 249

or green, a number of *disjunctive generalizations* can be obtained. Thus, for the three colors above from which the other colors of the spectrum can be obtained, a generalization can be made using disjunctive operators that could create a new color. Thus, blue and red may produce purple as a result (this would also have its own chromosome encoding different from all others).

In addition to being able to produce qualitative representations, classifier system provide parallelism which is crucial in building combinations of rules that would otherwise be impossible to configure in a serial system. Specifically, the firing of rules simultaneously greatly enriches the pool of facts (or *messages* in classifier systems) that exist in working memory and that can be accessed for the matching of classifier conditions.

Classifier systems use sub–symbolic processing, similar to connectionist systems. In such models of computation, each "unit of intelligence" is in itself not very meaningful [21]. For example, a single synapse in a neural network or a classifier in a classifier system by themselves do not have much informative power. But the combinations and interactions of these units at low–level produces very complex high–level representations, as supported by the schemata mechanism in classifier systems. On the other hand, knowledge can be represented in rule–base format which is more easily interpretable and accessible to investigation that the hidden layers of the connectionist systems. Classifier systems, hence, provide a middle ground to connectionist systems and traditional symbolic processing systems such as expert systems.

Furthermore, the schemata mechanism of genetic algorithms has been used to provide a better understanding of sub–symbolic processing in connectionist systems. Via the schemata mechanisms, hierarchies are built to represent different symbols. An excellent example of the is given by Dolan [19]. Indeed, understanding neural network hidden layers and what those layers represent (what is the network learning at any given time) can be quite complicated [20].

Antonisse and Keller [2] provide yet another example of how classifier systems can help build high–level representations via the schemata processing. They build a production system for building trees representing hierarchies.

1.4 Classifier Systems and Finite State Machines

A Finite State Machine (FSM) can be used to model a number of domains with classifier systems. The reason for this is that the environment with which the classifier system intermittently transacts can be represented as a state space system. Each state within the FSM can represent a state the environment can be in at a given time. Once the classifier system receives input from the environment, processes it, and outputs its behavior, the environment's *state* changes. FSM, hence, make the problem of *representation* easier. The classifier system is a computer program with no inherent cognitive traits. In order for it to be able to process the information it receives, it must know how the environment "looks like."

The use of the FSM in classifier systems is not a unique concoction. In fact, FSMs are the main design element of cellular automata. They have also been used in a number of new artificial life systems, underlining the fact that FSMs are an appropriate cognitive model, capable of representing environmental states in a simple while complete fashion. They are also used as a main counter–example of artificial neural networks.[2]

1.5 Classifier Systems, Finite State Machines, and Communication Protocols

It has been shown that classifier systems are an important tool in developing problem solving systems that can successfully handle tasks within a particular domain. One

2. For a good sampling of Finite State Machines and their use in intelligent systems, the Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity volume can be a good guide.

such area is that of protocol design. A protocol defines the functions of a communication system. Specifically, a protocol delineates all the rules and conventions that are used in communication among various systems that are connected to a network.

Communication protocols are specified as finite state machines, or *protocol machines* [25], and incorporate the various states that the communications system may be in at any given time. The FSM concept is especially appropriate in protocol specifications since a communications network consists of a number of input and output sequences that change the state of the network. Hence, at one point of communication between two *nodes* (such as two computer terminals, using *Data Connecting Equipment (DCE)*) in a network, one node may be sending a message or it may be waiting for an acknowledgement for a message that was sent to another node. An example of such a state diagram is seen in Figure 1.3, as defined by Bochmann [7]:

Very often the number of states specified in a protocol machine can be superfluous or redundant. In such a case, inefficiencies are created in communication since the protocol system has to transit to additional states when this not really necessary. To avoid such a plethora of states, a method has to be devised to analyze whether all the states are necessary and whether any of the states can be eliminated, while not comprising the functioning or efficiency of the protocol. The classifier system in this paper analyzes a protocol that defines the Integrated Services Digital Network link–access protocol on D–channel (LAPD), which is defined in the data–link layer of the Open System Interconnection (OSI) Reference Model of the International Organization for Standardization (ISO). In doing so, it learns which states appear to be bottlenecks within the protocol specification. In addition, to support empirical results, some randomly generated finite states machines are produced and tested to check the efficacy of the approach taken.

This thesis intends to support the concept of system reliability optimization. To produce systems that are reliable and perform well, redundancies have to be eliminated

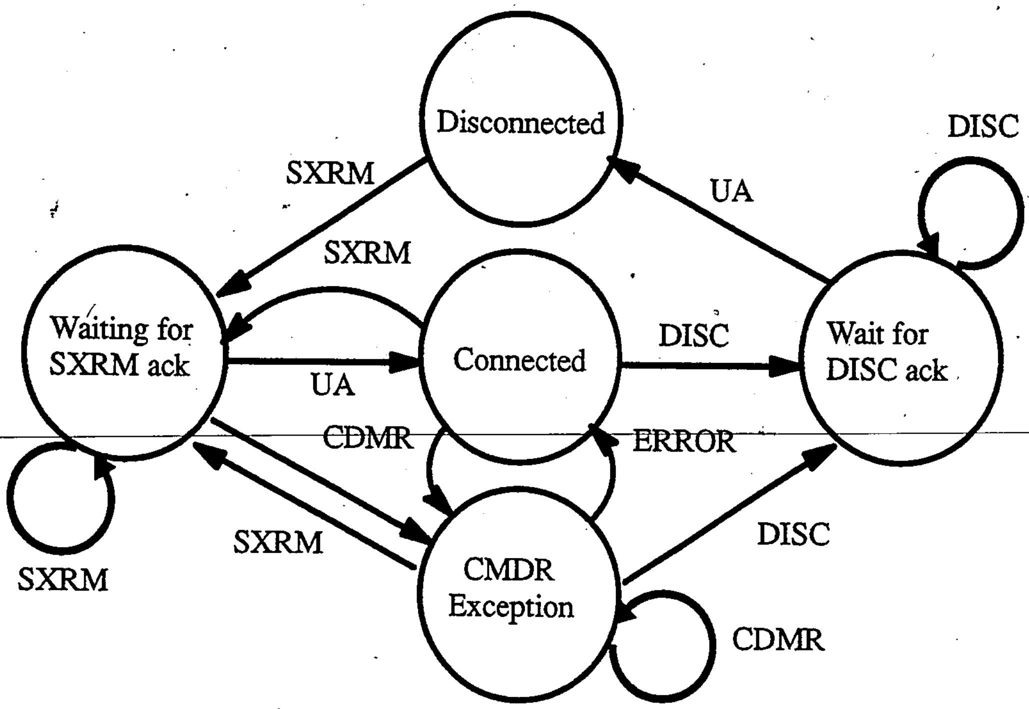


Figure 1.3 A Finite State Machine Representation of a Communications Protocol
From: Bochmann, Gregor V. "A General Transition Model for Protocols and Communication Service" in *IEEE Transactions on Communications*, vol. COM–28, no. 4, (April 1980)

or highly sophisticated and reliable components have to be manufactured to support the system. It seems far less expensive to take the former approach, minimizing costs in doing so [18]. The approach taken here is of optimizing protocol reliability.

CHAPTER 2

CLASSIFIER SYSTEMS

Classifier systems are dynamic, rule–based systems that incorporate mechanisms that permit learning via environmental feedback. Such a system is based on inferential learning aided by continuous interaction with the environment. By inference here we mean the capability of deriving knowledge from very broad concepts and applying them to specific situations. Learning in classifier systems is provided by credit assignment and discovery algorithms, which modify the repository of information at any given time and produce an updated knowledge base at every successive time step that is more adept at the problem being solved. The learning process follows a model of evolution whereby each knowledge "unit" or rule competes for survival within the knowledge base using a "fitness" value as its leading survival indicator. In doing so, the system develops an internal model reflecting what the environment looks like.

Classifier systems, in addition to being a good computational model of cognition, are also a good example of parallel architectures. This allows the knowledge search to be performed in a more efficient and accelerated fashion. Parallelism also supports the framework of emergent computation, where "units" of intelligence compete with each other to produce knowledge about the environment.

2.0 Classifier Systems and Induction

Classifier systems are examples of *inductive systems*, where learning occurs as a result of exposure to a given environment. An inductive system is defined in these terms, as explained by Holland et al. [36]:

1) contains a knowledge base that encodes environmental data; each knowledge rule (classifiers in classifier systems) can be represented in terms of conditions and actions resulting from those actions. Therefore, the system is capable of perceiving a certain event and reacts to that event in an appropriate fashion (system behavior). Note that chaining of actions is allowed here where the action of one condition–action rule leads to the fulfillment of the condition of another rule;

2) such rules encode a relation between the current state of the system (*diachronic relation*) and a modification of categories and their associations (*synchronic relation*) where diachronic relations are used in the search for new rules and synchronic relations permit their modification as a result of the learning process. Categories here represent certain common features encoded by the rules in the knowledge base;

3) a building block mechanism whereby categories can be refined and specialized. This occurs only after sufficient interactions with the environment;

4) the construction of *default hierarchies*, which represent the first state of any building block and that are refined given relations among various categories within the knowledge base and the environmental transactions;

5) an emergent model based on a number of diachronic and synchronic rules that permit system behavior (i.e., the reinforcement of information in the knowledge base);

6) the existence of parallelism, by which a number of rules are active at the same time in the system (their conditions are satisfied) and where these rules complement each other by providing further refinement of the categories they support;

7) a mechanism for modifying rule strength (credit assignment algorithm) and a mechanism for generating new rules given those strengths (discovery algorithm);

8) new rules are generated given their past failures in supporting a given category;

9) the inductive system must learn about any changes that occur in the environment and adjust to them appropriately, by altering the knowledge base.

As it will be shown, classifier systems comply to the specifications delineated by inductive systems, permitting dynamic behavior as a result of its interactions with the environment and successive modification of its knowledge base.

2.1 Elements of Classifier Systems

A classifier system is a rule–based system that interfaces with the outside world. In doing so, it learns from the environment and generates appropriate behavior. In expert system language, a classifier system is made of *condition/action* rules, where conditions are matched by *messages* detected from the environment and actions are *fired* as a consequence. The classifier system consists of the following element:

1) an *input interface* that extracts information from the environment and encodes it into binary form, the classifier system operating language. These binary strings are called *chromosomes* for their use in the *genetic algorithm* (the discovery algorithm used by classifier systems). These chromosomes are called *detector messages* since they detect the current state of the environment.

2) a *rule–base* or *classifier–list* consisting of the condition/action rules. The number of classifiers in the rule–base is called the *population*, once more in genetic algorithm terminology.

3) a *message list*, much like the *agenda* of an expert system, that contains the detector messages that have been taken from the input interface. The message list is updated constantly to represent the state of the environment. Note that the classifier system is capable of performing real–time operations thanks to its environmental interface.

4) an *output interface* that generates system behavior. The output is also a chromosome, called effector, which is then decoded to affect the environment in some way. For example, if in a packet–switched networking system a certain data packet is not received at a given

time, the classifier system must learn to request re–transmission of the data packet using its output interface.

2.1.1 Types of Classifier Systems

The rule–base of the classifier system consists of short–term memory of facts that lasts as long as the classifier system is operating to solve a certain task. Once the task is accomplished, or learning has occurred, this volatile memory is shut down. In order to prevent such loss, a classifier system with long–term memory has been developed [77] that allows old rules to be recalled. In order to do so, the classifier system matches its detector messages with rules residing in this long–term memory and builds a population of classifiers from those rules. This type of association can be referred to as learning by analogy, since the classifier system retrieves facts from previous experiences and uses them for current problems. Note also that this noticeably improves system performance since the system does not need to reconstruct those same rules again. Once the system has accomplished its tasks, it saves those rules in long–term memory in order to fetch them at some future time.

Figure 2.1 shows a classifier system with short–term memory. Note that whatever resides in the classifier list is then lost when the task is accomplished. The input interface gathers data from the environment in the form of a message which it places on the message list. The message list is also updated with any other messages that are fired from the classifier population. When the classifiers generate effector messages, these messages are delivered via the output mechanism of the system, generating system behavior.

Figure 2.2 shows a classifier system with long–term memory which is initialized with a classifier population stored from previous runs of the system. Associations are made between what is in the message list and what resorts in the long–term memory via the *matcher*. Specifically, as soon as the classifier system receives some initial environmental

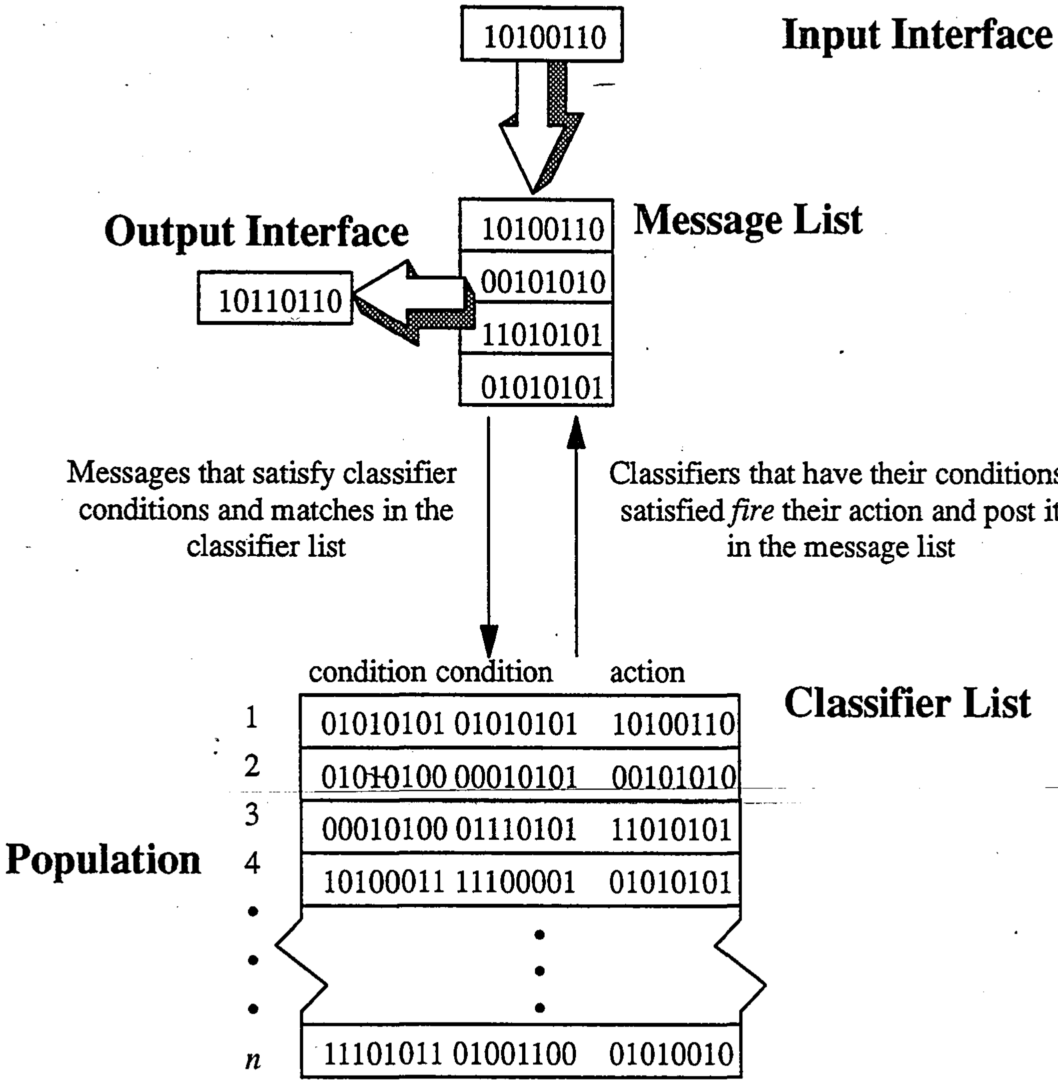

10100110
Input Interface
Output Interface
10110110
10100110
00101010
11010101
01010101
Message List
Messages that satisfy classifier conditions and matches in the classifier list
Classifiers that have their conditions satisfied *fire* their action and post it in the message list
condition condition action
Classifier List
Population
1 01010101 01010101 10100110
2 01010100 00010101 00101010
3 00010100 01110101 11010101
4 10100011 11100001 01010101
n 11101011 01001100 01010010

Figure 2.1: A Classifier System

data from the input mechanism, it compares the data to the rules in the long–term memory, which are grouped into clusters related to the different problem–solving tasks dealt with in the past, and initializes a population consisting of all the rules in that cluster. If the number of rules in that cluster is not as big as the limit of the classifier list, additional rules are generated randomly. The remaining operations are similar to the short–term memory classifier

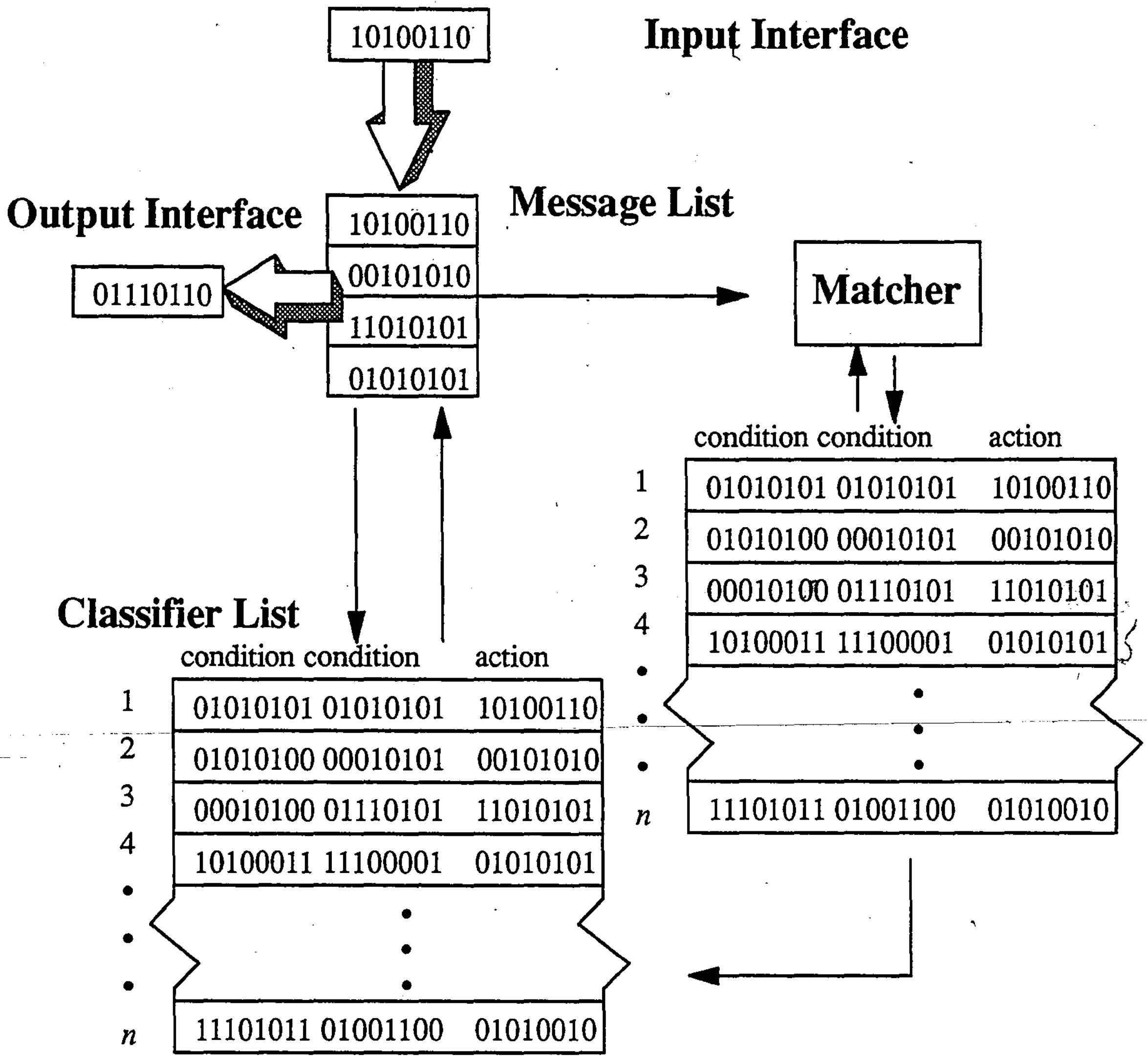


Figure 2.2: A Classifier System with Long–Term Memory

system.

The classifier system, like any other adaptive system, is supposed to be a dynamic system which changes over time given its interaction with the "outside world" and improves its performance as it learns more about it [17]. The adaptive mechanism is an encapsulated "black box" to which access is limited and all interfacing is done via a special *broadcast* or interface language. In addition, in order to improve performance, it has to be able to extract new environmental data into a format that it can understand (which is called *schema*). The formation of such abstract forms is an ongoing process since the environment is rarely in a static state, changing in time as a result of its interaction with the classifier system or other source.

2.1.2 The Rule–Base

All rules within a classifier system are composed of bits (1, 0) and a *don't care* symbol (#). The condition and action strings of a rule are composed of a sequence of such symbols. The syntax used in can be written in Bakus–Naur Form (BNF) [21]:

<classifier system> ::= <classifier>

<classifier> ::= <condition> <condition> => <action>

<condition> ::= $\langle\text{alphabet}\rangle_n$ | $\sim\langle\text{alphabet}\rangle_n$

<action> ::= $\langle\text{alphabet}\rangle_n$

<alphabet> ::= 1 | 0 | #.

Note here that we have used two condition strings which is the default number used by the software library [56] used in this thesis and have set n to 16. The tilda in the condition statement indicates that if no other strings matches that condition, then the rule is *fired*, or accepted, given that the first condition is also satisfied.

Therefore, a condition or action may look like 0010101010101011O or ~1010100001010101010 or #100##11010#11111. The latter case is that of a generalist message, which will match many different messages (approximately 4^3 messages) since there are 4 don't cares. The former messages, on the other hand, are specialists and will match only one other message, genotypically identical to them.

There are four different *tags* associated with the messages. A message[3] that starts with “00” in its left–most bits designates a detector message. Therefore, such a message will only match messages from the input interface. Messages that start with “11” or “01” are used only within the classifier list. Finally, those messages that start with a “10” in their left–most bits represent effector messages. For example, when a classifier fires an action with “11” or “01” in it, that action will be matched only by the condition of another classifier. If that action has “00” in its left–most bits, then it will match messages on the message–list. And if the action has “10” in left–most bits, then it will match effectors. In fact, there are different effectors that can be defined, each defining the system behavior that is desired. This will be discussed later on.

Tags are also used to produce networks of activations of classifiers. For example, a tag like 1001, embedded in the action of a classifier, can designate an action to be taken by the effector mechanism. But that action, when matched by a condition can generate another action with a different tag which would only occur if the previous rule had fired an action. In this fashion, a hierarchy of classifiers is formed giving the classifier system a knowledge structure that not only provides maneuverability of the knowledge and how it should be used, but also flexibility in how to use that knowledge. A more accurate account of networks is given by Booker et al. [10].

2.3 Schematas and Mental Models

A *schema* is the classifier mechanism for encoding categories. Schemata are the building blocks of the system. From the schema, the classifier system builds an *internal*

3. Note that terms such as message, chromosome, bit string are used interchangeably since they are standard classifier system terminology.

representation of the system. In other words, the schemata define the most relevant characteristics of the environment that are being considered. The schemata constitute a part of the conditions or actions of a classifier. For example, if one of the classifiers in the rule–base looks like the following: 00110**00#1#1**0101 ~101010**1#01#0**001111 → ##100**###0101**######, then the schemata (in boldface) designate those attributes that are significant in identifying a particular concept. Note also that the pound signs ("#") here serves as *pass–through* symbols since the message that the action of this rule will match will satisfy the message regardless of the values in those positions.

2.3.1 Schemata and Building Blocks

Each of the bits in a chromosome can represent a certain concept. For example, in a system that diagnoses communication network problems and a string is posted in the classifier system such as 0000000000000001, the right–most bit (turned on, or set to 1) may indicate that the last data packet in the a series has not been received. But one may need to capture more data once a problem has been detected. In such a case, a number of bits can be used to represent such a concept. For example, in the bit string 00000000001010, the sequence 1010 can represent the network state *network load high and rising*, where bit 2 could represent *rising load* and bit 4 could represent *load high*. Note that since the alphabet is also composed of a third symbol, the 1010 sequence of bits is actually only one in a set of four distinct possibilities ({1010, 1110, 1011, 1111}). This schema, 1#1#, permits the classifier system to represent many different concepts having a similar building block. Therefore, the network example could be expanded to represent *network load high, rising, and breakdown point reached*, using the schema 1110, for example. A schema can be of any length and appear anywhere in the chromosome. In the above example, the schema can be represented as ************1#1# since the other bits are not part of it. Therefore, in more generic terms, a schema can be in the set $\{1, 0, \#, *\}^n$, where n is the length of the string (set to 16 in this application). Radcliffe [52] uses this more generic definition of schemata theory to develop his *formae*.

This phenomena, where an n–bit string is an instance of 2^n schemata, has been called *implicit parallelism*. In a sense, the schemata and all the strings that can be derived from them, are active at the same time. This gives greater computational power to the classifier system and a greater number of concept formation strategies.

A schema, then, serves as a building block for the classifier system. Holland, however, suggests that there are various obstacles to adaption related to schemata that have to be overcome if learning is to be efficient [33]. First of all, the cardinality of the various schemata is very high. There can be a great number of such structures and the search for the most fit can be non–exhaustive. Second, due to the fact that biological evolution has no memory, the apportionment of credit may occur on the basis of incomplete knowledge, based solely on the information currently in the classifier list. Third, the strength (fitness) of the classifiers have high dimensionalities, removing much influence from the other variables that are at the basis of the optimization process provided by the genetic algorithm. The strength measure follows a nonlinear (chaotic) behavior due to epitasis, causing the optimization process to be locked in suboptimal areas of the total search space. Fifth, search and exploitation are not mutually exclusive. the search for new structures interferes with the generation of above–average schemata. Finally, there is more information than payoff values that the environment provides that has to be taken into consideration. This is especially true with such parameters as system rewards. Holland also explains that such problems can be overcome by progressively exploiting the best schemata, ensuring that the schemata being generated are the best ones, and testing a large number of new schemata in order to ameliorate the chances of exploiting the entire search space.

2.3.2 Mental Models and Q–Morphisms

Mental models are used to emulate the structures by which we represent the world in our brains. These structures, categories, or schemata are certainly not unique to classifier systems and genetic algorithms. Indeed, Roger Schank and Marvin Minsky developed most of the concepts related to such models. Schank developed the concept of *scripts* and later on Memory Organization Packages (MOPs) while Minsky improvised *frames* [62] [45].

Schemata and similar structures permit adaptive systems to gather environmental data from the outside world, decode them, and represent them in internal format. This is what the detectors do in a classifier system on a continual basis since the environment keeps on changing with time. Once a classifier system has detected the information from the environment, performed its classification, and output its behavior, the environment has clearly changed its state. A mathematical model of such state transition is given by Holland et al. [36]. This is composed of a transition function $T[S(t), O(t)]$, where T is the transition function, $S(t)$ is the current state of the environment at time t, and $O(t)$ is the output of the classifier system at that time step.

Once the time step is concluded, the transition will be $T[S(t + 1), O(t + 1)]$. This transformation process is represented in Figure 2.3. Note that information about the outside world is mapped into the classifier system at any given time step.

Since the environment is highly complex and difficult to convert into an internal state using an isomorphic function, only approximations can be made. This is done via a *homomorphism*. In doing so, the most relevant characteristics of the environment are extracted and mapped into the input interface of the classifier system. Recall that in the classifier system being used here, these characteristics are mapped into 16–bit messages. A better

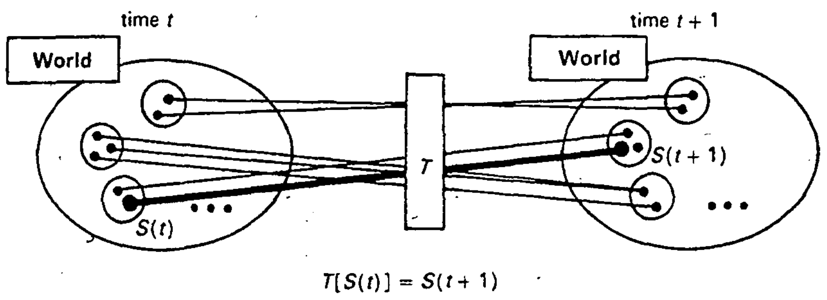


Figure 2.3: A transition function which detects the input from the environment and outputs some behavior that affects the environment.

From: Holland, John H., Holyoak, Keith J.,.Nisbett, Richard E., and Thagard, Paul R. <u>Induction: Processes of Inference, Learning, and Discovery</u>. Cambridge, MA: Massachusetts Institute of Technology Press (1986)

representation of this is given in Figure 2.4 Note that for the different characteristics, certain schemata are formed.

The schemata are the initial building blocks of the classifier systems. They constitute *default hierarchies* from which additional refinements can be made to fully represent the state of the environment. In order to do so, the current mathematical model has to be expanded to take into consideration the default hierarchies and the models that can be derived from them. Figure 2.5 displays a *quasi–morphism*, or *Q–morphism*, as defined by Holland et al. [36] which not only brings the previous two figures into one, but also models the internal state into two levels. Level 1 represents the default hierarchy while the Level 2 represents a specific instance of that hierarchy. Therefore, a condition within one rule in the classifier system could undergo the following changes (the schemata is in boldface):

	<u>Time *t*</u>	<u>Time *t+1*</u>
condition (Level 1):	01010**#1#00#**01010	01010**#0#00#**01010
condition (Level 2):	01010**11100**101010	01010**00100**101010

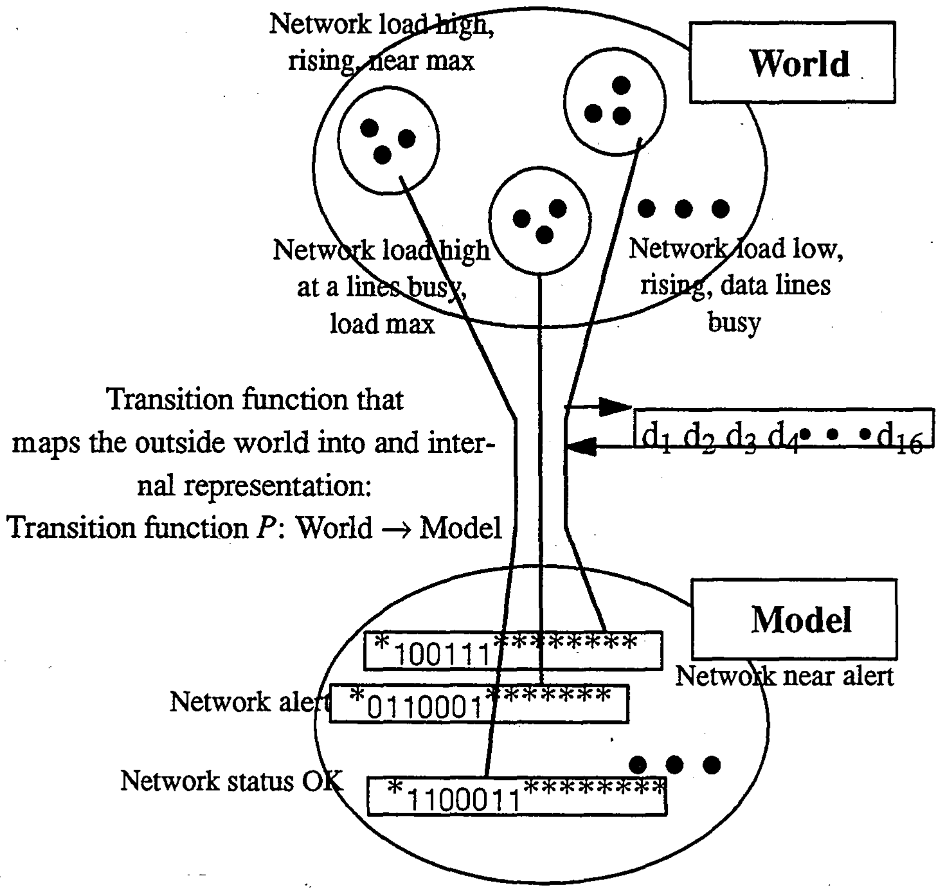

Network load high, rising, near max
World
Network load high at a lines busy, load max
Network load low, rising, data lines busy
Transition function that maps the outside world into and internal representation:
Transition function P: World → Model
d1 d2 d3 d4 • • • d16
Model
*100111*********
Network near alert
Network alert
*0110001*******
Network status OK
*1100011********

Figure 2.4: A transition function which detects the input from the environment and and produces the approximate internal category

Adapted from: Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R. Induction: Processes of Inference, Learning, and Discovery. Cambridge, MA: Massachusetts Institute of Technology Press (1986)

Note that pound signs have been replaced by bits, indicating that a specific concept has been formed.

Accordingly, mental models undergo a continuous process of alteration. It is via this mathematical process that learning takes place and refinement occurs. Schank [61] argues that learning occurs thanks to the iteration of interfaces with the environment. And this is what occurs in a classifier system since the more instances of a schema that are encountered, the stronger and less abstract that schemata becomes, representing the current state of the environment at any given time.

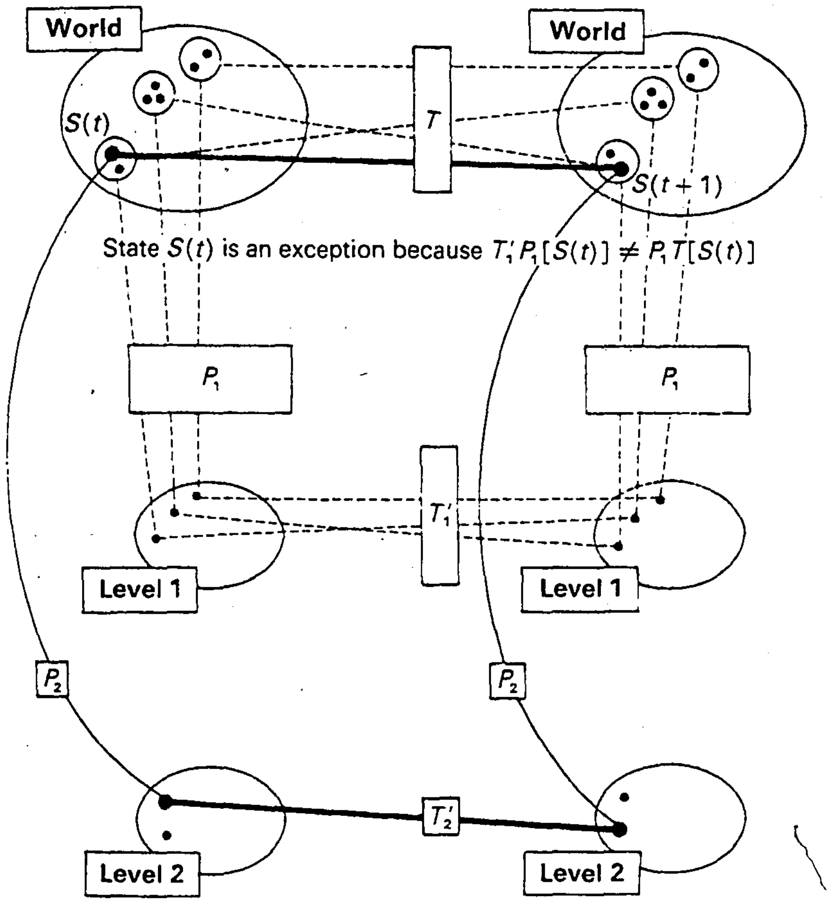


Figure 2.5: A Q–morphism showing the two levels of abstraction that are used to model the environment, using function P

From: Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R. Induction: Processes of Inference, Learning, and Discovery. Cambridge, MA: Massachusetts Institute of Technology Press (1986)

An interesting analogy can be derived from the works of Cariani and the Hertzian model of emergence he presents [13]. This model embraces the concept that emergence in a system takes place by encapsulating and encoding information from the environment, processing it via some rules (or functions), and then generating a prediction of what the environment would look like in a successive time step. This is also clearly a mental model of how a system behaves, with the difference that a predictive element is used rather than a deterministic, after the fact formula. The Hertzian model given by Cariani is displayed in

Figure 2.6 and shows how the data from the world is gathered and understood (semantics) and how it is transformed from one time step to the other via internal functions (syntax).

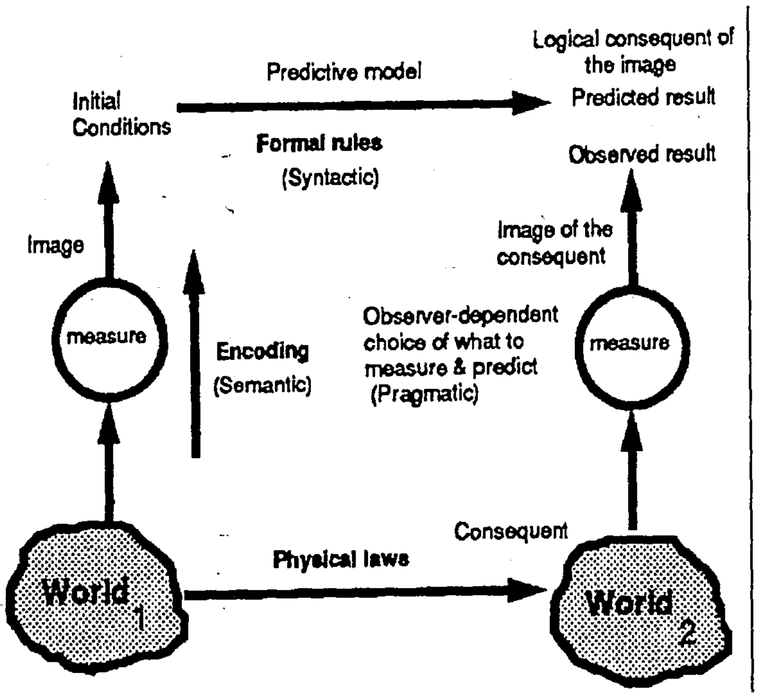


Figure 2.6: The Hertzian Model

From: Cariani, Peter "Emergence and Artificial Life" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp. 775–779

2.4 The Classifier System Major Cycle

The classifier system performs its task by adapting to the environment. In order to do so, it keeps on getting information from the environment and processing it. The classifier system, therefore, uses the following algorithm [56]:

1. **Add** messages that represent the current state of the environment to the message list;

2. **Compare** all the messages that are on the message list with the conditions of the classifiers in the classifier list;
3. **Calculate** bids for all the classifiers that have their conditions satisfied, run a competition among the bidding classifiers and select those classifiers that win;
4. **Generate** new message from the actions of those classifiers that won the bidding competition (add the messages to the message list);
5. **Process** the new messages to the output interface (effectors) and get the reward from the environment.;
6. **Apply** the credit assignment algorithm to reallocate the strength of the classifiers;
7. **Use** the discovery algorithm to modify the classifier rules given the new strengths; this will produce new rules representing the current state of the environment;
8. **Replace** of the message list with the new messages from the new classifiers;
9. **Go to** Step 1.

Messages are added to the message list by a classifier and have a value associated with them called *intensity*. The intensity of the message is the amount a classifier bids in order to get its message (the action it fired) posted. The bidding system will be discussed later, but it is important to keep in mind that the message intensities affect how much classifiers bid for their messages. After every major cycle, the messages are added to a list of messages along with their intensity values.

All classifiers undergo a Darwinian survival process, via the discovery algorithm, where they must survive given their use and strength. Therefore, the more a classifier is used (the greater its use), the more its strength will increase. Classifiers also have a *bidratio* associated with them. The higher the bidratio, the more the classifier will bid during the bidding competition. The bidratio can be seen as the classifier's willingness to post messages. The classifier specification can now be expanded to:

<condition> <condition> $\rightarrow$ <action>, bidratio, strength

The default value for the bidratio of a classifier is called the *specificity*, in case a bidratio is not assigned. The bidratio is computed in this fashion[4]:

$$specificity = \frac{(S(condition1) + S(condition2))}{(chromosome\ size * 2)} \tag{2.1}$$

where

$$S(condition\ \mathrm{i}) = \begin{cases} \text{number of 1 or 0 symbols in match conditions} \\ \text{number of \# symbols in non–match conditions (those with tildas (\textasciitilde)).} \end{cases}$$

Another value that is used in the bidding contest is the *support* value for a classifier. It is the summation of all the intensities of the messages on the message list that match the classifier's conditions. This can be viewed as:

$$V_i(t) = \sum_{m'} I_{m'}(t) \tag{2.2}$$

4. The parameters and formulae which appear are taken from Riolo in [56].

where m' represents all the messages that match classifier i at time step t and $I_{m'}$ is the intensity associated with those messages. Support is used to compute bids for classifiers. The greater the support for a certain classifier, the greater the probability that that classifier will still be active in the next time step.

The bidding process of the classifier system permits those classifiers that have their conditions satisfied to bid in order to become active. The bid for a classifier i at time step t is $B_i(t)$ and is computed in the following fashion:

$$B_i(t) = k * S_i(t) * bidratio_i^{power} * avgsupport_i(t) \qquad (2.3)$$

where k is the parameter that determines how much of a classifier's strength will be used in bidding, $S_i(t)$ is classifier i's strength at time step t, *bidratio* is classifier i's specificity, *power* increases the importance of the specificity in the bidding process, and $avgsupport_i(t)$ is the support for classifier i divided by the summation of the supports of the other bidding classifiers. Note that the bidding process heavily relies on the relevance of the classifier to the current situation the classifier system concerned with and also on how effective the classifier was in the past.

Once the bids for all the classifiers that want to be active are computed, a competition is run to determine which classifiers will become active, in case there is a threshold to the number of classifiers that can be active at any given time step. The probability that a classifier can then become active is:

$$P(i) = \frac{\beta_i(t)}{\sum_j \beta_j(t)} \qquad (2.4)$$

where $P(i)$ is the probability that classifier i will win the competition, j ranges over all the bidding classifiers at time step *t*, $\beta_i(t)$ is the effective bid of a classifier and is defined as:

$$\beta_i(t) = \beta_i(t)^{bidpower} * bidratio_i^{effpower} \qquad (2.5)$$

where *bidpower* is used to change the probability distribution used to selecting classifiers and *effpower* is used to allow specialist classifiers to win as opposed to generalists. The higher the bidpower, the higher the probability that those classifiers that have highest bid rates will win the competition. The effective bid is used to introduce some additional randomness into the selection process so that all classifiers get a chance to post messages, providing for a redistribution of strength among the bidding classifiers (otherwise the selection would be deterministic and biased). It is important to note that all classifiers should be given a chance to post messages since even though they may not have high strengths, they may contribute to providing a useful solution to the problem on hand.

2.5 Modifying the Classifier List

In order for adaption to occur, the classifier system must have a mechanism that will change its knowledge base. This knowledge encoded in the chromosomes has to be modified. Two mechanisms are used for this. One is the credit assignment algorithm and the other is the discovery algorithm. As mentioned earlier, each classifier has a strength parameter associated with it. The greater the strength, the greater the "survival probabilities" of that classifier. Classifiers with low strength values are considered as not being useful to the system's learning process and are thus discarded. In addition to credit assignment and discovery algorithms, taxes are applied to classifiers to reallocate strength.

2.5.1 Credit Assignment: the Bucket Brigade Algorithm

Classifier system have to reward those classifiers that have shown some utility to the task being learned. However, when doing this, two very important factors have to be taken into consideration [10]. First of, the classifiers in the classifier list in a time step (or major cycle) *t* are "stage–setters" for other classifiers in time step $t + 1, t + 2, ...$ In other words, current classifiers determine what the future classifier list will look like since they encode the attributes of any future chromosomes, as defined by the schema model. As the classifier list is refined, certain schemata become stronger than others and their presence within the knowledge base increases rapidly until there is complete convergence towards those schemata. The chromosomes that contain those schemata become stronger since they generate profits by paying out part of their strength in order to post messages but getting more heavily recompensed when they have posted their messages.

Another issue is that the environment with which the classifier system interfaces with is of a dynamic sort, making it difficult to maintain a knowledge base that can handle abrupt changes. This element of uncertainty is present within all of the rules and must be handled successfully.

The *bucket brigade algorithm* is designed to cope with these problems. The algorithm can be understood in economic terms. The economy is composed of a great number of classifiers where each classifier performs transactions with its peers. Specifically, there are certain suppliers for a classifier, those that post messages that satisfy the classifiers' conditions, and there are consumers for a classifier, those who have their conditions matched when the classifier fires its action and posts a message on the message list. Once a classifier wins a bidding competition, it has to pay some of its strength to those classifiers that supplied the matching messages. Once the classifier becomes active and posts messages, it is remunerated by those classifiers that will have their conditions satisfied by it.

In addition to the payments (bids) that the classifier gets when it posts messages, it also gets an environmental reward. In other words, the environment can be specified so that after producing a beneficial effect, the classifier system as a whole gets rewarded for that. For example, if a classifier system correctly detects a possible network failure before the disaster takes place, the system may get rewarded by an amount that compensates it for having prevented a significant loss. Hence, the strength of a given classifier is modified in the following manner:

$$S_i(t + 1) = S_i(t) + R(t) + P_i(t) - B(_i) \qquad (2.6)$$

where $S_i(t + 1)$ is the strength of the classifier at time step $t + 1$, $S(t)$ is the strength at the current time step, $R(t)$ is the reward received from the environment, $P_i(t)$ is the summation of the bids made to classifier i for having matched their conditions, and $B_i(t)$ is the amount the classifier has to pay to its suppliers. The above equation can also be seen in the following fashion:

$$S_i(t + 1) = S_i(t) + R_i(t) - B(_i) \qquad (2.7)$$

for the classifier posting messages and

$$S_j(t + 1) = S_j(t) + aB(_j), \qquad (2.8)$$

for the suppliers to the winning classifier.

Notice that a chain of producers and consumers is formed here [10]. Specifically, the classifier list becomes more robust and complete as consumers and their respective consumers build the classifiers and become stronger. The last consumers in this chain are those that transact directly with the environment and receive payoff from it. If this chain of consumers is not one that can successfully work with the environment, then that chain is broken up and eventually replaced by stronger, more profitable rules. Thus, the bucket brigade attempts to maintain a repository of highly profitable and remunerative rules in the knowledge base at all times.

2.5.2 Taxes

There are a number of taxes that classifiers have to pay at a given time step (after each major cycle) that cause classifiers to remain productive and generate actions appropriate to the learning process. Taxes are applied at a rate R and and the strength of a classifier is affected in the following manner after each time step:

$$strength_i(t+1) = strength_i(t) - [\ strength_i(t) * R\ ] \qquad (2.9)$$

where i is the classifier being taxed.

The first tax that is applied is the *head tax* which is used to stimulate classifiers that do not typically bid to do so. These classifiers are not contributing in any way to the task being solved. Therefore, their strength is reduced in order to warn them of their inefficiency. If such warning will be of no use, these classifiers will be replaced by other classifiers introduced by the discovery algorithms.

A *bid tax* is applied to all bidding classifiers at a given time step. The purpose of this tax is to promote specialized classifiers over generalized classifiers that may bid all the time since their conditions get satisfied easily at every major cycle. In this way, specialized classifiers that may actually contribute useful information about the current state of the environment will have chance to enter the bidding competition. This also controls the premature convergence problem towards the generalized rules by permitting other, more pertinent rules to participate.

The third tax that is applied, the *producer tax*, is paid by those classifiers that post messages during a time step. This tax tries to accomplish the same goal as the bid tax, attempting to discredit overly generalized classifiers that tend to remain in the classifier list for a long time without getting environmental rewards. This sort of permanent memory is eliminated by applying to classifier i the tax R:

$$R_i(t) = maxproducertax * \left[\frac{m_i(t)}{M}\right]^{producertaxpow} \tag{2.10}$$

where *maxproducertax* is the maximum producer tax that can be applied, m_i is the number of messages classifier i produced at time t, M is the maximum size of the message list, and *producertaxpow* is used to magnify the effect of the tax.

2.6 Genetic Algorithms: The Discovery Method

In order for learning to take place within the classifier system, there must be some mechanism to provide dynamic behavior. Genetic algorithms accomplish this by taking the classifier list at every time step and selecting those classifiers that are the strongest within the classifier list. Those classifiers are then modified, creating new ones which would replace the weakest ones.

Genetic algorithms are an appropriate discovery approach for classifier systems since they are robust and can handle a number of different problems [23]. In other words, genetic algorithms provide a trade–off between exploitation and exploration of various domains. While trying to be explore new concepts, genetic algorithms also attempt to exploit those that are available. In this way, more domains can be searched without worrying about problems that commonly face optimization methods. One such problem is that of premature convergence. Optimizing too quickly often causes the search to end surreptitiously without finding the desired optima. Holland [35] gives a good description of genetic algorithms and their function in the optimization of the schema search space.

2.6.1 Genetic Algorithms and their Operation

The classifier system consists of a *population* of classifiers, each composed of a condition and action strings, called *chromosomes*. The chromosomes are defined by the *genotype*. The genotype defines the structure of a chromosome (i.e., how the chromosome looks like, or its gene structure). Each classifier can be evaluated (or decoded) to a given value. This is called the *phenotype* of the classifier and is determined by the genotype. The phenotype is also used to determine the behavior of the classifier. Each position in the chromosome is called a gene and the value of the gene is the called the *allele* and the position of the gene is called the locus. Figure 2.7 shows such a relationship, that Stork et al. describe [69], among the genetic algorithm constituents and their effects on the functioning of the algorithm. Note how this figure shows the effects of adaption in the system. The system gets input from the environment, matches it with its internal "memory" which is encoded in the form of genotypes. These in turn produce system behavior which then affects the structure of the "memory".

In such a system, the classifiers within the population have to compete in order to survive (to preserve the best classifiers in the system). The genetic algorithm process consists mainly of three steps:

1. Select a pair of classifiers from the population according to their strength (select those classifiers that have greatest strengths). The genotype of the parents (those classifiers being selected) defines the offsprings' traits

2. Apply the genetic operators on the selected pairs.

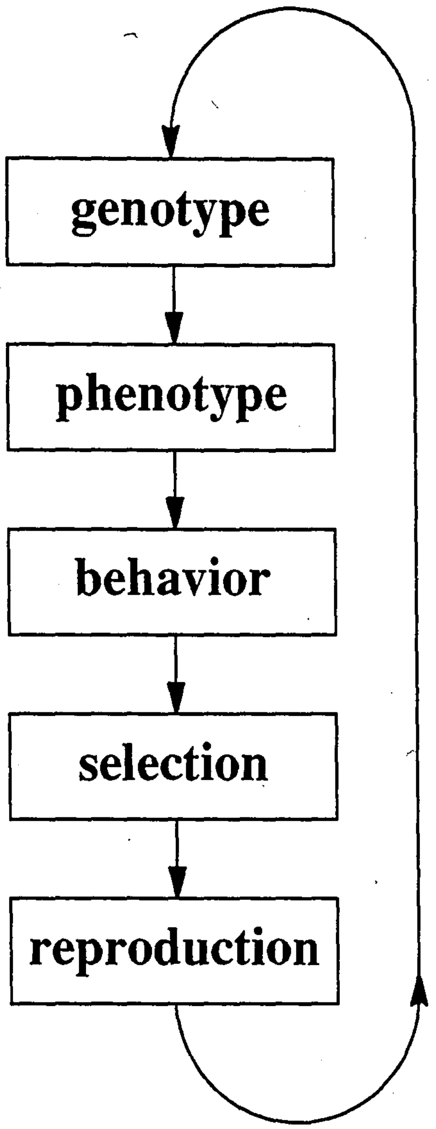


Figure 2.7 The Genetic Algorithm Behavior Mechanism
From: Stork, David G., Jackson, Bernie and Scott Walker "Non–Optimality" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992)

3. Replace the weakest classifiers by the resulting classifiers from Step 2.

Figure 2.8 gives a graphical illustration of how this is done. Assume that a set $G(t)$ of M classifiers $\{C_1, C_2, C_3, ..., C_M\}$ is in the classifier system at time t. Each classifier has strength

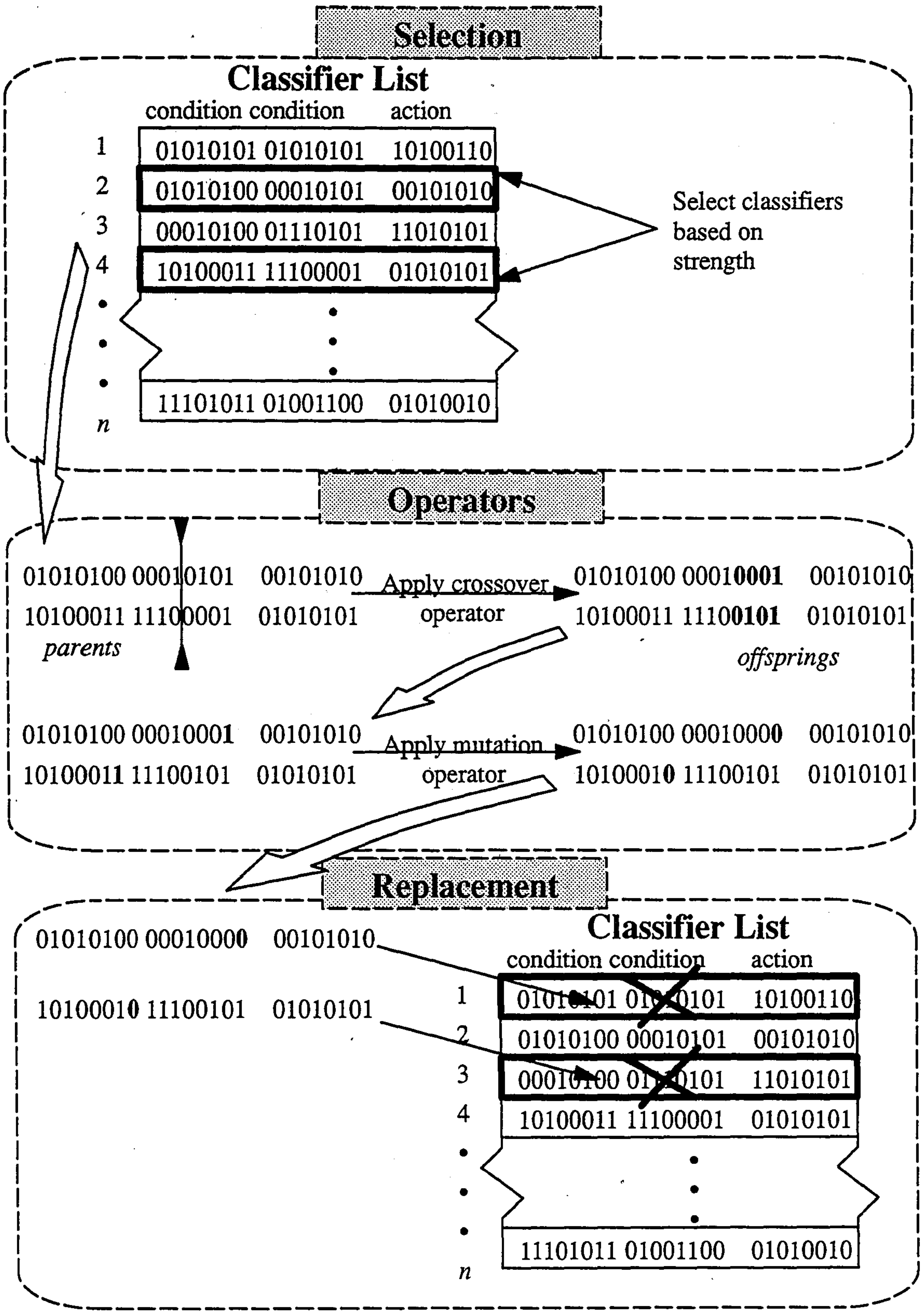


Figure 2.8 Genetic Algorithms in Classifier Systems

$S(C_i, t)$. The genetic algorithm then proceeds to modify the classifier list in the following manner [10]:

1. Compute the average strength $S(t)$ from all the classifiers in $G(t)$ and assign the value $S(C_i, t)/S(t)$ to the classifiers.
2. Assign a probability p_i proportional to the above value to each classifier in $G(t)$. Use p_i to select n pairs of classifiers from $G(t)$.
3. Apply the crossover and mutation operators on the pairs of classifiers selected. *Crossover* is applied by randomly selecting a locus j and then switching the remaining bits in the string between the two pairs. The CFS–C systems allows switching to occur in the entire classifier (so that the two conditions strings and action string are crossed over as one string) or just in the condition strings. Then apply the *mutation* operator which selects an allele and replaces it given a Poisson distribution.
4. Replace the classifiers in the classifier list that have the least strength with the new ones generated.
5. Go to Step 1.

By modifying the classifier list, genetic algorithms handle schemata and therefore affect the building block mechanism of classifier systems. A schemata H (sometimes called *hyperplane* in literature) may have m samples in the population $G(t)$ at any given time, denoted as $m = m(H, t)$. After each major cycle, probabilities (p_i's) of being selected for being crossed over and mutated are computed and applied to the schemata survival. Therefore, at the next time step $t + 1$ the number of schemata in the classifier list is going to be:

$$m(H, t + 1) = m(H, t) * S(H, t)/S(t) * n \qquad (2.11)$$

where n is the number of classifiers that are selected for replacement.

2.6.2 Other Operators that Complement the Genetic Algorithm

There are a number of other discovery algorithms that aid the genetic algorithm and the generation of classifiers. The reasoning here is that the classifier system must be able to be updated with environmental feedback at all times, and the genetic algorithm sometimes cannot handle novel situations.

The *Cover Detector Messages* algorithm is used when there are no detector messages that can be matched in a major cycle [56]. In such a case, which occurs especially when a population is initialized, the classifier list is modified to accommodate the detector messages. The task is managed by checking if at a major cycle rules bid for a message in the message list *M* but no message is matched by the conditions of the classifiers and then making a copy of all those classifiers and assigning a probability relative to the bid. If no classifiers bid for any messages in *M*, then select a parent classifier that matches more closely a message in *M*. To do this, a counter is kept of the number of loci that are similar between the message and the condition of the parent classifier. The higher the count for a classifier, the greater the chances that that classifier will be selected. Then, modify the conditions of the copied classifiers to match the detector message. This will, of course, occur in the successive time step.

Another algorithm that is used by Riolo is the *Cover Effectors Operator*. This operator is triggered when the system is incapable of producing behavior (matching any of the effectors) or when the actions of the classifiers are too similar. In this latter case, those actions would keep on matching the conditions of certain classifiers, making them the strongest and leading to the premature convergence of the knowledge base. When, during a time step, the classifier system fails to produce system behavior (i.e., emit an effector), the Cover Effectors Operator will select the classifiers that bid to have their actions posted. It copies them and randomly modifies the action chromosome by randomly changing a don't care symbol to a zero or one so that they will more closely match an effector.

There also two operators for chaining classifiers together. This concept was purported by Booker et al. [10] and is implemented in CFS–C. The chaining of classifiers is a useful building block strategy. It is not always possible to have one classifier represent an entire concept in a problem–solving situation. Rather, as supported by the concept of emergence in classifier systems, a number of classifiers permit the construction of "trains of thought." *Triggered Chaining Operators* cause one classifier to be coupled with another one. This means that the action of one classifier will satisfy the condition of another one. To accomplish this, a classifier C_2 must make a profit (which is the reward the classifier received minus how much it paid in bids) at time step t and another classifier C_1 must have been active at time step $t-1$. The chaining operator would thus modify the action string of C_1 to make it match one of the conditions of C_2.

Another operator that is used is the *Low–Bidders Operator*. The genetic algorithm modifies those rules that have the highest strengths. But there can be cases where classifiers exist that do not have much strength and are never modified, keeping on producing system behavior even when not desired. In such case, the classifier system will check to see which rules bid below a certain threshold (determined using the strength of the classifier). The amount a classifier bids is a reflection of its strength. Thus, in selecting the low–bidders, the Low–Bidders Operator selects the poorly performing classifiers and modifies their actions randomly. The reasoning here is that the classifiers' conditions may be adequate to the problem being solved, but the action that the classifier is generating is not appropriate and therefore does not produce acceptable behavior.

2.6.3 Genetic Algorithms and Appropriate Usage

Genetic algorithms are appropriate for usage in classifier systems for their robustness in optimizing a number of domains, as I explained before. However, genetic algorithms are also better suited for learning in nonlinear environments since they do not require a monitor during the learning process. Specifically, the environment often displays chaotic behavior and the learning systems must be able to deal with such realistic problems. Some systems, such as neural networks, require a monitor to evaluate the output of the network. A feedback mechanism is needed to analyze the output and send it back for re–processing in case the output was not acceptable (via such devices as *back–propagation*). Genetic algorithms, on the other hand, have no such need and perform well in those cases. This clearly adds to their efficacy and robustness.

Genetic algorithms have, in fact, been used in developing hybrid systems to aid connectionist systems in this aspect. Various models have been developed and seem to be very successful [6] [66]. The advantages tracked in such cases also include the fact that genetic algorithms are good for *global sampling* rather than *local sampling*. This is beneficial if large search spaces have to be searched, which is very possible especially in cases with neural networks with large hidden units, leading to a high dimensionality. Genetic algorithms have also been employed in optimizing the networks [51].

Finally, genetic algorithms are appropriate for usage in classifier systems and other artificial life (evolutionary) systems since they provide the needed flexibility and dynamics to develop complex systems. Other optimization techniques tend to be less efficient from the evolutionary standpoint, due to the data structures that genetic algorithms use and the operations that can be performed on them. The data structures allow the encapsulation of many different types of problems and their manipulation. This flexibility was desired by John von Neumann, claiming that for a complex system to evolve, the mechanisms on which it is built should not be delimiters 48]. Bec [5] and others followed the von Neumann ideal.

2.7 Parallelism and Emergence

Classifier systems fit the category of biological, physical, psychological computational systems found in connectionist systems, cellular automata, ecologies, etc. These systems purport the view that some mechanism exists that produces *growth* of knowledge in time. This view is further enhanced when the concept of sub–symbolic processing is understood. Classifier systems, like connectionist systems, are sub–symbolic systems since computation is performed on very low–level elements rather than high–level symbols as in expert systems. Specifically, classifier systems build their representations on sequences of bits rather than words.

To build these representations, the appropriate architecture is indispensable. The bits of information are to fit like pieces in a puzzle, and time is an important factor. Parallel architectures become important in classifier systems since they can process the classifiers in a distributed fashion. This leads to the emergence of knowledge within a given domain.

2.7.1 Emergence

The concept of emergence is not new. Roger Schank has addressed the issue in the concept of storytelling. According to him, the mind stored information in distributed fashion. Each bit of information is in itself not significant, but when recalled for telling stories, all of the information is put together in a perspicacious fashion, producing knowledge.

Marvin Minsky also confirmed such a resolution when he wrote:

> How can intelligence emerge from non–intelligence?... One can build a mind from many little parts each mindless by itself. I'll call "Society of Mind" this scheme in which a mind is made of smaller processes. We'll call them agents. Each agent by itself can do some simple thing which needs no mind or thought at all. Yet where we join those agents in societies–in certain very special ways–that leads to true intelligence.[5]

In classifier systems, each classifier is one intelligent unit. The classifier encodes information about the environment at a given time. However, the information that it provides is clearly not complete since one classifier cannot encode the state of the entire environment. This is due to the dynamics of the environment and the many possible state the environment could be in. The classifier list could, therefore, be thought of a short–term memory containing different representations (schemata) of how the environment might look like. In order to select the real or actual state of the environment and represent it in behavioral terms, the classifier system has to extract the appropriate representations and, in a sense, converge towards one solution. This is an *emergent* process since the system accumulates information and places it together in a comprehensible fashion.

Self–organization of the classifier system occurs via emergence, where information is gathered and converged to represent the environment as closely as possible. The classifier list is continuously updated until it is fully organized.

Emergence can be thought of a selective process. It is an operation which requires gathering those pieces of knowledge which are most appropriate for representing the environment and generating the appropriate behavior (matching conditions and firing actions). The more information is available for scrutiny, the more focused and precise will the

5. Minsky, Marvin. Society of Mind. p.17

representation be and more adequate the resulting behavior.

2.7.2 Parallelism

Parallelism in classifier systems occurs in a number of ways [21]. First of all, the bits of messages being matched are done so simultaneously. Second, the messages match classifier conditions at the same time (one or more messages on the classifier list matching one or more classifier conditions). Third, more than one classifier can be active at any given time step once the competition is run for the strongest classifiers. Fourth, parallelism occurs at the genetic algorithms level. Here, *implicit parallelism* identifies the situation where each n–gene chromosome can actually represent 2^n schematas. Specifically, consider a schema H from a population of classifiers $G(t)$ at time step t. If $m(H, t)$ denotes the number of schema H in $G(t)$ at time t, then the selection of the schema can be written as [27]:

$$M(H, t+1) = \frac{S(H,t)}{S(G,t)} M(H,t) \tag{2.12}$$

where $S(H, t)$ is the average strength of the chromosomes that are in both the schema H and the classifier population $G(t)$ and $S(G, t)$ are those bit strings that are present only in the population $G(t)$. This equation shows that the as chromosomes are found in H, H grows exponentially. Thus, for n–gene chromosomes in a population of N chromosomes, from 2^n to $N2^n$ schemata can be found.

The ideal architecture for a parallel implementation of a classifier system would be Multiple Instruction Stream Multiple Data Stream (MIMD) rather than Single Instruction Stream Multiple Data Stream (SIMD). Forrest [21] explains that SIMD architectures are "coarse–grained", where each processor (CPU) in the system is powerful enough and can compute large amounts of data[6] and the processing among units is done synchronously. In a MIMD (or "fine–grained") architecture, each processor is not very powerful and the inter–process communication is not rigidly defined. The MIMD architecture would favor the concept of emergence. Each processor is small enough to process some information and interact with other processes in reaching a cooperative solution. This removes the requirement that all classifiers have to be simultaneously recalled (due to the synchronous characteristic of SIMD systems). Only certain classifiers will be activated, given the nonlinear behavior of the environment and the interactions among the activated classifiers cannot be in any way anticipated. As a result, "fine–grained" implementation also tends to remove any bias that there may be in favor of specific classifiers in the system in reaching system behavior.

2.8 Trade–Off Between Knowledge and Search

Raymond Kurzweil [41] justly argues that the human brain is a very large repository of information, where such information is retrieved and accessed via the parallel architecture provided by the many neurons (100 billion) and the many more interconnections that exist among them (1,000). However, the analog computation provided by the brain falls short of the digital power of modern computers. This limitation forces the brain to retrieve information that is pertinent to the current situation being considered on an analogical (or associative) level in order to be able to face problems on a real–time basis. The brain, is in addition, capable of putting this knowledge together in a sensible fashion [62]. Computers, on the other hand, have a much computational power, capable of performing searches within a given state space much faster than the human brain. However, they lack the necessary

6. The nCUBE might represent such a machine with each processor having the power of an Intel 386, while the MIMD architecture might be more like the Connection Machine.

amount of information in order to produce applicable or significant results.

Classifier systems seem to provide a way in between these two extremes. First of all, classifier systems are capable of encoding vast amounts of knowledge within the rules. This is allowed by having a sequences of bits within each condition or action represent a number of concepts (schemata). Schemata augment the number of possible representations and provide for an even greater repository of information. Finally, chaining among various rules further enriches the knowledge base of the classifier system, providing for interrelationships among different concepts.

But classifier systems are also adept at performing fast searches. Through their parallel architecture, having many rules active simultaneously, classifier systems perform searches of the state space that consent real–time performance even given the large repository of knowledge. The rules appropriate to the problem being solved are extracted and applied. The genetic algorithm also aids in this search effort, modifying rules so that many possible alterations are analyzed in a rapid and efficient fashion. An example of a parallel implementation of classifier systems is *CFS, the implementation of the CFS–C program on Thinking Machine's Connection Machine [59]. In this case, 65,000 classifiers were generated and each one was processed on a single processor. Each processor in this system does not have extensive processing powers, but the combination of processors make *CFS gain noticeable performance, illustrating the concept of emergent computing.

Figure 2.9, modified from Kurzweil [41], shows where classifier systems might stand in relation to some of the more traditional AI approaches. They would move to a higher equiperformance isobar thanks to the greater computational power provided by the parallel mechanisms. Classifier systems would move up along the equicost isobars due to their additional capabilities in terms of the amount of the knowledge that can be manipulated. The outcome is greater search power accompanied by additional informative power, a goal artificial intelligence systems of the future have to achieve more and more.

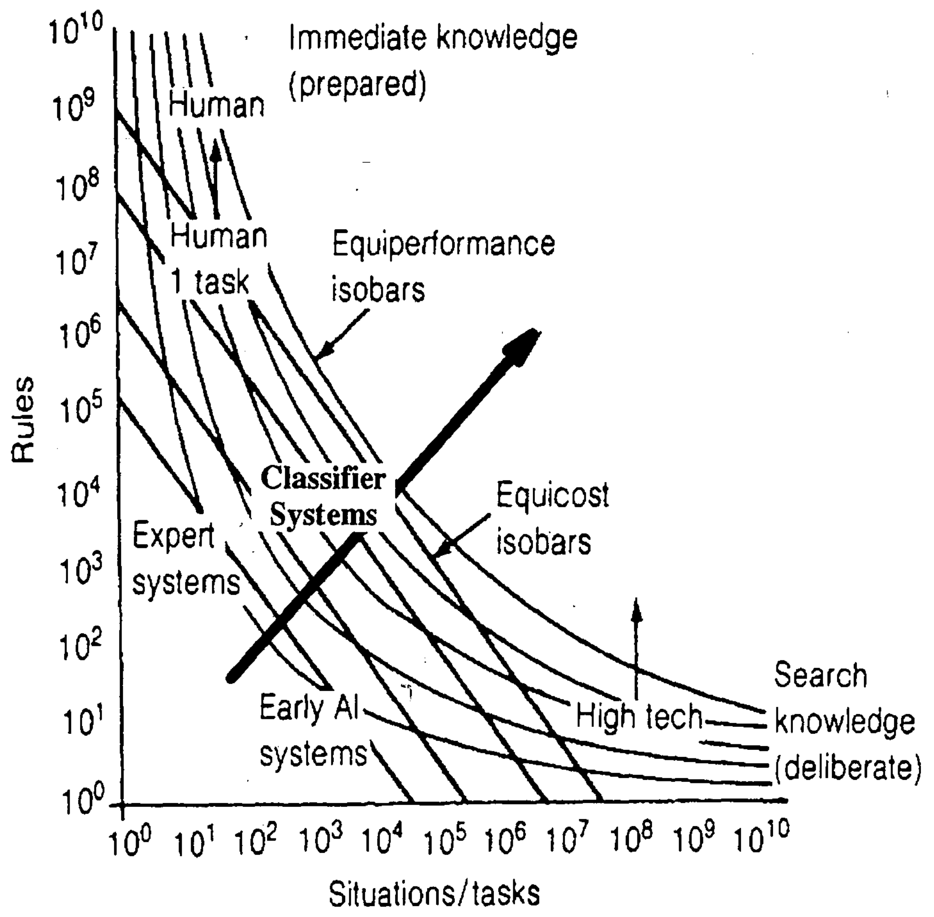


Figure 2.9: Trade–off between Knowledge (Rules) and Search (Tasks)

From: Kurzweil, Raymond. The Age of Intelligent Machines. Cambridge, MA: Massachusetts Institute of Technology Press (1990)

CHAPTER 3

FINITE STATE MACHINE REDUCTION AND COMMUNICATIONS PROTOCOL APPLICATION

Finite State Machines (FSMs) provide a good cognitive model of learning. The FSM mechanism describes a model where certain states designate the condition of the environment at any given time. It, thus, serves as an excellent example of how to simulate an environment for learning purposes. The FSM receives inputs which make it change the state it is in and produce an output (behavior). The desirable outcome in an FSM world is to have the FSM always move to those states which provide the highest rewards or that are simply more attractive. A reward can be viewed as a mechanism by which the environment encourages further desirable behavior. For example, if a person invests money in a company's stock option in the stock market (the environment), any improvement in the company's performance would lead to a rise in the stock's value. The stock market has generated a reward.

Accordingly, a classifier system interacting with an FSM would use the reward as an indication of how it should act in the future. In using such a scheme, where the classifier system can interact with an FSM, many domains can be simulated. Some interesting applications include Riolo's system where human category learning is studied [59] and Zhou's system which teaches a robot how to move in an FSM world and avoid obstacles [76].

Finite State Machines, however, sometimes tend to have a great number of states that may not be consequential or of any use. In addition, the environment contains a number of states that can be represented by other ones. It is important to attempt to design an environment (when possible) which performs better while not compromising its function. Various procedures exist for the minimization of FSMs such as the *k*–equivalence analysis

[40]. Methods such as this one, however, use procedures that analyze the machines *a priori* and are not capable of capturing the rather dynamic behavior of the environment. In order to accomplish this, the classifier system presented here attempts to analyze which states are more useful in the FSM and which ones are not. A simple heuristic is used to function as the judging factor in the FSM reduction task.

One such design task is found in communication protocol design. Communication protocols are designed as FSMs and often have a great number of states that represent the protocol status. The more states there are, the more expensive the design in terms of performance and cost. It is thus important to attempt to reduce as much as possible the design complexity. This is a concern in many such information systems, where the system at times reaches burdening levels of complexity.

3.0 The Finite State Machine World

The FSM used here is the FSW1 model designed by Riolo [55]. The FSM consists of a finite Markov process consisting of a number of states and payoff values associated with those states. The Markov process's current state is used as a detector message that is placed on the message list. Once the classifier system processes the input, it generates output which causes it to modify the transition it will take next. In order to do this, the classifier system will internally "bid" for certain transitions as opposed to others, affecting the movement in the world. Each state has a certain payoff associated with it and that payoff is given to the classifier system as system reward.

3.0.1 The Markov Process

The Markov process is described in the following fashion:

1. A set of n states S_i where $i = 0, 1, \ldots, n$

2. A set of payoff values $v(S_i) \in \Re$ assigned to the states S_i

3. An alphabet Σ defined by the value $r \in \Re$

4. A set of probability matrices $P(r)$ where each entry $p_{ij}(r)$ in $P(r)$ gives the probability of making a transition to state S_j, given that the classifier is in state S_i and the value r has been emitted by the classifier system

5. A set of transitions T_{ij} for each state S_i defining the transition from state S_i to state S_j

Each state S_i in the Markov process is defined as a bit string in the classifier system. Specifically, each state has a unique value associated with it that identifies it from the other states and this value is in bit–string (chromosomal) encoded format. This attribute string is used as the detection mechanism. At every time step, the current state's attribute string is placed on the message list indicating what state the system is in (of course, other messages that classifiers fire for satisfying conditions of other classifiers are also placed on the message list). Each state has also associated with it a next best state the system may move to. This is useful when designing a protocol since sequencing of states becomes important.

The system transits to a given state whenever the r value of the edge leading to that state is emitted. To see how this works, the state diagram, derived from the one in Chapter 1, can be used as an example. Figure 3.1 shows a FSM consisting of five states. Note that each state is defined by a bit string (attribute list) used for placing it on the message list.

Whenever the system activates a effector messages (those starting with "10" and serving as the system's output mechanism), the system will transit to a given state. Specifically, the effector message will be decoded into an integer value representing the r value.

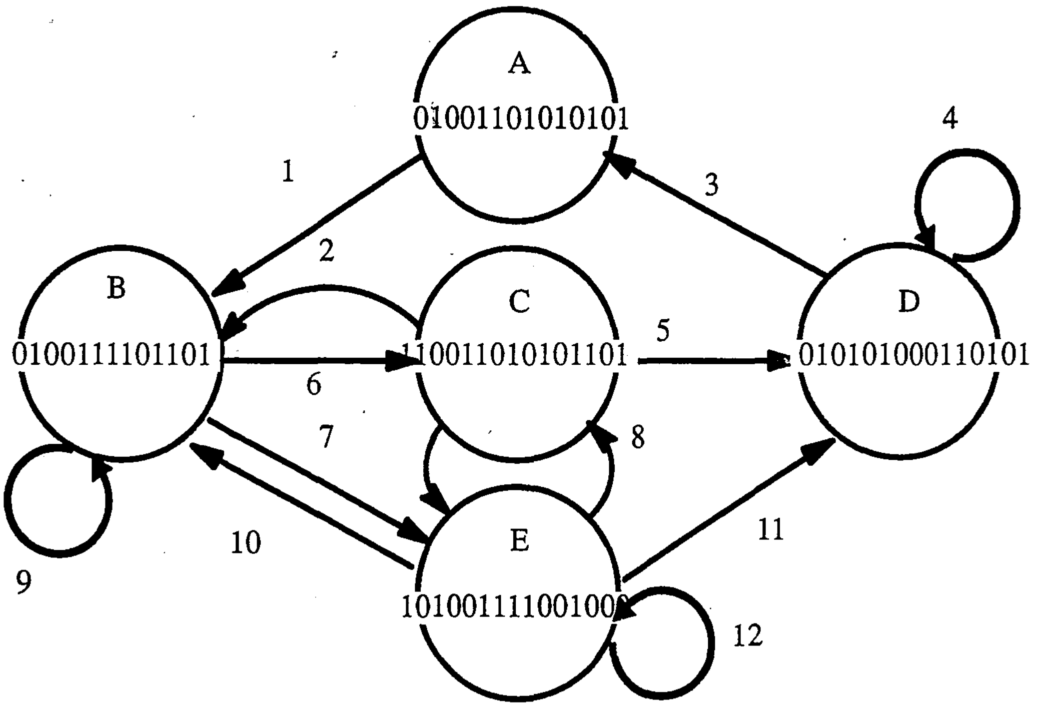


Figure 3.1 A Simple Finite State Machine Defining a Markov Process

Adapted from: Bochmann, Gregor V. "A General Transition Model for Protocols and Communication Service" in *IEEE Transactions on Communications*, vol. COM–28, no. 4, (April 1980)

Each two bits of the message are translated into one bit using the following scheme:

$$00 \rightarrow 0$$

$$01 \rightarrow 1$$

$$10 \rightarrow \#$$

$$10 \rightarrow \#.$$

Hence, when a detector is fired from the classifier list such as 1000000000010001, that translates to (using all the bits except the left–most two which serve as the tag for effector messages) 0000101, which decodes to the integer value 4 (using binary to decimal conversion). Note that if a don't care symbol were used, such as in the message 10000000000000110, that would produce more that one *r* value. This last effector could set *r* to a value in the set

{2, 3} since the effector decodes to 0000001#, allowing both "10" and "11".

The value $r = 4$ will be set only if a bidding competition is won. Since there may be a number of classifiers that will fire an effector message in any time step, the system must determine which r value is the most desired at that time. In order to accomplish this, the system uses a probabilistic approach whereby the amount each classifier bid to have its message posted (or its intensity) is used as a determinant in whether that r value will be used. As an example, if the system is in state 4, and two effectors are on the message list, one which would like to set r to 3 with an intensity of 100, while the other wants to set it to 4 with an intensity of 200, then it will be set to 4.

3.0.2 The Learning Process

The classifier system must learn what states are the most rewarding ones. In fact, when the system goes to a given state, it will receive reward in the form of payoff values. Each state has a reward associated with it. The system then distributes the reward to all the classifiers that are active (i.e., fire actions) during that time step. In doing this, the system is trying to stimulate classifiers to generate r values that move the system to the higher paying states within the FSM. Hence, once the detection mechanism inputs the attribute list of a state which is then matched by some classifier, the system must learn to generate an action string that supports setting the r value to the next reachable high–paying state. Each classifier could also set the action string to return to the same state in the condition string in case that state also provides substantial rewards.

The learning process is also reflect in the emergence of coupled chains [57]. Specifically, a number of classifiers are chained together when the action of one satisfies the condition of another. In the FSM domain, learning is reinforced when a number of classifiers satisfy the condition of another classifier that would fire an effector that would produce desirable *r* values. Hence, if a classifier C would set the *r* value to 4 by firing its effector, then classifier B might want to satisfy the condition of that classifier in order to make C fire its action. At the same time, another classifier A might want to satisfy B so that B will fire its action. These chains will last as long as their importance to the task is safeguarded. Once the states that classifier C supports lose relative payoff in relation to other states, the chain may break down.

As the system is exposed to more environmental feedback, it discovers which particular states provide the highest rewards of all. In such a case, it may well be that if there is one specific state that has rewards greater than anyone else's, then the system will always try to move to that state. This all depends on what the environmental (Markov process) input is. Since the classifier system's behavior affects the environment directly, the environment will tend to react in harmony to the classifier system, unless some chaotic element is introduced.

Such a behavior is clearly synchronous to a cognitive approach since the system attempts to model the environment according to its own needs. The more the system learns from the Markov process, the greater its ability to manipulate that process in its own favor.

An important factor in any learning process is that it should not be precipitous. The classifier system should not evaluate a subset of all states as the best states only after a few time steps. It should have enough time to evaluate all states. In addition, the environment can change any time due to anomalous behavior, heavily influencing the relevance of certain states to the Markov process. Hence, if 2 out of 5 states were important up to time step *t*, those same states may become completely useless in the problem–solving task after time step *t* due to a some event that modified the environment's behavior.

Classifier systems are equipped with a number of mechanisms that prevent rapid convergence to a solution. The most important of these is incorporated in the Bucket

Brigade Algorithm (BBA). The BBA increments the strengths of classifiers gradually in order to support incremental learning. Hence, the system will not only learn gradually, but it will also have time to modify its behavior if later on the environment modifies its behavior. A good description of the BBA and its traits as a graceful credit assignment method are given by Sutton [70].

3.1 The Finite Space Reduction Task

Any Finite State Machine may be composed of a number of states that perform a certain function. However, it often happens that some states might perform overlapping function as other states and their functionality can, therefore, be encapsulated by other states. Such a discovery would lead to a more simplified design of an FSM system, reducing not only the number of states, but also the transitions to those states.

In order to accomplish this task, a method has to be developed to learn which states are really important to the FSM and how can the FSM be reduced. The classifier system here is put to test this problem and designate those states in the FSM that may not be needed.

3.1.1 The Reduction Task Approach

To approach taken here is to make the states prove their usefulness. In order to do so, the state's visit count, or the number of times a state is transited to, is used to determine it worthiness. The classifier system increases the payoff attributed to each state, depending on how many times the state has been visited. Therefore, the payoff also determines state needs. The following equation shows how this occurs:

$$P_i = P_i + \left(Const * \frac{(V_i - V_{\min})^k}{\sum_j (V_i - V_{\min})^k}\right) \tag{3.1}$$

P_i is the payoff value for state i, *Const* is some constant value, V_i is the visit count for state i, V_{min} is the lowest visit count among all states, and k is some value used to calibrate increases in the payoff. The payoff value of the states are increased to give a qualitative estimate of the worthiness of that state. At the end of a task, the payoff values of the states are analyzed and those n states that have the greatest payoff values are classified as being the ones useful to the FSM. Note, however, that such a conclusion can only be preliminary, since there are a number of other facts that determine the legitimacy of a certain state in addition to the payoff value. One such momentous element is state transition. Our model attempts to reach a resolved based on the visits made to a state, which is not necessarily a conclusive factor. It suggests, however, that the FSM might have to be redesigned to incorporate the functions of less frequently visited states into those that are most often visited. There are a number of applications that use the FSM paradigm which may find such suggestion useful since there is a cost associated with having additional states.

3.2 The Protocol Environment

A communications protocol defines the rules of a network that connects various communicating systems together. There are various layers of communications that exist within a certain protocol specification. One such specification is defined by the Open Systems Interconnection (OSI) Reference model of the International Organization for Standardization (ISO) [38]. The functionality of the protocol is divided into 7 *layers*, each of which deals with specific aspects of a communication between two connected stations. Figure 3.2 shows how the OSI looks like and what the various layers perform.

Each layer performs a certain task. For example, the presentation layer defines the data formats and codes in an exchange, while the data–link layer assures reliable

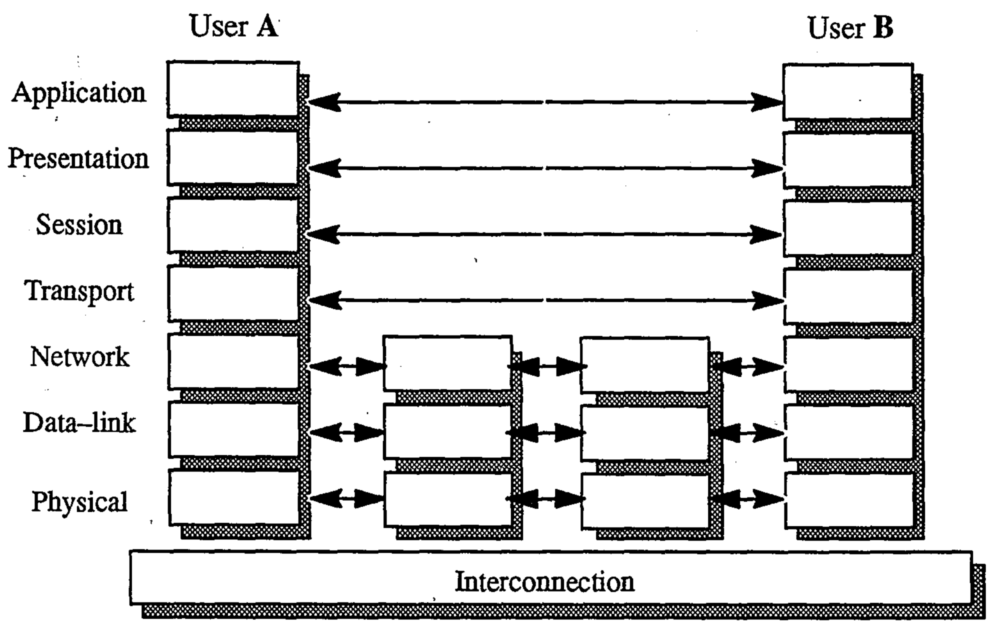


Figure 3.2 The Open Systems Interconnection

transfer of data via error detection and error recovery. A good description of the OSI model is given by Green [26].

Each protocol can be specified as a Finite State Machine, defining exactly in what stage of communication the system is in. Figure 3.1 illustrates such a protocol.

3.2.1 The Protocol Specifications

The communications protocol models the environment that the classifier system uses. The protocol is defined as a rule set that contains a series of input/output sequences. There are a number of possible inputs and outputs that can be generated, depending on the protocol, causing the environment to change its state as a consequence. The classifier system relies on such changes to display its behavior.

As an example, this thesis uses an FSM protocol that was developed by Hashem et al [31] which is used for conformance testing of the *data–link* layer of the ISDN D channel. The protocol defined by Hashem at al. consists of a total of 12 states, 3 of which exist only for testing the reliability of the protocol. There are a number of possible inputs to each state and a number of possible outputs. There is also a list of transitions. The transitions are specified as:

current_state: next_state WHEN *(input, output).*

The inputs and outputs represent actual data that the data–link layers manipulates. Therefore, the data must meet certain syntactic and semantic requirements. In a communication session, data is sent from one station to another in a sequential fashion, such as requesting to send data, sending it, and acknowledging it. The data also has to conform to certain specifications such as size and content.

Transitions can be either *valid, inopportune,* or *illegal.* Valid transitions exist when a certain input is expected, while inopportune inputs are received that are correct syntactically, but do not arrive at the desired time. Illegal inputs are received when the input does not contain the desired data. Appendix A defines some of the data and the state definitions.

3.2.2 Reduction of the Protocol

There are great number of protocols defining communication between two entities. The more complex the protocol for each entity, the more computational overhead there is and the higher delay in communication. It, thus, becomes necessary to reduce the protocol to some design that could provide a more efficient form to represent the protocol.

Such a design would focus on combining certain states that may be transited to given the same or similar conditions or combining a number of states infrequently visited. The classifier system must determine which states are not transited to frequently, and therefore do not provide high returns to the protocols performance. The task then involves deter-

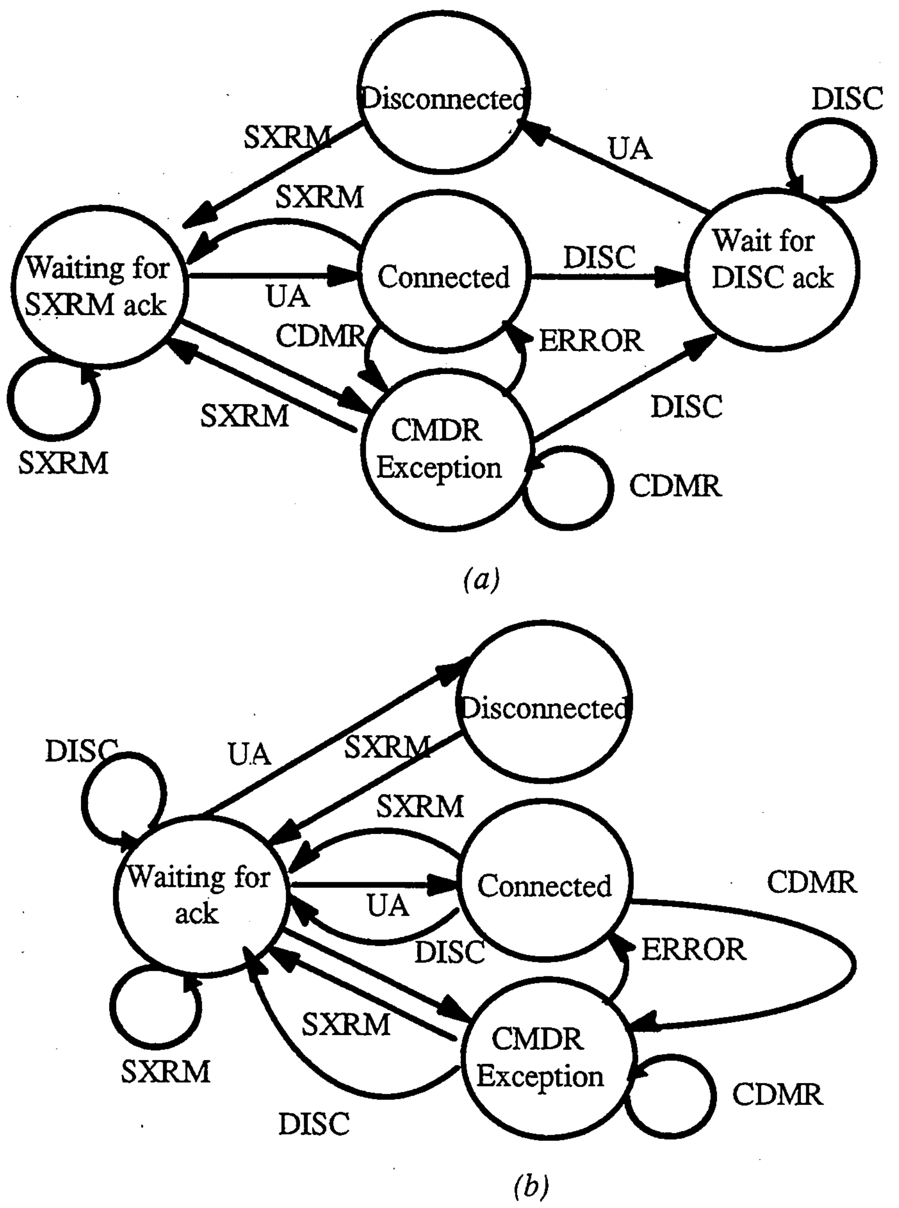


Figure 3.3 Reduction of a Communications Protocol
In *(a)* the original protocol is shown while in *(b)* the reduced protocol is derived.

Adapted from: Bochmann, Gregor V. "A General Transition Model for Protocols and Communication Service" in *IEEE Transactions on Communications*, vol. COM–28, no. 4, (April 1980)

mining whether those states that have been tagged as not generating high payoffs can be combined, as a result of similar transitions. It will also be able to study whether certain transitions can be modified so that the protocol still performs its functions as set forth initially, but does so with a fewer number of states. In other words, rather than having more states, it may be better to have one state performing the functions of a number of other states with additional transitions to that state. A very simple example, just for illustrative purposes, is the one shown in Figure 3.3. Here, the protocol that was shown in Chapter 1 is displayed in part *(a)*. This protocol has two states that define the acknowledgment of some message. However, the two states could be combined into one that captures the functionality of the two separate states. The transitions then are left to decide when to leave those states. Some logic could then be built within the state that would allow the input/output sequences that define the protocol to function as desired. In any case, unique input/output sequences exist for all the states of the protocol, and only the logic within the states defines which output to take given the input received.

CHAPTER 4

EMPIRICAL RESULTS FOR THE FINITE STATE MACHINE REDUCTION TASK

The concepts for which this work is done have now been laid out. It is appropriate to observe what results can be derived from the prescribed methodology. The classifier system is put to test to see if the Finite State Machines to which it presents can be reduced.

The FSM, which models a Markov process, serves as the environment with which the classifier system interacts. The Markov process uses probabilistic transitions which have been set to 1 in all cases in order to model a communications protocol, which is of deterministic nature.

This chapter presents a number of different results that were derived both from the Finite State Machine minimization perspective and also from classifier systems and their performance in this domain. Due to the fact that no other machine learning systems have been employed in this domain, the results that are derived cannot be compared to other learning systems and therefore, one cannot readily assess how classifier systems perform here. There are, however, a number of different test sets that have been generated to give a better grasp of the value and quality of the results gathered. Specifically, such test sets provide benchmarks to test the efficacy and robustness of the classifier system in this domain.

4.0 Testing Procedures for the Finite State Machine Task

A gradual refinement strategy was used in this paper to identify a method by which FSMs could be generated and appropriate tests could be performed on them. First of all, in order to generalize our contribution, the FSM is not restricted to a specific domain. In order to do that, FSMs have been generated randomly using uniform distributions. This includes random initial payoffs, random state attribute values, and random number of transitions.

4.0.1 Preliminary Tests

The test sets that were developed were based on randomly generated FSMs using uniform distributions. The number of states generated was set to 18 for computational purposes. The software was run on an IBM RS/6000 model 950 and could handle 18 states comfortably. The attribute strings for the 18 states were also randomly generated using binary values. The payoffs were generated using a uniform distribution [0, 327]. The number of transitions were generated using a uniform distribution from 0 to 24. Finally, the next best states were produced using a uniform distribution from 0 to 18.

Initially, various machines were generated to test the robustness of the classifier system in its interaction with the FSM environment. The classifier system's major cycle

Parameter Settings			
Population	*200*	*Sharerew*	*1*
Bid_k	*0.10*	*Crowdfax*	*1*
Bidpow	*1*	*Mutprop*	*0.05*
Brpow	*1*	*k*	*1.0*
Effpow	*6*	*Const*	*2.0*
Headtax	*0.0005*	*Maxproducertax*	*0.00*
Bidtax	*0.000*	*Producertaxpow*	*1.5*
Fbidtax	*0.02*	*Crossover prob.*	*0.5*

Table 4.1 Parameter Settings for the Classifier System

was run for 12,800 generation in order to make it interact sufficiently with the FSM domain. Some of the parameter settings that were used are shown in Table 4.1.

All the parameters are described in Chapter 2, except for *k* and *Const* which are described in equation 3.1. The crossover probability is for crossing over entire classifiers rather than each chromosome. Figure 4.1 shows the results for the 18 states that were tested.

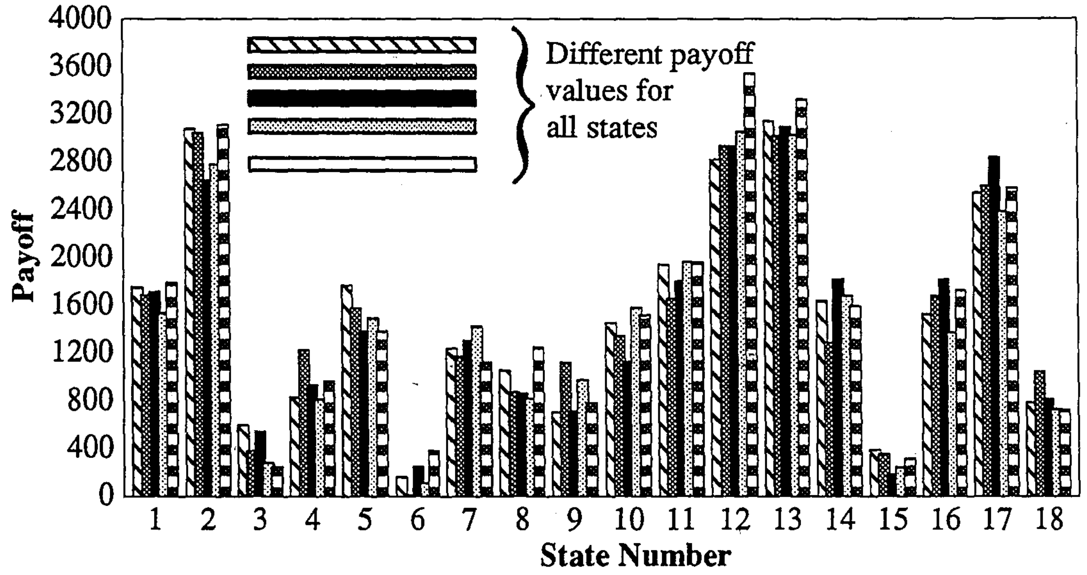


Figure 4.1 Final Payoff Values for a Finite State Machine Using Five Different Initial Payoff Values

The results reflect the payoff values at cycle 12,800 for a randomly generated FSM with five different randomly generated initial payoff values. One will notice that the classifier system demonstrates quite robust behavior to the initial payoff values.

The graph in Figure 4.1 helps in easily distinguishing useful states. For example, states 3, 6, and 15 appear to produce very low payoffs, raising questions as to their usefulness to the FSM. The FSM could be reduced by removing those states and attempting a new design that would incorporate the functions of those states in other states.

There are occasions, however, that its become difficult to select the states that may be removed. One such instance is illustrated in Figure 4.2. The results in this case show that there exists only one state (i.e., 18) that can be possibly discarded, but the automata is

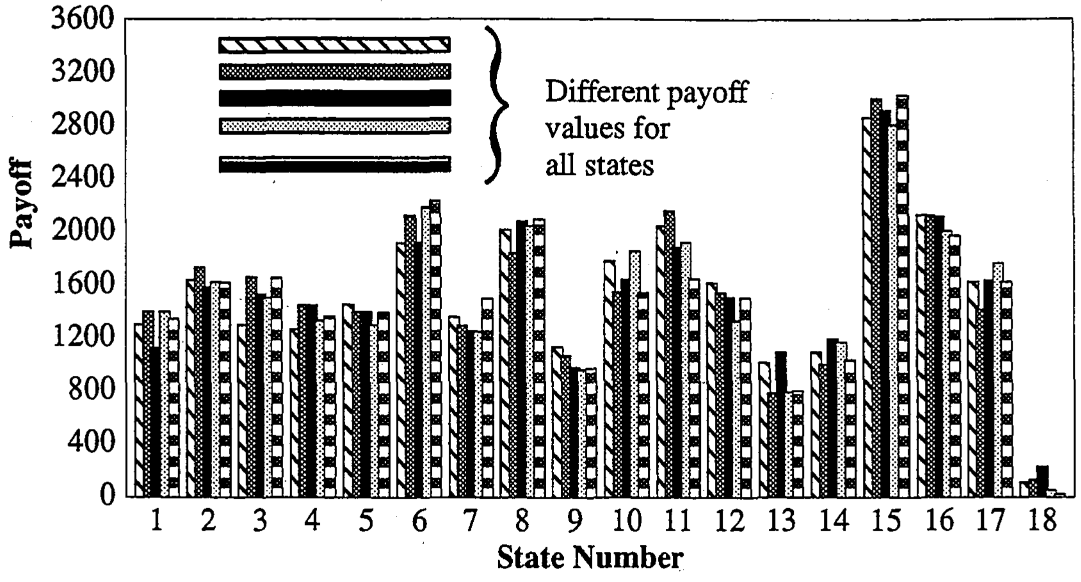


Figure 4.2 Final Payoff Values for a Finite State Machine Using Five Different Initial Payoff Values

quite robust otherwise. Therefore, such machine may already be reduced to start with.

4.0.2 Parameter Settings

In order to develop a method that is capable of confidently discriminating useful states from "useless" ones, the greatest difference in payoff values must be sought. In other words, the greater the difference between the payoff values of the high–payoff states and the low–payoff states, the greater the reliance in eliminating those states.

A number of tests were developed to provide this additional tool. The different test sets utilized a number of different parameter settings of the classifier system. These are shown in Table 4.2. The parameters were selected on the basis of their ability to successfully discriminate among states. They include:

1. *mutprop* – The mutation probability which controls how often a chromosome's allele will be mutated and therefore additionally diversified.

2. *bid_k* – The bid factor represents the percentage of the strength a classifier will lose in case it bids for another classifier but that classifier fails to post a message.

3. *crowdfac* – The crowding factor is the variable that controls how how many classifiers will be selected for each classifier to be replaced. When a weak classifier is to be removed from the classifier list, more than one can be chosen as a substitute.

Test Set #	Different Parameter Setting		
	mutprop	*bid_k*	*crowdfac*
1	0.05	0.10	1
2	0.05	0.10	6
3	0.05	0.20	1
4	0.05	0.20	6
5	0.10	0.10	1
6	0.10	0.10	6
7	0.10	0.20	1
8	0.10	0.20	6
9	Average of above results		

Table 4.2 Test Set for FSM

All three parameters have the capability of strongly affecting the premature convergence problem. In other words, there must be someway of avoiding the classifier list of quickly converging towards a solution. This criteria is indispensable if the best possible FSM must be designed. The environment exhibits nonlinear behavior and in order to capture any alterations in its conduct, the classifier system must be exposed to sufficient interaction. It may happen, for example, that a states x might be initially very relevant to the FSM, but then, due to some endogenous force, state x's strength drops due to changed circumstances. The classifier system must be able to adjust to such abrupt phenomena.

Eight different parameter settings have been generated as seen in Table 4.1. From these, the best settings have to selected.

4.0.3 Results with Modified Parameter Settings

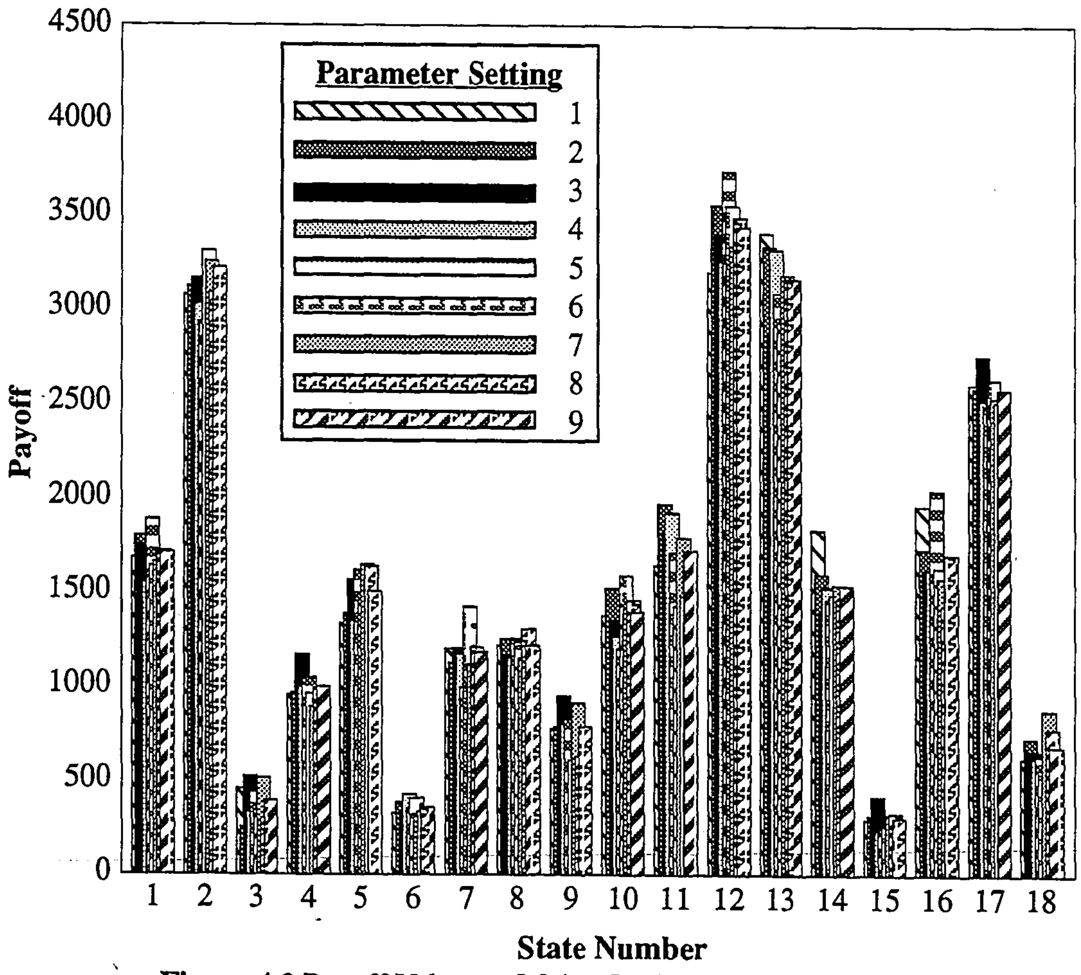


Figure 4.3 Payoff Values at Major Cycle 12,800 Using Eight Different Parameter Setting Combinations

The parameter settings strongly affect the behavior of the classifier system and its ability to determine what states are relevant to the FSM. Figure 4.3 gives the results

State #	Parameter Setting								
	1	2	3	4	5	6	7	8	9
1	1678	1794	1749	1548	1881	1634	1653	1714	1706
2	3071	3118	3159	3023	2929	3303	3246	3083	3216
3	454	249	521	332	439	340	511	297	392
4	954	964	1165	995	1045	961	889	986	994
5	1332	1384	1563	1347	1616	1491	1644	1632	1501
6	324	384	320	426	351	404	321	336	358
7	1200	1123	1203	1172	996	1421	1118	1205	1179
8	1217	1252	1165	1149	1254	1232	1148	1304	1215
9	772	783	948	797	831	610	912	629	785
10	1374	1521	1353	1266	1198	1587	1414	1457	1396
11	1639	1966	1814	1919	1709	1427	1788	1479	1717
12	3195	3551	3394	3255	3731	3548	3337	3487	3437
13	3403	3331	2874	3316	3087	2933	3182	3164	3161
14	1828	1595	1391	1526	1461	1485	1540	1466	1536
15	292	313	416	238	264	262	320	328	304
16	1957	1729	1619	1623	2039	1629	1574	1430	1700
17	2603	2584	2753	2523	2492	2630	2532	2511	2578
19	618	725	658	623	633	562	874	778	683

Table 4.3 Payoff Values for FSM States at Major Cycle 12800 Using Different Values for *mutprop*, *bid_k*, and *crowdfac*

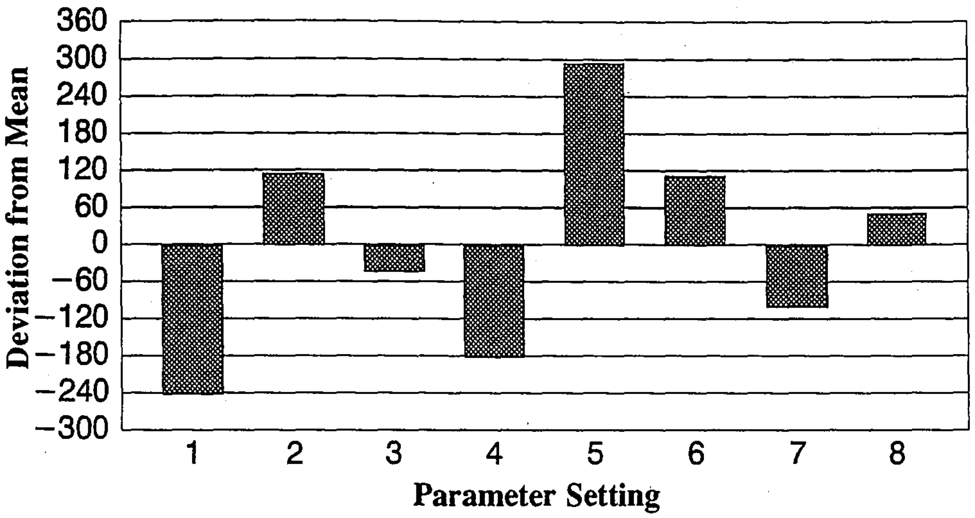


Figure 4.4 Deviations from the Mean of Payoff Values for FSM State 12

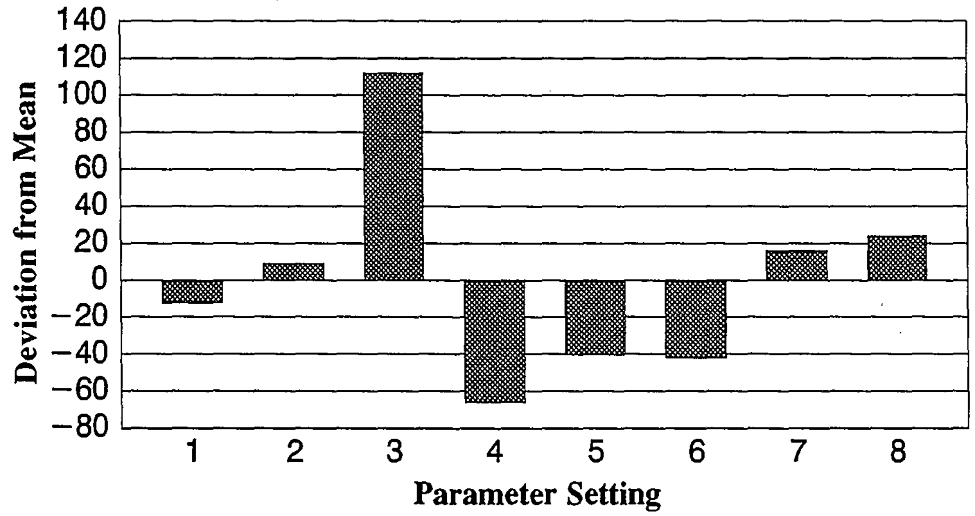


Figure 4.5 Deviations from the Mean of Payoff Values for FSM State 15

for a randomly generated FSM and its payoff values at major cycle 12,800 using the parameter settings in Table 4.2.Note that there are some observable differences among the different settings. To better view the above result, the mean value for the 8 combinations was computed and served as a basis for the variance analysis. Table 4.3 displays the numerical results for Figure 4.3.

Using the mean value (parameter setting 9 of Table 4.2), the deviations from that mean were computed. Column 9 of Table 4.3 gives the mean values. The deviation from the mean values for state 12, with the highest mean payoff value, and state 15, with the lowest mean payoff value, are shown in Figures 4.4 and 4.5, respectively. These figures show which parameter setting provided the greatest discriminatory effects. A complete analysis for all

Test Set	State #								
	1	2	3	4	5	6	7	8	9
1	–28	–145	62	–40	–169	–34	21	2	–13
2	88	–98	–143	–30	–117	26	–56	37	–2
3	43	–57	129	171	62	–38	24	–50	163
4	–158	–193	–60	1	–154	68	–7	–66	12
5	175	–287	47	51	115	–7	–183	39	46
6	–72	87	–52	–33	–10	46	242	17	–175
7	–53	30	119	–105	143	–37	–61	–67	127
8	8	–133	–95	–8	131	–22	26	89	–156
	10	11	12	13	14	15	16	17	18
1	–22	–78	–242	242	292	–12	257	25	–65
2	125	249	114	170	59	9	286	6	42
3	–43	97	–43	–287	–145	112	–81	175	–25
4	–130	202	–182	155	–10	–66	–77	–55	–60
5	–198	–8	294	–74	–75	–40	339	–86	–50
6	191	–290	111	–228	–51	–42	–71	52	–121
7	18	71	–100	21	4	16	–126	–46	241
8	61	–238	50	3	–70	24	–270	–67	95

Table 4.4 Deviations from the Mean of Payoff Values for the FSM at Major Cycle 12800

18 states is shown in Table 4.4. Using this latter table and computing the sum of deviation from the mean, three of the 8 parameter setting combinations were chosen to guide future experimentation. These are shown in Table 4.5.

Payoff	Parameter Setting							
	1	2	3	4	5	6	7	8
Above Mean	901	1211	976	438	1106	746	790	487
Below Mean	1130	446	769	1218	1018	1145	595	1059
Variance	2031	1657	1745	1656	2124	1891	1385	1546

Table 4.5 Best Parameter Settings

4.0.4 Selecting the best Parameter Settings

Now that the number of parameter setting combinations has been narrowed down, the best one among them has to be selected. Specifically, the three parameter setting combinations were put to the test again using three randomly generated FSMs and three different randomly generated payoff values for each FSM. The best combination of settings is found by the testing.

First of all, a heuristic was used here to discriminate among states. Out of the 18 states used, those ten with the highest mean payoff values were regarded as useful and the remaining eight as not contributing to the FSM design. This formed two sets of states.

Figures 4.6 and 4.7 show the results for the first FSM tested using two out of three of the initial payoff values used. It is easy to see that no matter what the initial payoff values, almost the same results are achieved. This result was observed in all three automata. From these results, a variance analysis was carried out to identify the best parameter setting combination. Table 4.6 shows the deviation results for the first FSM used and its first set of payoff values. The deviations are from the mean values in the fifth column. Using this table, all the positive

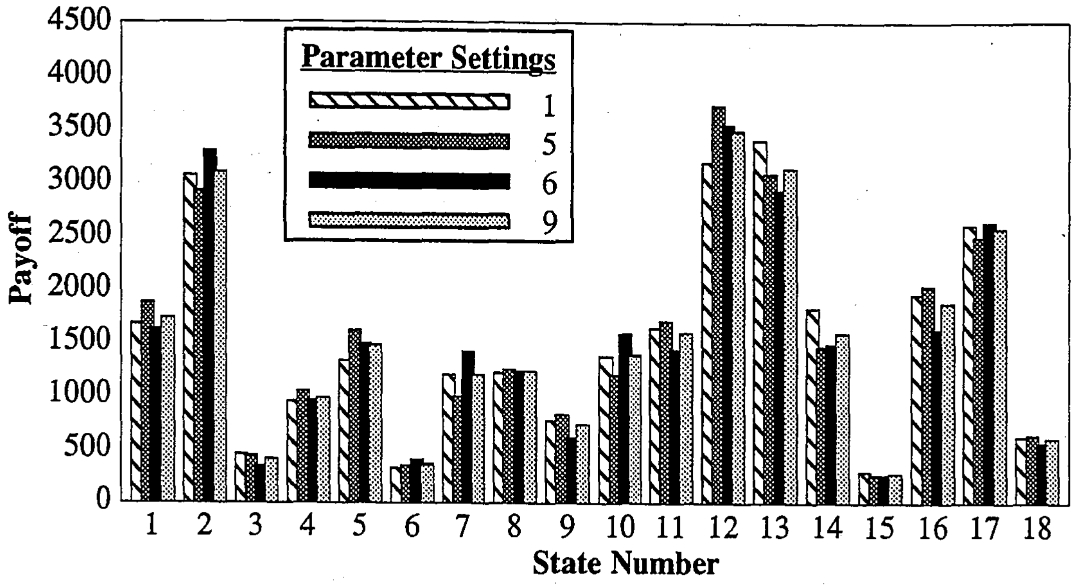


Figure 4.6 Payoff Values at Major Cycle 12,800 for FSM 1 with Payoff Set 1

deviations and negative deviations were summed up for both sets of states (i.e., the best 10 and worst 8) and for each of the different parameter settings. The mean values were then computed for the summations. Finally, deviations from the mean were calculated.

The results for the same FSM of Table 4.6 are displayed in Table 4.7 . For each

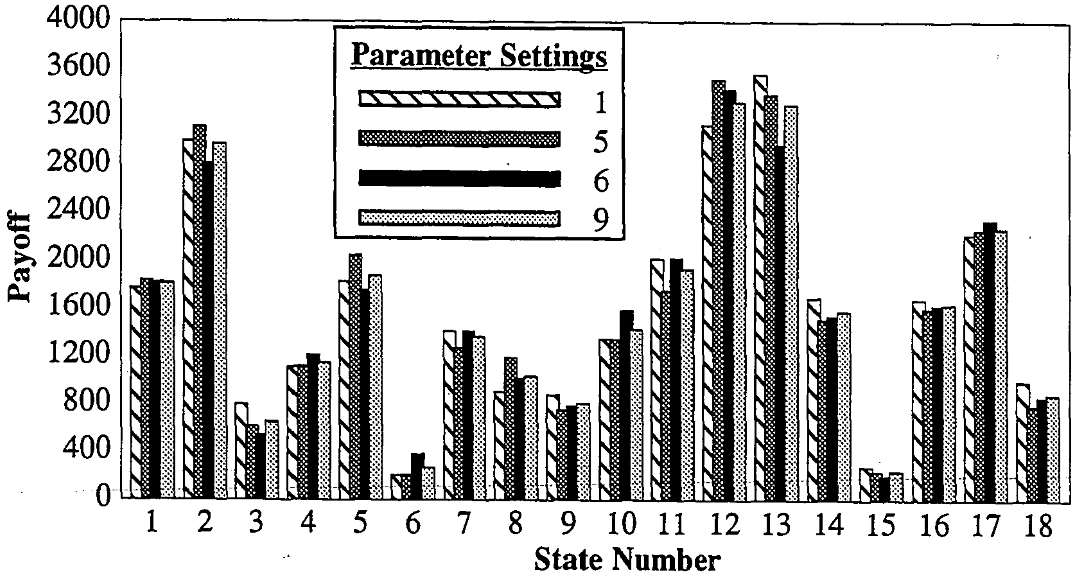


Figure 4.7 Payoff Values at Major Cycle 12,800 for FSM 1 with Payoff Set 1

parameter setting combination, 1, 5, and 6, the mean values where computed for the 8 worst states, the 10 best, and the two sets combined. The deviations were then used to carry out the discriminatory analysis.

State #	Parameter Setting Combination				Deviations from the Mean		
	1	5	6	Mean	1	5	6
1	1678	1881	1634	1731	–53	150	–97
2	3071	2929	3303	3101	–30	–172	202
3	454	439	340	411	43	28	–71
4	954	1045	961	986	–32	59	–25
5	1332	1616	1491	1479	–147	137	12
6	324	351	404	359	–35	–8	45
7	1200	996	1421	1205	–5	–209	216
8	1217	1254	1232	1234	–17	20	–2
9	772	831	610	737	35	94	–127
10	1374	1198	1587	1386	–12	–188	201
11	1639	1709	1437	1595	44	114	–158
12	3195	3731	3548	3491	–296	240	57
13	3403	3087	2933	3141	262	–54	–208
14	1828	1461	1485	1591	237	–130	–106
15	292	264	262	272	20	–8	–10
16	1957	2039	1629	1875	82	164	–246
17	2603	2492	2630	2575	28	–83	55
18	618	633	562	604	14	29	–4

Table 4.6 Parameter Settings for FSM 1 Problem 1, Including Deviations from the Mean

Thus, for parameter setting 1 and state set 8, the mean payoff was 728 and the deviation 369. The deviation ranged from 359 (728 – 369) to 1097 (728 + 369). For the same parameter setting but state set 10, the mean was 2208 with a standard deviation of 787.

Parameter Settings	State Set	Mean Payoff	Standard Deviation	Range of Deviation		Overlap
1	8	728	369	359	1097	324
	10	2208	787	1421	2995	
	18	1550	977	573	2527	
5	8	719	349	370	1068	342
	10	2219	809	1410	3028	
	18	1553	992	561	2545	
6	8	724	431	293	1155	182
	10	2167	840	1337	3007	
	18	1526	997	529	2523	

Table 4.7 Results of Discriminatory Analysis for Different Parameter Settings and State Sets from FSM 1 Problem 1

The range in this case was from 1421 to 2995. The upper range of state set 8 was subtracted from the upper range of state set 10 (1421 – 1097) giving the gap of deviation between the two states which was 324 in this case. The greater the gap, the more the classifier system discriminates between the higher–payoff value states and the lower–payoff value states. This is, of course, a desired factor since the more discrimination there is, the better the confidence in eliminating states. Hence, in Table 4.7, parameter setting 5 gives the best results since it differentiates the most between the two sets.

Table 4.8 contains the discrimination results for all three FSMs and the different payoff values. Note that parameter setting 1 provides the greatest amount of discrimination between the two state sets since the gap in the payoff values of the two sets is the greatest using that setting. In addition, parameter setting 1 also proves to be the most robust since the variation in gap sizes is more constant than in the other two cases. For example, parameter settings 5 and 6 have values as low as 9 and 19 for problem 3–2, while setting 1 remains much higher.

Parameter Setting	Test Set #									Sum
	1–1	1–2	1–3	2–1	2–2	2–3	3–1	3–2	3–3	
1	324	291	345	230	205	201	116	124	50	1886
5	342	246	340	172	122	249	105	9	37	1622
6	182	306	350	153	288	282	47	19	83	1710

Table 4.8 Results of Discriminatory Analysis for all Tests

Therefore, the best parameter setting for the tests thus conducted consists of using a *bid_k* of 0.10, a *mutprop* of .05, and a *crowdfac* of 1. Using these values, an FSM can be designed containing only essential states. The states that the classifier system will identify as "below mean", should be eliminated, if possible.

4.0.5 Using Different Heuristics

The machines that have been used up to now have used a number of heuristics. In order to increase the confidence in the results provided by the classifier system, a number of different machines were generated using a number of heuristics.

First of all, up to now the states of the FSM were divided into two sets consisting of those with the lower payoff values (8 of them) and those with higher payoff values (10 of them). In order to modify the above configuration of sets, the combination was modified to include 10 states in the low payoff set and 8 states in the high payoff set. Such a diversification would allow checking the robustness of the classifier system in discriminating states, even when they are subdivided into variable sets.

In addition, different uniform probabilities were used for generating the FSMs. Table 4.9 displays the two different set sizes using the uniform probability used up to now, namely, 0 to 327 for the payoff values and 0 to 24 for the number of transitions. Table 4.10, on the other hand, uses a range of 0 to 250 for payoff values and 0 to 18 for the number

FSM	Gap Sizes								
	1–1–1	1–1–2	1–1–3	1–2–1	1–2–2	1–2–3	1–3–1	1–3–2	1–3–3
8/10	386	486	157	154	79	55	415	442	341
10/8	486	439	259	212	48	126	389	344	311

Table 4.9 Gap Sizes for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 327] for Payoffs and [0, 24] for Transitions

of transitions. Finally, Table 4.11 uses a uniform probability of 0 to 150 for the payoff values and 0 to 12 for the number of transitions. Each of the tables are divided into three sub–groups of machines representing different, randomly generated FSMs. Hence, in Table 4.9, FSMs 1–1–1, 1–1–2, and 1–1–3 all use the same randomly generated machine but have different payoff values, while FSMs 1–2–1, 1–2–2, and 1–2–3 use a different randomly generated machine, and so on.

FSM	Gap Sizes								
	2–1–1	2–1–2	2–1–3	2–2–1	2–2–2	2–2–3	2–3–1	2–3–2	2–3–3
8/10	114	157	202	574	595	365	347	320	351
10/8	109	164	221	460	458	255	367	348	424

Table 4.10 Gap Sizes for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 250] for Payoffs and [0, 18] for Transitions

The results show that whether the 8/10 combination is used (which is the one used in all the tests) or the 10/8 combination, the gap sizes remain positive. However, the

10/8 combination (which reduced the FSM even further) provides greater discriminatory power. Thus, it could be expected that the more the FSM is reduced, the more the discrimination increases.

FSM	Gap Sizes								
	3–1–1	3–1–2	3–1–3	3–2–1	3–2–2	3–2–3	3–3–1	3–3–2	3–3–3
8/10	82	107	87	102	154	110	351	271	243
10/8	193	301	184	198	222	200	677	568	523

Table 4.11 Gap Sizes for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 150] for Payoffs and [0, 12] for Transitions

Notice that the results also show that the 8/10 combination, which would reduce the FSM from 18 states to 10, generates sufficient discrimination, not requiring additional reduction, which would most probably compromise the functioning of the machine (i.e., with the 10/8 combination).

4.1 Using the Classifier System Appropriately

There are a number of factors to consider when carrying out the tests. The classifier system must be set up appropriately so that it will carefully consider all possible environmental data.

One way to assure that this happens is to avoid the premature convergence problem. If the system very quickly discriminates among states, then it may not have gathered all the needed data from the environment. To achieve some control over this problem, the classifier system provides a few tools. One is that of crowding factors.

When the weakest classifier in the classifier list is to be replaced, a stronger classifier will be used as a replacement. But this may be problematic in some problems since that would cause the strongest classifiers to quickly dominate all the coupled sequences and default hierarchies, since those classifiers would keep on being chosen as replacements. To avoid this problem, a number of rules can be selected for replacement randomly, so that the bias is reduced and other classifiers that may appear to be trivial to the task being solved will have a chance to remain in the classifier list for future use. These latter classifiers could indeed become very useful at some future time. The crowding factor specifies how many of these classifiers will be entered in the replacement pool.

Figure 4.8 shows how the different parameter settings fared with the conver-

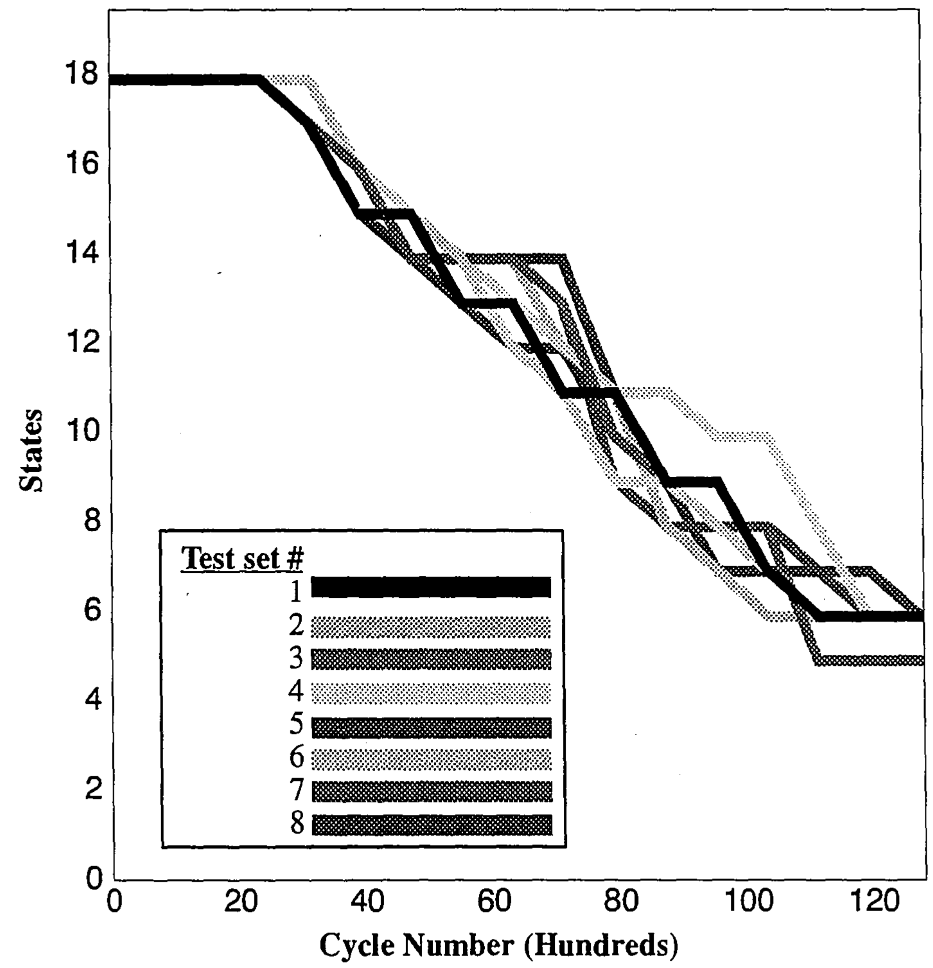


Figure 4.8 Convergence of State Payoff Values Using a Threshold of 1000

gence problem. The graph shows the number of states whose payoff value exceeded a threshold of 1000 at each cycle step. Thus, the more the graph is skewed to the left, the worse the premature convergence problem. From the figure, it is possible to see that the winning parameter setting combination (in black) converges nicely, representing almost an average of the other ones. Another tool is that of sharing rewards. Each time the classifier system produces a desired output, such as going to a state that has a higher payoff value, then the environment generates a reward so that the system will continue to produce such behavior. The classifier system can be controlled so that the reward will be shared only by those classifiers that posted the message and activated an effector or it will be shared among all the classifiers that were active when the system produced its behavior. Sharing among all active classifiers may help some of the weaker classifiers to remain in case they may be needed for future use.

4.2 Protocol Analysis

Now that the parameter setting combination has been selected, the testing can be done on the communications protocol. Appendix A gives a brief description of the protocol and its design. There are 12 states in the model. A number of the states represent timers that are activated when data is not received. The states that represent these timers are clearly not as important as those that actually represent the current state of the communication, but must be included in the analysis since they still represent the status of the protocol (e.g., data not received).

4.2.1 Protocol Reduction

The classifier system, using the parameter settings 1, was run using the protocol as its environment. Two different initial payoff strategies were used. One with fixed payoff values (set to 100.00) and the other using a random uniform strategy from 0 to 327. The results are shown in Figure 4.9. Note that the difference between the two is almost trivial, signalling that the classifier system can recognize useful states even in the presence of some

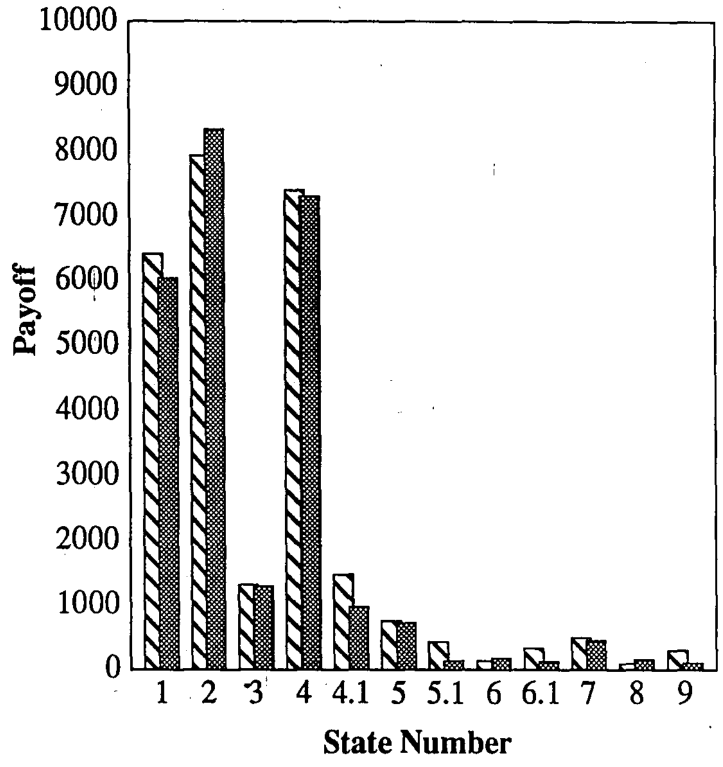


Figure 4.9 Payoff Values at Major Cycle 12,800 using the Communications Protocol

▨	Random initial payoff values
▩	Non–random initial payoff values

“noise”. This robust behavior can be attributed to the fact that certain states are rarely used, no matter what their initial payoff values are.

The LAPD protocol defined here has a number of states that could be classified as being used infrequently. States 5 through 9 in this case might be considered as not contributing to the protocol design as a whole. These states are transited to when a data packet is not received in sequence (states 5 and 5.1), when a busy signal is received from the receiving end (states 6 and 6.1), or when a timer expires (states 7, 8, and 9). The timer could probably be combined into a single state and a logic could be set up to still provide the appropriate transitions. Minimizing the protocol would save on design costs in this case.

The result here achieved can also be used as a test for the classifier system’s learning ability. It would be logical to assume that states such as the ones that have been pinpointed by the classifier system would be rarely visited. Hence, the classifier system has learned to identify them correctly. The learning that has taken place can be observed by the number of schemata in the classifier list that support each state. Figure 4.10 shows the num-

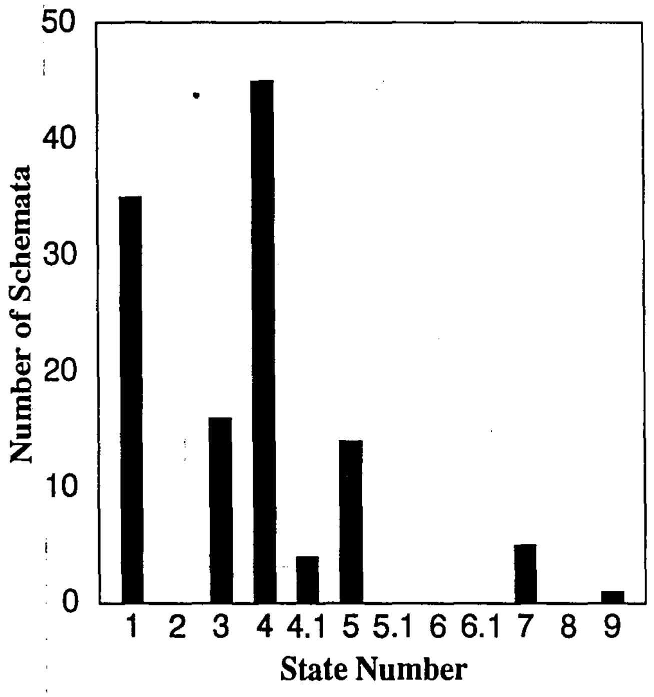


Figure 4.10 Number of Classifier Schemata in the Classifier List Supporting Each State at Cycle Step 1000

ber of schemata supporting the protocol states while Figure 4.11 shows the same figure superimposed payoff values in the non–random case. Figure 4.11 shows that once learning is complete, the classifier list converges towards the states with the highest payoff values. Again, we must specify that the result that is produced by the classifier system is based on the transitions and visitations to those states. The "support" for the winning states can then be used for actually deciding whether the reduction of those states would be acceptable and plausible.

Related to the support for certain states by the classifiers is the increase in the strengths (or fitnesses) of the classifiers. A growth in strength is a sign that learning is taking

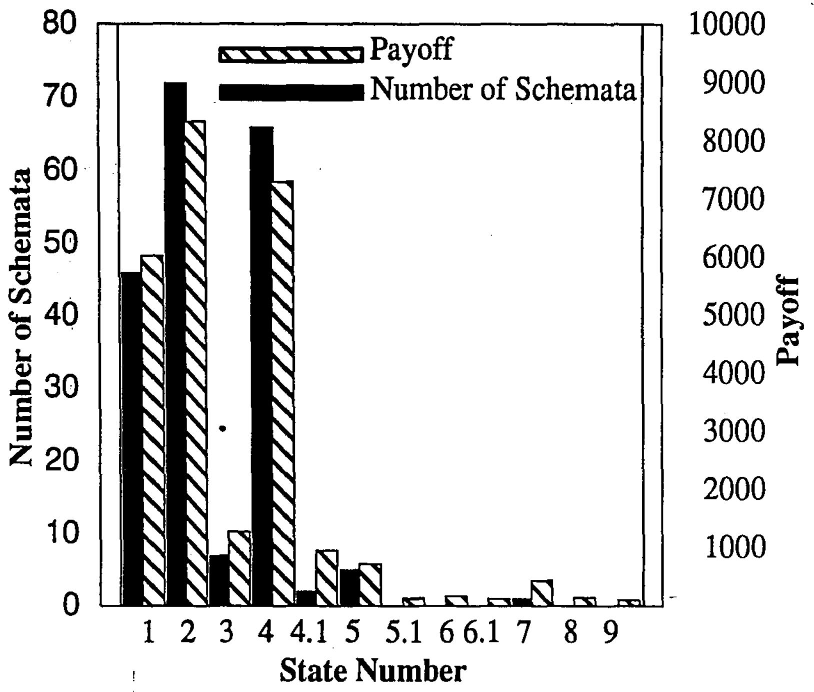


Figure 4.11 Number of Classifier Schemata in the Classifier List Supporting Each State at Cycle Step 12,000 Superimposed with the Payoff for those States at the Same Time

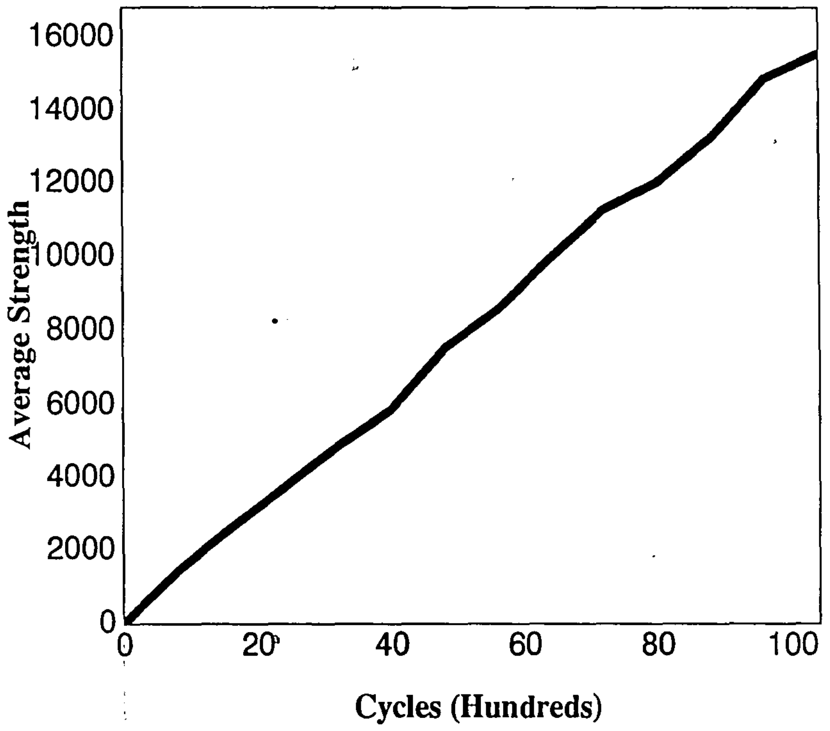


Figure 4.12 Flow of Strength in Classifier List

place and that the classifier system is "confident" in the solution it is reaching. The growth in classifier strength is mapped in Figure 4.12. Note that the graph increases constantly with no disruption, marking the confidence the classifier system has in the problem solving task.

4.2.2 Protocol State Convergence

As it was with the generic automata that were tested earlier in this chapter, in minimizing the communications protocol attention must be paid to rapid convergence. Figure 4.13 shows the convergence of the classifier list towards the final set of states using

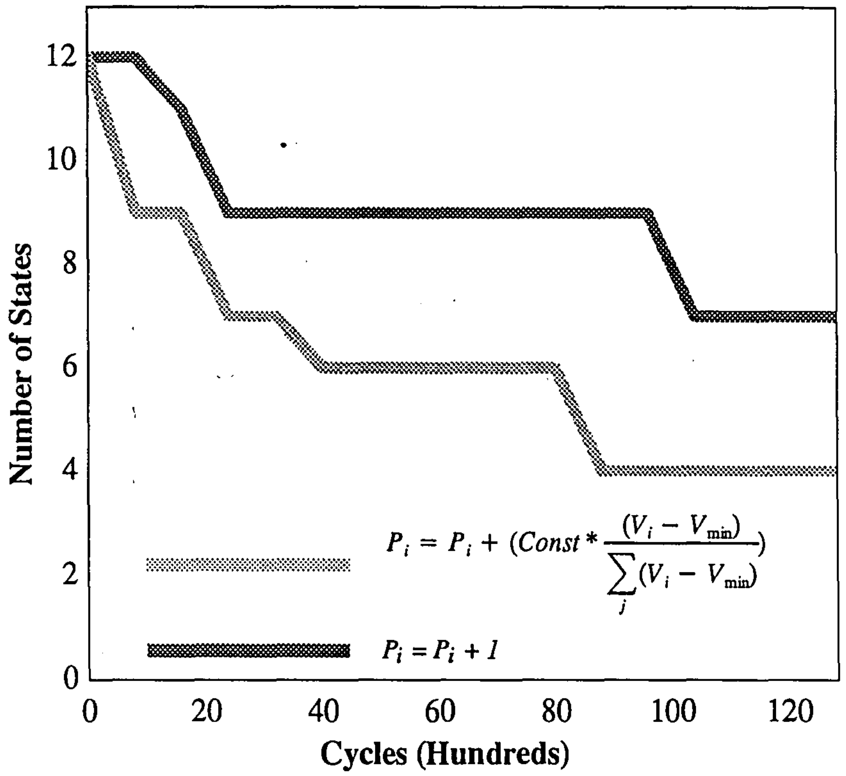


Figure 4.13 Convergence of Payoff Values for Protocol Using Three Different Heuristics

a threshold value of 500.

The figure shows two different curves each representing the heuristic that was used in increasing the payoff value. The first one, which uses equation 3.1, is used in all the testing done so far. It is the one that provides a very gradual convergence and does not discriminate as much as the other one. In fact, the other equation, $P_i = P_i + 1$, causes the system to very quickly converge towards a solution which would eliminate most of the states. Such a solution would not be very realistic. It is desirable to reduce the number of states in the design, but up to a certain limit. Otherwise, the functionality would certainly be compromised. Figure 4.14 gives a second way of looking at the convergence problem. The figure

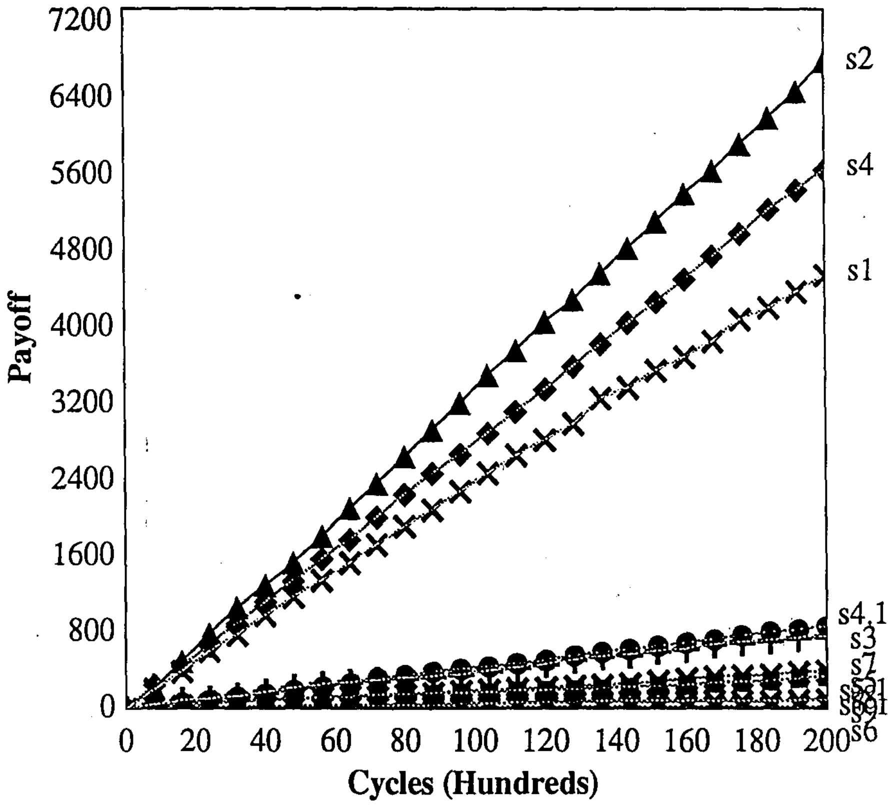


Figure 4.14 Increase in Payoff Values for Protocol

shows how quickly the payoff values for the states increase. Note that three states very quickly receive high payoffs for they are the most commonly visited ones. Others increase gradually while a few remaining ones have almost no growth in payoff.

4.2.3 Perturbation and its Effects

It was mentioned that the environment at times displays nonlinear behavior, which affects the performance of the classifier system. The performance is especially affected when the system has been interacting with the environment for a long time and learning has taken place to a great extent. If the environment is perturbed, the classifier system must adapt to the change and learn again, even though the classifier list may have already converged to a certain extent.

To see how such an event can affect the system's behavior, the payoff values of the states were randomly changed and re-mapped using the same probabilistic approach used to assign them initially. The *simulated* perturbation was injected half way through the run. Figure 4.15 shows the growth in fitness of classifiers that support two states. State 2 is

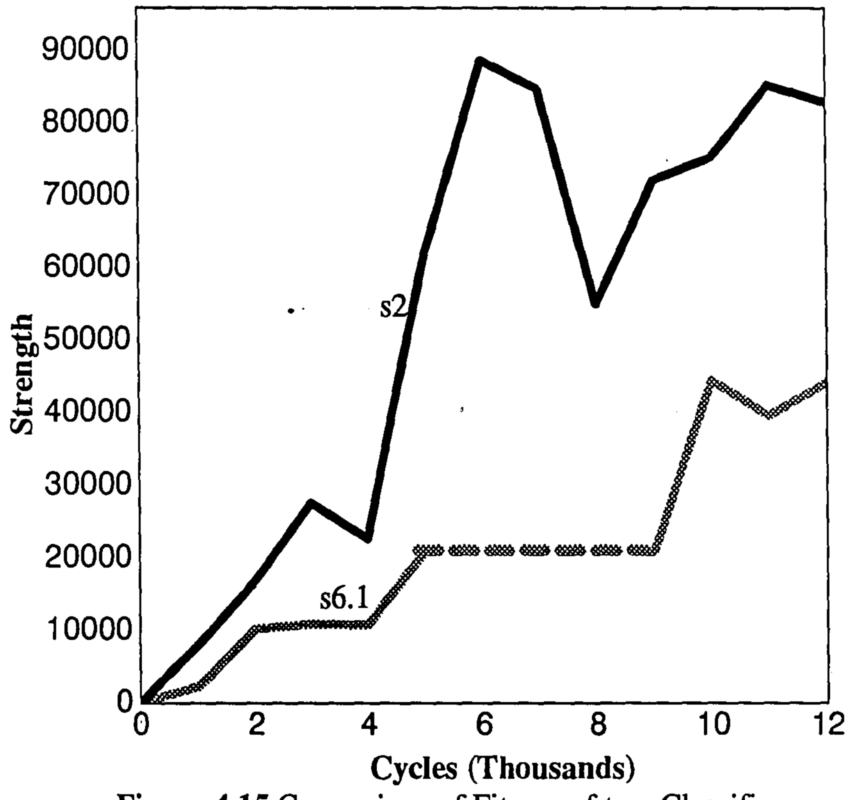


Figure 4.15 Comparison of Fitness of two Classifiers Supporting States 2 and 6.1 with no Perturbation

the state that receives the highest payoff during a normal run while state 6.1 is one of the low–payoff states. State 2 receives payoff quickly and its fitness increases rapidly while state

6.1 increases more slowly. In addition, the area that is dashed represents a discontinuous growth in state 6.1's curve since no classifiers existed in the classifier list in support of that state.

Figure 4.16 shows the same two states but with the perturbation introduced

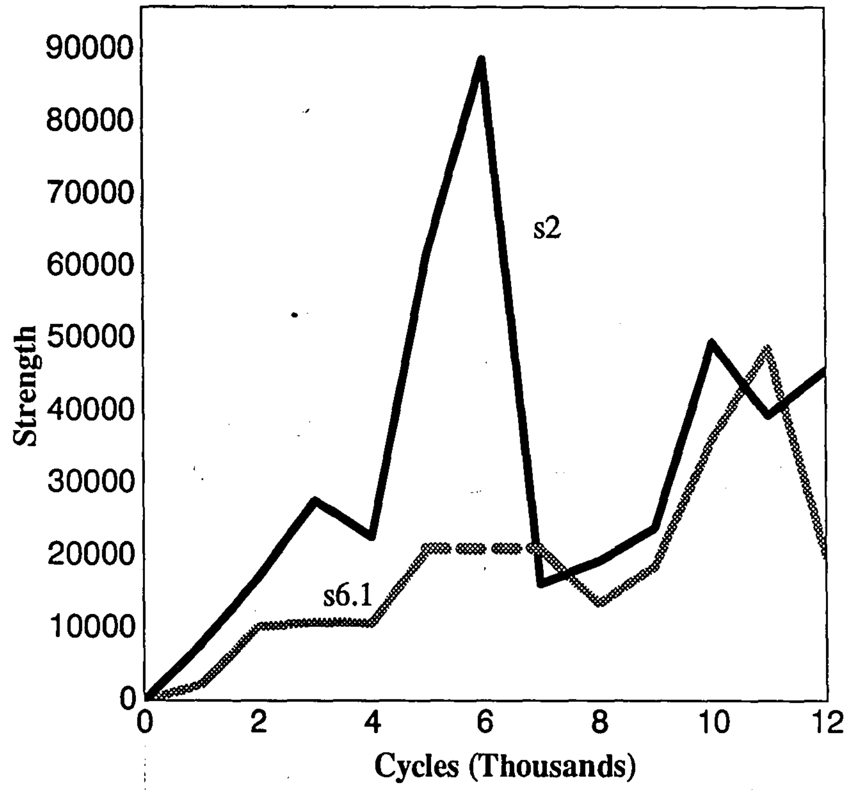


Figure 4.16 Comparison of Fitness of Classifiers Supporting States 2 and 6.1 in Presence of Perturbation

after cycle 6000. Note that the fitnesses fall to almost their initial values and then start increasing again. But, in addition, state 6.1's curve rises almost as fast as state 2. That signifies that the initial payoff may have moved in favor of state 6.1. But the classifier soon learns of the greater importance of state 2 and that curve starts growing faster.

4.2.4 Rewards

Rewards are essential for the classifier system to carry out its task properly. If no reward is received, the classifier system cannot determine whether it is discriminating properly among states. This is a problem that exists in many learning systems. Connectionist systems, for example, include a "tutor" that check the output that the network produces and see if the output is correct.

Figure 4.17 shows how the average fitness of the classifiers in the classifier

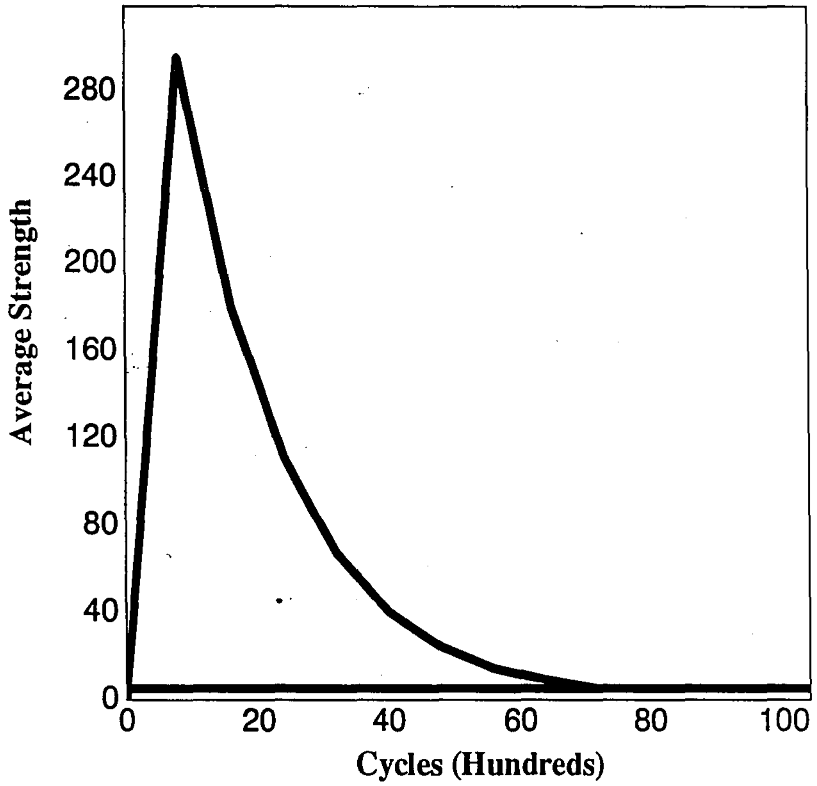


Figure 4.17 Flow of Strength in Classifier List with no Rewards Received

list in the absence of rewards. Clearly, when no rewards are received, the system loses a sense of the worthiness of states and cannot successfully determine which states should be allowed into the protocol design. If on the other hand a reward mechanism exists and guides the system, then the design benefits from the advice that the system produces.

Sharing rewards can also affects system performance to. When rewards are shared among all active classifiers in the classifier list, the fitness of the list is enhanced on the average, allowing classifiers that are weaker to remain in the list for a longer period of time. A more aggressive strategy would allocate the reward only to classifiers that post their actions and produce effectors. Such a "greedy" approach would quickly eliminate weak classifiers and lead to a much more rapid convergence of the classifier list. This may be desired in environments that are deterministic and are not affected by abrupt changes. Such environments are not too commonplace, however.

The communications protocol, in fact, can often be affected by sudden changes and the classifier system must adapt to them. Figure 4.18 shows the difference in the average fitness of the classifiers between an approach that recompenses all active classifiers and one that does not. For the most part, sharing rewards allows the average fitness to be higher than in the case were no rewards are shared. The tests that have been performed

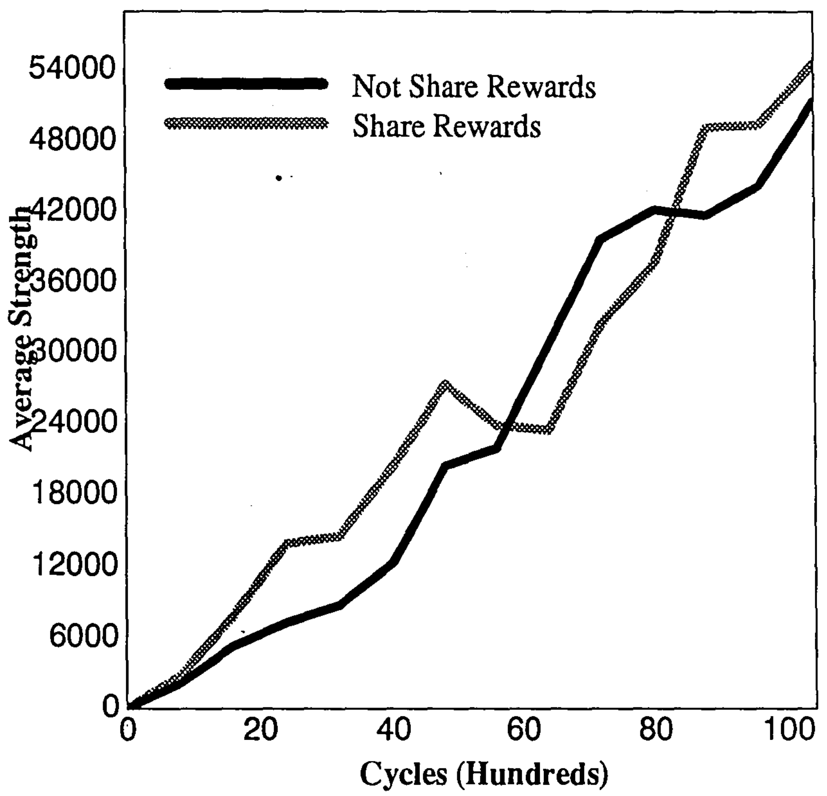


Figure 4.18 Difference in Fitness Between a Strategy which Shared Rewards Among all Active Classifiers and One that Does Not

up to now have utilized the greedy approach since the intent has been to discriminate the most among protocol states. In addition, in this simulated environment, the amount of perturbation is trivial, even though it could be induced, as was shown before.

CHAPTER 5

CONCLUSION

The results thus achieved demonstrate that the classifier system learning methodology can successfully identify states within a Finite State Machines that can be eliminated, reducing the complexity of the machine. As specified earlier, the classifier system functions more as an advisor than an implementor. In other words, the classifier system can only advise an FSM designer (for a specified domain such as communication protocols) what are the states in the system that could be removed without compromising the functioning of the machine. If the machine is then affected in its functionality, then the reduction cannot take place.

5.0 Results

The reduction procedure consisted of identifying what parameter settings would provide the greatest discriminatory results between the two sets of states (low payoff and high payoff) in the FSM.

Successful reduction requires that the classifier system be able to bias more towards those states that generate the higher payoffs. The greater the discrimination, the more confident the results. In Chapter 4 a discriminatory analysis was carried out which served to test this confidence. The mean payoff were computed for 9 different FSM, each with three different payoff sets. From these, the standard deviation were computed and the gap separating the lower range of the high payoff states and the higher range of the low payoff states was computed. The gap served to identify the amount of discrimination displayed by the classifier system.

Another way of analyzing this information is to consider what percentage of the range of deviation for all 18 states the gap covers. In other words, how much does the classifier system separate the two sets of states from one another? The greater the gap, the more the two sets of states are apart and the more confident the solution.

This type of analysis is carried out in Tables 5.1, 5.2, and 5.3 which are related to Tables 4.9, 4.10, and 4.11, respectively. The tables show the percentage that the discriminatory gap covers range of deviation for the payoff values for the 18 states of the FSM. Hence, if the mean for one of the machines for all 18 states were 1000 which a standard deviation of 250, then the total range of deviation would be (1000 – 250) + (1000 + 250) = 2000.

FSM	Percent of Deviation								
	1–1–1	1–1–2	1–1–3	1–2–1	1–2–2	1–2–3	1–3–1	1–3–2	1–3–3
8/10	12	15	5	5	3	2	13	14	11
10/8	15	14	8	7	2	4	12	11	10

Table 5.1 Gap Sizes as a Percentage of the Range of Deviation of Payoff Values for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 327] for Payoffs and [0, 24] for Transitions

If the gap were 200, then the percentage that the gap covers is 10%. This number gives a qualitative estimate of how much discriminatory power the classifier system produces.

FSM	Percent of Deviation								
	2–1–1	2–1–2	2–1–3	2–2–1	2–2–2	2–2–3	2–3–1	2–3–2	2–3–3
8/10	3	5	6	19	19	12	11	10	11
10/8	4	5	7	15	15	8	11	11	13

Table 5.2 Gap Sizes as a Percentage of the Range of Deviation of Payoff Values for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 250] for Payoffs and [0, 18] for Transitions

Hence, the greater the percentage, the greater the discrimination. The 10/8 combination provides more discriminatory power than the 8/10 combination, on the average.

Once these tests were completed and the best parameter setting was selected for usage on the FSM, the communications protocol was tested. This machine consisted of 12 states. The minimization procedure that was carried out showed that there are a number of states in the protocol that could be tentatively eliminated, given that the functionality of the protocol is not compromised. Any reduction in the size of that protocol could contribute to a reduction in the cost of design and a noticeable increase in performance.

FSM	Percent of Deviation								
	3–1–1	3–1–2	3–1–3	3–2–1	3–2–2	3–2–3	3–3–1	3–3–2	3–3–3
8/10	3	3	3	4	4	4	12	9	8
10/8	6	10	6	5	7	5	24	20	19

Table 5.3 Gap Sizes as a Percentage of the Range of Deviation of Payoff Values for Three Different FSMs Using Two Different Payoff Sets with Random Uniforms [0, 150] for Payoffs and [0, 12] for Transitions

Hence, it was suggested that some of the states representing timers in the protocol be reduced to one state generalizing the functions of all of them, but producing the same outputs.

5.1 Communication Protocols Used as the Environment

Clearly, the method developed here can be expanded to usage in other FSMs and protocols. Another type of environment that could be constructed would consist of a number of different communicating protocols (i.e., each representing a different node in a communication network). Hence, each protocol would represent a separate state in the environment. The system could then find the optimal number of nodes in such a communication system. The classifier system could determine what the load is on such a system and try to find a better configuration. The system could then learn to identify such problems in a number networks and attempt to provide similar solutions, if acceptable.

A more challenging task, derived from the above, would be to find out how the network could be expanded in case the network is already at high capacity. The classifier system would have to analyze the load and determine an appropriate new configuration.

5.2 Classifier Systems Revisited

The above results that have been achieved are encouraging and demonstrate the capabilities that classifier systems have in interacting with different environments and solving different types of problems. However, there are a number of ways to make them work more effectively.

First of all, a number of different strategies could be used for crossover and mutation, that have recently shown greater success than the traditional operators used in CFS–C. A number of different crossover operators have been proposed, including *single arithmetic crossover, whole arithmetic crossover* [44], *partially mapped crossover, order crossover, cycle crossover,* and *position–based crossover* [67]. Some mutation techniques include *boundary mutation, non–uniform mutation* [44], *swap, reverse,* and *remove–and–reinsert* [41]. These techniques could prove to be more appropriate for the learning process, generating more robust behavior.

A second way of improving classifier system performance might be of that of incorporating the concepts of synthetic systems [52]. Classifier systems demonstrate simulation systems, where the genetic classifiers and the environment with which they interact are simulated. Specifically, classifier systems are designed *a priori* with an understanding of what the internal configuration of the system is and what the environment looks like. But this may not be sufficient. Rather, there may be a need to introduce some elasticity, allowing classifier systems to be independent of the design specifications. One example might be that of the population size in the classifier system, which is fixed. But fixing the number of classifier allowed might greatly influence the performance of the system, because at a given time the classifier system may need some of the classifiers that are being replaced. This is especially true when an *elitist* strategy is used, attempting to maintain the strongest classifiers at every time step. But as it was mentioned, very often weak classifiers may be needed in the future, and a simple heuristic that randomly keeps a few for future use may not be sufficient. Rather, the classifier list should be able to grow and shrink, depending on the status of the environment with which it interacts and the relevance of the classifiers.

The environment can also be synthesized. The environment does not change only its state or condition, but also its appearance. In such an environment, the classifier system must be able to update its representation of the environment at all times. This is one way of learning too, characterized by learning to recognize diverse phenomena.

Third, the incorporation of long–term memory would be helpful in the classifier systems. Zhou [77] already implemented such a system using an associative memory. Another mechanism that can be used is that developed by Beale et al. [4]. This implementation uses a distributed associative memory for storing information that is classified during a run of their connectionist system. The memory is composed of a matrix that consist of a number of vectors (representing the size of the memory) to which the input vector (or in the classifier system, the detector) would be matched. If a perfect match occurs between the input vector and a vector that was stored, that vector is recalled. Otherwise, the closest match is used by counting the number of bits that match between the two vectors. The vector with the highest match is then recalled. Beale et al. describe the memory in the following fashion:

1. A memory matrix M_{ij}

2. An input vector (detector) A_i

3. A class vector (the vector in memory) C_j

When the system is learning, the memory stores all the vectors in memory:

$$M_{ij} = \begin{cases} 1 & A_i, C_j = 1 \\ 0 & otherwise \end{cases} \quad \forall\, i, j$$

and during recall, the vectors are fetched back:

$$Recalled\ vector\ R_j = \sum_{p=0}^{i} M_{pj} * A_p \quad \forall j$$

This can easily be applied to classifier systems since the population could be initialized using the above method. Rather than randomly generating the population, a detector could capture the current status of the environment (used as the input vector to the memory) and recall stored classifiers. The above equations would have to be modified to include the don't care symbol #. If enough classifiers cannot be recalled using this method, the remaining members of the population could be generated randomly. Even in such case, the learning process could be greatly improved.

These and other extensions to classifier systems may prove to greatly increase their learning capability and enhance their robustness in a number of different domain. The ultimate goal would be to have a classifier system that can adapt to a number of environments. The technology to date does not provide such flexibility, even though its generality could be a powerful tool.

REFERENCES

[1] Ackley, David, and Micheal Littman "Interactions Between Learning and Evolution" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp. 487–510

[2] Antonisse, Hendrik James and K. S. Keller "Genetic Operators for High–Level Knowledge Representations" in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[3] Antonisse, Hendrik James "A Grammar–Based Genetic Algorithm" in Gregory J.E. Rawlins (Editor) Foundations of Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers (1991)

[4] Beale, Russell, Finlay, Janet, Austin, James, and Micheal Harrison "User Modelling by Classification: a Neural–based Approach" in Taylor, J. G. and C. L. T. Mannion (Editors) New Developments in Neural Computing IOP (1989), pp. 103–110

[5] Bec, Louis "Eléments d'Epistemologie Fabulatoire" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp. 799–812

[6] Belew, Richard K., McInerney, John, and Nicol N. Schraudolph "Evolving Networks: Using the Genetic Algorithm with Connectionist Learning" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp.511–548

[7] Bochmann, Gregor V. "A General Transition Model for Protocols and Communication Service" in *IEEE Transactions on Communications*, vol. COM–28, no. 4, (April 1980)

[8] Booker, Lashon B. "Classifier Systems that Learn Internal World Models" in *Machine Learning*, *3*, (1988), pp.161–192

[9] Booker, Lashon B. "Improving the Performance of Genetic Algorithms in Classifier Sys-

tems" in Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufman Publishers (1989)

[10] Booker, Lashon B., Goldberg, David E., and John H. Holland "Classifier Systems and Genetic Algorithms" in *Artificial Intelligence, 40* (1989) pp. 235–282

[11] Booker, Lashon B. "Representing Attribute–Based Concepts in a Classifier System" in Gregory J.E. Rawlins (Editor) Foundations of Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers (1991)

[12] Carbonell, J.G. "Learning by Analogy: Formulating and Generalizing Plans from Past Experience" In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Editors). Machine Learning: An Artificial Intelligence Approach, Vol. 1, San Mateo, CA: Morgan Kaufman Publishing (1983)

[13] Cariani, Peter "Emergence and Artificial Life" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp. 775–779

[14] Cohoon, J. P., Hegde, S.U., Martin, W. N., and D. Richards "Punctuated Equilibria: A Parallel Genetic Algorithm" in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[15] Davidor, Yuval. Genetic Algorithms and Robotics: A Heuristic Strategy for Optimization. Teaneck, NJ: World Scientific (1991)

[16] Davis, Lawrence. Handbook of Genetic Algorithms. New York: Van Nostrand Reinhold (1991)

[17] DeJong, Kenneth A. "Genetic–Algorithm–Based Learning" in Michalski, R.S., and Y. Kodratoff (Editors). Machine Learning: An Artificial Intelligence Approach, Vol. 3, San Mateo, CA: Morgan Kaufman Publishing (1990)

[18] Dhillon, Balbir S. Reliability Engineering in Systems Design and Operation. New York, NY: Van Nostrand Reinhold (1983)

[19] Dolan, Charles P. and Micheal G. Dyer "Towards the Evolution of Symbols" in John J.

Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[20] Elman, Jeffrey L. "Distributed Representations, Simple Recurrent Networks, and Grammatical Structures" in *Machine Learning*, *7*, (1991), pp. 195–225

[21] Forrest, Stephanie. Parallelism and Programming in Classifier Systems. San Mateo, CA: Morgan Kaufmann Publishers (1991)

[22] Frey, Peter W. and David J. Slate "Letter Recognition Using Holland–Style Adaptive Classifiers" in *Machine Learning*, *6*, (1991), pp. 161–182

[23] Goldberg, D. E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison–Wesley (1989)

[24] Goldberg, David E., Earickson, Jeff A., and Robert E. Smith "SGA–Cube: A Simple Genetic Algorithm for nCUBE 2 Hypercube Parallel Computers" The Clearinghouse for Genetic Algorithms TCGA Report No. 91005, Department of Engineering Mechanics, University of Alabama, Tuscaloosa, Alabama (1991)

[25] Goldberg, S.H. and J. A. Mouton, Jr. "A Base for Portable Communications Software" in *IBM Systems Journal*, vol. 30 no. 3 (1991)

[26] Green, Paul E. (Editor) Computer Network Architectures and Protocols. New York, NY: Plenum Press (1982)

[27] Greene, David Perry and Stephen F. Smith "A Genetic System for Learning Models of Consumer Choice" in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[28] Grefenstette, John J. "Genetic Algorithms and Their Applications" in Kent, A. and J.G. Williams (Editors). The Encyclopedia of Computer Science and Technology, volume 21, Marcel Dekker (1990)

[29] Grefenstette, John J., Ramsey, Connie L., and Schultz, Alan C. "Learning Sequential Decision Rules Using Simulation Models and Competition" in *Machine Learning*, *5* (1990), pp. 355–381

[30] Grefenstette, John J. "Strategy Acquisition with Genetic Algorithms" in Handbook of Genetic Algorithms. New York: Van Nostrand Reinhold (1991)

[31] Hashem, Mostafa S. and M. Ümit Uyar "Protocol Modeling for Conformance Testing: Case Study for the ISDN LAPD Protocol" in *AT&T Technical Journal*, vol 69, no 1. (January/February 1990)

[32] Hillard, M. R., Liepins, G. E., Palmer, Mark, Morrow, Micheal, and Jon Richardson "A Classifier–based System for Discovering Scheduling Heuristics" in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[33] Holland, John H. Adaption in Natural and Artificial Systems. Ann Arbor, MI: The University of Michigan Press (1975)

[34] Holland, John J. "A Cognitive System with Powers of Generalization and Adaption". Unpublished manuscript, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI (1977)

[35] Holland, John H. "Escaping Brittleness: The Possibilities of General Purpose LearningAlgorithms Applied to Parallel Rule–Based Systems" in Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (Editors). Machine Learning: An Artificial Intelligence Approach, Vol. 2, San Mateo, CA: Morgan Kaufman Publishing (1986)

[36] Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R. Induction: Processes of Inference, Learning, and Discovery. Cambridge, MA: Massachusetts Institute of Technology Press (1986)

[37] Holland, John H. "Genetic Algorithms and Classifier Systems: Foundations and Future Directions" in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[38] ISO/TC97/SC21, "Information processing systems – Open systems interconnection, Basic Reference Model" in ISO 7498. Geneva, Switzerland: International Organization for Standardization (1984)

[39] Jog, Prasanna, and Dirk Van Gucht "Parallelisation of Probabilistic Sequential Search

Algorithms" in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[40] Kohavi, Zvi Switching and Finite Automata Theory. New York, NY: McGraw–Hill Book Compay (1978)

[41] Kurzweil, Raymond. The Age of Intelligent Machines. Cambridge, MA: Massachusetts Institute of Technology Press (1990)

[42] Manderick, Bernard, de Weger, Mark, and Pief Spiessens "The Genetic Algorithm and the Structure of the Fitness Landscape" in Belew, Richard K. and Lashon B. Booker (Editors). Proceedings of the Fourth International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers (1991), pp. 324–333

[43] Matwin, Stan, Szapiro, Tom, and Haigh, Karen "Genetic Algorithms Approach to a Negotiation Support System" *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, No. 1 (January/February 1991), pp. 102–114

[44] Michalewicz, Zbigniew, and Cezary Z. Janikow "Handling Constraints in Genetic Algorithms" in Belew, Richard K. and Lashon B. Booker (Editors). Proceedings of the Fourth International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers (1991), pp. 324–333

[45] Minsky, Marvin. "A Framework for Representing Knowledge" in P.H. Winston (Editor). The Psychology of Computer Vision. New York: McGraw–Hill (1975)

[46] Minsky, Marvin. Society of Mind. New York: Simon and Schuster Publishers (1986)

[47] Montana, David J. "Empirical Learning Using Rule Threshold Optimization for Detection of Events in Synthetic Images" in *Machine Learning*, *5*, (1990), pp. 427–450

[48] von Neumann, John Theory of Self–Reproducing Automata.. Edited by Arthur W. Burks. Urbana, IL: University of Illinois Press (1966)

[49] Nussbaumer, Henri. Computer Communication Systems. Volume 1, Data Circuits, Error Detection, Data Links. New York: John Wiley & Sons (1990)

[50] Pettey, Chrisila B., Leuze, Micheal R., and John J. Grefenstette "A Parallel Genetic Al-

gorithm" in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[51] Radcliffe, Nicholas J. "Genetic Set Recombination and its Application to Neural Network Topology Optimization" Techinal Report EPCC TR9121 of the Edinburgh Parallel Computing Center: University of Edinburgh (1991)

[52] Radcliffe, Nicholas J. "Equivalence Class Analysis of Genetic Algorithms" Techinal Report EPCC TR9003 of the Edinburgh Parallel Computing Center: University of Edinburgh (1990)

[53] Ray, Thomas S. "An Approach to the Synthesis of Life" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp. 371–408

[54] 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: University of Michigan (1988)

[55] Riolo, Rick L. "CFS–C/FSW1: An Implementation of the CFS–C Classifier System in a task Domain that Involves Learning to Traverse a Finite State World" Logic of Computers Group: University of Michigan (1988)

[56] Riolo, Rick L. "CFS–C: A Package of Domain Independent Subroutines for Implementing Classifier Systems in Arbitrary, User–Defined Environments" Logic of Computers Group:University of Michigan (1988)

[57] Riolo, Rick L. "The Emergence of Coupled Sequences of Classifiers" in Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufman Publishers (1989)

[58] Riolo, Rick L. "The Emergence of Default Hierarchies in Learning Classifier Systems" in Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufman Publishers (1989)

[59] Riolo, Rick L. "Modeling Simple Human Category Learning with a Classifier System" in Belew, Richard K. and Lashon B. Booker (Editors). Proceedings of the Fourth Interna-

tional Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers (1991), pp. 324–333

[60] Robertson, George G. “Parallel Implementation of Genetic Algorithms in a Classifier System” in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[61] Sannier, Adrain V. II, and Erik D. Goodman “Genetic Learning Procedures in Distributed Environments” in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[62] Schank. Roger C. Real and Artificial Memories. New York, NY: Scribner’s & Sons (1990)

[63] Shaefer, Craig G. “The ARGOT Strategy: Adaptive Representation Genetic Optimizer Technique” in John J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[64] Smith, S.F. “A Learning System Based on Genetic Adaptive Algorithms” Ph.D. Dissertation, Department of Computer Science, University of Pittsburgh, Pittsburg, PA. (1980)

[65] Smith, S.F. “Flexible Learning of Problem Solving Heuristics Through Adaptive Search” in Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, Germany (1983)

[66] Spears, William M. and Kenneth DeJong “Using Neural Networks and Genetic Algorithms as Heuristic for NP–complete Problems” in International Joint Conference on Neural Networks, volume I, Hillsdale, NJ: Lawrence Erlbaum Associates (1990), pp. 118–125

[67] Starkweather, T, McDaniel, S. Mathias, K. Whitley D., and C. Whitley “A Comparison of Genetic Sequencing Operators” in Belew, Richard K. and Lashon B. Booker (Editors). Proceedings of the Fourth International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers (1991), pp. 324–333

[68] Stein, Richard M. “Real Artificial Life” *Byte* (January 1991), pp. 289–297

[69] Stork, David G., Jackson, Bernie and Scott Walker "Non–Optimality" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp. 409–430

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

[71] Tanese, Reiko "Parallel Genetic Algorithm for a Hypercube" in ohn J. Grefenstette (Editor). Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates (1987)

[72] Taylor, Charles E. "'Fleshing Out' Artificial Life II" in Langton, Christopher G., Taylor, Charles, Farmer, J. Doyne, and Steen Rasmussen. Artificial Life II: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity. Redwood City, CA: Addison–Wesley (1992), pp. 25–38

[73] Wilson, Stewart W. "Knowlegde Growth in an Artificial Animat" in Proceedings of the Fourth Yale Workshop on Applications of Adaptive Systems (1985)

[74] Wilson, Stewart W. "Classifier System Learning of a Boolean Function" Research Memo RIS–27r, Cambridge, MA: Rowland Institute for Science (1986)

[75] Wilson, Stewart W and David E. Goldberg "A Critical Review of Classifier Systems" in Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufman Publishers (1989)

[76] Zhou, Harry H. "CSM: A Computational Model of Cumulative Learning" *Machine Learning, 5* (1990), pp. 383–406

[77] Zhou, Harry H. "CSM: A Genetic Classifier System with Memory for Learning by Analogy" Ph.D. Dissertation, Computer Science Department: Vanderbilt University (1987)

APPENDIX A

PROTOCOL ENVIRONMENT SPECIFICATION

The specifications for the FSM used in this thesis are given by Hashem et al [31]. Here a few relevant features are described.

The protocol defines a networking system where information is carried via packets, or *frames*. However, different kinds of frames exist. One frame is sent to the receiver (i.e., user A sends to user B) to establish connection, called *sabme*. Another frame, the *I–frame*, contains the actual packet of information being sent. Each I–frame is usually part of sequence of such frames part of an entire message being sent across the network. Each I–frame contains a number, *N(S)*, identifying the frame's position in the sequence of frames. Finally, a *disc* frame exists to disconnect the communication, or *logical link*, which defines the medium where the transfer of the data actually occurs. There exist also a number of timers that assure system functioning. The T200 timer assures that once an I–frame is sent to the receiver, the receiver acknowledges its receipt after some time. The T203 timer is used to prevent the system from remaining idle indefinitely. Bad frames can also be transmitted for a number of reasons [31]. Table A.1 defines the states as they are defined by Hashem et al.

A number of frames are defined for this protocol. There also a number of input and output service primitives that designate the state of the communication at any stage. A number of other frames are added by Hashem et al. for the testing of the protocol. The specifics of the above will not be discussed here. The intent, rather, is to give a general idea of what the protocol looks like and how it is defined in the classifier system.

The text that follows shows how the environment is defined in CFS–C and for this particle application. Please note that in the transitions section, the text in italics is not part of the file designating the environment. It is included here to show the transitions defined in the protocol. All the state specifications and the transitions are derived from Hashem et al. [31].

State		Description
1	TEI ASSIGN	This is the start state and defines the assignment of the address for the data–link communication.
2	AWAIT EST	This state is entered when a *sabme* is issued.
3	AWAIT REL	This state is entered when a *disc* command is issued.
4	MF EST NORM	The data link is established in this state after the receipt of the *sabme* command, moving here from state 1.
4.1	MF EST NORM WIND CLOS	This state is used for the testing of the protocol and is similar to state 4, except that the "window" through which the frame has to pass is closed.
5	MF EST REJ	If an I–frame is received in an incorrect sequence, as defined by *N(S)*, this state is entered from state 4.
5.1	MF EST REJ WIND CLOS	This state is used for the testing of the protocol and is similar to state 5, except that the "window" through which the frame has to pass is closed.
6	MF EST BUSY	This state is entered when a local–busy condition occurs while in states 4 or 5.
6.1	MF EST BUSY WIND CLOS	This state is used for the testing of the protocol and is similar to state 6, except that the "window" through which the frame has to pass is closed.
7	TM REC NORM	Once timer T200 is expired in state 4, there is a move from state 4 to this state.
8	TM REC REJ	This state is entered if a T200 timer is received in state 5 or an out of sequence I–frame results.
9	RM REC BUSY	This state is entered if the T200 timer expires in state 6 or a local–busy condition results in states 7 or 8.

Table A.1 Data–Link Layer State Descriptions

Each state is represented as a chromosome consisting of 16 bits. The left–most two bits, which are set to '00' define the type of the chromosome, which in this case are detectors since the system updates its classifier list and by checking what the current state of the FSM is.

Each state definition is given as *num1, num2, chromosome*, where *num1* is the identification of the state, *num2* is the visit count, which is set to 0 for all states except for the first one, which is the initial state. The *chromosome* follows.

The payoffs follow as a pair *state, payoff1*, where the payoff in this case is equal across all states since all the states in the protocol are important to start with.

The transitions of the FSM are defined as *s1, r, s2, p*, where *s1* is the current state the system is in, *r* is the *r* value that leads to that state (edge value), *s2* is the state the edge leads to, and *p* is the probability that the system will move to that state.

Finally, the effector values are given. The first effector defines the effector that usually determines the *r* value. There are cases, however, that the system will not generate an effector. In such cases, a default effector is used that will keep the classifier system interacting with the environment.

```
; The number of states followed by the initial state followed by a threshold value
12, 0, 500
; These are the states and their equivalents in the LAPD model:
;  Classifier System State   LAPD State    Description
;  ______________________   __________   __________
;          0                   s1          Initial state
;          1                   s2          Await EST
;          2                   s3          Await REL
;          3                   s4          MF EST NORM
;          4                   s4.1         MF EST NORM WITH WINDOW CLOSED
;          5                   s5          MF EST REJ
;          6                   s5.1         MF EST REJ WITH WINDOW CLOSED
;          7                   s6          MF EST BUSY
;          8                   s6.1         MF EST BUSY WITH WINDOW CLOSED
;       9           s7          TM REC NORM
;       10          s8          TM REC REJ
;       11          s9          TM REC BUSY
0, 1, 0000 0000 0010 0001
1, 0, 0000 0000 0100 0001
2, 0, 0000 0000 0101 0101
3, 0, 0000 0101 0101 0101
4, 0, 0000 0101 0001 0100
5, 0, 0001 0001 0100 0101
6, 0, 0001 0101 0001 0101
7, 0, 0000 0100 0001 0101
8, 0, 0001 0101 0001 0101
9, 0, 0000 0000 0010 0110
10, 0, 0000 0000 0100 0001
11, 0, 0000 0000 0100 0011
ENDSTATES
0,100/ 1,100/ 2,100/ 3,100/ 4,100/ 5,100/ 6,100/ 7,100/ 8,100/ 9,100/ 10,100/ 11,100
ENDPAYOFFS
; State s1
0,1, 0,1      WHEN (disc_p0, dm_f0)                       inopportune
0,2, 0,1      WHEN (disc_p1, dm_f1)                       inopportune
0,3, 3,1      WHEN (sabme_p1, ua_fl_13_e8)                valid
0,4, 3,1      WHEN (sabme_p0, ua_f0_13_e8)                valid
0,5, 1,1      WHEN (13_el, sabme_pl)                      valid
0,6, 1,1      WHEN (dm_f0, sabme_pl)                      valid
0,7, 0,1      WHEN (timeout_t200, null)                   timer
0,8, 0,1      WHEN (timeout_t203, null)                   timer
; State s2
1,9, 0,1      WHEN (dm_f1, 13_e7)                         valid
1,10, 1,1     WHEN (sabme_p0, ua_f0)                      inopportune
1,11, 1,1     WHEN (sabme_p1, ua_f1)                      inopportune
1,1, 1,1      WHEN (disc_p0, dm_f0)                       inopportune
1,2, 1,1      WHEN (disc_p1, dm_f1)                       inopportune
1,12, 3,1     WHEN (ua_f1, 13_e9)                         valid
1,13, 1,1     WHEN (timeout_t200, sabme_pl)               timer
1,14, 0,1     WHEN (timeout_t200_N200, 13_e7)             timer
; State s3
2,15, 0,1     WHEN (ua_f1, 13_e10)                        valid
2,16, 0,1     WHEN (dm_f1, 13_e10)                        valid
2,17, 2,1     WHEN (sabme_pl, dm_f1)                      inopportune
2,18, 2,1     WHEN (sabme_p0, dm_f0)                      inopportune
2,19, 2,1     WHEN (disc_p1, ua_f1)                       inopportune
2,20, 2,1     WHEN (disc_p0, ua_f0)                       inopportune
2,21, 2,1     WHEN (timeout_t200, 13_e7)                  timer
```

2,22, 0,1	*WHEN (timeout_t200_N200, disc_p1)*	*timer*
; State s4		
3,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
3,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*valid*
3,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*valid*
3,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*valid*
3,5, 1,1	*WHEN (13_e1, sabme_p1)*	*valid*
3,25, 2,1	*WHEN (13_e2, disc_p1)*	*valid*
3,26, 5,1	*WHEN (13_e3, rnr_p0)*	*valid*
3,27, 3,1	*WHEN (13_e5, i_p0)*	*valid*
3,28, 4,1	*WHEN (13_e6_max, i_p0)*	*valid*
3,29, 3,1	*WHEN (13_e6, i_p0)*	*valid*
3,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
3,30, 2,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
3,31, 1,1	*WHEN (frmr_f0, sabme_p1)*	*inopportune*
3,32, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*
3,33, 3,1	*WHEN (i_p1, rr_f1_13_e11)*	*valid*
3,34, 3,1	*WHEN (i_p0, rr_f0_13_e11)*	*valid*
3,35, 3,1	*WHEN (i_p0_cond, i_p0_13_e11)*	*valid*
3,36, 4,1	*WHEN (i_p0_unexp_ns, rej_f0)*	*inopportune*
3,37, 4,1	*WHEN (i_p1_unexp_ns, rej_f1)*	*inopportune*
3,38, 3,1	*WHEN (rr_p0, null)*	*valid*
3,39, 3,1	*WHEN (rr_f0, null)*	*valid*
3,40, 3,1	*WHEN (rr_f1, null)*	*inopportune*
3,41, 3,1	*WHEN (rr_p1, rr_f1)*	*valid*
3,42, 3,1	*WHEN (rnr_p0, null)*	*valid*
3,43, 3,1	*WHEN (rnr_f0, null)*	*valid*
3,44, 3,1	*WHEN (rnr_f1, null)*	*inopportune*
3,45, 3,1	*WHEN (rnr_p1, rr_f1)*	*valid*
3,46, 3,1	*WHEN (rej_p0, null)*	*valid*
3,47, 3,1	*WHEN (rej_f0, null)*	*valid*
3,48, 3,1	*WHEN (rej_f1, null)*	*inopportune*
3,49, 3,1	*WHEN (rej_p1, rr_f1)*	*valid*
3,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
3,51, 9,1	*WHEN (timeout_t200, rr_p1)*	*timer*
3,52, 3,1	*WHEN (timeout_t203, rr_p1)*	*timer*
; State s4.1		
4,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
4,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*valid*
4,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*inopportune*
4,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*inopportune*
4,5, 1,1	*WHEN (13_e1, sabme_p1)*	*inopportune*
4,25, 2,1	*WHEN (13_e2, disc_p1)*	*inopportune*
4,26, 8,1	*WHEN (13_e3, rnr_p0)*	*inopportune*

4,53, 4,1	*WHEN (13_e6, null)*	*inopportune*
4,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
4,30, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
4,31, 1,1	*WHEN (frmr_f0, sabme_p1)*	*inopportune*
4,32, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*
4,33, 4,1	*WHEN (i_p1, rr_f1_13_e11)*	*inopportune*
4,34, 4,1	*WHEN (i_p0, rr_f0_13_e11)*	*inopportune*
4,54, 3,1	*WHEN (i_p1_max, rr_f1_13_e11)*	*valid*
4,55, 3,1	*WHEN (i_p0_max, rr_f0_13_e11)*	*valid*
4,56, 3,1	*WHEN (rr_p0_max, null)*	*valid*
4,57, 3,1	*WHEN (rr_f0_max, null)*	*valid*
4,58, 3,1	*WHEN (rr_p1_max, null)*	*valid*
4,59, 3,1	*WHEN (rnr_p0_max, null)*	*valid*
4,60, 3,1	*WHEN (rnr_f0_max, null)*	*valid*
4,61, 3,1	*WHEN (rnr_p1_max, null)*	*valid*
4,62, 3,1	*WHEN (rej_p0_max, null)*	*valid*
4,63, 3,1	*WHEN (rej_f0_max, null)*	*valid*
4,64, 3,1	*WHEN (rej_p1_max, null)*	*valid*
4,36, 6,1	*WHEN (i_p0_unexp_ns, rej_f0)*	*inopportune*
4,37, 6,1	*WHEN (i_p1_unexp_ns, rej_f1)*	*inopportune*
4,38, 4,1	*WHEN (rr_p0, null)*	*inopportune*
4,39, 4,1	*WHEN (rr_f0, null)*	*inopportune*
4,40, 4,1	*WHEN (rr_f1, null)*	*inopportune*
4,41, 4,1	*WHEN (rr_p1, rr_f1)*	*inopportune*
4,42, 4,1	*WHEN (rnr_p0, null)*	*inopportune*
4,43, 4,1	*WHEN (rnr_f0, null)*	*inopportune*
4,44, 4,1	*WHEN (rnr_f1, null)*	*inopportune*
4,45, 4,1	*WHEN (rnr_p1, rr_f1)*	*inopportune*
4,46, 4,1	*WHEN (rej_p0, null)*	*inopportune*
4,47, 4,1	*WHEN (rej_f0, null)*	*inopportune*
4,48, 4,1	*WHEN (rej_f1, null)*	*inopportune*
4,49, 4,1	*WHEN (rej_p1, rr_f1)*	*inopportune*
4,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
4,51, 6,1	*WHEN (timeout_t200, rr_p1)*	*timer*
4,52, 4,1	*WHEN (timeout_t203, rr_p1)*	*timer*
; State s5		
5,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*valid*
5,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*valid*
5,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
5,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*valid*
5,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
5,50, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
5,31, 1,1	*WHEN (frmr_f0, sabme_p1)*	*inopportune*
5,32, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*

5,5, 1,1	*WHEN (13_e1, sabme_p1)*	*valid*
5,25, 2,1	*WHEN (13_e2, disc_p1)*	*valid*
5,26, 7,1	*WHEN (13_e3, rnr_p0)*	*valid*
5,27, 5,1	*WHEN (13_e5, i_p0)*	*valid*
5,28, 5,1	*WHEN (13_e6, i_p0)*	*valid*
5,29, 6,1	*WHEN (13_e6_max, i_p0)*	*valid*
5,33, 3,1	*WHEN (i_p1_rr_f1_13_e11)*	*valid*
5,34, 3,1	*WHEN (i_p0, rr_f0_13_e11)*	*valid*
5,35, 3,1	*WHEN (i_p0_cond, i_p0_13_e11)*	*valid*
5,65, 5,1	*WHEN (i_p0_unexp_ns, null)*	*inopportune*
5,66, 5,1	*WHEN (i_p1_unexp_ns, rr_f1)*	*inopportune*
5,38, 5,1	*WHEN (rr_p0, null)*	*inopportune*
5,39, 5,1	*WHEN (rr_p1, rr_f1)*	*inopportune*
5,40, 5,1	*WHEN (rr_f0, null)*	*inopportune*
5,41, 5,1	*WHEN (rr_f1, null)*	*inopportune*
5,42, 5,1	*WHEN (rnr_p0, null)*	*inopportune*
5,43, 5,1	*WHEN (rnr_f0, null)*	*inopportune*
5,44, 5,1	*WHEN (rnr_f1, null)*	*inopportune*
5,45, 5,1	*WHEN (rnr_p1, rr_f1)*	*inopportune*
5,46, 5,1	*WHEN (rej_f0, null)*	*inopportune*
5,47, 5,1	*WHEN (rej_f1, null)*	*inopportune*
5,48, 5,1	*WHEN (rej_p0, null)*	*inopportune*
5,49, 5,1	*WHEN (rej_p1, rr_f1)*	*inopportune*
5,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
5,51, 10,1	*WHEN (timeout_t200, rr_p1)*	*timer*
5,52, 5,1	*WHEN (timeout_t203, rr_p1)*	*timer*
; State s5.1		
6,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*inopportune*
6,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*inopportune*
6,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
6,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*valid*
6,54, 5,1	*WHEN (i_p1_max, rr_f1_13_e11)*	*valid*
6,55, 5,1	*WHEN (i_p0_max, rr_f0_13_e11)*	*valid*
6,56, 5,1	*WHEN (rr_p0_max, null)*	*valid*
6,57, 5,1	*WHEN (rr_f0_max, null)*	*valid*
6,58, 5,1	*WHEN (rr_p1_max, null)*	*valid*
6,59, 5,1	*WHEN (rnr_p0_max, null)*	*valid*
6,60, 5,1	*WHEN (rnr_f0_max, null)*	*valid*
6,61, 5,1	*WHEN (rnr_p1_max, null)*	*valid*
6,62, 5,1	*WHEN (rej_p0_max, null)*	*valid*
6,63, 5,1	*WHEN (rej_f0_max, null)*	*valid*
6,64, 5,1	*WHEN (rej_p1_max, null)*	*valid*
6,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
6,30, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*

6,31, 1,1	*WHEN (13_e1, sabme_p1)*	*inopportune*
6,25, 2,1	*WHEN (13_e2, disc_p1)*	*inopportune*
6,26, 8,1	*WHEN (13_e3, rnr_p0)*	*inopportune*
6,53, 6,1	*WHEN (13_e6, disc_p1)*	*inopportune*
6,33, 4,1	*WHEN (i_p1, rr_f1_13_e11)*	*inopportune*
6,34, 4,1	*WHEN(i_p0, rr_f0_13_e11)*	*inopportune*
6,65, 6,1	*WHEN (i_p0_unexp_ns, null)*	*inopportune*
6,66, 6,1	*WHEN (i_p1_unexp_ns, rr_f1)*	*inopportune*
6,38, 6,1	*WHEN (rr_p0, null)*	*inopportune*
6,39, 6,1	*WHEN (rr_p1, rr_f1)*	*inopportune*
6,40, 6,1	*WHEN (rr_f0, null)*	*inopportune*
6,41, 6,1	*WHEN (rr_f1, null)*	*inopportune*
6,42, 6,1	*WHEN (rnr_p0, null)*	*inopportune*
6,43, 6,1	*WHEN (rnr_f0, null)*	*inopportune*
6,44, 6,1	*WHEN (rnr_f1, null)*	*inopportune*
6,45, 6,1	*WHEN (rnr_p1, rr_f1)*	*inopportune*
6,46, 6,1	*WHEN (rej_f0, null)*	*inopportune*
6,47, 6,1	*WHEN (rej_f1, null)*	*inopportune*
6,48, 6,1	*WHEN (rej_p0, null)*	*inopportune*
6,49, 6,1	*WHEN (rej_p1, rr_f1)*	*inopportune*
6,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
6,51, 10,1	*WHEN (timeout_t200, rr_p1)*	*timer*
6,52, 6,1	*WHEN (timeout_t203, rr_p1)*	*timer*
; State s6		
7,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*valid*
7,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*valid*
7,5, 1,1	*WHEN (13_e1, sabme_p1)*	*valid*
7,25, 2,1	*WHEN (13_e2, disc_p1)*	*valid*
7,67, 3,1	*WHEN (13_e4_cond1, rr_p0)*	*valid*
7,68, 5,1	*WHEN (13_e4_cond2, rej_p0)*	*valid*
7,69, 5,1	*WHEN (13_e4_cond3, rr_p0)*	*valid*
7,70, 7,1	*WHEN (13_e5, i_p0)*	*valid*
7,71, 7,1	*WHEN (13_e6, i_p0)*	*valid*
7,72, 8,1	*WHEN (13_e6_max, i_p0)*	*valid*
7,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
7,30, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
7,31, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*
7,32, 1,1	*WHEN (frmr_f0, sabme_p1)*	*inopportune*
7,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
7,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e11)*	*valid*
7,33, 7,1	*WHEN (i_p1, rnr_f1_13_e11)*	*inopportune*
7,34, 7,1	*WHEN (i_p0, rnr_f0_13_e11)*	*inopportune*
7,73, 7,1	*WHEN (i_p0_unexp_ns, rnr_f0)*	*inopportune*
7,74, 7,1	*WHEN (i_p1_unexp_ns, rnr_f1)*	*inopportune*

7,38, 7,1	*WHEN (rr_p0, null)*	*valid*
7,39, 7,1	*WHEN (rr_f0, null)*	*valid*
7,40, 7,1	*WHEN (rr_f1, null)*	*valid*
7,41, 7,1	*WHEN (rr_p1, rnr_f1)*	*valid*
7,42, 7,1	*WHEN (rnr_p0, null)*	*valid*
7,43, 7,1	*WHEN (rnr_f0, null)*	*valid*
7,44, 7,1	*WHEN (rnr_f1, null)*	*valid*
7,45, 7,1	*WHEN (rnr_p1, rnr_f1)*	*valid*
7,46, 7,1	*WHEN (rej_f0, null)*	*valid*
7,47, 7,1	*WHEN (rej_f1, null)*	*valid*
7,48, 7,1	*WHEN (rej_p0, null)*	*valid*
7,49, 7,1	*WHEN (rej_p1, rnr_f1)*	*valid*
7,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
7,75, 7,1	*WHEN (timeout_t203, rnr_p1)*	*timer*
7,76, 11,1	*WHEN (timeout_t200, rnr_p1)*	*timer*
; Stat s6.1		
8,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*valid*
8,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*valid*
8,5, 1,1	*WHEN (13_e1, sabme_p1)*	*inopportune*
8,25, 2,1	*WHEN (13_e2, disc_p1)*	*inopportune*
8,67, 4,1	*WHEN (13_e4_cond1, rr_p0)*	*inopportune*
8,68, 6,1	*WHEN (13_e4_cond2, rej_p0)*	*inopportune*
8,69, 6,1	*WHEN (13_e4_cond3, rr_p0)*	*inopportune*
8,53, 8,1	*WHEN (13_e6, null)*	*inopportune*
8,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
8,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*valid*
8,54, 7,1	*WHEN (i_p1_max, rr_f1_13_e11)*	*valid*
8,55, 7,1	*WHEN (i_p0_max, rr_f0_13_e11)*	*valid*
8,56, 7,1	*WHEN (rr_p0_max, null)*	*valid*
8,57, 7,1	*WHEN (rr_f0_max, null)*	*valid*
8,58, 7,1	*WHEN (rr_p1_max, null)*	*valid*
8,59, 7,1	*WHEN (rnr_p0_max, null)*	*valid*
8,60, 7,1	*WHEN (rnr_f0_max, null)*	*valid*
8,61, 7,1	*WHEN (rnr_p1_max, null)*	*valid*
8,62, 7,1	*WHEN (rej_p0_max, null)*	*valid*
8,63, 7,1	*WHEN (rej_f0_max, null)*	*valid*
8,64, 7,1	*WHEN (rej_p1_max, null)*	*valid*
8,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
8,30, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
8,31, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*
8,32, 1,1	*WHEN (frmr_p0, sabme_p1)*	*inopportune*
8,33, 8,1	*WHEN (i_p1, rnr_f1_13_e11)*	*inopportune*
8,34, 8,1	*WHEN (i_p0, rnr_f0_13_e11)*	*inopportune*
8,73, 8,1	*WHEN (i_p0_unexp_ns, rnr_f0)*	*inopportune*

8,74, 8,1	*WHEN (i_p1_unexp_ns, rnr_f1)*	*inopportune*
8,38, 8,1	*WHEN (rr_p0, null)*	*inopportune*
8,39, 8,1	*WHEN (rr_p1, rr_f1)*	*inopportune*
8,40, 8,1	*WHEN (rr_f0, null)*	*inopportune*
8,41, 8,1	*WHEN (rr_f1, null)*	*inopportune*
8,42, 8,1	*WHEN (rnr_p0, null)*	*inopportune*
8,43, 8,1	*WHEN (rnr_f0, null)*	*inopportune*
8,44, 8,1	*WHEN (rnr_f1, null)*	*inopportune*
8,45, 8,1	*WHEN (rnr_p1, rr_f1)*	*inopportune*
8,46, 8,1	*WHEN (rej_f0, null)*	*inopportune*
8,47, 8,1	*WHEN (rej_f1, null)*	*inopportune*
8,48, 8,1	*WHEN (rej_p0, null)*	*inopportune*
8,49, 8,1	*WHEN (rej_p1, rr_f1)*	*inopportune*
8,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
8,75, 8,1	*WHEN (timeout_t203, rnr_p1)*	*timer*
8,79, 11,1	*WHEN (timeout_t200, rnr_p1)*	*timer*
; State s7		
9,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*valid*
9,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*valid*
9,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
9,30, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
9,31, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*
9,32, 1,1	*WHEN (frmr_p0, sabme_p1)*	*inopportune*
9,5, 1,1	*WHEN (13_e1, sabme_p1)*	*valid*
9,25, 2,1	*WHEN (13_e2, disc_p1)*	*valid*
9,26, 11,1	*WHEN (13_e3, rnr_p0)*	*valid*
9,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
9,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*valid*
9,33, 9,1	*WHEN (i_p1, rnr_f1_13_e11)*	*valid*
9,34, 9,1	*WHEN (i_p0, rnr_f0_13_e11)*	*valid*
9,36, 10,1	*WHEN (i_p0_unexp_ns, rej_f0)*	*inopportune*
9,37, 10,1	*WHEN (i_p1_unexp_ns, rej_f1)*	*inopportune*
9,38, 9,1	*WHEN (rr_p0, null)*	*inopportune*
9,39, 9,1	*WHEN (rr_p1, rr_f1)*	*inopportune*
9,40, 3,1	*WHEN (rr_f0, null)*	*valid*
9,41, 9,1	*WHEN (rr_f1, null)*	*inopportune*
9,42, 9,1	*WHEN (rnr_p0, null)*	*inopportune*
9,43, 9,1	*WHEN (rnr_f0, null)*	*inopportune*
9,44, 3,1	*WHEN (rnr_f1, null)*	*valid*
9,45, 9,1	*WHEN (rnr_p1, rnr_f1)*	*inopportune*
9,46, 9,1	*WHEN (rej_f0, null)*	*inopportune*
9,47, 9,1	*WHEN (rej_f1, null)*	*inopportune*
9,48, 3,1	*WHEN (rej_p0, null)*	*valid*
9,49, 9,1	*WHEN (rej_p1, rr_f1)*	*inopportune*

9,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
9,77, 1,1	*WHEN (timeout_t200_N200, sabme_p1)*	*timer*
9,78, 9,1	*WHEN (timeout_t200, rr_p1)*	*timer*
; State s8		
10,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
10,30, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
10,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*inopportune*
10,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*inopportune*
10,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*inopportune*
10,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*inopportune*
10,31, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*
10,32, 1,1	*WHEN (frmr_p0, sabme_p1)*	*inopportune*
10,5, 1,1	*WHEN (13_e1, sabme_p1)*	*valid*
10,25, 2,1	*WHEN (13_e2, disc_p1)*	*valid*
10,26, 11,1	*WHEN (13_e3, rnr_p0)*	*valid*
10,33, 9,1	*WHEN (i_p1, rnr_f1_13_e11)*	*valid*
10,34, 9,1	*WHEN (i_p0, rnr_f0_13_e11)*	*valid*
10,65, 10,1	*WHEN (i_p0_unexp_ns, null)*	*inopportune*
10,66, 10,1	*WHEN (i_p1_unexp_ns, rr_f1)*	*inopportune*
10,38, 10,1	*WHEN (rr_p0, null)*	*inopportune*
10,39, 10,1	*WHEN (rr_p1, rr_f1)*	*inopportune*
10,40, 5,1	*WHEN (rr_f0, null)*	*valid*
10,41, 10,1	*WHEN (rr_f1, null)*	*inopportune*
10,42, 10,1	*WHEN (rnr_p0, null)*	*inopportune*
10,43, 10,1	*WHEN (rnr_f0, null)*	*inopportune*
10,44, 5,1	*WHEN (rnr_f1, null)*	*valid*
10,45, 10,1	*WHEN (rnr_p1, rnr_f1)*	*inopportune*
10,46, 10,1	*WHEN (rej_f0, null)*	*inopportune*
10,47, 10,1	*WHEN (rej_f1, null)*	*inopportune*
10,48, 10,1	*WHEN (rej_p0, null)*	*valid*
10,49, 10,1	*WHEN (rej_p1, rr_f1)*	*inopportune*
10,50, 1,1	*WHEN (bad_frame, sabme_p1)*	*illegal*
10,77, 1,1	*WHEN (timeout_t200_N200, sabme_p1)*	*timer*
10,78, 10,1	*WHEN (timeout_t200, rr_p1)*	*timer*
; State s9		
11,23, 0,1	*WHEN (disc_p0, ua_f0_13_e7)*	*valid*
11,24, 0,1	*WHEN (disc_p1, ua_f1_13_e7)*	*valid*
11,6, 1,1	*WHEN (dm_f0, sabme_p1)*	*inopportune*
11,30, 1,1	*WHEN (dm_f1, sabme_p1)*	*inopportune*
11,31, 1,1	*WHEN (frmr_f1, sabme_p1)*	*inopportune*
11,32, 1,1	*WHEN (frmr_p0, sabme_p1)*	*inopportune*
11,4, 3,1	*WHEN (sabme_p0, ua_f0_13_e8)*	*valid*
11,3, 3,1	*WHEN (sabme_p1, ua_f1_13_e8)*	*valid*
11,5, 1,1	*WHEN (13_e1, sabme_p1)*	*valid*

```
11,25, 2,1     WHEN (13_e2, disc_p1)                          valid
11,67, 9,1     WHEN (13_e4_cond1, rr_p0)                      valid
11,68, 10,1    WHEN (13_e4_cond2, rej_p0)                     valid
11,69, 10,1    WHEN (13_e4_cond3, rr_p0)                      valid
11,33, 11,1    WHEN (i_p1, rnr_f1_13_e11)                     valid
11,34, 11,1    WHEN (i_p0, rnr_f0_13_e11)                     valid
11,73, 11,1    WHEN (i_p0_unexp_ns, rnr_f0)                   inopportune
11,74, 11,1    WHEN (i_p1_unexp_ns, rnr_f1)                   inopportune
11,38, 11,1    WHEN (rr_p0, null)                             inopportune
11,39, 7,1     WHEN (rr_p1, rr_f1)                            valid
11,40, 11,1    WHEN (rr_f0, null)                             inopportune
11,41, 11,1    WHEN (rr_f1, null)                             inopportune
11,42, 11,1    WHEN (rnr_p0, null)                            inopportune
11,43, 7,1     WHEN (rnr_f0, null)                            valid
11,44, 11,1    WHEN (rnr_f1, null)                            inopportune
11,45, 11,1    WHEN (rnr_p1, rnr_f1)                          inopportune
11,46, 10,1    WHEN (rej_f0, null)                            inopportune
11,47, 7,1     WHEN (rej_f1, null)                            valid
11,48, 10,1    WHEN (rej_p0, null)                            inopportune
11,49, 11,1    WHEN (rej_p1, rr_f1)                           inopportune
11,50, 1,1     WHEN (bad_frame, sabme_p1)                     illegal
11,79, 11,1    WHEN (timeout_t200, rnr_p1)                    timer
11,80, 1,1     WHEN (timeout_t200_N200, sabme_p1)             timer
ENDTRANS
 10## #### ## ## ## ##, Effector1, 0, 0
~10## #### ## ## ## ##, Default, 0, 0
END-EFFS
```

APPENDIX B

THE CLASSIFIER LIST

The following text shows the classifier list at major cycle 1000. The first number is an identification number assigned to the classifier. It is followed by the first condition string, the state in the Finite State Machine that chromosome supports, the second condition string, and the state that chromosome supports. The action sting follows the arrow and is followed by the strength (fitness) of the classifier, its bidratio, and the support.

Current Classifiers (cycle–step 1000):

```
 819> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
 987> 00 00#0 0100 #001 [1,10]; 01 1000 1#11 1101 [] → 10 1100 0010 1101
 405> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
 373> 00 00#0 0100 #001 [1,10]; 10 1100 0010 1101 [] → 01 1000 1011 1101
 578> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
 877> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
 648> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 10 1110 1010 1111
 282> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
 333> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 531> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
 278> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
 992> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 626> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
 835> 01 0001 0##0 0101 [5]; 01 0001 010# 0101 [5] → 11 0011 1010 1111
1131> 00 0101 0101 01#1 [3]; 11 1001 1111 0111 [] → 11 0000 0011 0110
 431> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 10 1110 1010 1111
 520> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 891> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 747> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
 217> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 10 1110 1010 1111
 785> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
1018> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
1012> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 11 0110 1010 1111
1064> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 11 1110 1011 1111
 553> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
1023> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 311> 00 00#0 0100 #001 [1,10]; 10 1100 0010 1101 [] → 01 1000 1011 1101
 799> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
 953> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 00 1110 1110 11#1
 787> 00 00#0 0100 #001 [1,10]; 10 1100 0010 1101 [] → 10 1111 1001 1101
```

```
1043> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
1135> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 11 1110 1111 1111
 532> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 372> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 997> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 352> 00 00#0 0100 #001 [1,10]; 10 1010 0101 0011 [] → 11 1100 0101 1101
 986> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
 622> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 10 1010 1111 11#1
 671> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 909> 01 0001 0##0 0101 [5]; 01 0001 010# 0101 [5] → 11 1010 0110 1111
 985> 00 0101 0001 01#0 [4]; 00 #101 0#01 0100 [4] → 00 0101 1101 1001
 749> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 10 1010 1111 11#1
 988> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 436> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 10 0001 1111 1111
1080> 00 0101 0001 01#0 [4]; 00 #101 0#01 0100 [4] → 00 1101 0111 0111
1099> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
1081> 00 0101 0001 01#0 [4]; 00 #101 0#01 0100 [4] → 01 0001 0000 0001
1020> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 11 0011 0111 1111
 378> 00 00#0 0100 0001 [1,10]; 10 1010 0101 0011 [] → 11 1100 0101 1101
 376> 00 0000 0010 0001 [0]; 00 000# 0010 000# [0] → 10 1010 0101 0011
1151> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 10 1001 1010 1011
 812> 00 00#0 0100 #001 [1,10]; 10 1100 0010 1101 [] → 10 1111 1001 1101
 769> 00 00#0 0100 #001 [1,10]; 10 1100 0010 1101 [] → 01 1000 1011 1101
 644> 00 00#0 0010 #110 [9]; 00 0000 0010 0##0 [9] → 10 0011 0001 0111
 435> 00 00#0 0010 #110 [9]; 00 0000 0010 0##0 [9] → 10 0011 0001 0111
 822> 00 00#0 0100 #001 [1,10]; 10 1100 0010 1101 [] → 10 1111 1001 1101
1106> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 0100 1011 0011
 366> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
 811> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
1048> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 0011 1101
1161> 00 0101 0101 01#1 [3]; 00 1111 0000 111# [] → 10 1111 1111 0110
1160> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 00 1111 0000 1111
1154> 00 00#0 0010 #110 [9]; 00 0000 0010 0##0 [9] → 10 0011 0001 0111
1152> 00 00#0 0100 #001 [1,10]; 0# 0#01 0101 0#01 [] → 10 1100 0010 1101
1156> 00 0000 0010 0001 [0]; 10 0011 0010 0111 [] → 10 1110 1010 1111
1153> 00 0101 0101 01#1 [3]; 01 1000 1011 1101 [] → 00 1101 1011 0111
1157> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 00 0001 0100 0111
1147> 00 0101 0101 01#1 [3]; 11 1001 1111 0111 [] → 11 0000 0011 0110
 959> 01 0101 0001 01#1 [6,8]; 01 0101 0001 0101 [6,8] → 10 1010 0100 11#1
1143> 00 0101 0101 01#1 [3]; 11 0011 1010 1111 [] → 11 1001 0001 0100
 386> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 10 1111 1111 0110
1136> 00 00#0 0100 #001 [1,10]; 01 0100 1011 0011 [] → 01 1000 1011 1101
 575> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 10 1010 1111 11#1
1159> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 10 1100 0010 1101
1155> 00 00#0 0100 #001 [1,10]; 10 1100 0010 1101 [] → 01 1000 1011 1101
1158> 00 00#0 0100 #001 [1,10]; 01 1000 1011 1101 [] → 00 1100 0010 1101
1055> 00 00#0 0100 #001 [1,10]; 01 1000 1#11 1101 [] → 11 0010 0011 0111
1126> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 00 0001 0100 0111
1128> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 10 0010 0101 11#1
1127> 00 0000 0101 01#1 [2]; 00 0000 0101 0101 [2] → 11 0000 0111 0001
1063> 00 00#0 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1111 1010 0001
```

```
1137> 00 00#0 0100 #001 [1,10];  01 1000 1#11 1101 [] → 10 1100 0010 1101
1112> 00 0101 0100 #1#1 [];  0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
1113> 00 10#0 0101 0001 [];  00 0000 0100 0##1 [1,10–11] → 01 10#0 0000 0000
1111> 00 00#0 0100 #001 [1,10];  01 0000 0010 000# [] → 10 1110 1010 1111
1110> 00 0000 0010 0001 [0];  00 1000 1011 1101 [] → 10 1100 0011 1101
1116> 00 0101 0101 01#1 [3];  0# 0#01 0100 0##1 [5] → 01 1000 1011 1111
1146> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
1122> 00 0000 0101 01#1 [2];  00 0000 0101 0101 [2] → 00 1001 1001 0000
 359> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 10 1100 0010 1101
1124> 00 0000 0101 01#1 [2];  00 0000 0101 0101 [2] → 11 1000 0011 0001
 794> 00 00#0 0100 #001 [1,10];  10 1010 0101 0011 [] → 11 1100 0101 1101
1101> 00 0101 0101 01#1 [3];  11 0000 0101 1101 [] → 11 1001 1111 0111
1093> 00 0000 0010 0001 [0];  10 1111 0100 1101 [] → 10 1110 1010 1111
1115> 00 00#0 0100 #001 [1,10];  10 1100 0010 1101 [] → 01 1000 1011 1101
1109> 00 00#0 0100 #001 [1,10];  10 1100 0010 1101 [] → 10 1111 1001 1101
1123> 00 0000 0101 01#1 [2];  00 0000 0101 0101 [2] → 00 1011 0000 1110
 943> 00 00#0 0010 #110 [9];  00 0000 0010 0##0 [9] → 11 0000 1011 0111
1089> 01 0001 0##0 0101 [5];  00 1101 0111 #110 [] → 11 0011 1010 1111
1141> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 00 0100 0011 #001
1120> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 11 0000 0011 0110
1138> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 00 1001 0101 0010
1142> 01 0001 0##0 0101 [5];  01 0001 010# 0101 [5] → 11 0011 1010 1111
1133> 01 0001 0##0 0101 [5];  01 0001 010# 0101 [5] → 11 0110 1110 1000
 885> 00 00#0 0100 #001 [1,10];  10 1100 0010 1101 [] → 01 1000 1011 1111
 964> 00 0000 0101 01#1 [2];  00 0000 0101 0101 [2] → 00 1011 0110 11#1
1100> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 0000 0101 1101
1150> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 00 1110 0010 0001
1075> 00 0101 0101 01#1 [3];  0# 1000 0101 0101 [] → 10 1110 1111 11#1
1074> 00 0101 0101 01#1 [3];  11 0#01 0101 0#01 [] → 11 1001 1111 0111
1094> 00 0101 0001 01#0 [4];  00 #101 0#01 0100 [4] → 10 0100 0011 1010
1062> 00 10#0 0100 #001 [];  00 0000 0100 0##1 [1,10–11] → 01 10#0 0000 0000
1088> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 00 1101 0111 0110
1144> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 01 0001 0000 0111
1046> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 01 1100 1110 1111
1061> 00 0101 0101 01#1 [3];  00 0111 0011 #110 [] → 11 1001 1111 0111
1077> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 1000 1010 1101
1149> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 01 0101 1010 1111
1148> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 10 0101 0111 1010
1049> 00 0101 0101 01#1 [3];  01 1000 0011 1101 [] → 11 1001 1111 0111
1130> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 10 0101 1001 0111
1134> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
1132> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
1035> 00 0000 0010 0001 [0];  00 0101 1001 111# [] → 00 0000 0101 0011
1086> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 11 1001 1111 0111
1033> 00 0101 0101 01#1 [3];  01 0001 0100 111# [] → 11 1001 1111 0111
1031> 00 0000 0010 0001 [0];  11 0010 0001 1101 [] → 10 0001 1111 1111
1076> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 10 1100 0011 1101
1041> 00 0000 0010 0001 [0];  11 0110 1010 111# [] → 10 1110 1010 1111
 678> 00 00#0 0100 #001 [1,10];  10 1010 0101 0011 [] → 11 1100 0101 1101
1119> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 10 1101 1111 0001
1087> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 10 1111 1111 0110
```

```
1103> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 00 0011 0110 0100
1060> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 00 0111 0011 0110
1054> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 00 1010 1100 1010
1011> 00 0101 0101 01#1 [3];  11 1000 0101 0101 [] → 10 1110 1111 11#1
1010> 00 0000 0101 01#1 [2];  00 0001 1111 0111 [] → 11 1010 1110 0101
1040> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 11 0110 1010 1111
1098> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
1118> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 10 0101 1001 0111
 991> 00 00#0 0100 #001 [1,10];  01 0101 0110 1101 [] → 01 1000 1011 1101
1117> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 00 1101 1011 0111
 993> 00 00#0 0010 #110 [9];  00 0000 0010 0##0 [9] → 11 0000 1011 0111
 989> 00 00#0 0100 #001 [1,10];  10 1100 0010 1101 [] → 10 1111 1001 1101
 977> 01 0001 0##0 0101 [5];  10 0100 1111 1111 [] → 01 1000 1010 1100
 569> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 10 1100 0010 1101
 981> 00 0001 0##0 0101 [];  01 000# 0010 000# [] → 01 0000 0000 1100
 978> 00 00#0 0011 01#1 [];  11 1001 1111 0111 [] → 10 1010 1111 0110
 796> 00 00#0 0100 #001 [1,10];  10 1100 0010 1101 [] → 01 1000 1011 1101
 917> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 10 1100 0010 1101
 911> 01 0001 0##0 0101 [5];  01 0001 010# 0101 [5] → 00 1111 0101 1111
1036> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 00 0111 1110 1101
 505> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
 938> 00 0000 0010 0001 [0];  10 0011 0010 0111 [] → 10 1110 1010 1111
 901> 00 0101 0101 01#1 [3];  11 0011 1010 1111 [] → 11 1001 0001 0100
 928> 00 0000 0101 01#1 [2];  00 0010 1111 #110 [] → 00 0111 0010 0010
1104> 00 0101 0101 01#1 [3];  0# 0#01 #101 0#01 [3] → 11 1001 1111 0111
 983> 01 0001 0##0 0101 [5];  01 0001 010# 01#1 [5] → 00 1100 0111 1111
 509> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 10 1100 0010 1101
 974> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 10 1100 0010 1101
1070> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 01 1010 0011 1111
 904> 00 0000 0010 0001 [0];  11 1010 1101 11#1 [] → 10 0001 1111 1111
1027> 00 00#0 0100 #001 [1,10];  01 1000 1#11 1101 [] → 10 1100 0010 1101
1009> 00 00#0 0100 #001 [1,10];  10 1100 0010 1101 [] → 10 0000 1010 0100
 886> 01 0001 0##0 0101 [5];  01 0100 0110 1111 [] → 00 1100 0111 1101
 888> 00 0101 0101 01#1 [3];  11 1000 1111 0111 [] → 10 1111 1111 0110
 961> 01 0001 0##0 0101 [5];  01 0001 010# 0101 [5] → 11 0110 1110 1000
 976> 01 0001 0##0 0101 [5];  01 0001 010# 0101 [5] → 10 0100 1111 1111
 918> 00 00#0 0100 #001 [1,10];  10 1100 0010 11#1 [] → 00 1000 1011 1101
1068> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 11 1100 1111 0010
 676> 00 0000 0101 01#1 [2];  00 0000 0101 0101 [2] → 10 1010 0100 11#1
1006> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 10 0100 0010 0110
 875> 00 00#0 0100 #001 [1,10];  10 1010 0101 0011 [] → 10 0101 0101 1101
 874> 00 0000 0101 11#1 [];  01 0001 0000 11#1 [] → 10 1010 1111 11#1
 862> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 11 1010 1110 0101
 277> 00 00#0 0100 #001 [1,10];  11 1100 0101 1101 [] → 01 1000 1011 1101
 915> 01 0001 0##0 0101 [5];  01 0001 010# 0101 [5] → 10 0100 1011 1001
1030> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 11 0010 0001 1101
 844> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 10 1111 1111 0110
 927> 00 0101 0101 01#1 [3];  11 1001 1111 0111 [] → 00 0010 1111 0110
 990> 00 00#0 0100 #001 [1,10];  10 1100 0010 1101 [] → 01 0101 0110 1101
 274> 00 0000 0010 0001 [0];  00 000# 0010 000# [0] → 10 1010 0101 0011
 828> 00 0000 0010 0001 [0];  01 0101 0010 111# [] → 10 1101 0111 0011
```

```
1082> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 10 0101 0001 10#0
 916> 01 0001 0##0 0101 [5];  01 0001 010# 0101 [5] → 01 0101 1101 1010
1004> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 01 0001 1100 1111
1121> 00 0101 0101 01#1 [3];  11 0000 0011 #110 [] → 11 1001 1111 0111
1059> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 01 0000 1000 1001
 903> 00 0000 0101 01#1 [2];  00 0000 0101 0101 [2] → 11 1010 1101 11#1
 522> 00 0000 0010 0001 [0];  00 000# 0010 000# [0] → 10 1010 0101 0011
1032> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 01 0001 0100 1111
1067> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 01 1101 0010 1011
1038> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 00 1101 0100 0110
 994> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 10 0010 1100 0100
1025> 00 0000 0010 0001 [0];  00 0000 0010 000# [0] → 00 0001 1110 1101
 718> 00 00#0 0100 #011 [11];  00 0000 0100 0011 [11] → 10 1100 0010 1101
 292> 00 00#0 0100 #001 [1,10];  01 1000 1011 1101 [] → 10 1100 0010 1101
 770> 00 0000 0010 0001 [0];  00 000# 0010 000# [0] → 10 1010 0101 0011
 857> 00 00#0 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
 829> 00 0000 0010 0001 [0];  00 000# 0010 000# [0] → 01 0110 1001 1011

Number of Classifiers: 200. Ave. strength 2129.47 (total 425894.94).
```

The following portion shows the classifier list at major cycle 12,000. Note that the list has converged towards a much smaller number of states.

```
Current Classifiers (cycle–step 12000):

11058> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
 9279> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
10906> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
10719> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11238> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 1010 1011 1111
11435> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
10994> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11574> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11363> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
11436> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11324> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
10736> 00 0000 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
11478> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11303> 00 0000 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
11125> 00 00#0 0101 #101 [2];  00 0000 0101 010# [2] → 11 1010 1011 1111
11349> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11332> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
 9861> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 1010 1011 1111
11576> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
 9176> 00 00#0 0101 #101 [2];  00 0000 0101 010# [2] → 11 1010 1011 1111
10348> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11388> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
```

```
10251> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11337> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11212> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11246> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
10576> 00 0000 0100 #001 [1,10]; 00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
11358> 01 0001 0100 01#1 [5]; 01 0001 0100 0101 [5] → 10 0000 1111 1011
11567> 00 0101 0101 01#1 [3]; 00 0100 1111 0111 [] → 10 0000 1111 1011
11538> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11376> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11442> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11495> 00 00#0 0100 #001 [1,10]; 11 1010 1011 111# [] → 11 1010 1011 1111
11142> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 11 1010 1011 1111
11560> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
10835> 00 00#0 0101 #101 [2]; 00 0000 0101 010# [2] → 11 1010 1011 1111
11440> 00 0101 0101 01#1 [3]; 00 0100 1111 0111 [] → 10 0000 1111 1011
11476> 00 0101 0101 01#1 [3]; 00 0100 1111 0111 [] → 10 0000 1111 1011
10213> 00 0000 0100 #001 [1,10]; 01 1000 1011 1101 [] → 11 1010 1011 1111
10474> 00 0101 0101 01#1 [3]; 00 0100 1111 0111 [] → 10 0000 1111 1011
11329> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11526> 00 0000 0100 #001 [1,10]; 11 1010 1011 111# [] → 11 1010 1011 1111
11432> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11610> 00 0000 0010 0001 [0]; 00 0000 0010 000# [0] → 11 1010 1011 1111
11005> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11578> 00 0000 0100 #001 [1,10]; 11 1010 1011 111# [] → 11 1010 1011 1111
11566> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
11326> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11345> 00 0000 0100 #001 [1,10]; 01 1000 1011 1101 [] → 11 1010 1011 1111
11260> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11499> 00 00#0 0101 #101 [2]; 00 0000 0101 010# [2] → 11 1010 1011 1111
11579> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11564> 00 0000 0100 #001 [1,10]; 11 1010 1011 111# [] → 11 1010 1011 1111
 9699> 00 00#0 0100 #001 [1,10]; 11 1010 1011 111# [] → 11 1010 1011 1111
11028> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11447> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11503> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
10852> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11130> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
 9596> 00 0101 0101 01#1 [3]; 00 0100 1111 0111 [] → 10 0000 1111 1011
 9538> 00 0000 0100 #001 [1,10]; 01 1000 1011 1101 [] → 11 1010 1011 1111
11545> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11571> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11620> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
10985> 00 00#0 0101 #101 [2]; 00 0000 0101 010# [2] → 11 1010 1011 1111
11249> 00 0000 0100 #001 [1,10]; 01 1000 1011 1101 [] → 11 1010 1011 1111
11581> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11535> 00 0000 0010 0001 [0]; 11 1010 1011 111# [] → 11 1010 1011 1111
11516> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
11612> 00 0000 0100 #001 [1,10]; 11 1010 1011 111# [] → 11 1010 1011 1111
11540> 00 00#0 0100 #001 [1,10]; 11 1010 1011 111# [] → 11 1010 1011 1111
10673> 00 0101 0101 01#1 [3]; 10 0000 1111 1011 [] → 00 0100 1111 0111
11187> 00 0101 0101 01#1 [3]; 0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
```

```
11623> 00 0000 0010 0001 [0];  11 0011 1110 1011 [] → 11 1010 1011 1111
11622> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 11 0011 1110 1011
11625> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11624> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11544> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11615> 00 0101 0001 0100 [4];  00 0011 1100 1011 [] → 01 1000 0101 0101
11609> 00 0000 0100 #001 [1,10];  10 1001 0101 #011 [] → 01 0101 1110 1000
10266> 00 0000 0010 0110 [9];  00 0000 0010 011# [9] → 11 1010 1011 1110
11554> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11605> 00 0101 0101 01#1 [3];  01 1010 1101 111# [] → 00 0100 1111 0111
11616> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 00 1011 0110 0110
11392> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11599> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 1010 1011 1111
11597> 00 0000 0100 #001 [1,10];  01 1100 0010 1011 [] → 01 1000 1011 1101
11595> 00 0000 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 10 1010 0111 #011
11548> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11621> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 11 1100 0101 0001
11617> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0111 1010 1011
11570> 01 0001 0100 01#1 [5];  01 0001 0100 0101 [5] → 10 0000 1111 1011
11569> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11594> 00 0000 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 0101 1110 1000
11583> 00 0101 0101 01#1 [3];  01 0001 0100 0101 [] → 10 0000 1111 1111
11582> 01 0001 0100 01#1 [5];  00 0101 0110 111# [] → 00 0100 1111 0011
11472> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11614> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 00 0011 1100 1011
11568> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11618> 00 0000 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 00 0111 1101 #000
11561> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
10956> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11078> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
10831> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11533> 00 0000 0100 #001 [1,10];  00 1111 1100 1011 [] → 00 0100 1111 0111
11529> 00 0000 0100 0001 [1,10];  11 1010 1011 1101 [] → 11 1010 1011 1111
11528> 00 0000 0010 #001 [0];  01 1000 1011 111# [] → 11 1010 1011 1111
11530> 00 0000 0100 #001 [1,10];  01 1000 0101 0#01 [] → 11 1010 1011 1111
11448> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11525> 00 0000 0100 #001 [1,10];  00 1010 1011 1011 [] → 01 1000 1011 1101
11438> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11613> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11606> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 1000 1011 0111
11523> 00 0000 0100 #001 [1,10];  11 1010 0000 1011 [] → 01 1000 1011 1101
11519> 00 0101 0101 01#1 [3];  11 1010 1111 1011 [] → 00 0100 1111 0111
11521> 00 0000 0010 #001 [0];  01 1000 1011 111# [] → 11 1010 1011 1111
11512> 00 0000 0010 0001 [0];  11 0001 1011 1101 [] → 11 1010 1011 1111
11602> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11506> 00 0101 0101 01#1 [3];  00 0101 0110 111# [] → 00 0100 1111 0111
11443> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11598> 00 0000 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 01 1000 1011 1101
11607> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 01 1000 0101 1010
11604> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 01 1010 1101 1111
11493> 00 0101 0101 01#1 [3];  01 1000 1111 1011 [] → 00 0100 1111 0111
```

```
11562> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11539> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
10947> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11464> 00 00#0 0100 #001 [1,10];  10 1001 0101 1011 [] → 11 0101 1010 1001
11549> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11234> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11454> 00 0000 0100 #001 [1,10];  10 0000 1011 1101 [] → 11 1010 1011 1111
11593> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 01 0001 1000 0011
11288> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11343> 00 0000 0100 #001 [1,10];  00 1111 1100 1011 [] → 01 1000 1011 1101
11603> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11600> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 01 1001 0110 1010
11515> 00 0101 0001 0100 [4];  0# 0101 0001 010# [4,6,8] → 01 1000 0101 0101
11527> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 1010 1011 1111
11194> 00 0101 0101 01#1 [3];  0# 0#01 0101 0#01 [3] → 00 0100 1111 0111
11592> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 1111 1001 01#0
11477> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 1010 1011 1111
11589> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11588> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11601> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 00 1010 0000 0011
11423> 00 00#0 0101 #101 [2];  11 1010 0101 010# [] → 11 1010 1011 1111
11409> 00 0000 0100 #1#1 [];  00 0100 1111 0111 [] → 01 1000 1011 1101
11608> 00 0000 0100 #001 [1,10];  00 0000 0100 0##1 [1,10–11] → 10 1001 0101 #011
11585> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1000 1111
11041> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11552> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 10 1110 0101 0111
10886> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11415> 01 0001 0100 01#1 [5];  01 0001 0100 0101 [5] → 11 1010 1010 1001
11584> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1001 0011 1111
11395> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11586> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11587> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11517> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11522> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11416> 01 0001 0100 01#1 [5];  01 0001 0100 0101 [5] → 01 0001 1001 1011
11441> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11417> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 01 1110 1110 1101
11406> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 1010 1011 1111
11580> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11439> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11590> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11235> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11460> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11299> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11396> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11543> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11563> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11354> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11456> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11397> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11471> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
```

```
11553> 00 0000 0100 #001 [1,10];  11 101# 1011 111# [] → 10 0100 0001 0000
11450> 00 0101 0101 01#1 [3];  10 0000 1111 1011 [] → 00 0100 1111 0111
11558> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 10 0010 1111 0101
11551> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 10 0011 1110 0001
11550> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 01 0011 0001 0100
11158> 00 00#0 0101 #101 [2];  00 0000 0101 010# [2] → 00 0110 1001 1111
11497> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 10 0000 1111 1011
11559> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 10 0011 0000 0000
11458> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 11 1100 0101 0001
11403> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11556> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11141> 00 0000 0100 #001 [1,10];  01 1000 1011 1101 [] → 11 0000 0110 #101
11485> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 0000 0000 0101
11536> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11489> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1001 0000 1111
11486> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 10 1001 1010 0101
11484> 00 00#0 0100 #001 [1,10];  11 1010 1011 111# [] → 00 0001 1101 0011
11268> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
10836> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
11502> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1001 1110 0101
11228> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 11 1010 0000 1011
11429> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 11 0101 0100 1100
11371> 00 0101 0101 01#1 [3];  00 0100 1111 0111 [] → 11 1100 0101 0001
11106> 00 0000 0100 #001 [1,10];  11 1010 1011 111# [] → 11 1010 1011 1111
11333> 00 0000 0010 0001 [0];  11 1010 1011 111# [] → 11 1010 1011 1111
```

Number of Classifiers: 200. Ave. strength 44151.51 (total 8830302.00).

VITA

Mr. Bijan Marjan was born in Teheran, Iran on June 5, 1967 to Amir Houshang Marjan and Rosaria Salomone Marjan. He earned a B.A. in Computer Science and a B.A. in Economics from Lehigh University in 1990. He has worked on a number of projects while at Lehigh. These include computer–aided design for Ford Corporation while with the Intelligent Systems Lab and computer–aided software engineering for McNEIL Consumer Products Company, a Johnson and Johnson Company while with the Computer Integrated Manufacturing Lab. Mr. Marjan intends to join Technology Systems Corporation in Bethlehem, Pennsylvania and return to his home, Rome, Italy, after the completion of some training in this country. Mr. Marjan's interests reside in artificial intelligence, machine learning, and intelligent information systems. Mr. Marjan received a number of awards while at Lehigh including the Graduate Student Leadership Award and is a member of IEEE, IEEE Computer Society, ACM, and ACM–SIGART, for which he served as Vice–Chairman of the Lehigh Chapter.