

# **Developing the Capability to Scale and Reuse Learned Knowledge and Functionality in Learning Classifier Systems**

by

Isidro M. Alvarez

A thesis  
submitted to the Victoria University of Wellington  
in fulfilment of the  
requirements for the degree of  
Doctor of Philosophy  
in Computer Science.

Victoria University of Wellington  
2017



## **Abstract**

Learning is an important activity through which humanity has incrementally improved accomplishing tasks by adapting knowledge and methods based on the related feedback. Although learning is natural to humans, it is much more difficult to achieve in the technological world as tasks are often learned in isolation. Software is capable of learning novel techniques and algorithms in order to solve these basic, individual problems, however transferring said knowledge to other problems in the same or related domains presents challenges. Solutions often cannot be enumerated to discover the best one as many problems of interest can be intractable in terms of the resources needed to successfully complete them. However, many such problems contain key building blocks of knowledge that can be leveraged to achieve a suitable solution. These building blocks encapsulate important structural regularities of the problem. A technique that can learn these regularities without enumeration, may produce general solutions that apply to similar problems of any length. This implies reusing learned information.

In order to reuse learned blocks of knowledge, it is important that a program be scalable and flexible. This requires a program capable of taking knowledge from a previous task and applying it to a more complex problem or a problem with a similar pattern. This is anticipated to enable the program to complete the new task in a practical amount of time and with reasonable amounts of resources.

In machine learning, the degree of human intervention in solving problems is often important in many tasks. It is generally necessary for a human to provide input to direct and improve learning. In the field of Developmental Learning there is the idea known as the Threshold Concept

(TC). A TC is transformative information which advocates learning. TCs are important because without them, the learner cannot progress. In addition, TCs need to be learned in a particular order, much like a curriculum, thus providing the student with viable progress towards learning more difficult ideas at a faster pace than otherwise. Therefore, human input to a learning algorithm can be to partition a problem into constituent sub-problems. This is a principal concept of Layered Learning (LL), where a sequence of sub-problems are learned. The sub-problems are self-contained stages which have been separated by a human. This technique is necessary for tasks in which learning a direct mapping from inputs to outputs is intractable given existing learning algorithms.

One of the first artificial learning systems developed is Learning Classifier Systems (LCSs). Past work has extended LCSs to provide more expressivity by using richer representations. One such representation is tree-based and is common to the Genetic Programming (GP) technique. GP is part of the Evolutionary Computation (EC) paradigm and produces solutions represented by trees. The tree nodes can contain functions, and leaf nodes problem features, giving GP a rich representation. A more recent technique is Code Fragments (CFs). CFs are GP-like sub-trees with an initial maximum height of two. Initially, CFs contained hard-coded functions at the root nodes and problem features or previously learned CFs at the leaf nodes of the sub-trees. CFs provided improved expressivity and scalability over the original ternary alphabet used by LCSs. Additionally, CF-based systems have successfully learned previously intractable problems, e.g. 135-bit multiplexer.

Although CFs have provided increased scalability, they suffer from a structural weakness. As the problem scales, the chains of CFs grow to intractable lengths. This means that at some point the LCS will stop learning. In addition, CFs were originally meant to scale to more complex problems in the same domain. However, it is advantageous to compile cross-domain solutions, as the regularities of a problem might be from different domains

to that expressed by the data.

The proposed thesis is that a CF-based LCS can scale to complex problems by reusing learned solutions of problems as functions at the inner nodes of CFs together with compaction and Layered Learning. The overall goal is divided into the following three sub-goals: reuse learned functionality from smaller problems in the root nodes of CF sub-trees, identify a compaction technique that facilitates reduced solution size for improved evaluation time of CFs and develop a layered learning methodology for a CF system, which will be demonstrated by learning a general solution to an intractable problem, i.e. n-bit Multiplexer.

In this novel work, Code Fragments are extended to include learned functionality at the root nodes of the sub-trees in a technique known as *XCSCF*<sup>2</sup>. A new compaction technique is designed, which produces an equivalent set of ternary rules from CF rules. This technique is known as *XCSCF*<sup>3</sup>. The work culminates with a new technique *XCSCF*<sup>\*</sup>, which combines Layered Learning, Code Fragments and Transfer Learning (TL) of knowledge and functionality to produce scalable and general solutions, i.e. to the n-bit multiplexer problem.

The novel ideas are tested with the multiplexer and hidden multiplexer problems. These problems are chosen because they are difficult due to epistasis, sparsity and non-linearity. Therefore they provide ample opportunity for testing the new contributions.

The thesis work has shown that CFs can be used in various ways to increase scalability and to discover solutions to complex problems. Specifically the following three contributions were produced: learned functionality was captured in LCS populations from smaller problems and was reused in the root nodes of CF sub-trees. An online compaction technique that facilitates reduced evaluation time of CFs was designed. A layered learning method to train a CF system in a manner leading to a general solution was developed. This was demonstrated through learning a solution to a previously intractable problem, i.e. the n-bit Multiplexer. The

thesis concludes with suggestions for future work aimed at providing better scalability when using compaction techniques.

# Publications

- **I. M. Alvarez**, W. N. Browne, and M. Zhang, "Compaction for Code Fragment Based Learning Classifier Systems - Redux," *World Congress on Computational Intelligence/ IEEE Congress on Evolutionary Computation (WCCI/CEC)*, 2016
- **I. M. Alvarez**, W. N. Browne, and M. Zhang, "Human-inspired Scaling in Learning Classifier Systems: Case Study on the n-bit Multiplexer Problem Set," *Genetic and Evolutionary Computation Conference GECCO '16*, 2016 , [available online] <http://dx.doi.org/10.1145/2908812.2908813>.
- **I. M. Alvarez**, W. N. Browne, and M. Zhang, "Compaction for Code Fragment Based Learning Classifier Systems," *2nd Australasian Conference on Artificial Life and Computational Intelligence ACALCI*, 2016, [available online] <http://dx.doi.org/10.1007/978-3-319-28270-1>.
- **I. M. Alvarez**, W. N. Browne, and M. Zhang, "Reusing Learned Functionality to Address Complex Boolean Functions," *10th International Conference on Simulated Evolution and Learning SEAL*, 2014, [available online] <http://dx.doi.org/10.1007/978-3-319-13563-2>.
- **I. M. Alvarez**, W. N. Browne, and M. Zhang, "Reusing learned functionality in XCS: code fragments with constructed functionality and constructed features," *Genetic and Evolutionary Computation Conference GECCO '14 Companion*, 2014, [available online]

<http://doi.acm.org/10.1145/2598394.2611383>.



# Acknowledgments

I thank the infinite, ubiquitous beneficence that may or may not permeate all, that may or may not go by many names and that may or may not be knowable.

After that, I am thankful to my supervisors, Dr Will Browne and Prof Mengjie Zhang, for their professional supervision. They have spent many dedicated hours reading many of my drafts throughout the course of the PhD. Their corrections and suggestions contributed positively to the readability and quality of the papers. I am specifically grateful to them for reading my thesis under a very tight schedule, while not compromising on the quality of the text. They have been available for questions or discussions at any time during the entire PhD, especially Will Browne, in spite of his being on a different continent at times. Will Browne never objected, even if he were busy with his own important tasks. *Thanks a Million Will!*

I acknowledge the effectiveness of the friendly environment and especially the regular meetings at the Evolutionary Computation Research Group (ECRG), led by Mengjie Zhang, and the cooperative behaviour of the group members.

I am thankful to Victoria University of Wellington for having awarded me the Victoria PhD Scholarship and to Will N. Browne and Mengjie Zhang for supporting my PhD scholarship application. I am also thankful for the support afforded me through the Marsden Grant (VUW1209). The road leading to this point has at times been circuitous and fraught with obstacles, however thanks to the help of Will Browne and Mengjie Zhang,

I was able to navigate those troubled waters successfully.

I am very grateful to my father, whose incredibly valuable lessons I am just beginning to understand. I thank my son, for having inspired me to pursue graduate studies and for providing constant moral support. I also thank my mother and the rest of my family for providing support and understanding, specially during the final month of writing.

I am thankful to various reviewers of my papers for their constructive criticism, suggestions and appreciation of my work submitted to various conferences.

I would also like to thank the Hillside Public Library and various cafes where I was able to write and use their internet facilities when fate decided I needed a temporary, awakening change in scenery.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Scope . . . . .	1
1.2	Motivation . . . . .	7
1.3	Thesis Statement . . . . .	10
1.4	Research Goals . . . . .	10
1.5	Major Contributions . . . . .	12
1.6	Organization of Thesis . . . . .	14
<b>2</b>	<b>Literature Review</b>	<b>17</b>
2.1	Evolutionary Computation . . . . .	17
2.2	Learning Classifier Systems . . . . .	18
2.3	Representation . . . . .	25
2.4	Code Fragments . . . . .	29
2.5	Code Fragment Based Systems . . . . .	31
2.6	Other Representations . . . . .	37
2.7	Layered Learning . . . . .	41
2.8	Transfer Learning . . . . .	43
2.9	Chapter Summary . . . . .	44
<b>3</b>	<b>Research Methodology</b>	<b>47</b>
3.1	Research Methodology . . . . .	47
3.1.1	Overview of Methods . . . . .	48
3.2	Experimental Design . . . . .	52

3.2.1	The Problem Domains . . . . .	53
3.2.2	Experimental Setup . . . . .	56
3.2.3	Test Data-set Size . . . . .	57
3.3	Chapter Summary . . . . .	58
<b>4</b>	<b>Reusing Learned Functionality</b>	<b>61</b>
4.1	Introduction . . . . .	61
4.1.1	Chapter Goals . . . . .	63
4.2	Reusing Rule-sets as Functions . . . . .	65
4.2.1	Method . . . . .	65
4.2.2	Results . . . . .	74
4.2.3	Interpretation of Results . . . . .	82
4.2.4	Summary of Reusing Rule-sets as Functions . . . . .	85
4.3	Reusing Code Fragments to address the Hidden Multiplexer	85
4.3.1	Method . . . . .	85
4.3.2	Results . . . . .	88
4.3.3	Interpretation of Results . . . . .	97
4.3.4	Summary of Reusing Code Fragments to address the Hidden Multiplexer . . . . .	98
4.4	Chapter Summary . . . . .	99
<b>5</b>	<b>Distilled Rules for CF-based Systems</b>	<b>103</b>
5.1	Introduction . . . . .	103
5.1.1	Chapter Goals . . . . .	107
5.2	Offline Distilled Rules Extraction . . . . .	108
5.2.1	Method . . . . .	108
5.2.2	Results of Offline Distilled Rules Experiments . . . . .	114
5.2.3	Interpretation of Results . . . . .	121
5.2.4	Summary of Offline Distilled Rules Extraction . . . . .	129
5.3	Online Distilled Rules Extraction . . . . .	129
5.3.1	Method . . . . .	130
5.3.2	Results of the Online Distilled Rules Experiments . . . . .	137

5.3.3	Interpretation of Results . . . . .	141
5.3.4	Summary of Online Distilled Rules Extraction . . . .	144
5.4	Chapter Summary . . . . .	145
<b>6</b>	<b>Layered/Transfer Learning for CF-based Systems</b>	<b>149</b>
6.1	Introduction . . . . .	149
6.1.1	Experiments Chosen . . . . .	156
6.1.2	Chapter Goals . . . . .	158
6.2	Solving the n-bit Multiplexer Problem Set . . . . .	159
6.2.1	Method . . . . .	159
6.2.2	Hidden Multiplexer . . . . .	172
6.2.3	Results of the n-bit Multiplexer Experiments . . . .	173
6.2.4	Interpretation of Results . . . . .	199
6.3	Chapter Summary . . . . .	204
<b>7</b>	<b>Conclusions and Future Work</b>	<b>207</b>
7.1	Achieved Objectives . . . . .	207
7.2	Main Conclusions . . . . .	209
7.2.1	Reusing CFs in the Root Nodes of CF Sub-trees . . .	209
7.2.2	Develop a New Compaction Technique . . . . .	210
7.2.3	Develop a Layered Learning Methodology for a CF System . . . . .	212
7.3	Future Work . . . . .	213
7.3.1	Distilled Rules . . . . .	213
7.3.2	n-Bit Multiplexer . . . . .	215
	<b>Bibliography</b>	<b>219</b>



# List of Figures

1.1	An example of a code fragment. . . . .	7
1.2	An example of a don'tCare code fragment. . . . .	7
2.1	XCS framework - major parts. . . . .	21
2.2	XCS schematic. . . . .	22
2.3	An example of a code fragment. . . . .	30
3.1	6-bit Multiplexer . . . . .	54
3.2	Example of the Hidden Multiplexer. . . . .	55
4.1	Code Fragment rules reusability . . . . .	66
4.2	Code Fragment and Function Rule-set reuse. . . . .	66
4.3	An example of a code fragment. . . . .	68
4.4	Rules matching at the root nodes. . . . .	69
4.5	Original design of the Code Fragments-based XCSCFC. . . .	72
4.6	Results of the Boolean problems using XCSCF <sup>2</sup> . . . . .	75
4.7	Comparison between XCSCF <sup>2</sup> with XCSCFC for the 6-20 Bit Multiplexer Problems. . . . .	77
4.8	Comparison between XCSCF <sup>2</sup> with XCSCFC for the 6-20 Bit Multiplexer Problems. . . . .	79
4.9	Comparison between XCSCF <sup>2</sup> with XCSCFC for the 37-Bit Multiplexer. . . . .	80
4.10	Comparison between XCSCF <sup>2</sup> with XCSCFC for the 70-Bit Multiplexer. . . . .	81

4.11	Comparison between XCSCF <sup>2</sup> with XCSCFC for the 135-Bit Multiplexer. . . . .	83
4.12	Example of the Hidden Multiplexer. . . . .	87
4.13	Training flow of learned functions for the system. . . . .	89
4.14	18-bit Hidden Multiplexer - three training paths. . . . .	90
4.15	18-bit Hidden Multiplexer - comparison with two other systems. . . . .	93
4.16	18-bit Hidden Multiplexer (Boolean, mux). . . . .	94
4.17	18-bit Hidden Multiplexer (Boolean, parity, mux). . . . .	95
5.1	OR - Final CFs and their corresponding DRs. . . . .	106
5.2	CF 43 Sub-Tree . . . . .	107
5.3	Training regimen for XCSCF3. . . . .	110
5.4	Training regimen for XCSCFC for the multiplexer problems. . . . .	112
5.5	Training regimen for XCSCFC for the hidden multiplexer problems. . . . .	113
5.6	Offline Distilled Rules creation process. . . . .	114
5.7	Results for the 6-bit and 11-bit multiplexer problems. . . . .	117
5.8	Results of the 20-bit and 37-bit multiplexer problems using XCSCF3, XCS and XCSCFC. . . . .	118
5.9	Comparison between XCSCF3, XCS and XCSCFC for the 70-bit Multiplexer problem. . . . .	119
5.10	Comparison between XCSCF3, XCS and XCSCFC for the 3x6 bit Hidden Multiplexer Problems. . . . .	120
5.11	. . . . .	122
5.12	Overview of the process. . . . .	133
5.13	Comparison between XCSCF3, XCS and XCSCFC (70-bit Hidden Mux). . . . .	139
5.14	Comparison between XCSCF3, XCS and XCSCFC (135-bit Hidden Mux). . . . .	140
5.15	Comparison between XCSCF3, XCS and XCSCFC (3x6 Hidden Mux). . . . .	142



5.16 Comparison between XCSCF3, XCS and XCSCFC (3x11 Hidden Mux). . . . .	143
6.1 Human method for multiplication shortcut. . . . .	150
6.2 Humans can recognize patterns. . . . .	157
6.3 6 Bit Multiplexer . . . . .	161
6.4 Training regimen . . . . .	164
6.5 Multiplexer training flow . . . . .	165
6.6 3x6 bit Hidden Multiplexer . . . . .	173
6.7 Results of the Kbits - AddressOf sub-problems . . . . .	179
6.8 Result for the ValueAt sub-problem . . . . .	180
6.9 135-bit Multiplexer Solution . . . . .	183
6.10 264 Bit and 521 Bit Multiplexer Solution . . . . .	184
6.11 264, 521 and 1034-bit Multiplexer solution . . . . .	185
6.12 2059, 4108 and 8205-bit Multiplexer solution . . . . .	186
6.13 6 Bit Multiplexer Solution Rule . . . . .	187
6.14 6-bit Mux Solution Tree . . . . .	190
6.15 18-bit Hidden Mux Solution . . . . .	196
6.16 33-bit Hidden Mux Solution . . . . .	197
6.17 33-bit Hidden Mux Solution . . . . .	198
7.1 CFs viewed as monolithic objects. . . . .	216



# List of Tables

2.1	A sample of classifiers from the final solution in XCSCFA . . .	36
2.2	A sample of classifiers for S-XCS . . . . .	40
4.1	Evaluation of a Code Fragment. . . . .	69
4.2	Sample CFs produced by XCSCF <sup>2</sup> and XCSCFC for the 20 bit Mux problem. The tags $N, M, c, m, r$ and $d$ stand for previously learned functions. The tags CF_55, CF_44, CF_56, CF_25, CF_31, CF_28, CF_47 and CF_36 stand for CFs associated with the previously learned functions. . . . .	73
4.3	Function M learned ternary rules produced by XCSCF <sup>2</sup> . . . .	78
4.4	Sample rules produced by XCSCF <sup>2</sup> and XCSCFC for the 135 bit Mux problem. . . . .	84
4.5	Number of rules and CFs produced by XCSCF <sup>2</sup> for three different training paths. . . . .	91
4.6	Final Code Fragments produced by the three different training paths. . . . .	92
4.7	Number of classifiers and training instances for XCSCF <sup>2</sup> . . .	96
4.8	Number of classifiers and training instances for XCSCF <sup>2</sup> , XCSCFC, and XCS for the 3x6 Hidden Multiplexer. . . . .	97
5.1	Distilled Rules produced by XCSCF3 for the 6-bit Multiplexer problem. . . . .	125

5.2	Distilled Rules produced by XCSCF3 for the 3x6 Hidden Multiplexer problem. . . . .	127
5.3	Distilled Rules produced by XCSCF3 for the 3x11 Hidden Multiplexer problem. . . . .	128
6.1	Functionality Provided (Hard-coded functions) . . . . .	167
6.2	Constant(s) Provided . . . . .	167
6.3	Input and Output Types for all the sub-problems and for the Mux problem. The data type 'Variant' can accept all other types. . . . .	167
6.4	Training data for the KBits sub-problem. It offers a mapping from a possible multiplexer length to the corresponding number of address bits. . . . .	169
6.5	Training data for the kBitString sub-problem. . . . .	170
6.6	Training data for the Bin2Int sub-problem. . . . .	170
6.7	Training data for the AddressOf sub-problem. . . . .	171
6.8	Training data for the ValueAt sub-problem. . . . .	172
6.9	Population size for the fundamental sub-problem experiments. . . . .	174
6.10	Final rules for the Bin2Int sub-problem (Experiment 8, arbitrarily picked as it is representative of the 30 experiments). <i>L</i> , <i>m</i> and \$ represent the length constant, learned KBitString function and the BinaryString axiom respectively. . . . .	176
6.11	Final rules for the AddressOf sub-problem (Experiment 8). . . . .	177
6.12	Final rules for the ValueAt sub-problem (Experiment 8). . . . .	178
6.13	Final rules learned while solving the 6-bit Multiplexer. The condition parts are composed of don'tCares, meaning that the rules are useful for solving any 6-bit multiplexer. Each rule is a complete solution to the problem. . . . .	191
6.14	Settings for p_don'tCare for the 135-521 bit multiplexer problems. The training and testing phases were involved. . . . .	192

6.15	Final rules learned while solving the 135, 264 and 521-bit Multiplexer for run number 8. The condition part is composed of don'tCares, meaning that the rule is useful for solving any size multiplexer. The letter b stands for learned ValueAt function. It is used by all three problems. . . . .	192
6.16	Final Code Fragment Actions from maximally general rules produced by 264-8205 Multiplexer tests (No training was involved). . . . .	194
6.17	Final rules learned while solving the 18-bit Hidden Multiplexer (Even Parity). The function denoted by the letter b represents the learned ValueAt function. It returns the value of the data bit addressed by the hidden multiplexer. . . . .	195



# List of Algorithms

1	XCSCFC: Execution . . . . .	32
2	XCSCFC: Matchset [49] . . . . .	33
3	XCSCFC: IsMoreGeneral [49] . . . . .	34
4	XCSCF <sup>2</sup> : evaluateCF . . . . .	71
5	Rules Compaction - Offline . . . . .	111
6	Creation of DRs Network - Online . . . . .	132
7	Rules Compaction - Reconcile Network DRs . . . . .	135
8	Rules Compaction - Subsumption with more general DRs . . .	136





# Chapter 1

## Introduction

“We know that the greatest works can be represented in model, and that the universe can be represented by the same means. The same principles by which we measure an inch or an acre of ground will measure to millions in extent. A circle of an inch diameter has the same geometrical properties as a circle that would circumscribe the universe. The same properties of a triangle that will demonstrate upon paper the course of a ship will do it on the ocean, and when applied to what are called the heavenly bodies, will ascertain to a minute the time of an eclipse, though those bodies are millions of miles distant from us.”

Thomas Paine, *The Age of Reason*, 1938 (p. 168)

### 1.1 Scope

Learning can be defined as, “the improvement of performance in some environment through the acquisition of knowledge resulting from experience in that environment” [88]. This is a sought-after property for software agents because it holds the promise for durable and progressive change in our everyday lives. While human beings have a natural ability for learning, software agents face several challenges in this endeavor. First of all, unlike humans, software agents are often unable to generalize

from a single example, they are liable to require many more [57]. In addition, software agents find it more difficult to use learned concepts using a richer representation than conventional algorithms [57]. Conventional algorithms tend to use simpler representations which translate into an easier computation and smaller requirement of resources.

The accumulation of knowledge learned through experience can also present challenges for software agents. Chief among these is the notion of having too many hypotheses or having too few. If there are too many, learning can become difficult due to a large search space. On the other hand, with too few hypotheses, the system may be incapable of learning quickly enough or not at all, due to missing functionality. Therefore accumulated knowledge is a crucial aspect of learning and plays a critical role in knowledge reusability, or experience, in human terms. Reusability is important because once new knowledge is stored in an appropriate form, it can then be utilized to solve problems in the same or a related domain. Although this type of processing is not a priority for many computer systems [11], there exist possible avenues of research to explore its usefulness.

The original work presented here is part of the paradigm known as Evolutionary Computation (EC). It is a group of techniques that rely on Darwinian principles to simulate the process of natural selection to find highly fit solutions to a problem [89]. There exist many pressures in this type of system, which force the evolution of a population that can be the solution to the problem. These solutions can be evolved by progressive exploration of the sample space. The sample space is composed of all the possible instances of a problem. EC is a good paradigm for symbolic learning because its general solutions can be encapsulated in compact representations. Moreover, EC has been used successfully to evolve solutions to intractable problems.

Depending on the approach used to represent the environment symbolically, the learning can be impacted positively or negatively. For example, using a simple ternary alphabet, such as  $\{0, 1, \#\}$ , can be quick and

easy to compute an answer, however there is a lack of expressivity in this type of representation. Using a richer representation can yield solutions that can cover the problem space more completely and with more expressivity. However, this type of representation can increase the search space and this can hinder learning. The search space denotes all the possible combinations of available functionality, i.e. the learned knowledge.

A powerful EC technique that has been used for learning solutions is Learning Classifier Systems (LCSs) [13]. They have been in use for over 40 years. Originally they were formulated as cognitive systems, however they have changed to become classification techniques. Cognitive systems were used to model part of the brain functionality. LCSs are composed of rules, or classifiers, which are known as individuals in the population of the system. The rules are composed of two main parts, the condition and the action. The condition and action can be thought of as a series of 'If Then Statements' [82]. When particular conditions are met, a valid action will be the output. Therefore the condition can also be regarded as equivalent to the stimulus from the environment, i.e. the message, when the condition matches the environmental message. The action is the response from the agent or the class.

There are two main types of LCSs, the Michigan based and Pittsburgh based varieties. The main difference between both types is that Michigan based LCSs tend to produce one population of classifiers that together constitute the solution to the problem. The Pittsburgh variety produce sets of solutions where each set can be a solution to the problem. Also, Michigan based LCSs can be used for solving online as well as offline problems while Pittsburgh based LCS are mainly for offline problems [88].

The classifiers of an LCS consist of two main parts, the conditions part and the action part. What these indicate is that a particular number of conditions can map to a valid action or classification. The classifiers constitute rules that can be applied to a problem in a domain in order to map all possible classifications. For example, the conditions could communi-

cate that if a car has four doors, four wheels and uses gasoline, then it is classified as a sedan. The purpose of the classifiers is to niche – or partition – the problem space into sub-parts that are easier to handle by the agent [79]. However, it is not possible to exhaustively enumerate some problem spaces because they may be too large and hence require an impractically long time to search. The LCSs address this type of situation by introducing generality into the classifier rules. This indicates that the representation used in the classifiers is very important in solving the problems.

Generally the major parts of an LCS are the encoding of the environment, the learning agent, the population of classifiers and the discovery component. In the Michigan style of LCS, the environment provides a stimulus that is encoded by the agent. The agent then finds or creates the set of classifiers that match the environment signal [79]. When the agent creates new classifiers, this is known as *covering* [95]. Depending on whether the current phase is training or testing, the agent chooses an action out of the set of possible actions. If the current phase is testing then the best possible action is chosen. If training, a random action is chosen. At that point the classifiers matching the predicted action are chosen out of the previously created matchset. These are the classifiers that make up the actionset. The action is executed and the environment provides a reward. This is a normal aspect of reinforcement learning, and it has implications on the population set. A Genetic Algorithm may be activated during the testing phase, such that new classifiers are created by choosing two useful parents and cross breeding them [63]. Mutation may also be applied, which plays an important part in enforcing diversity among the population of classifiers.

The measure of classifier worthiness has changed over the decades during which the LCS technique has been used [79]. Two of the main ways for this determination are strength and accuracy. The Zeroth Level Classifier (ZCS) is a strength based LCS, which exemplifies simplicity over

its predecessors. However ZCS exhibits unsatisfactory performance due to the abundance of over-general classifiers [79], [88], [94], [95]. The technique eXtended Classifier System (XCS) differs from other types of LCSs in that the fitness of the classifiers is based on the accuracy of the prediction of the reward and not on the prediction itself. This means that the population of classifiers is pressured towards being accurate, however, specific classifiers that satisfy certain niches are not readily eliminated. This often results in a compact population of classifiers that is maximally general and accurate [22], [95].

Learning Classifier Systems have provided numerous benefits to the field of machine learning such as human interpretable solutions and arguably, ease of implementation, but they suffer from a number of weaknesses. They tend to produce large final populations, which can be difficult to process when solving problems. Also, the original ternary representation of the classifiers can not address complex domains due to its simplistic nature. Another major flaw in LCSs is that ordinarily they tend to throw away any learned knowledge and must begin anew when solving a new problem.

The introduction of aspects of Genetic Programming (GP) into LCSs has enabled richer alphabets. Genetic Programming is part of the Evolutionary Computation (EC) paradigm and produces solutions represented by trees. The tree nodes can contain functions, giving the LCS the capability of addressing complex domains [80], [102]. More recently, Code Fragments (CFs) were introduced into the XCS framework enabling knowledge to be maintained during successive problems in the same domain. This was achieved by storing all the learned knowledge for later usage. This increased scalability and increased the power of expression in LCSs. CFs are GP-like sub-trees which utilize a given function-set in the root nodes. The function-set is problem dependent, e.g.  $\{+, -, *, /\dots\}$  for symbolic regression problems, and  $\{\text{AND}, \text{OR}, \text{NAND}, \text{NOR}\dots\}$  for binary classification problems [49]. CFs also utilize a set of terminals at the leaf

nodes, i.e.  $\{D_0, D_1, D_2, \dots, D_{n-1}\}$ , where  $n$  is the length of the environmental input state [49]. The terminals map to the problem features, see Figure 1.1. Just like there is a don'tCare symbol in the original ternary alphabet of classifiers, there is a don'tCare CF. It always returns 1 and serves the same purpose as the original '#' in the ternary alphabet, see Figure 1.2. CFs have an initial maximum depth of two [44]. This limit was chosen because it was determined to help limit bloating, i.e. the introduction of large numbers of introns. Introns are seemingly useless genetic code that accumulates throughout the run of a system. In EC introns are a liability because they increase the search space. Analysis suggests that there is an implicit pressure for parsimony [48].

CFs have been combined successfully with a number of techniques. A system where XCS was combined with CFs is XCS with Code Fragment Condition (XCSCFC), see Section 2.5 (page 31). This technique was instrumental in solving until-then intractable problems, e.g. the 135-Bit Mux, i.e a Boolean problem. Another CF-based system is XCS with Code Fragment Action (XCSCFA) [42]. This type maintained the ternary alphabet for the condition part, but substituted a CF for the action part. The system provided more expressivity for the action part while providing an unanticipated benefit. In certain XCSCFA configurations the final population of classifiers was autonomously divided into optimal and sub-optimal sub-populations [45]. This eased the process of simplification, see Chapter 2. Normally, LCSs do not produce final populations in this manner, hence the need arises for a compaction algorithm to streamline the final classifiers.

The above stated techniques and expansions have introduced many improvements in scalability. However, inherent limitations in the same, provide opportunity for new techniques to help an LCS scale better during increasingly larger problems.

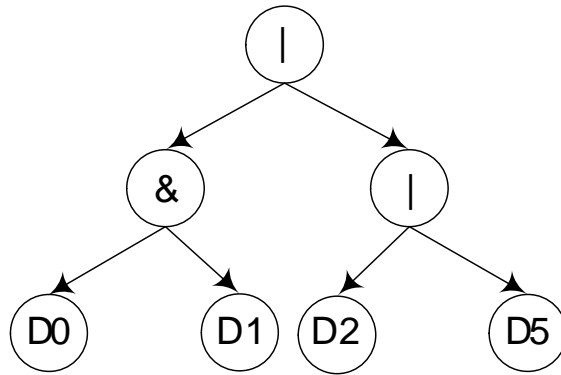


Figure 1.1: An example of a code fragment. Here D0, D1, D2 and D5 represent classifier condition bits. Also, '&' represents logical AND, and '|' represents logical OR.

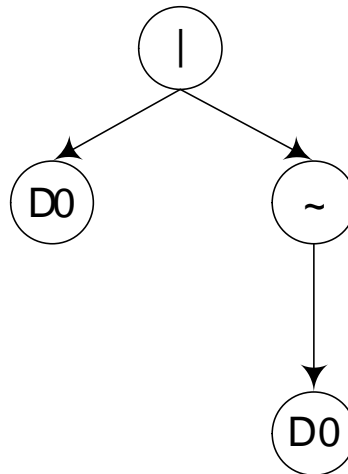


Figure 1.2: An example of a don'tCare code fragment. The don'tCare CF always outputs a 1. This is important because in order for a classifier to take part in a matchset all of its condition CFs have to return a 1

## 1.2 Motivation

Although XCSCFC fulfilled its potential in terms of increased scalability and an increased power of expression, it suffers from a systemic weakness.

Since the CF sub-trees reuse other CFs at the leaf nodes, the chains of CFs grow to intractable lengths. At some point, depending on the problem, the system stops learning as the search space increases. While XCSCFC was the first LCS to successfully solve the 135-bit multiplexer, it was not capable of solving any of the more difficult multiplexer problems.

Ironically, one of the main reasons for XCSCFC's lack of scalability for the more difficult multiplexer problems was its very own advantage, its increased power of expression. Although the original ternary alphabet was limited and incapable of expressing complex domains, CFs suffer from too much expressivity. Since they are compact, they contain much more functionality packed in them than a ternary based alphabet. This increases the search space to the point that learning can be impacted negatively as the problem scales [49].

Other versions of the CF-based techniques have also been unsuccessful at learning specific problems, e.g. the 264-bit multiplexer. XCSCFA, another CF-based technique, produced well divided final populations where the useful and not useful classifiers were grouped together, however it was not capable of evolving a general solution to the multiplexer problem either. It is hypothesized that the reason is that the multiplexer problem is very difficult. It contains epistasis, which means that the importance of certain bits depends on the values of other bits in the message. The multiplexer problem is also highly non-linear and this makes it difficult for an LCS to discover the patterns that will solve it successfully in a tractable amount of time. More importantly, some of the functionality that could be very useful in solving this type of problem is not part of the Boolean domain [41]. This suggests that a system needs to learn information from multiple domains, store that knowledge and successfully use it in the solution of further problems. New techniques are needed to increase scalability and learning of LCSs.

Code Fragments enable adaptation of their terminal set because of their rich and dynamic expressions, i.e. created CFs are added to the termi-



nal set. Furthermore, All other common EC approaches predefine their function set, which is not linked to the terminal set [49]. This includes discovered building blocks, see Chapter 4. A novel method is needed to enable functions within an LCS to be adapted and linked to the associated discovered building blocks. It is hypothesized that this will provide increased scalability by reducing the search space. There is a similarity between the arguments and return values of functions and the condition part and actions returned by rule sets that could be exploited. This could provide benefits in the form of modularity for the final rule-sets produced by an LCS, similar to a function. During covering, only certain groups of classifiers could be processed as opposed to conducting a search on the entire final population.

The choice of alphabet plays a major role not just in the learning capability of a system, but also in its practicality within a problem domain. As such, there is a trade-off between sample space and search space. Low-level alphabets, such as the ternary alphabet, are fast to execute, compact and straightforward to interpret, but the search space is poor such that the algorithm may become trapped in local optima, see Section 2.1. High-level alphabets, such as GP trees are expressive [81], so they can avoid local optima by redesigning the search space, see Section 2.1. On the other hand, they are slow to execute, are susceptible to bloating and are difficult to interpret [101]. It is hypothesized that it could be possible to transform the high-level CF solutions, retaining their benefits, to a low-level alphabet, thus reaping their benefits in future processing, see Chapter 5.

Learning Classifier Systems excel at creating a complete state-action map, these are rules that cover the entire input space. They accomplish this by discovering underlying patterns that link features to classes in the training data. This is also true if there exists an observable, computable relationship between features and classes in the data. For example, a weighted sum approach of the raw features [97]. However, if the relationships are not linearly separable and the appropriate functionality is

not available in the function set, the learning systems will struggle to discover such patterns. Furthermore, even if the functionality is available in the function set, it is not clear what building blocks of learned information should be used as input to the functions. Humans overcome this problem by using threshold concepts where a teacher provides a curriculum of learning to ensure that a new learner has appropriate functionality available to solve successive problems that build on each other. Concepts from layered learning and transfer learning will be considered to adopt a similar approach into LCS, see Chapter 6. Layered Learning (LL) pertains to the successive, bottom-up learning of a problem that has been broken into sub-problems by a human. This could be a useful approach, considering the limitations of CFs. Transfer Learning (TL) pertains to the technique where learned information from a problem domain is used to solve a problem in a target domain. This new domain may be the same or a related domain [106].

### 1.3 Thesis Statement

The proposed thesis is that a CF-based LCS can scale to complex problems by reusing LCS populations as functions at the inner nodes of CFs, using online compaction and using layered learning in the same or related domain.

### 1.4 Research Goals

The overall practical goal following from this thesis is to increase the scalability of learning classifier systems. The overall goal is divided into the following three sub-goals:

- \* **Reuse** learned functionality captured in LCS populations from smaller problems in the root nodes of CF sub-trees.

- \* **Create** an online compaction technique that facilitates reduced evaluation time of CFs.
- \* **Develop** a method to train a CF system in a manner leading to a general solution. This will be demonstrated through an intractable problem, i.e. the n-bit Multiplexer.

The three objectives are:

- **Show** that inner node-based functionality is beneficial. This objective will be evaluated by testing it with progressively more difficult multiplexer problems up to and including the 135-bit multiplexer. The technique will also be tested with the 3x6 hidden multiplexer problem. Here there will be three separate training paths for the new system. These will be used to determine the benefit of including or omitting certain functions from the training.
- **Create** a compaction technique to simplify the final population of CF rules. The technique will be online and will translate the CFs into an equivalent rules-set using a ternary alphabet. The benefit of the technique will be measured by its performance during a series of multiplexer problems. The system will also be tested with the 3x6 and 3x11 hidden multiplexer problems.
- **Show** that Layered Learning can provide benefits in scaling for a CF based LCS by solving an intractable problem. The n-bit multiplexer problem will be partitioned into constituent sub-problems by a human. The technique will then be trained on these sub-problems in series. Following this, rules for the 6-bit multiplexer will be produced by the proposed system. At this point, the rules are anticipated to be maximally general and able to cover the entire problem space. To determine this, the rules will be tested against progressively more difficult multiplexer problems of very large size, e.g. 264-bits and above.

XCS will be used to implement the techniques listed above; it is a well tested and studied implementation of the LCS technique. Additionally, Layered Learning is flexible, accurate as it has full functionality and is easy to adapt to Transfer Learning. For these reasons the techniques mentioned above are well suited for the purpose of this thesis.

## 1.5 Major Contributions

The novel work in this thesis is based on a body of knowledge that includes CFs. These were rooted in ADFs from GP but were not predetermined structures [43]. CFs have been used in the condition or the action of a classifier [49], [42]. Subsequently, the concept of generative representation in LCS was extended to finite state machines [46]. CFs could only be used at the leaf nodes of CF sub-trees, which meant that knowledge could be transferred but crucially not functionality. A scaled approach to learning thus could be adopted, but not a layered learning approach. This meant that learned functionality could not be continually expanded. The novel work used CFs at the inner root nodes of CF sub-trees as well as the leaf nodes. The resulting problem of long chains of learned knowledge and functionality can be reduced using the novel compaction technique which reduces a rich alphabet, that is slow to process, into a fast but simple alphabet. The novel technique utilizing a layered learning approach in LCS with CFs at the root nodes as well as the leaf nodes of CF sub-trees is crucial in increasing the scalability of an LCS. In addition it transforms EC practitioners from specifiers of problems to specifiers of curricula in order to solve simple problems that lead to a solution of more complex problems that could not have been solved previously.

This thesis has led to the following major contributions to the field of Evolutionary Computation in general and to the field of LCS-based learning in particular.

- (a) A main contribution of this work was showing that a growing set of

learned functions reused in the inner nodes of a code fragment tree can be beneficial. Using this approach, the system was capable of solving up to the 135-bit multiplexer and the 3x6 hidden multiplexer.

Parts of this contribution have been published in:

IWLCS 2014 - Reusing Learned Functionality in XCS: Code Fragments with Constructed Functionality and Constructed Features [3]

SEAL 2014 - Reusing Learned Functionality to Address Complex Boolean Functions [4]

- (b) A compaction technique was created which facilitated the solution to numerous problems with high epistasis and non-linearity. The technique transformed the representation of knowledge from CFs to a ternary alphabet. The novel work culminates in an innovative on-line method to produce Distilled Rules (DRs). The new technique was capable of producing DRs for the 70-bit multiplexer problem. This is something that the original DRs technique was incapable of accomplishing.

Parts of this contribution have been published in:

ACALCI 2016 - Compaction for Code Fragment Based Learning Classifier Systems [5]

CEC 2016 - Compaction for Code Fragment Based Learning Classifier Systems - Redux [6]

- (c) The scalability capability of a CF-based LCS was increased by using Layered Learning as well as Transfer Learning. The novel work reused learned knowledge and learned functionality to scale to complex problems by transferring them from simpler problems. Progress was demonstrated on the benchmark Multiplexer (Mux) domain.

Parts of this contribution have been published in:

GECCO 2016 - Human-inspired Scaling in Learning Classifier Systems: Case Study on the n-bit Multiplexer Problem Set [7]

## 1.6 Organization of Thesis

The remainder of this thesis is organized as follows.

Chapter 2 provides a detailed description of LCSs along with an overview of related evolutionary machine learning and knowledge transfer learning approaches. This chapter also describes various encoding schemes that have been used by the LCS community to represent classifier rules.

Chapter 3 describes two research methodologies adopted in this work, to achieve the overall goal, and briefly describes the systems designed and implemented following these research methodologies. The details of each implemented system are provided in a separate contribution chapter, from Chapter 4 to Chapter 6. It also provides details of the problem domains experimented here and the experimental setup used for testing and evaluating the developed systems. This chapter also clarifies various differences between the evaluation of an LCS and a traditional evolutionary machine learning approach.

In Chapter 4, building blocks of knowledge are successfully extracted from small-scale problems and reused at the root nodes of CF sub-trees to learn more complex, large-scale problems in the domain. By utilizing this novel learning approach, the resulting system readily solves problems of a scale that existing classifier system and genetic programming approaches find troublesome, e.g. the 135-bit MUX problem, hidden multiplexer.

In Chapter 5, the compaction capability of a classifier system is increased beyond what normal subsumption and deletion can normally achieve. This helped the system to learn a series of Boolean operators without the use of hard-coded functions and subsequently more difficult problems such as the 3x11 hidden multiplexer.

In Chapter 6, the scalability of CF-based classifier systems is increased by using a layer learning approach to the training phase. A difficult problem is broken into major constituent parts. The parts are solved in succession while reusing previously learned knowledge during the next part. The type of systems used here maintained the usual ternary alphabet for the condition part while substituting a CF for the binary action part. The developed system produced general solutions of any scale  $n$  for the multiplexer problem.

Chapter 7 presents the achieved objectives, main conclusions from each contribution chapter, and the future work that stems from this research work.





# Chapter 2

## Literature Review

This chapter provides an overview of Evolutionary Computation (EC), Learning Classifier Systems (LCSs) and Code Fragments (CFs). This chapter will also be devoted to describing Transfer Learning (TL), Layered Learning (LL) as well as other methods for expressing rules in LCSs. This outline structure follows closely the progressive development of the thesis work.

### 2.1 Evolutionary Computation

Evolutionary Computation (EC) is a paradigm for problem-solving techniques. Characteristically, these techniques are based on principles of Darwinian evolution, such as natural selection and biological principles such as genetic inheritance [24], [52], [93]. EC is deemed a suitable approach to this thesis because it encompasses numerous techniques which have been found beneficial for specific tasks such as learning and scalability [48], [49], see Chapter 4, Chapter 5 and Chapter 6. One such technique is Learning Classifier Systems (LCS) [13], [62].

Another EC member is Genetic Programming (GP). It is a technique that genetically evolves a population of programs that can produce a series of algorithmic steps. These steps can constitute an answer to a prob-

lem [54]. In GP, the program is usually represented in the form of trees [16]. The tree nodes consist of two sets of members: functions and terminals. The goal is to evolve programs that could compute a solution [65]. The functions are drawn from the reservoir of functions provided *a priori* [74]. These could take the form of arithmetic operators such as  $\{x, -, +, /\}$ , or boolean operators such as  $\{\text{AND, OR, NOT, XOR}\}$  [75]. The terminals can take the form of constants, such as integers or real numbers  $\{1, 3, 2.2, 6.9\}$ , or they could represent variables, such as distance  $x$  or height  $y$ . Normally the inner nodes will be comprised of functions from the function set while the leaf nodes are comprised of terminals from the terminal set. GP allows for complex representations of solutions by leveraging a rich alphabet composed of the terminal set and the function set. These programs are flexible and powerful but suffer from bloating, i.e. constant growth of the trees. However, there exist different mechanisms for limiting the creation of introns, i.e. apparently useless genetic code [104], [105].

GP is an integral part of this work since its representation is key in extracting reusable building blocks of information. GP assists in translating the representation of the classifier condition into a more abstract, complex form [11], [52], [70]. This representation forms the basis for much of the new work presented here and has been crucial in enabling the extraction of building blocks of knowledge as well as increasing the scalability to more complex problems [49], see Section 6.1. The disadvantage of using GP is that it has a very large search space caused by the rich alphabet and would make it unfeasible for solving some of the more complex problems such as the 135-bit multiplexer [49], [103].

## 2.2 Learning Classifier Systems

The initial description of the Learning Classifier System (LCS) was of a cognitive system [34]. The concept was that this kind of system could learn about its environment and about its state, thus enabling it to execute bene-

ficial actions on its environment [99]. Cognition is a term from psychology that is defined as the process of acquiring knowledge and understanding through thought, experience, and the senses [29]. This implies the ability to consider ostensibly disparate information and then apply it towards a desired goal [98].

Holland and Rietman proposed the first Learning Cognitive System (CS-1); this was the first implementation of an LCS [13]. The Learning System One (LS-1) developed by Smith exhibited an advantage over CS-1 in that it could learn from two different problem domains simultaneously [36], [73]. This capability afforded the benefit of building a repository of learned information, which was more complex than previous learned functionality. This in turn was useful when attempting more difficult problems that required complex information to achieve their solutions. In spite of this advantage, currently a number of LCSs (and a majority of EC techniques) only consider one domain at a time for the sake of simplicity. The LS-1 work is important – even though the technique used was the Pittsburgh as opposed to the Michigan technique used here – because it demonstrates that an LCS is capable of learning from different domains, a quality that will be indispensable in the proposed work.

Recent Learning Classifier Systems (LCSs) come in two main varieties, the Pittsburgh and Michigan approaches [79]. The work here is based on the Michigan style LCS and hence the reader is directed to [61] for a review of the former. Figure 2.1 depicts the main highlights of a Michigan based LCS. This figure is based on the XCS system developed by Wilson [15], [95]. XCS is notable for its popularity in the study of LCSs and its extensive body of knowledge. XCS differs from its predecessors in a number of key ways: 1) XCS uses the accuracy of predictions instead of the amount of the reward, this promotes a solution encompassing a complete state-action map of the problem via accurate rules [51]; 2) mutation and a GA are applied to niches – subsets of the population – instead of panmictically – across the entire population; 3) unlike the traditional LCS, XCS has no

message list and therefore it is only suitable for learning Markov environments [22], [95].

XCS has been described as an evolutionary, adaptive agent, which is used to learn information via a set of mutually cooperative rules [89], [95]. XCS learns by interacting with the environment, see Figure 2.1. The population of classifiers is typically initialized by covering the individual data patterns from the environment input and eventually generalizes the population by removing irrelevant information [37], [42], [96]. This type of technique is composed of three major parts: the environment, the agent and the population of rules [38]. The environment provides stimuli to the agent, which in turn provides a viable action based on the message presented by the environment. The environment then provides appropriate feedback, depending on the worth of the action proposed by the agent [78]. The actionset is updated. The Genetic Algorithm (GA) may be activated at this time [32], [90], it may insert new classifiers into the population [12], [30]. If the number of classifiers in the population exceeds the maximum population allowed, unworthy classifiers will be chosen for deletion [22].

XCS is a good method for solving certain problems because it produces a complete state-action map. Since the solution is a population of classifiers, each classifier represents a portion of the overall solution [61]. The action of each classifier represents one possible decision that could be taken by the system based on the environmental stimulus. Accuracy based LCSs, e.g. XCS [61], [88], attempt to produce the final classifier population as a collection of general classifiers with optimal accuracy and fitness [21]. The concept is that the suboptimal individuals would have been eliminated by preferential selection of optimal individuals through the process of evolution [18].

There are two types of subsumption in XCS, GA subsumption and action set subsumption. GA subsumption normally occurs when creating a new offspring classifier. If it can be subsumed by an accurate and suffi-

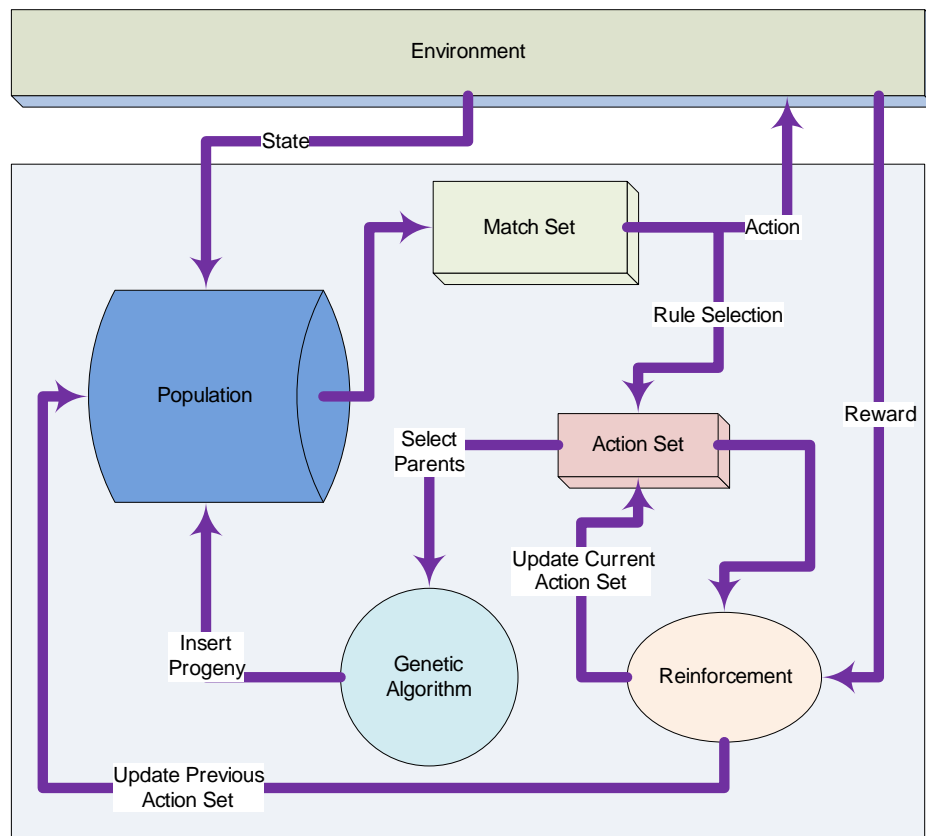


Figure 2.1: XCS framework. A match set is created from the population of classifiers. These are all the classifiers that match the environmental state. Each available action is assigned a possible payoff by the classifiers. Based on this array of predictions, an action is chosen. An action set is created from the matchset and the action is executed. Depending on the reward, the action set is updated and the GA may be applied [22], [95]. GA subsumption takes place right before the offspring are added to the population. If the new population size is larger than the maximum population limit, then classifiers are chosen to be deleted until the population is within the maximum population size.

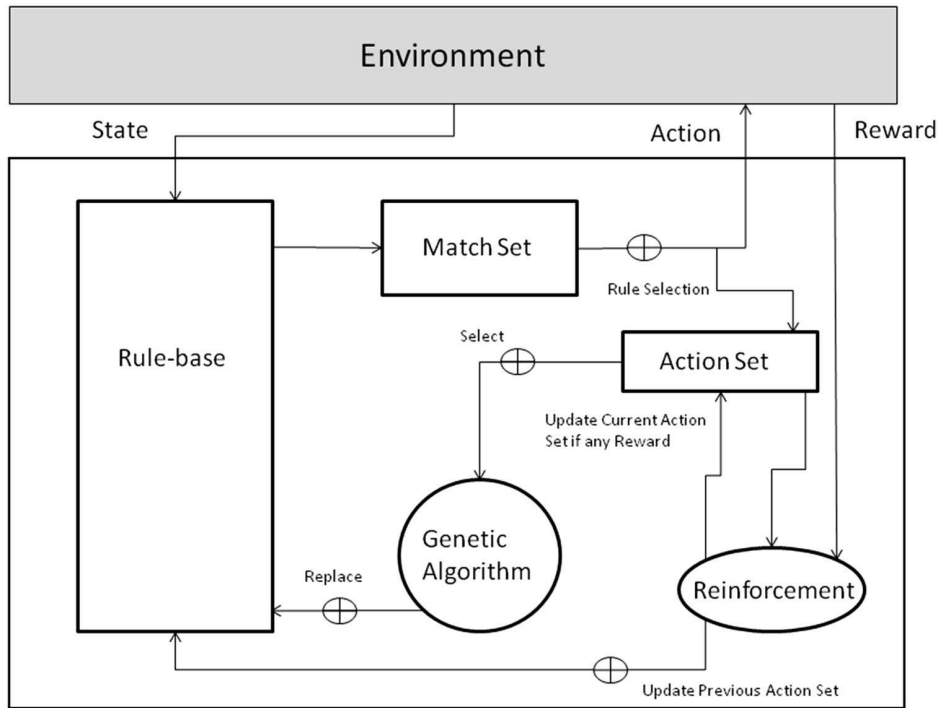


Figure 2.2: Schematic of XCS [13]. The population, matchset and actionset are three of the major components.

ciently experienced parent, it is not added to the population and the subsuming parent's numerosity is increased by one. The motivation behind this is that less redundancy, in terms of problem space coverage, translates into greater efficiency. Action set subsumption typically happens on the action set after having gone through the discovery phase. The principal idea of the mechanism is to identify classifiers that are both accurate and well experienced. The less general classifiers meeting this criteria are removed (subsumed) from the population and the subsumer has its numerosity counter increased. This method also provides benefits by reducing the computation costs associated with the subsumed classifiers and by producing a more condensed final population [22]. However, action set subsumption is often too harsh and as a result its turned off on many problems.

A similar subsumption technique has been used to extend a CF based XCS system as described in [39]. In this technique the object is to determine if a classifier  $c1$  subsumes a classifier  $c2$ . This is done by a process which invokes each CF and compares the result of the corresponding bit positions from both classifiers being compared. The more general classifier, the subsumer, will have all its CFs evaluating to 1. For example, if  $c1$  were determined to be more general than  $c2$ , then  $c2$  would get subsumed by  $c1$ . However, all of the CFs in classifier  $c2$  also need to evaluate to 1.

The first technique for compaction was proposed by Wilson [95]. The objective of the technique was to increase efficiency in XCS. With this in mind, an approach utilizing condensation was put forth. The method removes unnecessary classifiers from the final population, i.e. those superfluous to adequately cover the problem space. This shrinks the population of classifiers increasing efficiency.

Subsequently, Kovacs [50] proposed a seminal condensation technique known as Subset Extraction. The technique analyzes the system at a particular instant. Unlike condensation, it does not rely on incremental calculations or a delay. Therefore it can be applied at any interval in order

to reduce processing resource usage. The technique relies on finding a well-evaluated subset of  $[P]$ , i.e. the population. This subset must have a good accuracy rating, it must completely cover the input/action space and must not overlap. If found, this subset is considered a candidate optimal population. The subset must also have as few members as possible. Although this technique is effective in eliminating unneeded classifiers, it is very complex. In spite of this, Subset Extraction has served as inspiration for other techniques.

Dixon proposed a compaction technique that builds on previous methods [25]. This technique performs the compaction as a post-processing step that condenses the population and is apart from the routine steps that exert pressure on the population, such as niche mutation and subsumption. LCSs keep alternative hypothesis, e.g.  $\{\# 1 \# 1 \# 1 : 1\}$ ,  $\{1 1 \# \# \# 1 : 1\}$ . Both are accurate and maximally general, but it is desirable to only keep one hypothesis. The three main steps [25], [92] consist of the following: 1) a subset of classifiers that achieve 100% performance is identified; 2) any classifiers that are not good for performance are extricated from the set; 3) the classifiers are ordered by the number of inputs matched, this continues until all the inputs are matched.

This is where a GA plays a crucial role in the classifier; by preferentially selecting fitter individuals. The GA is applied if the average time period since the last GA application is greater than the threshold  $\theta_{GA}$ . Two parents are chosen by roulette wheel selection. These are based on fitness, and the offspring are created from them. The offspring may undergo crossover and mutation. At this point the new classifiers are inserted into the population, followed by corresponding deletions (if applicable). If GA subsumption is used, then each new offspring is checked to see if it can be subsumed by either parent. If this is the case, then the numerosity of the parent is increased by one, and the new child is not inserted into the population [22].

Although standard XCS encapsulates many benefits, it still needs im-



provements in order to solve more complex problems, e.g. the 135-bit multiplexer. One of the drawbacks is that although it can scale in certain domains, it still has to relearn from the beginning each time. Any increase in the dimensionality of the problem increases the search space, the hardware demands, and training time [42]. To address these pressures, different types of representation have been developed.

## 2.3 Representation

In an LCS the rules, i.e. classifiers, are composed of two main parts, the condition and the action. Originally the condition part utilized a ternary alphabet composed of:  $\{0, 1, \#\}$  and the action part utilized the binary alphabet  $\{0, 1\}$  [88]. This representation is simple and efficient to process and is beneficial in certain types of problems, such as boolean domains.

LCSs can select/deselect features using the generality inherent in the {don't care} operator [34]. Originally the don't care utilized a '#' hash-mark to mean that it could take the value of 0 or 1; this comprised part of the ternary alphabet  $\{0, 1, \#\}$ , [34]. Since the initial introduction of LCSs, the number of applicable alphabets has been expanded to include more representations such as Messy Genetic Algorithms (mGAs), S-Expressions, Hyper-ellipsoids [20] and Code Fragments.

Messy GAs process variable-length strings that may be missing values for some of their conditions, they may also have multiple values for particular conditions [31]. The concept is to build longer building blocks from smaller ones until a solution is discovered. They emphasized a different viewpoint on Holland's *Adaptation in Natural and Artificial Systems* [36]. Contrary to what many researchers hold as true, Holland was not against a "scruffy" type of GA. Accordingly, his work was put forth as a guide and not as a strict formulation of what a GA should be [31]. Subsequent researchers acknowledged this influence and developed richer representations based on a messy encoding.

Lanzi introduced an extension to XCS in which variable length messy chromosomes replace the original bitstring representation [59]. The sensory inputs are still translated into bitstring but the bits in the classifier's condition are not bound to the position of sensory input bits anymore. The technique had success with the Woods1 and Maze4 environments. In these environments, the systems tested were able to converge to a solution that was quite near the optimum but never reached it [59]. This was due to certain limitations in the representation.

Bitstring representation of classifier conditions has two major limitations: 1) it can result in a loss of information about the environment structure, i.e. not capable of capturing regularities in an environment; 2) the fixed correspondence between the position of bits in the classifier condition and the position of sensor bits can result in the incapability to represent complex environments, e.g. discrete problem domains [59]. A different representation was needed to obtain better scalability.

Regular S-Expressions was Lanzi's next foray into rich encoding of the condition part. S-Expressions are LISP-like expressions which have been used to express the LCS condition or action [58]. XCSL, as the system is known, uses LISP S-Expressions. The classifier conditions are restricted to the set of possible boolean functions that can be generated by combining the logical operators {AND, OR, and NOT} with elementary conditions over sensory inputs [58]. Though this technique is effective in eliminating the two limitations of a ternary encoding (mentioned above), it makes it difficult to determine if one condition is more general than another one. For this reason, the subsumption operator cannot be employed efficiently anymore and different heuristics must be developed, such as condensation techniques [58]. Another problem is that the complexity of the classifier conditions tends to grow as the learning proceeds. One important caveat is that both of Lanzi's techniques mentioned above aimed to replace the entire condition of the classifiers, while the work here aims to replace individual features of the condition with GP sub-trees [58]. Finally, while

Lanzi's work did not reuse the learned functionality, the proposed work does. However, some of Lanzi's later work involved concepts that are currently used in XCSCFC and in the proposed work. One of these contributions is described in [67], while another is the XCS with stack-based GP technique. The Simple Compact Genetic Algorithm (SCGA) has similarities with the novel compaction work in this thesis. SCGA utilizes a probability vector from the action-set to create new offspring. This is similar to the vectors used by the online compaction technique described in this work. However the vectors here are used to help identify an optimal population of CF-based classifiers.

Stack-based GP extended XCS with conditions based on stack-based Genetic Programming [60]. The condition part of the classifiers are linear programs written using Reverse Polish Notation (RPN) and are interpreted by a virtual machine. The application of tree-based genetic programming is imbued with computational costs. These are related to the genetic operators, crossover and mutation. Usually they have to examine a large part of the parent conditions before progeny can be created, which is due to the fact that GP expressions can grow long due to bloating. This technique provides the benefit of reducing the computational costs associated with the genetic operators adapted from tree-based Genetic Programming. This technique shares some common aspects with CF-based systems.

Bull and Ahluwalia proposed an alternative representation of the classifier [1]. In the system known as GP-CS they used a binary alphabet for the condition part and an S-Expression for the action. Combined with the K-nearest neighbor algorithm, this approach was successful in the classification of various data-sets [1].

Koza devised a technique known as Automatically Defined Functions (ADFs) [55]. ADFs are a technique for reusing generic sub-parts of GP trees. These are defined once in a top-down manner, along with place holders for their parameters. These then receive a sub-expression, or their

input parameters, when they are called from within a GP tree. ADF is one of the precursor techniques of the proposed work and therefore is an important development.

Another method proposed by Bull and Ahluwalia integrated an LCS with ADFs. They utilized ADFs to develop a filtering program to reduce the number of features. In [14] the ADFs were the feature preprocessors/extractors and the result was fed to a K-nearest-neighbor (Knn) classifier. This technique showed that explicit feature selection at the level above the coevolving extractor functions provided better performance than traditional approaches. This was because unneeded features could be ignored more effectively [14]. A number of key findings emerged from Bull's further work described in [2] and [14]. It was found that all coevolutionary approaches performed better than the standard approaches. Also, the use of feature selection along with feature extraction performed best of all. The technique uses automatically defined functions (ADFs) and assigns an independent sub-population to each one. These sub-populations co-evolve with other ADF sub-populations and a population of the main program trees.

Bull and Ahluwalia also introduced the technique called Evolutionary Defined Functions (EDFs). EDFs in turn contained two new mutation operators: compression and expansion [2]. These operators, along with the coevolutionary method, enabled the automatic specification of EDF sub-populations via compression. EDFs also contained a counter which was useful in determining the EDF's worth to the evolutionary process. This form of credit assignment could be useful for future work, however in the proposed work the number of axiomatic functions and those to be learned by the system are not numerous enough to warrant this method. There is opportunity of using a similar grading method for the CFs evolved in the proposed work. It is anticipated that the new system will combine learned functionality based on the effect that reinforcement exerts on the evolutionary process. More importantly, the types of inputs and outputs

of the different functions will be a gatekeeper, disallowing any illegal combinations of functions. Further, the problem will be presented as a series of sub-problems, so there can be a sequential measure of the usefulness of the axioms and sub-problems presented to the system [2], [14].

While Bull and Ahluwalia's work added a level of abstraction to the representation of the classifier, as well as insight into not just the creation of functions, but also their usefulness to the production of a solution, it lacked a very important component of the proposed work. It did not reuse the learned building blocks to solve a more complex problem. The concept of the ADFs is important here because of its obvious relationship to function learning, however the proposed work aims to learn brand new functions online, along with the number of their parameters. The number of parameters will be set by a human at the start of the training for each function.

## 2.4 Code Fragments

Code Fragments (CFs) were introduced in the form of GP-like subtrees [43]. They have an initial maximum depth of two, as this number was deemed important to limit bloating, see Figure 2.3. CFs can utilize previously generated CFs in their terminal nodes as the problem scales. CFs have enabled numerous solutions to previously intractable problems, such as the 135-bit multiplexer. They have also helped to find optimal populations in discrete domain problems as well as in continuous domain problems [49]. Importantly, CFs have enabled the re-usability of learned information in LCSs in novel ways. For example, CFs have enabled the solution to previously intractable problems like the 135-bit multiplexer [49]. CFs have also been used to produce compact rule sets that can be easily converted to the optimum population [42].

The Code Fragment technique shares a number of properties with ADFs. Both techniques used human-defined functions, including a def-

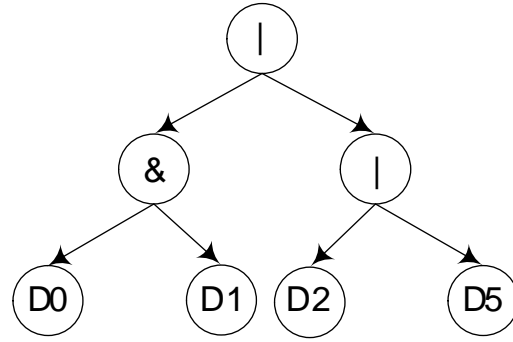


Figure 2.3: An example of a code fragment. Here D0, D1, D2 and D5 represent classifier condition bits. In addition, ‘&’ represents logical AND, and ‘|’ represents logical OR.

inition for the number of arguments these would take. In ADFs the functions can be used by establishing a constrained syntactic structure for the individual expressions in the population [56]. For example, the number of arguments each ADF can take must be defined *a priori* and cannot be changed during evolution [28]. This argument constraint also applies to CF-based learned functions, with the exception that CF functions are learned online. While CFs do increase their number of arguments, each new problem has a limit of arguments dictated by the number of CFs allowed for the problem. A CF is an expression, similar to a tree generated in Genetic Programming [44]. CFs generate small blocks of code in binary trees. CFs have also been expressed using sub-trees with more than two terminals, with varying degrees of success [4]. Analysis suggests that there is an implicit pressure for parsimony as only the CFs found to be useful survive to form part of the final population [47], [48].

Numerous CF-based XCS systems have been created, where CFs (often built upon previously learned CFs) contain building blocks of knowledge. These building blocks of knowledge enable feature selection and feature construction by including important message bits at the terminals. These bits would have been found important for the current problem. For exam-

ple, CFs could determine that the address bits of a message are important for determining the data bit of a multiplexer problem. In addition, CFs use a rich alphabet meaning that their relationships are functional instead of linear. These CF-based systems will be described in detail in the following section.

## 2.5 Code Fragment Based Systems

Current LCSs can be utilized to extract building blocks of knowledge. These blocks of knowledge can then be used to solve more difficult problems in the same domain. The past work showed that the reuse of knowledge through the integration of CFs into the XCS framework could assist in scaling [49].

The first investigation involving code fragments was the introduction of GP-tree like expressions to represent condition bits in a terse classifier rule, which was named *code-fragment conditions* XCSCFC [49]. See Algorithm 1 for an overall view of the execution. There are similarities with the XCS technique, however there exist differences at the matchset and classifier comparison levels. Algorithm 2 depicts the matchset creation process; the matching is incumbent on a code fragment evaluating to 1. All the code fragments in a classifier must meet this requirement for the classifier to be added to the matchset. Algorithm 3 shows how XCSCFC determines if one classifier is more general than another, this becomes useful when comparing two classifiers. In this approach, the conditions in a classifier were replaced with code fragments. Thus enabling feature construction in the condition of the classifiers. The action part used the binary alphabet  $\{0, 1\}$  [49]. An important benefit inherent in CFs is the decoupling between a CF and a position within the condition, i.e. the ordering of the CFs is unimportant. In fact, the number of CFs present in the condition does not have to be the same as the number of features in the environment as these are available in the terminal set, i.e. for a multiplexer problem with a con-

dition of length 6, 3 CFs could be enough to find the solution. Although XCSCFC exhibited better scalability than XCS, eventually, a computational limit in scalability was reached [46]. The reason is that as multiple CFs can be used successively at the terminals, as the problem domain is scaled, then a depth of tree of  $2^{\text{number of problems scaled}}$  could be created.

---

**Algorithm 1:** XCSCFC: Execution

---

**Data:** The currently observed input state  $s$ .

**Result:** Final population of accurate and general classifiers.

```

1  $x \leftarrow$  Maximum number of training instances
2  $i \leftarrow 0$ 
3 while  $i < x$  || not converged do
4   Create matchset [M]
5   Create actionset [A]
6   Execute action
7   Get reward
8   Update [A]
9   Execute GA
10   $i \leftarrow i + 1$ 
11 end
12 return true

```

---

Initially, there was a separate population of code fragments used. This population was randomly created and kept static throughout the learning process [43]. Currently the CF population is housed simply within the rules [48]. Each time a new classifier is created, a random CF is also created for each of the required bit positions [49]. This means that the number of code fragments to be reused from a particular level is governed by the unique CFs in good classifiers.

By using CFs, XCSCFC increased in scalability and was able to reuse knowledge learned in a simpler problem in a more complex one of the same domain. For example, in the 135-bit multiplexer problem, XCSCFC



---

**Algorithm 2:** XCSCFC: Matchset [49]

---

**Data:** The current input state  $s$  and a classifier  $cl \in [P]$  where  $[P]$  is the population of classifiers.

**Result:** If the classifier  $cl$  matches the observed state  $s$  then the result is true otherwise false.

```

1 for  $i = 1$  to  $n$  do
2    $cf \leftarrow$  code fragment from  $[q]$  indexed at  $cl.condition[i]$ 
3   if  $cf \neq \text{'don't care' code fragment}$  then
4     /* Environment feature */
5     load terminal symbols in  $cf$  with corresponding binary bits
      from state  $s$ 
6      $val \leftarrow$  evaluate value of  $cf$ 
7     if  $val \neq 1$  then
8       | return false
9 end
10 return true

```

---

takes only  $2 \times 10^6$  instances to successfully solve the problem compared to the search space of  $4 \times 10^{40}$ . The standard XCS was not able to solve the same problem [49].

The next step in CF-based systems was to replace the static binary action by a code fragment while using the ternary alphabet in the condition of the classifier rules to create XCSCFA (XCS fragment actions) [42]. Each code fragment was a binary tree of depth up to  $d$ . The value of  $d$  was dependent on the length of the condition in a classifier [42]. The action value of the classifier was determined by evaluating the action CF [47]. In order to achieve this, it was necessary to populate the CFA's terminal symbols [42], [45]. The terminals in the CF tree could be replaced with either the corresponding bits from the environment message or with bits from the classifier condition. In the later case a don'tCare symbol, i.e. '#', in the condition was randomly treated as 0 or 1 [42].

---

**Algorithm 3: XCSCFC: IsMoreGeneral [49]**


---

**Data:** Two classifiers  $cl1$  and  $cl2$ .

**Result:** If classifier  $cl1$  is more general than classifier  $cl2$ , then the result is true otherwise false.

```

1  $n \leftarrow$  'number of don't care' code fragments in  $cl1$ 
2  $s \leftarrow$  'number of don't care' code fragments in  $cl2$ 
3 if  $n \leq s$  then
4   | return false
5  $N \leftarrow$  set of all 'non-don't care' code fragments in  $cl1$ 
6  $S \leftarrow$  set of all 'non-don't care' code fragments in  $cl2$ 
7 if  $N \not\supset S$  then
8   | return false
9 return true

```

---

Subsumption deletion was effectively disabled, leading to a larger search space [42]. The reason that subsumption was disabled is that a phenotype could map to multiple genotypes. Since the CF action is populated with terminals from the condition, any don'tCares were treated randomly as 1 or 0. This produced an inconsistency in the value of the CF action which made it difficult to determine the generality of the classifiers. One way to get around this is by comparing the CF actions on a bit by bit basis but this can be troublesome [42]. Whereas in standard XCS with binary action, subsumption deletion is fully enabled. This means that the numerosity of the general classifier in a niche gets higher values as it subsumes the less general classifiers in the niche. In code fragment based XCS, the multiple genotypes to a single phenotype issue disables the subsumption deletion function. Therefore, fitness in a niche is distributed among multiple equally general classifiers, all having a relatively small fitness value as compared to the binary action-based XCS [42]. However, the lack of subsumption deletion was compensated by the fact that code fragments contained useful knowledge, such as the importance of the address bits in

the multiplexer problems.

Another advantage of utilizing CFs in the action of a classifier is the autonomous separation of optimal and sub-optimal classifiers in the final population, which is due to the consistency of action formed by these classifiers [45]. This eventually results in the optimum rule set of the maximally general, compact and accurate classifiers, see Table 2.1. This is useful because it produces a solution that can be easily converted to the optimum population [48].

Consistency of action is an obscure but interesting phenomenon of XCSCFA. It arises whenever don'tCares are effectively in the action, which is a highly unusual practice for a classifier system. That is, the same classifier can effect different actions depending on the features covered by the condition. This occurs when the CF terminals are filled from a condition part containing don'tCares. The input message is matched by a classifier, which has to convert its CF in the action to an output. xxx This could be achieved by substituting the environmental message into the CF of the the action part. However, if a '0' or '1' is substituted randomly for the don'tCare, an interesting effect occurs. There is a complete separation between the classifiers that are optimally fit and the newly created/sub-optimal ones, as inconsistent actions are more likely to be incorrect and thus removed from the population. This means that it is possible to separate optimal from sub-optimal classifiers without condensation, see Chapter 5.

XCSCFA does not scale to very large problems, even if the problems had repeated patterns in the data. This shortfall was due to the inconsistencies in the values of the CF action. For the larger problems, e.g. 70-bit multiplexer and beyond, the sample space is too large for the system to learn the problem effectively.

For many problems, e.g. the multiplexer problem has multiple niches, there is not a single fully general rule that would provide the solution using a ternary condition. It is possible to have don'tCares in the address bits

Table 2.1: A sample of classifiers from the final solution in XCSCFA for the 20-bit multiplexer problem. XCSCFA has the advantage of producing the optimal classifiers separated from the sub-optimal ones with respect to the numerosity [45].

No.	Condition	CF Action	n	p
1	1011#####1####	D2 D0 & D0 &	9	1000
2	0010##0#####	D6	9	1000
3	1100#####1###	D1 D13   ~	9	0
4	1001#####0####	D13 D9 d D13 D9 d d	9	1000
5	0111#####1#####	D0 D0 d D0 D0 d r	9	0
6	1000#####1#####	D2 D2 d D2 d	8	1000
7	1000#####0#####	D6 D0 ~ d	8	0
8	0111#####0#####	D11 D11 D13 & r	8	0
9	0101#####0#####	D11 D0 & D11 D0 & &	8	1000
10	1100#####1###	D3 D2 d ~	8	0
11	1111#####1	D3	8	1000
12	1011#####1####	D0 ~ ~	8	1000
13	1011#####1####	D0 ~ ~	8	1000
14	1011#####0####	D15 D15 d	7	0
15	1010#####1####	D14 D1 r D1 r	7	1000

that map onto multiple data bits to provide the correct answer. It would not be an eloquent solution, but it would be a solution nevertheless. Therefore a classifier would have optimal and suboptimal rules mixed within the population.

Prior to the work for this thesis, any new code fragments that were learned while solving a problem could be reused by a CF in the terminal set in any subsequent problems. However, the functions were predefined and once a problem was finished it was not possible to adapt the function set.

## 2.6 Other Representations

It has been shown above that evolving general solutions using rich representations can increase scalability. This is an important characteristic which will be useful in the new work. Previously, other Boolean problems have been solved successfully by using techniques similar to this thesis work, see Chapter 6. In addition to CFs, a number of representations were developed previously. One of these representations was XCS with State-Machine Action (XCSSMA). XCSSMA was introduced with the capability to generate state machines to encapsulate repeating patterns. In other words, it discovered recurrences and loops in order to repeat useful behaviors. This was accomplished by replacing the numeric action in XCS with a Finite State Machine [46].

XCSSMA evolved compact and easily interpretable general solutions for the even-parity and the carry problem domains. The parity problem is considered difficult because parity is non-linear and non-monotonic [40]. The even-parity domain does not allow generalizations if the standard ternary alphabet is used with static numeric action. So each bit must be specific for a rule to be accurate [46]. Since this type of problem has been studied extensively, producing a general solution is a notable accomplishment [40]. Compact and general solutions are important for this work

because they increase scalability.

XCSSMA produced useful solutions to some boolean domain problems. However, it could not improve the generalization for the multiplexer (Mux) domain. This was because the state machines needed for this domain are more complex than the other domains mentioned above [46]. Furthermore, the multiplexer problem domain does not contain repeating patterns at the domain level. However, there are repeating patterns, loops, at the underlying problem description layer, but these were not accessible to XCSSMA. Therefore this technique was not chosen for this novel work.

Self-Modifying Cartesian Genetic Programming (SMCGP) is an alternative to LCSs for scaling. SMCGP is a developmental form of CGP that allows an individual program to modify itself using self-modifying functionality. SMCGP evolves a computer program that could generate an arbitrary sequence of computer programs, each of which solves a particular problem [49]. The drawback to this technique is that the solutions for large scale problems are difficult to interpret. Furthermore, as the problem scales, the size of the solution becomes very large; eventually learning will be intractable [46]. This limitation as well as the increasing difficulty in interpreting the solutions, precluded this technique from being chosen.

The multiplexer problem is a complex and difficult problem due to epistasis and its large search space at large scales. An early attempt at scaling was the S-XCS system that utilizes optimal populations of rules, these are learned in the same manner as classical XCS [41]. These optimal rules are then imported into S-XCS as messages thus enabling abstraction. The system uses human constructed functions such as Multiply, Divide, PowerOf, ValueAt, AddrOf, among others [41], see Table 2.2. Although these key functions provide the system with the scaffolding to piece together the necessary knowledge blocks, they have an inherent bias and might not be available to the system in large problem domains. For example, in the Boolean domain the Log and Multiplication functions do not exist. It also assumes completely accurate populations, whereas the pro-

posed system is required to learn both, the population and functionality, from a *tabula rasa*. The novel work will utilize similar functions but will learn them from basic functions provided by a human, e.g. Multiplication, Log. This will provide a shortcut to reduce the search space. But more importantly, CFs will facilitate the learning of the aforementioned functions in progressively richer stages. Meaning, that once the system has learned Multiplication, Addition, Subtraction and others, it could then attempt to learn the function that provides the address of the important bit in the problem. The reason is that the CFs would then contain all of the learned functionality acquired throughout the training stages. This is something that S-XCS is incapable of accomplishing [41].

If supervised learning is permitted (unlike in this work), the heterogeneous approach of ExSTraCS scales well, up to the 135 Bit Mux [87]. This technique uses a mechanism that is similar to memory. It is designed for Michigan style LCSs using supervised learning. During training a vector of accuracy scores is maintained for each of the training instances. After training, the scores are used to identify association regularities in the data-set. In addition, the mutation and crossover mechanisms are directed probabilistically while creating new rules. This rule generation is based on an instance's tracking scores [87]. This technique successfully linked individual instances in the data-set to etiologically heterogeneous subgroups by using attribute tracking. Also, it was found that attribute feedback significantly improves test accuracy, generalization, run time and the capability to discriminate between predictive and non-predictive attributes in the presence of heterogeneity. One drawback to this technique is that Michigan style LCSs do not typically scale well. Also, the results presented in [87] are computationally expensive. This means that large-scale analysis (using large numbers of attributes) is currently impractical. However, in its present form, this technique would be applicable to studies with a filtered set of factors [87]. The technique described above relates to this original work dealing with compaction algorithms; compaction techniques are

Table 2.2: A sample of classifiers from the final solution in S-XCS with optimal pupulation for the 3-bit, 6-bit, 135-bit and 1034-bit multiplexer problems [41].

Mux	Condition	Length
3	VALUEAT OR 2 2	4
3	VALUEAT AND 2 2	4
3	VALUEAT ADDROF 2 2	4
3	VALUEAT AND 2 POWEROF 1	5
3	VALUEAT OR POWEROF 2 2	5
3	VALUEAT OR POWEROF 1 2	5
6	VALUEAT ADDROF 4 5	4
6	VALUEAT ADDROF 5 4	4
6	VALUEAT ADDROF 4 POWEROF 5	5
6	VALUEAT ADDROF POWEROF 3 4	5
6	VALUEAT ADDROF POWEROF 5 4	5
135	VALUEAT ADDROF 128 134	4
135	VALUEAT ADDROF 134 128	4
135	VALUEAT ADDROF POWEROF 22 128	5
135	VALUEAT ADDROF 128 PLUS 133 134	6
1034	VALUEAT ADDROF 1033 1024	4
1034	VALUEAT ADDROF PLUS 1029 1029 1024	6
1034	VALUEAT ADDROF MULTIPLY 1025 324 1024	6
1034	VALUEAT ADDROF PLUS 1029 1024 1024	6
1034	VALUEAT ADDROF MULTIPLY 1029 324 1024	6
1034	VALUEAT ADDROF PLUS 1033 1033 1024	6



primarily used to reduce the number of classifiers based on a given criteria [92]. In particular, the online nature of the technique and the usage of a vector might be useful, see Chapter 5. However, this novel work relies on reinforcement learning and therefore ExSTraCS is not a viable candidate.

## 2.7 Layered Learning

Layered Learning (LL) is a hierarchical machine learning paradigm proposed in [83]. LL applies to tasks for which learning a direct mapping of inputs to outputs is intractable. The tasks are decomposed in a bottom-up manner into subtasks and there is separate learning at each subtask. The learning at each subtask directly facilitates the learning of the next higher layer. LL assumes that the appropriate aspects of the task to be learned are determined as a function of the specific domain. It does not include an automated hierarchical decomposition of the task. In this original work, this composition will be done by a human. Since each subtask may have specific characteristics, an appropriate algorithm is applied to learn each subtask layer [83].

LL has been used to learn several problems. Stone and Veloso used LL to decompose the activity of robotic soccer [83]. For example, the task of the agent retrieving a moving ball and deciding what to do with it was decomposed into: ball interception, pass evaluation, and pass selection. LL has also been applied to the evolution of goal scoring behavior in soccer players [9]. The authors compared their technique to standard GP programs with positive results. They determined that LL is on average able to evolve goal scoring behavior comparable to standard GP, more reliably and in a shorter time. Despite these advantages, the quality of solutions found by LL did not exceed those of standard GP. LL in this fashion requires a large amount of domain specific knowledge and programmer effort to engineer an appropriate layer [9]. Another usage of LL involved a study of the interactions between evolution, development and lifelong

layered learning [33]. The technique used a developmental tree-adjoining grammar guided GP. It was capable of solving GP problems that lie well beyond the scaling limits of standard GP. The solutions it found were simple, succinct, and highly structured [33].

Another implementation of LL was based on four different layers [77]. These layers were inspired by the biological hierarchy of the human nervous system. The two stage training was designed to learn bipedal walking skills. In this application of LL, the technique demonstrated several advantages compared to previous methods. It required a minimum convergence time. However the higher average training error presented a challenge [77]. In [8], the goal was to improve the generalization of GP for symbolic regression problems. In that approach, several data-sets, called primitive training sets, were derived from the original training data. The sequence of the generated sets was from less complex to more complex. However, the last data-set was still less complex than the original training set [8]. The results indicated that the technique can improve the performance of GP, both on the training and on the test data. Furthermore, it decreases overfitting, and finds solutions with less complexity [8]. The main limitation of the technique is that it was designed only for symbolic regression problems, therefore it can not be applied to classification problems, for example [8].

LL relates to this thesis work in several ways. One of the aims is to evolve a solution for the  $n$ -bit multiplexer problem. This is a difficult problem that can be partitioned into a number of sub-problems. Each sub-problem can then be learned in sequence while each subsequent sub-problem relies on the solutions discovered by the previous ones. The learned functionality is anticipated to be compact and maximally general. This is due to the power of expression present in CFs. In addition by using LL and CFs, the negative effects of bloating are anticipated to be attenuated.

## 2.8 Transfer Learning

Another approach which is helpful in learning more than one task is Transfer Learning (TL). The core idea of TL is that experience gained in learning to perform one task can help improve learning performance in a related, but different, task [66], [84]. In essence, transfer learning aims to extract the knowledge from one or more source tasks and apply the knowledge to the target task [27], [85]. According to [85], it is also reasonable to frame TL as one agent learning in a source task and a different agent learning in the target task, where transfer is used to share knowledge between the two agents. This is analogous to using LL combined with CFs, with a different agent learning each sub-problem. TL is multi-directional and less subject to human bias (compared to LL), as the learned information is gleaned from the problem domain by the agent. The success of the learned knowledge in the new domain is only determined by its usefulness in the new domain. The benefits of TL are actualized when learning from one domain and transferring the learned knowledge to a similar or related domain. The field of machine learning defines TL as transferring the underlying model [69].

In many real-world applications, the training and test data do not have the same distribution. For example, there may be a classification task in one domain of interest, but there may only be sufficient training data in another domain of interest. Furthermore, the latter data may be in a different feature space or follow a different data distribution. In such cases, knowledge transfer, if done successfully, would greatly improve the performance of learning by avoiding much expensive data-labeling efforts [69], [84]. Transfer learning is classified into three different settings: inductive transfer learning, transductive transfer learning, and unsupervised transfer learning [69]. Most previous works focused on the first two settings. Unsupervised transfer learning may attract more and more attention in the future [69]. Furthermore, each of the approaches to trans-

fer learning can be classified into four contexts based on what to transfer in learning. They include the instance-transfer approach, the feature-representation-transfer approach, the parameter-transfer approach, and the relational-knowledge-transfer approach, respectively [69].

Transfer Learning (TL) is related to this proposed work due to its strengths in advocating knowledge reuse. In addition, one of the aims is to evolve a general solution for the  $n$ -bit multiplexer problem. This type of problem has a nature which precludes an EC system from learning it in one step, see Chapter 6. Therefore, TL forms an integral part of the triumvirate used in this novel work which consists of TL, TL and CFs. Since the target task must be partitioned into sub-problems by a human, this means that TL alone would be ineffective in learning the problem.

## 2.9 Chapter Summary

Evolutionary Computation is a paradigm, which is suited to large scale search, therefore it forms a component of this new work. Genetic programming provides flexibility and power of expression, which should enable it to extract reusable building blocks of knowledge. The disadvantage to GP is that it suffers from bloating and has a large search space.

Learning Classifier Systems (LCSs) are techniques based on evolutionary principles whose goal is to produce an optimal population of classifiers. Together, these classifiers, or rules, form a solution to the problem. LCSs come in two major varieties, Michigan and Pittsburgh. The novel work presented in this thesis is based on XCS, a Michigan style LCS. XCS can be used to learn online tasks, something that Pittsburgh styled LCSs do not perform normally. XCS also learns a complete state-action map, meaning that it produces rules that are always accurate and correct, as well as rules that are always accurate and incorrect. In addition, XCS evolves rules that are maximally general, which is important because this increases scalability.

The rules that are produced by an LCS have two main parts, the condition and action. The original alphabet used to represent the condition of the classifiers consisted of  $\{0, 1, \#\}$  while the action consisted of  $\{0, 1\}$ . Although these alphabets are simple and quick to use, they lack the expressivity and flexibility required to represent complex problem domains. This indicates that the representation is very important for LCSs.

Several representation schemes have been explored in order to improve generalization and to cover the problem space better and with more expressivity. Each of these representations has strengths and weaknesses. CFs have introduced increased scalability and expressivity to the XCS technique. This has enabled the solution to previously intractable problems, but as the problem scaled, all the techniques stopped learning at some point. This thesis investigates various methods for improving scalability and learning in XCS-based systems extended with CFs.

Layered Learning is promising because it has been used successfully in various settings. For example, a difficult problem was decomposed into 4 different layers in [77]. The aim of this work was to learn bipedal walking skills. Another successful implementation of LL was [83]. The task of an agent retrieving a moving ball was decomposed into several subtasks. This means that at least one of the aims of this thesis work could benefit from the advantages of LL.



# Chapter 3

## Research Methodology

### 3.1 Research Methodology

The proposed research methodology adopted to investigate the thesis will be presented in this section.

The overall goal of this thesis is to improve the scalability and re-usability of learned knowledge in Learning Classifier Systems (LCSs). In order to achieve these goals, the following three research methodologies are adopted: (1) identify and extract useful functionality from simpler problems in a domain and then use that learned information at the nodes of CFs, (2) convert rules – formed with a rich alphabet – that are effective at identifying complex patterns into equivalent rules – formed with a simple alphabet – that are efficient to process, (3) identify underlying functionality and knowledge by using Layered Learning and human intervention to separate a complex problem into constituent sub-problems, in order to evolve a general solution.

In addition to improving the scalability and re-usability of learned knowledge in LCSs, this thesis also aims at understanding the inner workings of function re-usability and combination by fully tracing the construction of solutions. To illustrate this method, the solutions to a multiplexer problem will be evaluated manually, as autonomous enumeration is im-

practical.

The rest of this section briefly describes the systems to be designed and implemented in this novel work following the above mentioned research methodologies. The details of each implemented system will be provided in separate contribution chapters, from Chapter 4 to Chapter 6.

### 3.1.1 Overview of Methods

The proposed systems extend Wilson's XCS described in [22] and in [95], which is an online accuracy-based LCS model. This system has been successfully extended with CFs in previous work and therefore presents itself as a viable candidate for this original work [49]. XCS is a good basis for this work because the classifier fitness is based on the accuracy of a classifier's payoff prediction instead of the prediction itself. This prevents overly strong classifiers from overtaking the population and edging out less strong ones which could be very useful in small niches. Also, the GA takes place in niches instead of panmictically. These aspects lead XCS to evolve maximally general classifiers. In addition, XCS uses reinforcement learning, which emphasizes the formation of complete mappings [95]; XCS produces a complete state-action map; the system evolves rules that are always correct and rules that are always incorrect [22], [95]. Last, the methods could be converted to supervised learning in a straightforward way [88].

The developed systems will be tested with Boolean problems such as the multiplexer and hidden multiplexer. These problems have been shown to contain epistasis and are highly non-linear. These characteristics make them very difficult to learn by standard algorithms. The multiplexer has been studied extensively and therefore is the subject of a large body of knowledge. Other types of problems such as real numbers could also be studied with the proposed technique. CFs together with LL and TL make it possible to represent complex domains in a scalable manner.



With enough time, it would just be a matter of identifying the correct sub-problems and their training sequence.

The criteria used to measure performance will be the rate of convergence over the number of training instances. The convergence measurement will appear as the Y-axis on the results plots, while the time will appear as the X-axis. This is a standard method for measuring LCS performance and therefore is a valid choice for this work.

#### **3.1.1.1 Reuse of Learned Functionality from Smaller Problems in the Nodes of CFs**

In the first proposed system, the typically used CFs will be modified to reuse learned information at the root nodes, as well as the leaf nodes, see Chapter 4. The work will extend the XCSCFC system, which uses CFs in the condition part, instead of the customary ternary alphabet  $\{0, 1, \#\}$ . XCSCFC was shown to increase scalability by reusing rich building blocks of knowledge, which is not possible in the standard XCS [49]. For these reasons it will be a starting point for the proposed work on developing reusable functionality.

The CF sub-trees will initially be considered as per the usual formulation of CFs, see Figure 4.1. In the novel work, they will be extended to also include learned rule-sets in the nodes. This departs from the standard CF definition that only includes previously learned CFs at the leaf nodes and hard-coded functions at the root nodes. The learned rule-sets will constitute functions in that they will be used to evaluate the connected terminals and nodes during the learning phase, e.g. matching and covering operations, instead of hard-coded functions.

### 3.1.1.2 Identify a compaction technique to simplify the final population of CF rules

XCS often utilizes a simple, ternary alphabet that is easy and quick to compute. However this simplicity means that difficult problem domains are not addressed very well. The condition bits are dependent on their position and there is a lack of expressivity, such that multiple rules are needed to describe a single niche in the domain. CFs have been shown to provide a richer representation as well as location independence, such that a single rule can describe a niche, which is often faster to evolve. Also, CFs are independent of the problem length, which reduces the search space together with training times, e.g. the 6-bit multiplexer could be solved with just three CFs. However, this power of expression is difficult to compute and is resource intensive. For these reasons, a new compaction technique will be developed. There will be an offline version that will be post-processed and executed once the final population has been evolved. There will also be an online version that will be mostly in-process. This technique will translate the rich CF-based rules into a ternary form. These will be used to evaluate the root nodes during the matching and covering phases.

It is envisioned that the proposed compaction technique will reduce the time requirements for evaluating CFs. This will be accomplished by reducing the search space and using the distilled-rulesets in place of learned functions<sup>1</sup>. The technique is anticipated to be effective because along with the new ternary rules to be created, the learned CF-based rules will also be available for feature construction during the covering phase.

---

<sup>1</sup>Note that functions contain CFs, which contain other CFs and functions, which can be time consuming to unravel.

### 3.1.1.3 Show that Layered Learning can provide benefits in scaling for a CF based LCS by solving an intractable problem

In addition to developing the above mentioned scalable classifier systems, this new work will introduce and implement a method to learn the underlying patterns in a domain, such that any problem in it can be solved. The problem will be partitioned into constituent sub-problems by a human. The new system will be trained with these sub-problems. Each solution will be used by the next sub-problem until the system has learned general rules comprising all the functionality in the sub-problems.

The system will be guided throughout the training via the sub-problems. This is a prominent characteristic of Layered Learning, which also includes the bottom-up construction of solutions. The new work will represent a novel combination, compared to the standard EC techniques.

This work will shift the normal emphasis of standard EC techniques by leveraging the benefits of LL. Therefore the EC researcher will not be required to specify the nature of the domain or define the algorithm's parameter values, terminal set and function set. Instead, the method will specify the order of the problems or domains. The system will be allowed to automatically adjust the terminal set through feature construction and selection, and thereby develop the function set. In this manner, the technique will differ from the self-contained stages of LL and will be closer to TL.

The overlap between the new technique and TL will consist of how the CFs will be reused. For example, part-solutions (CFs) will be available for propagation between sub-problems. Similarly, complete solutions — learned functions — could form part-solutions for future problems. Since the system will have the capability to use learned rule-sets as functions, along with the associated building blocks, i.e. CFs, that capture any associated patterns, this will be an advantage over pre-specifying functionality in EC and LL.

This method changes the fundamental problem from finding an over-

arching ‘single’ solution that covers all instances or features of a problem to finding the structure (links) of sub-problems that construct the overall solution. Learning the underlying patterns that describe the domain is anticipated to be more compact and reusable as they do not grow as the domain scales (unlike individual solutions that can grow impractically large as the problem grows, e.g. DNF solutions to the multiplexer problem).

It is anticipated that once the training is complete, the system will have all the functionality necessary to solve any  $n$ -bit multiplexer problem where  $n$  refers to the length of the problem. Once the system has learned the fundamental sub-problems, the rules will be used on the 6-bit multiplexer problem. Subsequently, the rules learned by the system during the 6-bit multiplexer will be tested on progressively more difficult problems. These will include the 264-bit through the 8205-bit multiplexer problems. The astronomically large sample space of the more complex problems will preclude any solutions relying on enumeration and therefore will be sufficient for determining the scalability and generality of the rules produced. In addition, the test problems will consist of only the testing phase, meaning that there will be no need for further training.

## 3.2 Experimental Design

This section provides details of the problem domains explored here, as well as the experimental setup used for the experiments. This section also clarifies certain details concerning the size of the training data-sets related to the problem sample space. The evaluation methods will be the number of training instances required to converge fully. This is a standard criteria used in LCS experimentation and therefore will be appropriate for this work.

### 3.2.1 The Problem Domains

This original thesis work focused on Boolean problems because they have a measurable search space and also contain identifiable building blocks. This domain also lends itself to analysis and interpretation. Other domains such as integer or real were not chosen for this new work because the boolean domain problems are sufficient to demonstrate the benefits of the new technique. However, the technique could be extended to address other domains. In addition, some of the new techniques address domains other than Boolean, see Chapter 6.

The following Boolean problem domains will be used in the experimentation: the multiplexer, the hidden multiplexer, Parity, NAND, AND, OR, XOR, NOR. The following sub-problems are composed of other domains such as real and integers: Kbits, KbitString, Bin2Int, AddressOf, ValueAt, see Chapter 6. The techniques developed as part of this thesis will not be fitted to the multiplexer problem domain; they will be transferable – with additional work – to alternative domains.

#### 3.2.1.1 The Multiplexer Problem Domain

A multiplexer can be thought of as a logic circuit where a particular number of bits provide the address of the output bit. Therefore the address bits are intrinsically bound to the structure in that they determine the importance of the corresponding data bit [45], [93]. The number of address bits, being a function of the length of the multiplexer, grows as the length of the input string scales to larger problems. Assuming  $L$  is the length of the input, then the equation:

$$L = k + 2^k \tag{3.1}$$

encapsulates the relationship between the length of the input and the number of address bits required (length  $k$ ) [50].

The search space of the multiplexer is considered adequate to demonstrate the benefits of this new work. For example, the search space of the 70-bit multiplexer contains  $2^{70}$  possible combinations [54]. Therefore this domain presents opportunities for a seamless construction of learned functionality via Boolean operators.

An example of a multiplexer is shown in Figure 3.1. Here the two address bits, A0 and A1, translate to 1 in real form. The real number points to the data bit D1 that contains the value to be returned. The index begins at D0 and proceeds from the left towards the right, as shown in Figure 3.1 <sup>2</sup>.

Condition						: Action
0	1	1	1	0	0	: 1
A0	A1	D0	D1	D2	D3	
Index:						
0	1	2	3	4	5	

Figure 3.1: 6-bit Multiplexer showing the address bits and the data bits of the condition.

The multiplexer is a difficult and interesting problem because it has epistasis and is highly non-linear. These characteristics are expressed by the Equation 3.1. Also, this problem scales as  $k$  increases. The larger problems are intractable using enumeration.

### 3.2.1.2 The Hidden Multiplexer Problem Domain

The Hidden Multiplexer is a two tier problem involving a lower layer composed (in this case) of 3-bit parity problems. The lower level is evaluated

<sup>2</sup>Note that the distinction between address bits and data bits is not provided to the LCS tasked with learning the input to output relationship.

first, in order for the results to be used by the top layer, which is composed of one multiplexer problem, see Figure 3.2. There is a fixed correspondence, the address bits are irrelevant as they are dependent on the evaluation of the lower level parity problems. Once all the upper level bits are known, the multiplexer part is evaluated and the class is determined. The Hidden Multiplexer provides a difficult testing scheme for the technique due to low sparsity [17]. The problem also provides the LCS with the opportunity of learning how to combine learned knowledge blocks to effectively solve both layers.

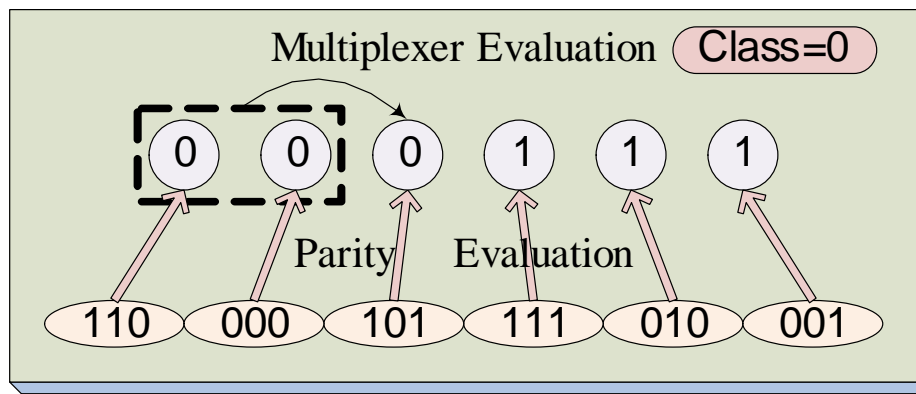


Figure 3.2: Example of the Hidden Multiplexer. The lower level consists of parity problems and the upper level consists of a multiplexer problem. The parity problems are solved first, then the multiplexer.

The hidden multiplexer problem is suitable for this work because it exhibits difficulty due to its hierarchical nature. The systems attempting this type of problem were faced with a lower level composed of Parity problems and an upper level composed of a multiplexer problem [17]. This was anticipated to pose an interesting challenge due to its multi-level nature and it requires effective building block processing. For example, The 37-bit multiplexer has seven specified bits per rule, the 3x6 (18)-bit hidden multiplexer (3-parity and 6-multiplexer) has nine specified bits, therefore proportionally there are more specified bits to discover in the hidden mul-

tiplexer. The LCS does not interact with its hierarchical nature in standard systems. This means that a classifier must identify the parity building blocks and then organize them in a manner leading to a solution of the overall problem.

As an added topic of interest, both 2-bit Parity and 3-bit Parity problems were included in the training regimen. That is to say, after the Boolean operators were learned, the proposed system, as well as XCSCFC, were both trained with 2-bit Parity and 3-bit Parity problems. The motivation behind this was to determine if exposing the system to a portion of the problem it would ultimately face was conducive to better learning, i.e. leading to better building blocks for the more difficult problems, i.e. the 3x11 hidden multiplexer. This is plausible as in previous work, it was determined that including the Parity problems in the training of a CF-based system provides better performance for the 3x6 hidden multiplexer problem [4].

Another task that was to be included in the training sequence was the 3-bit multiplexer. Again, the motivation was to determine if performance increased, i.e. number of correct classifications *vis a vis* the number of training instances required, when the system's training schedule included functionality for lower scale than the primary problem composition. The next problem is to include the 6-bit multiplexer; this will test whether including a problem at the same level as part of the hidden multiplexer improves performance.

### 3.2.2 Experimental Setup

Unless stated otherwise, the following parameter values, commonly found in the literature, are used for the experiments here, as suggested by Butz and Wilson in [22]. These values are well tested and will serve as a good starting point for trials: learning rate  $\beta = 0.2$ ; fitness fall-off rate  $\alpha = 0.1$ ; fitness exponent  $\nu = 5$ ; prediction error threshold  $\epsilon_0 = 10$ ; thresh-



old for GA application in the action set  $\theta_{GA} = 25$ ; mutation probability  $\mu = 0.04$ ; two-point crossover with probability  $\chi = 0.8$ ; experience threshold for classifier deletion  $\theta_{del} = 20$ ; fraction of mean fitness for deletion  $\delta = 0.1$ ; classifier experience threshold for subsumption  $\theta_{sub} = 20$ ; probability of don'tCare symbol in covering  $P_{don'tCare} = 0.33$ ; reduction of the prediction error  $predictionErrorReduction = 0.25$ ; reduction of the fitness  $fitnessReduction = 0.1$ ; and the selection method is tournament selection with a tournament size ratio 0.4. Both GA subsumption and action set subsumption are activated. This is good because it helps to evolve general classifiers, however it also increases processing time. Explore and exploit problem instances are alternated. The reward scheme used is 1000 for a correct classification and 0 otherwise<sup>3</sup>.

All the results obtained in this new work are the average of 30 independent runs, each with an independent seed; this will minimize the chance that the same series of random numbers are used in the experiments. In graphs presented, the X-axis is the number of problem instances used as training examples and the Y-axis is the classification performance measured as the moving average over the last 1000 test problem instances.

### 3.2.3 Test Data-set Size

An LCS evolves a population of classifier rules to generate a model for the problem. This is to be achieved by randomly choosing problem instances from the whole data-set during the learning/training process. This is different from standard machine learning and EC practice where a training and unseen test set are created. This tests for overfitting in order to provide an estimate of performance when implemented in real-life. In learning small data-sets, using an LCS approach, the generated model is only tested against the known optimum solution and no conclusions from testing performance can be made for unseen data. In large data-sets, e.g.

---

<sup>3</sup>Note  $P_{don'tCare}$  and population size are critical as a problem scales, see Chapter 6

MUX greater than 70, the number of training instances is (much) less than the number of problem instances, so generalization is also tested. In some problems, which have an astronomically huge sample space, the rules produced by the techniques cannot be fully tested by using all the possible instances from the unseen data-set, as enumeration is intractable. In these cases, a test data-set of 1 000 000 instances is small but it is large enough to expose many deficiencies.

In the sub-problems of the work dealing with human-inspired scaling, see Chapter 6, the training set is much smaller than the problem sample space. For example, the Kbits sub-problem has a sample space of 11 training instances. However, sufficient functionality is available – e.g. log, addition, subtraction, floor, ceiling and others – for the LCS to evolve a population of classifiers that covers the problem.

### 3.3 Chapter Summary

The overall goal of this thesis is to improve the scalability and reusability of knowledge blocks in Learning Classifier Systems (LCSs). To achieve this goal, three research methodologies are adopted: (1) identify and extract useful functionality from simpler problems in a domain and then use that learned information at the nodes of CFs, (2) convert rules that are effective at identifying complex patterns into equivalent rules that are efficient to process, (3) identify underlying functionality and knowledge by using Layered Learning and human intervention to separate a complex problem into constituent sub-problems, in order to evolve a general solution.

Additionally, this chapter provides details of the problem domains explored here and the experimental setup used for the experiments. This chapter also clarifies certain details concerning the size of the training data-sets related to the problem sample space. The evaluation methods will be the number of training instances required to converge fully. This is a standard criteria used in LCS experimentation and therefore will be

appropriate for this work.



# Chapter 4

## Reusing Learned Functionality

### 4.1 Introduction

This chapter introduces novel work where standard CFs are expanded. Originally CFs reused learned information at the leaf nodes only. This new work expands CFs to reuse learned knowledge at the root nodes as well as the leaf nodes. The implementation consists of CFs combined with Learning Classifier Systems (LCSs). LCSs are powerful classification techniques, due to their variety, e.g. XCS, and flexibility, e.g. representation through different alphabets. LCSs are usually formulated as an input (conditions) providing an outcome (action). Originally, the conditions were expressed using a ternary alphabet, i.e.  $\{0, 1, \#\}$ : the hash mark could take the value of 0 or 1. The action part used a binary alphabet, i.e.  $\{0, 1\}$ . These alphabets, although simple and efficient to compute, posed limitations on the type of problems that could be addressed, as well as on the richness in expression. Many additions to the LCS technique have been developed since their inception, one of the most prominent ones being the eXtended Classifier System (backronym) (XCS).

The power of LCSs was limited up until 2011 because they discarded information as the problem scaled. Code Fragments (CFs) were developed to address this limitation and to increase the expressivity of this technique.

CFs are GP-like sub-trees that have a rich and flexible representation of environmental patterns. CFs can use many types of functions in their root nodes and usually take environment features or other CFs in their terminal nodes. This makes CFs capable of addressing many types of problems in very complex domains. Code Fragments combined with XCS (XCSCFC), have been demonstrated to be scalable by solving the 135 bit multiplexer problem, something that until then had been considered intractable [49]. Although XCSCFC fulfilled its potential for increased scalability with an expansive power of expression, it suffers from an integral weakness. Since the CF sub-trees reuse other CFs at the leaf nodes, the chains of CFs grow to intractable lengths. At some point, depending on the problem, the system stops learning.

CFs and their reuse at leaf nodes helped but still would reach a limit. The reason behind this is that as the CFs grew, they eventually led to very long chains of CFs. To overcome XCSCFCs propensity to grow large CF chains, it was hypothesized that using CF information at the root nodes as well as at the leaf nodes, could provide increased scalability. It was also desired to capture the underlying patterns through learned functionality and skills. For these reasons, a new system based on XCS and CFs will be developed. The technique, known as *XCSCF<sup>2</sup>*, will be utilized later to solve a number of problems in the Boolean domain, see Section 4.2.1. The technique will capitalize on the benefits of reusing learned knowledge at the root nodes. It will also show that knowledge learned from simple domains can then be transferred to a more complex domain, thus showcasing increased scalability. Past work has shown that learning from simple domains can lead to the solution of more complex problems in the same or a related domain [49].

The hypothesis is that reusing learned functionality at the inner nodes can provide increased scalability. This is because the root nodes will have a tight linking between the function and the CFs associated with that function, i.e. inner node. This is anticipated to decrease the search space,

thus providing better scalability. Previously, only terminals could be replaced by the constructed code fragments. This was a design choice, as the building blocks of knowledge were at the terminal level. This being where the discovery of useful combinations of features and constants took place. This is analogous to feature construction using GP trees. The reason that code fragments were used only at the terminals in XCSCFC is because CFs can accept any number of arbitrary inputs, where a function takes in a set number of inputs. As the chains of CFs grow, the number of leaf nodes present also grows. This means that if one were to swap a function with a code fragment it might involve dissimilar objects. This is because different functions could have different number of inputs.

#### 4.1.1 Chapter Goals

A genetic programming approach may have many pre-programmed functions. But these are unrelated to each other and to useful building blocks of knowledge. In *XCSCF*<sup>2</sup>, it is intended that functions and building blocks, i.e. CFs, will be linked in order to guide genetic operations for improved search. Coupling CFs and learned functions together is anticipated to improve learning over just supplying user defined functions without previous useful input knowledge. The aim of the work presented in this chapter is to create methods to enable rule-sets as functions, achievable through the following methods.

- \* **Reuse** functions at the inner nodes. Function reuse at the inner nodes is important because by reusing the learned rule-sets as functions (termed Function-RuleSets) and their associated code fragments, it is possible to provide a tight linking between the two. This will aid in the performance of the system as well as provide valuable and reusable building blocks of learned knowledge. The analogy is as follows:

$$'If < Conditions > Then < Actions >' \quad (4.1)$$

$$'If < Input > Then < Output >' \quad (4.2)$$

$$Function(Arguments < Input > Return < Output >) \quad (4.3)$$

Equation 4.1 is the standard way that a classifier would process its conditions to achieve an action, which is analogous to 4.2. Equation 4.3 is the analogy with a function. These functions will take a number of arguments as their input and will return an output.

- \* **Reduce** the search space. This is anticipated to be achieved by linking the new CFs with the new rule-sets learned. This will be shown with performance graphs, which will depict the number of training instances required to solve a series of problems. The usage of graphs is a suitable measure as it is a standard method for gauging LCS performance. The final output will provide a composite function which at the time of evaluation will have to search only among those CFs that have been assigned to it. This 'shortcut' is anticipated to produce an increase in scalability and should facilitate the solution to problems in related domains [3].

According to [91], there are a finite number of common patterns in the world. In addition, natural, human and artificial systems tend to fall into these repeating patterns. If a pattern can be recognized in one system along with the ability to solve the problems related to that system or domain, it should be possible to reuse those techniques in a related domain.

The technique is explained fully in the following sections. Its applications will also be described. Afterward, the results from these experiments will be presented and a discussion will be conducted describing the implications of the findings.



## 4.2 Reusing Rule-sets as Functions

### 4.2.1 Method

The new technique proposed here builds on the strengths of XCSCFC and improves on them by expanding the reuse of CFs. Specifically, rule-sets that contain classifiers containing CFs can be reused as functions in the inner nodes of higher level CFs. For instance, Figure 4.1 shows that the leaf nodes can contain learned CFs as well as environment features. In addition, the root nodes of the CF sub-tree contain learned functions (rule-sets). This combination provides a link between the CFs and the new ternary rules. In this case the function rule-sets utilize the ternary alphabet  $\{1, 0, \#\}$ . In Figure 4.1 the function rule-set maps to the AND Boolean operator which would have been learned previously. When the next task is presented to the system, the learned Function Rule-set could be utilized to construct new CFs as part of the covering process. This facilitates learning of a series of functions that could then participate in learning a more difficult problem in a related domain.

Boolean operators have been found useful in learning problems and can be learned in series. For these reasons the proposed technique, known as XCSCF<sup>2</sup>, will initially be presented with several Boolean tasks. Figure 4.2 illustrates the step by step methodology. The series of objects with the letters "FS" inside signify the Function Sets currently learned; the circles at the bottom of Figure 4.2 illustrate the necessary rule-sets that will need to form as the function is being learned. The four sets of axes show the expected performance by the system while learning the current problem. The Y axes represent the percentage of the problem learned by the system, e.g. level of convergence, while the X axes depict the amount of instances required for the system to learn the task.

The training problems used by the novel system are part of the Boolean domain. The system is initially given the NAND Boolean operator rules. With these rules, it is then trained with the OR Boolean operator. The

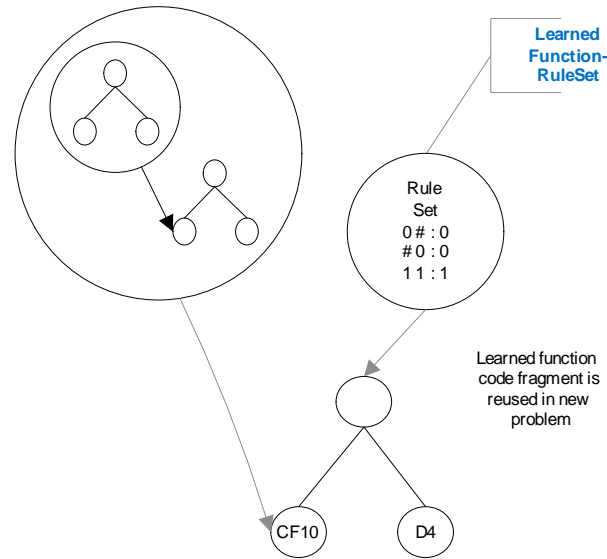


Figure 4.1: Code Fragment and Function Rule-set reuse. The root node illustrates the ternary rules of the 'AND' function that have been learned previously. These are then reused when learning a new problem.

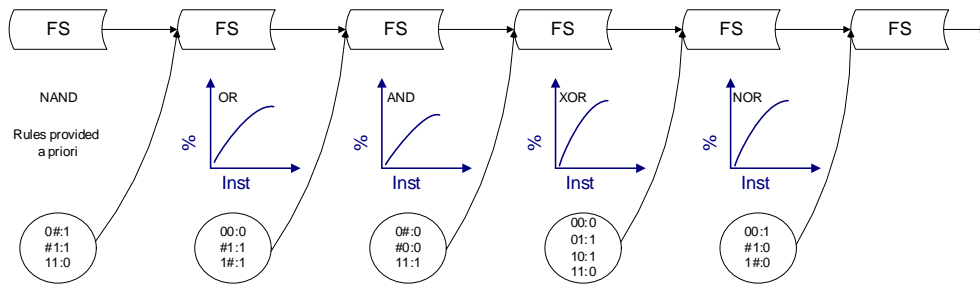


Figure 4.2: Code Fragment and Function Rule-set reuse. Each learned problem feeds its complement of learned functionality to the next problem.

NAND operator is deemed essential because with it, other Boolean operators such as OR, AND or XOR could also be learned [86], [68]. The concept is that as the system learns each function, it accumulates the essential rules and CFs that will solve that problem. For instance, after the system learns the OR function it tackles AND, XOR and finally NOR. With the exception

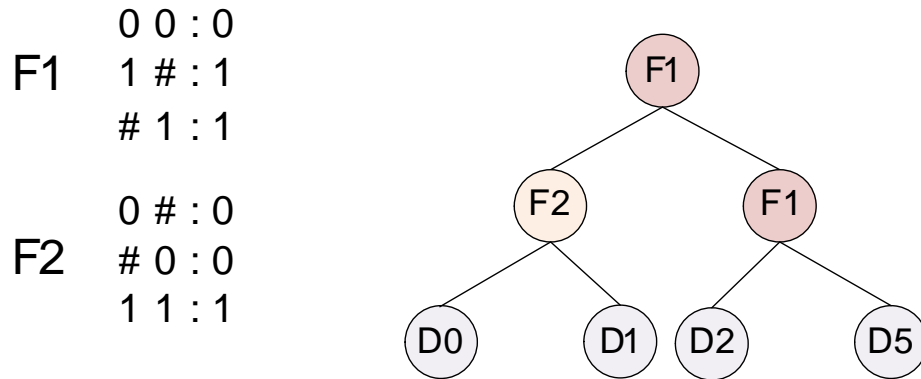
of the NAND function, the sequence of Boolean operators is not important since it is possible to learn all of them as long as NAND is present. Once this basis of learned functionality is in place, it can then be used to learn useful patterns in the multiplexer problem domain. The system is not given more Boolean functions to learn due to its intractable memory requirements. Importantly, the system does not reuse the learned CFs at the leaf nodes until it begins learning the 11-bit multiplexer problem. This was a design decision taken to minimize the memory requirements. This was due to the growing chains of CFs. The stopping criteria is when the system has processed the given number of training problems. The next step is to determine if the learned knowledge can be transferred to a more difficult problem in the same domain and possibly into a related domain such as the multiplexer problem.

The multiplexer problem is suitable for these experiments because it is a difficult problem which has been studied extensively in research [17]. It is highly non-linear in the sense that some of the bits are crucial in determining the importance of other bits in the string. The number of address bits is a function of the length of the message string and grows as the problem size increases. This means that the search space of the problem is also adequate to show the benefits of the proposed technique. For example, for the 70-bit multiplexer the sample space consists of  $2^{70}$  combinations [54].

#### 4.2.1.1 Evaluation of CF Sub-trees

In order to evaluate a CF tree, the system uses the Reverse Polish Notation method to traverse the tree. For example, the CF depicted in Figure 4.3, when presented with the environment message: 1 0 1 1 0 1, is then evaluated in the steps illustrated in Table 4.1. Step 1 begins by mapping the environment features to the corresponding terminal, the right hand side column shows this. Then the function is evaluated by comparing the inputs with the rules associated with the function. When a matching rule is found, the action of that rule is returned, see Figure 4.4. Step 2

evaluates the right hand side branch as shown in the middle column, with the right hand side column showing the mapped features. The root node produces the final answer for the CF tree by using the results from the two branches along with the function in the root node. When the top root node is evaluated – step 3 – the learned rule-sets for the function at that node are compared with the inputs produced by the two branches. When a matching rule is found, the action is returned as the overall answer for the CF tree.



## F1 & F2 : Learned functions

Figure 4.3: An example of a code fragment.

This evaluation process is further illustrated in Algorithm 4. Here  $n$  is the length of the CF,  $cf$  in a classifier rule. The CF is evaluated, as in the methods used for Reverse Polish Notation. If a CF evaluates to a number which falls within the length of the problem, then the corresponding environment feature is returned. In the case of a previously learned CF, the *evaluateCF* function is called recursively. On the other hand, if a CF matches a previously learned function, e.g AND, OR, then there is a comparison between the current inputs and the rules linked to that function. Depending on the number of inputs of the learned function, the same number of inputs are expected from the CF currently being evaluated. The

Table 4.1: Evaluation of a Code Fragment. Reverse Polish Notation is used to traverse the left branch, then the right branch and finally the top root node of the CF in Figure 4.3. The right hand side column shows the environment features mapped to the corresponding CF terminals.

Step	Original	Mapped Features
1	D0 D1 F2	1 0 F2
2	D2 D5 F1	1 1 F1
3	0 1 F1	1

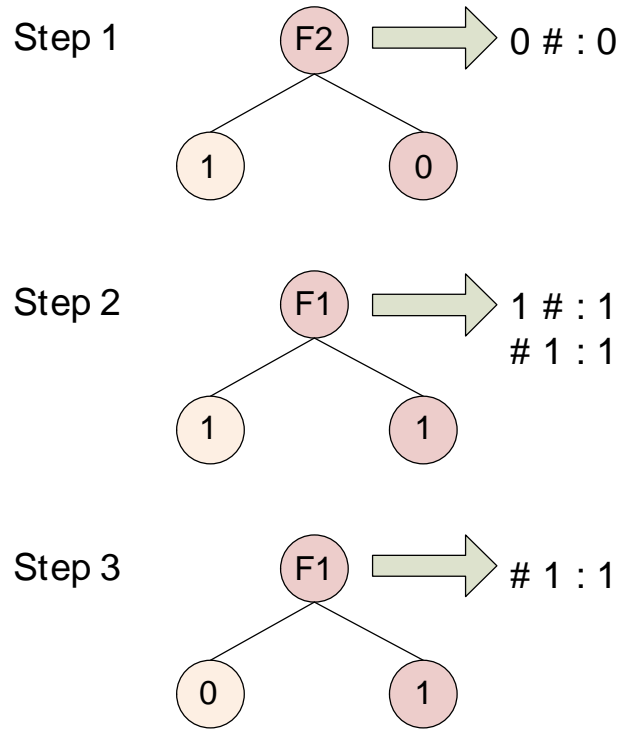


Figure 4.4: Rules matching at the root nodes.

CF creation process ensures that new CFs are allotted the correct number of inputs, thereby avoiding mismatches. These inputs are compared with each of the ternary rules linked to the learned function until a match is

found. At this stage of development, there will always be a match because the functions being learned are simple; they contain at most four rules each. It is plausible that as the functions increase in complexity there will be mismatches due to the quality of the ternary rules learned. This will need to be handled in future systems. When the match occurs, the action linked to the matching rule is returned. When the entire CF has been evaluated, the final value in the calculations is the result of the CF.

#### 4.2.1.2 Comparison with XCSCFC

It is important to tell apart the proposed system from XCSCFC, on which it is based, in order to understand the novel technique better. XCSCF<sup>2</sup> is based on XCSCFC and as such, shares many properties with this type of system. First and foremost, the proposed system uses CFs in the condition just like XCSCFC. The CF trees are built similarly, with the root nodes containing functions and the leaf nodes containing either an environment feature or a previously learned CF. Both systems also introduce a don'tCare CF, a regular CF, or an environmental feature for each of the condition positions in the problem. A don'tCare CF is one that always returns a one. Importantly, both systems can have a CF count that is equal to or less than the length of the problem. For example, a problem of length 11 could be solved by less than 11 CFs.

In addition, just like XCSCFC, the new system creates a matchset by including classifiers where all the conditions, i.e. CFs, produce a one. This means that all CFs created during covering, match the associated condition bit. Determining if two classifiers are equal is also accomplished like in XCSCFC, CFs are compared node by node; if all the nodes are the same, then the CFs are equal to each other. This is important when determining copies of CFs during the discovery phase. The equality between CFs is not altogether exact, because unlike XCSCFC, the proposed technique uses rule-sets for functions. This means that one CF with a root node containing Function  $x$  could be different from another CF which also con-

---

**Algorithm 4:** XCSCF<sup>2</sup>: evaluateCF

---

**Data:** The currently observed input state  $s$  and a CF  $cf \in [c]$  where  $[c]$  is the set of conditions in a classifier.

**Result:** When the top root node of the CF tree is evaluated, the final answer of the CF is produced.

```

1   $n \leftarrow$  Length of Code Fragment cf
2  for  $i = 1$  to  $n$  do
3       $code \leftarrow$  code indexed at  $cf[i]$ 
4      if  $code \geq 0 \ \&\& \ code < condLength$  then
5          /* Environment feature */
6          return  $s[code]$ 
7      else if  $code \geq condLength$  then
8          /* Previously learned CF */
9          /* Call evaluateCF function recursively */
10         return evaluateCF( $code, s$ )
11     else
12         /* Previously learned Function */
13          $x \leftarrow$  Number of learned functions
14         for  $idx = 1$  to  $x$  do
15             if  $code \equiv Function$  then
16                  $rNum \leftarrow$  Number of rules in function
17                 for  $rdx = 1$  to  $rNum$  do
18                     if  $CFrule \equiv Function \ rule$  then
19                         /* Function rule match */
20                         return Function rule action
21                 end
22         end
23 end
24 return true

```

---

tains Function  $x$  at the exact same position. The first function might never match the same rule number as the second instance of the same function. In this manner, determining if CFs are equal is only done at the Function level, otherwise it could become very time-consuming to determine if one Function is the exact copy of another. In addition, it would not serve any benefit to allow this granularity, because the matching rules are dependent on the inputs being passed up the CF sub-tree. For these reasons, XCSCF<sup>2</sup> only tests at the function level and therefore is not performing a complete comparison.

While both systems use CFs in the condition part and a binary alphabet in the action part, it is in the structure of the CF sub-trees where the main difference is found. Figure 4.5 illustrates how XCSCFC reuses either learned knowledge or problem features in the leaf nodes. The functions provided for the system are predefined, which means that they include some bias from the environment.

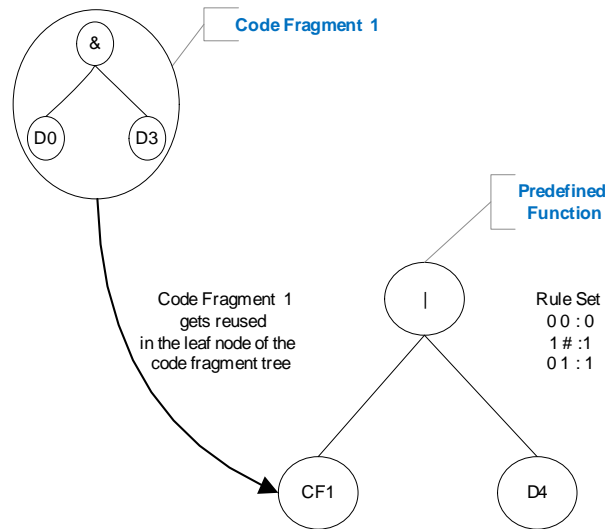


Figure 4.5: Original design of the Code Fragments-based XCSCFC. Code Fragment 1 uses the hard-coded AND function denoted by the &, while D0, D3 and D4 address features in the environment message.



The structural differences between both systems lead to different executions during the evaluation of CFs. As was described above, XCSCF<sup>2</sup> contains learned rule-sets at the root nodes. These learned rule-sets perform the role of atomic values, designed to shorten the search space. The NAND boolean rules are provided for the new system *a priori*, therefore they constitute a bootstrap. When XCSCF<sup>2</sup> is learning the first function, in this case OR, a new function is created with all the corresponding rules for the NAND operator and their corresponding actions. This new NAND function is then used to learn the OR Boolean operator. When learning subsequent Boolean functions, the OR and NAND functions will be available for those tasks. All the learned functions in the new technique are assigned a unique tag by the system described here. For example, in Table 4.2 the proposed system reuses previously learned functions; these have been tagged by the system as ‘M’, ‘N’, ‘c’, and ‘m’, while XCSCFC uses its hard-coded functions.

Table 4.2: Sample CFs produced by XCSCF<sup>2</sup> and XCSCFC for the 20 bit Mux problem. The tags  $N$ ,  $M$ ,  $c$ ,  $m$ ,  $r$  and  $d$  stand for previously learned functions. The tags CF\_55, CF\_44, CF\_56, CF\_25, CF\_31, CF\_28, CF\_47 and CF\_36 stand for CFs associated with the previously learned functions.

CF Number	XCSCF <sup>2</sup>	XCSCFC
1	D2 D2 M D0 D1 N c	D1 D5 & ~
2	D2 D3 M D1 D2 M N	D0 D1 & D1 D0 &
3	D0 D3 N D1 D5 N m	D2 D5   D2 D1   d
4	CF_55 CF_44 N CF_56 CF_25 m c	CF_31 CF_28 r CF_47 CF_36 r &

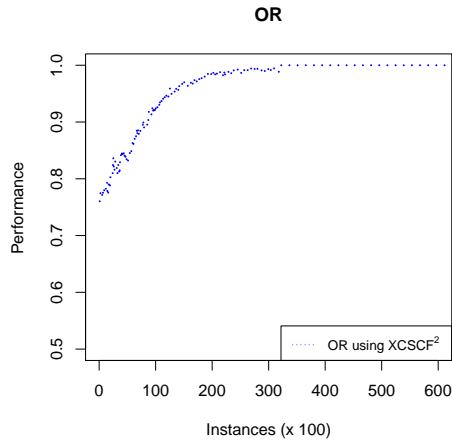
There exists a structural overlap between the systems in that both use a similar number of functions. XCSCFC uses the hard coded functions NAND, OR, AND, NOT and NOR, while XCSCF<sup>2</sup> uses the following functions NAND, OR, AND, XOR and NOR. The proposed system uses the given NAND rules to learn the rule-sets for the rest of the functions. Although XCSCF<sup>2</sup> does not learn any additional rule-sets past the NOR operator, it does continue to learn CFs, just like XCSCFC. Future versions

of XCSCF<sup>2</sup> will be capable of learning functions with more than two inputs. In addition, similar to XCSCFC, XCSCF<sup>2</sup> begins to reuse CFs at the leaf nodes starting with the 11-bit multiplexer problem. This makes it an extension of the original XCSCFC system. In other words, after the NOR operator is learned, the system uses all the rule-sets learned up to that point when evaluating the root nodes of the trees. At the same time, it also uses any previously learned CFs, beginning with the 11-bit multiplexer problem. In essence, the population of CFs will continue to grow, unlike the population of rule-sets associated with each learned function, which will remain static after the NOR function is learned. The novel system is unable to learn past the NOR boolean operator due to its serious memory requirements. In addition, the proposed system is incapable of learning functions with more than two inputs. For example, all the Boolean operators learned here have at most two inputs. In order to handle more inputs, the structure of the CFs needs to change to accommodate the extra nodes. This will have to be a dynamic property so that memory is not wasted.

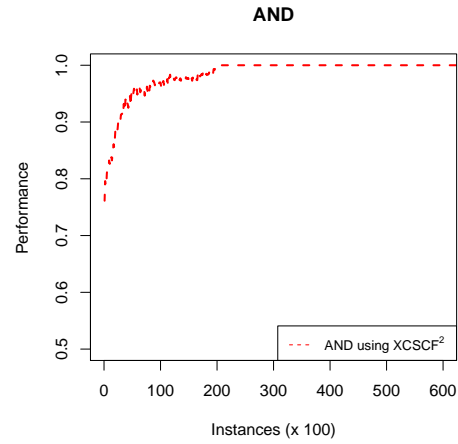
## 4.2.2 Results

### 4.2.2.1 Boolean Operators

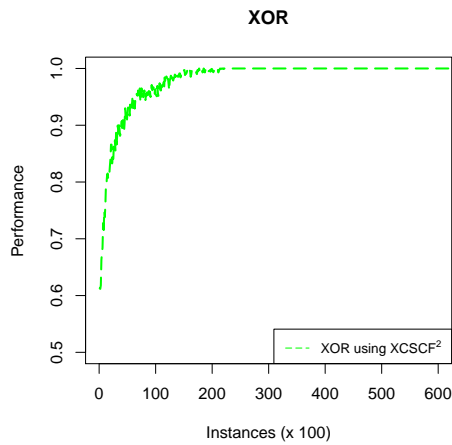
The proposed system was trained with the data-sets corresponding to the Boolean operators NAND, OR, AND, XOR and NOR in sequence, following are the results of these experiments. XCSCF<sup>2</sup> was capable of learning the {OR, AND, XOR and NOR} operators, see Figure 4.6. The NAND operator became a boot-strap function which was used to learn OR, and then the system learned the AND operator, continuing in this manner until it had learned all the necessary Boolean operators. XCSCF<sup>2</sup> was able to converge rapidly as it was learning the new rules and CFs. This was because the functions being learned are not difficult, consisting of at most four rules each, with each rule consisting of two bits.



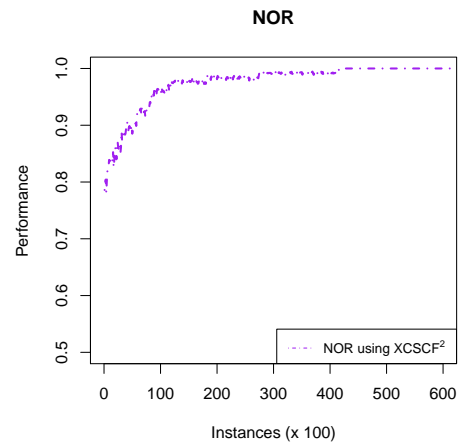
(a) OR Boolean Operator.



(b) AND Boolean Operator.



(c) XOR Boolean Operator.



(d) NOR Boolean Operator.

Figure 4.6: Results of the Boolean problems using XCSCF<sup>2</sup>. The figures illustrate learning by reusing previously learned information. The OR Boolean operator is learned by using the rules for the NAND operator, these are provided for the system. The AND Boolean operator is learned by using the learned OR operator and the NAND operator, and so on and so forth. The sample period for the instances is every 100th instance, i.e. each data point on the graph represents 100 instances.

#### 4.2.2.2 Multiplexer Problems

The results for the multiplexer experiments are illustrated in Figure 4.7, which shows the performance curves for both XCSCF<sup>2</sup> and XCSCFC. The experiments spanned the 6 bit Mux through the 20 bit Mux problems. It is evident that the proposed system performed better than XCSCFC here, in terms of a lower number of training instances needed for learning the problem. Although it is not clear what caused this advantage, it is suggested that having learned the XOR Boolean operator instead of the NOT operator – XCSCFC had the NOT operator hard-coded instead – was beneficial to the proposed system. This is because the XOR function contains at most four rules while the NOT operator contains at most two. Also, XOR contains two inputs while NOT only contains one. This means that XOR can describe more of the problem domain and therefore can help to learn faster.

Learning was slightly different between the CF-based systems used in the experiments. Both systems produced a similar number of CFs for the 20 bit Mux problem. For example, for run 8, XCSCF<sup>2</sup> produced 888 CFs while XCSCFC produced 751 CFs. This indicates that the functionality that XCSCF<sup>2</sup> was learning and that XCSCFC had hard-coded, was adequate to learn the problem. Also, XCSCF<sup>2</sup> was learning more efficiently than XCSCFC; this is supported by Figure 4.7. The proposed system converged with about 40,000 training instances while XCSCFC accomplished that with about 70,000 instances.

Table 4.2 illustrates a sample of the rules produced. Rules 1-3 demonstrate that both systems made use of their functions. Also, XCSCF<sup>2</sup> and XCSCFC both used features from the environment at the terminal nodes. The fourth rules exhibit learned CFs at the terminal nodes. The prefix 'CF\_' stands for Code Fragment, followed by the Id assigned by the system. Table 4.3 illustrates the learned rules associated with function M. These rules do not contain '#' characters, meaning that they have not been optimized by the system. This will be an improvement in future versions. The "M"

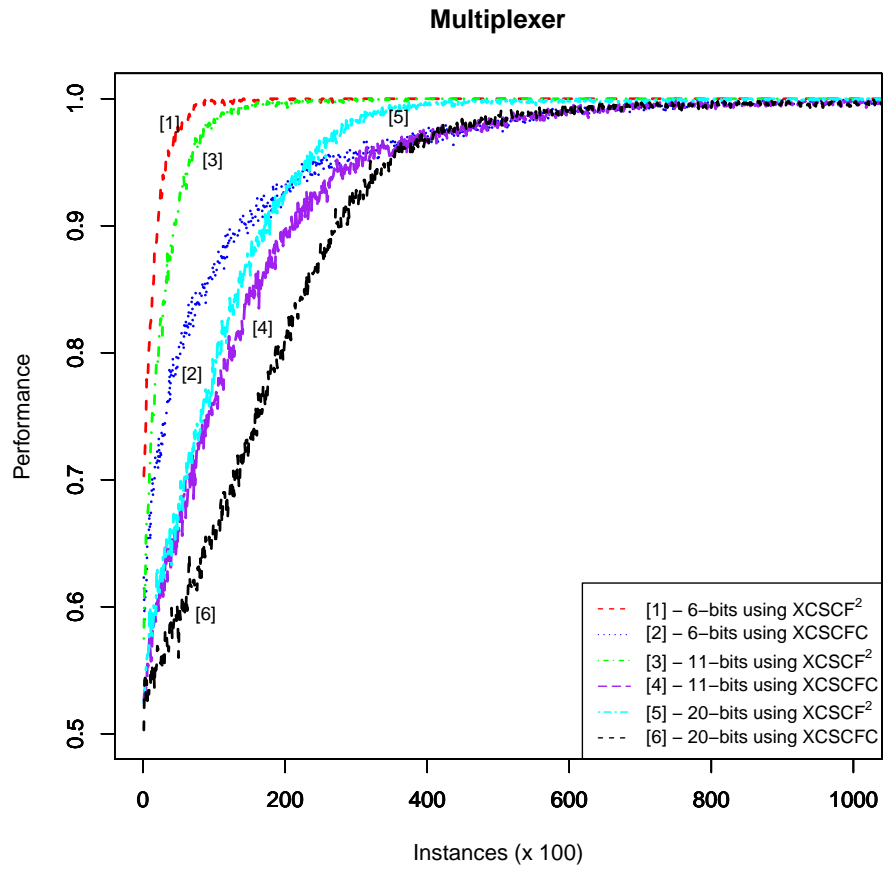


Figure 4.7: Comparison between XCSCF<sup>2</sup> with XCSCFC for the 6-20 Bit Multiplexer Problems. The numbers between the brackets are for identifying the curves.

Table 4.3: Function M learned ternary rules produced by XCSCF<sup>2</sup>. These correspond to the NOR boolean operator. The system does not provide pressure to learn *don'tCares*.

XCSCF <sup>2</sup> Rules			
1	1	:	0
1	0	:	0
0	1	:	0
0	0	:	1

is just a tag that the new system assigned to the NOR boolean operator as it was learning.

The XOR function provided benefits that were missing when the NOT function was given instead. This is illustrated in Figure 4.8. These experiments replaced the XOR function with the NOT function for XCSCF<sup>2</sup>. All the curves for XCSCF<sup>2</sup> are located closer to the corresponding curves of XCSCFC, in comparison to those of Figure 4.7. This indicates that the system has taken more training instances to fully converge for all the experiments.

The results for the 37-bit and 70-bit Mux experiments comparing XCSCF<sup>2</sup> with XCSCFC are illustrated in Figures 4.9 and 4.10. It is evident that during these experiments, XCSCF<sup>2</sup> required more training instances than XCSCFC in order to learn the problem. It is possible that XCSCF<sup>2</sup> was increasingly at a disadvantage due to not having access to the NOT operator. It is also useful to keep in mind that although XCSCF<sup>2</sup> learned new CFs and reused them during the subsequent problem(s), it did not learn new rule-sets with each new problem, past the NOR stage. This also appears to be a contributing factor of this disparity in performance. During the 6-bit to 20-bit series of multiplexer experiments, the binary rule-sets were sufficient to cover the search space of the problem. In essence, the learned rules were used to match short segments of the environmental message string. This facility was not enough for more complex problems

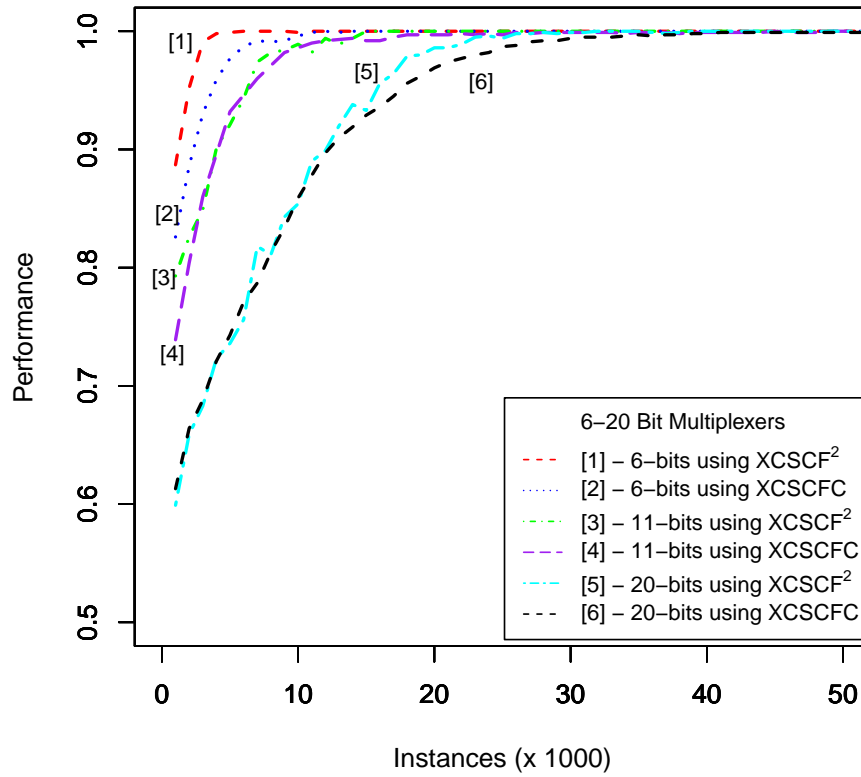


Figure 4.8: Comparison between XCSCF<sup>2</sup> with XCSCFC for the 6-20 Bit Multiplexer Problems. Instead of the XOR function, the system was given the NOT function to learn. The numbers between the brackets are for identifying the curves.

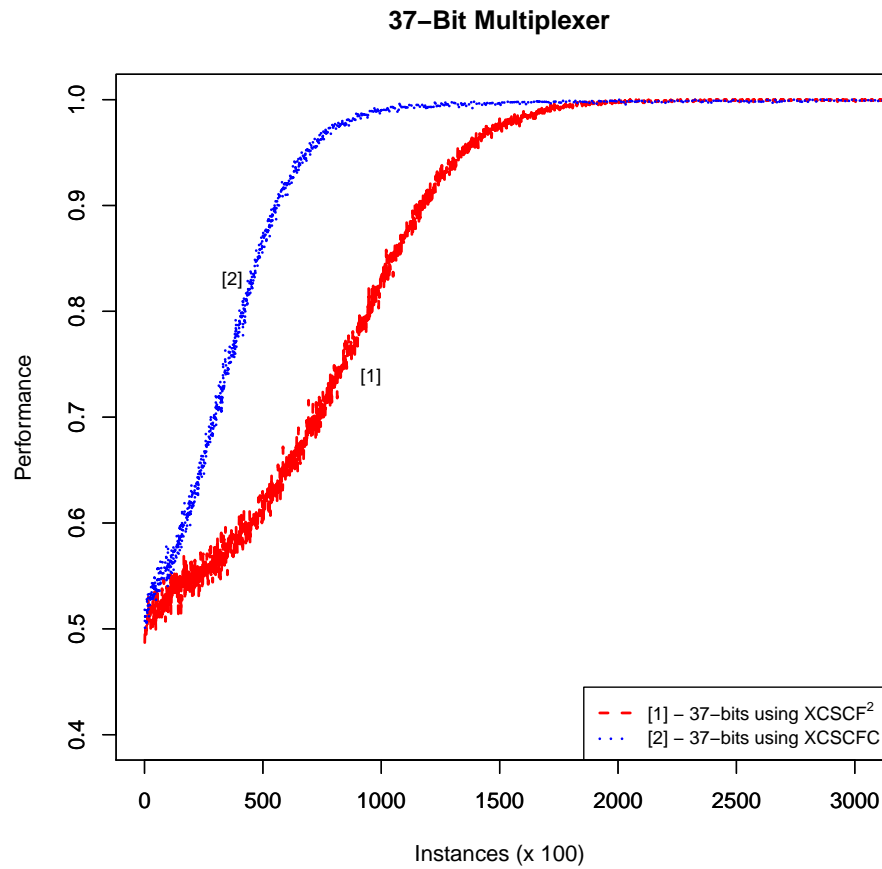


Figure 4.9: Comparison between XCSCF<sup>2</sup> with XCSCFC for the 37-Bit Multiplexer.



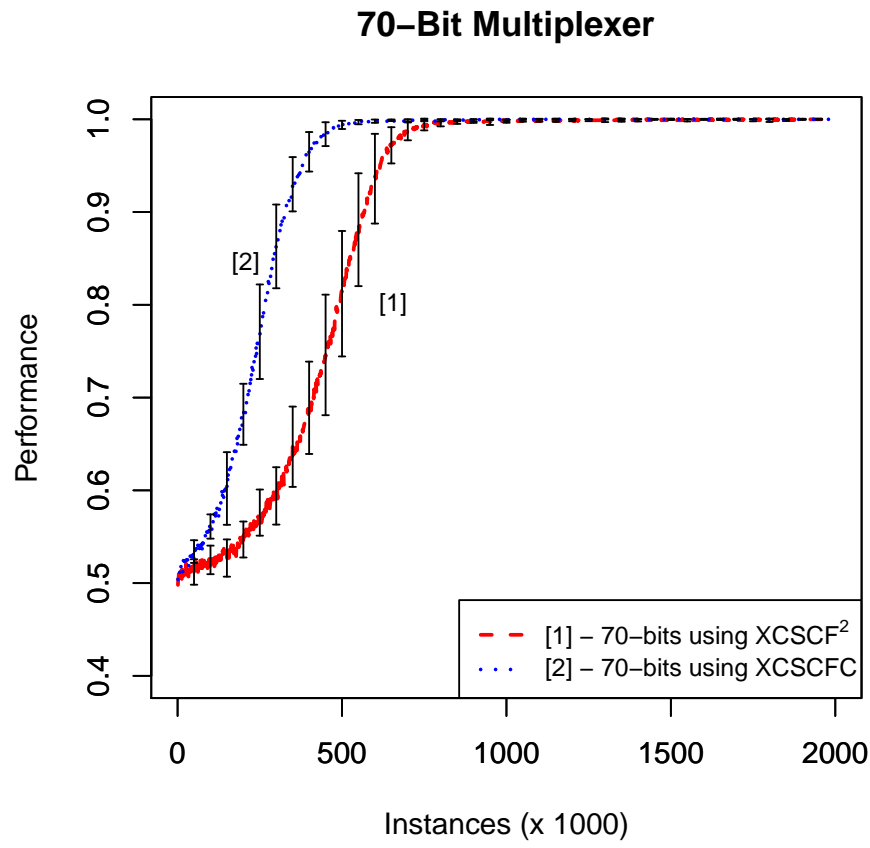


Figure 4.10: Comparison between XCSCF<sup>2</sup> with XCSCFC for the 70-Bit Multiplexer.

such as the 37-bit and 70-bit problems graphed here. This occurred because there were not enough learned functions to form useful building blocks. It could be argued that XCSCF<sup>2</sup> must work harder to find the useful combination of CFs and rules that will express the Multiplexer address bits as well as the corresponding data bit. Therefore the system must labor to niche the problem space and in this way find the general classifiers that will contain the appropriate environment features.

The results for the 135-bit Multiplexer experiments are illustrated in Figure 4.11. Similar to the 70 bit experiments, XCSCF<sup>2</sup> required more training instances than XCSCFC, however the difference was proportionally similar in both cases, if taken over the total number of training instances for the experiments. In addition, the standard deviation is slightly larger for XCSCF<sup>2</sup> as compared to XCSCFC. This indicates that the experiments had very different learning performances during the individual experiments.

It is not known *a priori* which functions will be useful for the problem being learned. There is a need to balance between too many functions and not enough. Both extremes can hinder learning by providing too large a search space or not enough *grist for the learning mill*. Our concern here is to test this on objective one and objective two.

### 4.2.3 Interpretation of Results

The results indicate that the proposed technique has potential for increased scalability. Experiments 5.1-5.2 demonstrated that there exist benefits in providing a CF system with the capability of reusing learned knowledge at the root nodes, as well as at the terminal nodes. This capability has translated into better performance by the proposed system for the 6-20 bit Mux problems. Although, these are not very difficult problems, by today's standards, they do help to illustrate the benefits of the technique. The CFs produced by the systems share similarities in the fact that

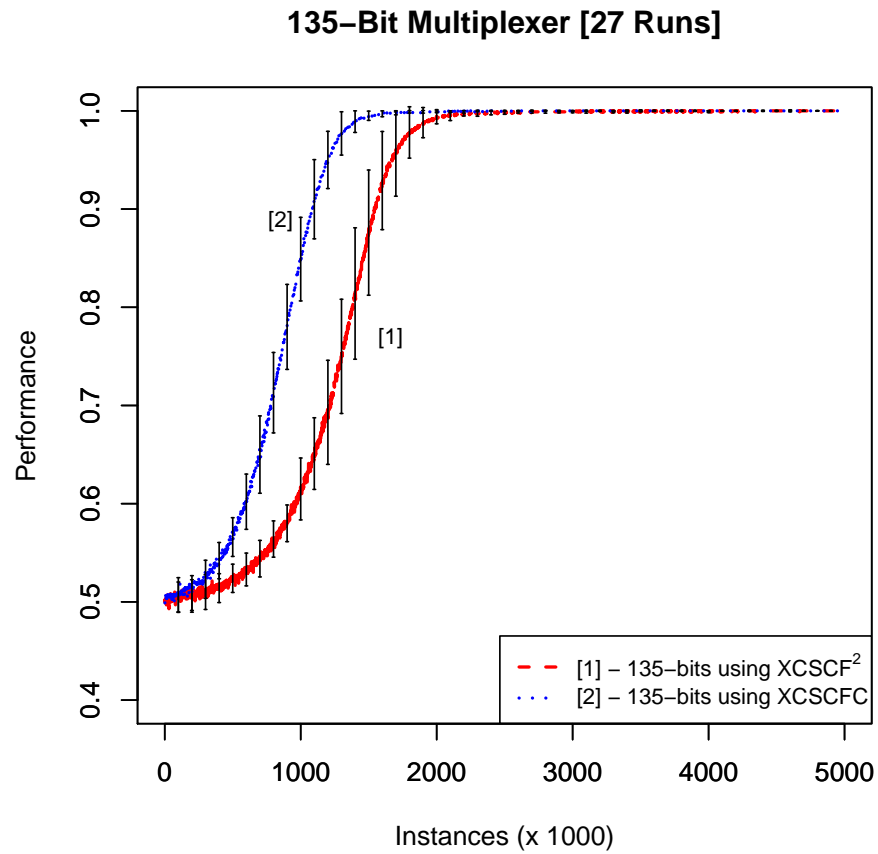


Figure 4.11: Comparison between XCSCF<sup>2</sup> with XCSCFC for the 135-Bit Multiplexer.

Table 4.4: Sample rules produced by XCSCF<sup>2</sup> and XCSCFC for the 135 bit Mux problem.

Rule Number	XCSCF <sup>2</sup>	XCSCFC
1	$N(c(D3, D1), m(D3, D5))$	$r(\&(D0, D2),  (D0, D3))$
2	$d(N(D5, D3), N(D3, D0))$	$\&(\sim(D2), d(D4, D0))$
3	$c(m(D1, D0), d(D0, D0))$	$r(d(D5, D0), d(D1, D5))$
4	$m(N(CF\_137, D2), d(D0, CF\_163))$	$\&( (CF\_147, D9), r(D1, CF\_160))$

both techniques reuse learned knowledge in a consistent manner. In other words, XCSCF<sup>2</sup> uses a combination of CFs coupled with learned rule-sets while XCSCFC reuses its learned CFs at the leaf nodes. The two techniques differ in that the functions produced by XCSCF<sup>2</sup> refer back to learned function rule-sets while in XCSCFC, the functions are hard-coded by the user. Table 4.4 illustrates the two variations of CFs.

Although the proposed work performed well against the state of the art technique, there is opportunity for improvement. First of all, the proposed system only learns new binary rule-sets up to the NOR Boolean operator. This presents a handicap that would ultimately become very apparent during the harder problems. It is hypothesized that a lack of more relevant rule-sets limits the scalability of the novel system as the problems become more complicated. By ‘relevant’ it is meant rule-sets that are closer to the problem being attempted at the time. For example, it may be helpful to have learned rule-sets for the 6-20-bit Mux problems when the system finally attempted the 37-bit Mux.

Another limitation faced by XCSCF<sup>2</sup> was a holdover from XCSCFC, namely the global storage of learned CFs in a single repository. Needless to say, this promoted lengthy searching for any matching CFs being used in the classifiers as the number of CFs can quickly number in the thousands, depending on the problem. This could be addressed in future versions by changing the way in which learned CFs are stored.

#### 4.2.4 Summary of Reusing Rule-sets as Functions

The new work showed that building blocks of knowledge can be useful at the root nodes as well as the terminal nodes of the CF trees. By using XCSCF<sup>2</sup> it was possible to learn building blocks of functionality from basic boolean rules. The learned function rule-sets were stored and used to solve the subsequent problems. It was also shown that knowledge learned in the boolean logic domain could be used to solve problems in a related domain such as the multiplexer. This was demonstrated with the solutions to the 6-20-bit multiplexer problems. It was also determined that as the problem scales, the time requirements for the technique also grow, however they are not prohibitive.

It is anticipated that further scaling is achievable by providing a linking between the learned rule-sets and their associated CFs. This will reduce the search space by limiting the number of CFs that must be searched during the normal execution of the technique. This has been identified as a critical step in enabling the cross-domain knowledge transfer that includes function rule-sets as well as their respective set of code fragments.

### 4.3 Reusing Code Fragments to address the Hidden Multiplexer

#### 4.3.1 Method

It has been shown that reusing rule-sets as functions at the root nodes of CF trees is feasible and beneficial. This was accomplished in the Boolean domain as well as a related domain, i.e. Multiplexer. However, there are problems which are more difficult than the aforementioned ones, which could benefit from this type of process. Such problems present enough difficulty that a reduction in the search space is a pre-requisite to find their solution. In this work, the proposed technique will provide a tight linking

between the learned rule-sets and the learned CFs. It is anticipated that this will reduce the search space facilitating the solution to these problems.

The system involved uses two-point crossover for the rule discovery; according to Butz [17], uniform crossover tends to have deleterious effects on the population, making XCS unable to solve the 18-bit hidden multiplexer problem. According to Butz [17], this happens because XCS is unable to process the building blocks at the lower level, but rather disrupts them. On the other hand, two-point crossover helps XCS learn the problem [17]. Butz also addresses the need to provide a larger population to XCS due to its inability to efficiently handle building blocks for this type of problem, see Section 4.3.2.

The problem domain in this new work is the hidden multiplexer. The basic structure of the problem is shown in Fig 4.12. Since the solutions can be composed of Boolean functions, this makes it a feasible domain to learn [17], [56]. Also, their multi-tiered nature, non-redundant features and high epistasis have made them appropriate for this type of experiments by providing enough difficulty to show the benefits of the new technique.

The hidden multiplexer is a two tier problem involving a lower layer composed – in this case – of 3-bit parity problems. The lower level must be evaluated first, in order for the results to be passed up to the top layer, which is composed of one multiplexer problem. There is a fixed correspondence, and the address bits are irrelevant. Once all the upper level bits are known, the multiplexer part is evaluated and the class is determined.

This problem is considered very difficult due to its two level structure. Unlike a standard multiplexer, the rules produced by the solution must account for this hierarchical property. The system must exercise efficient usage of the building blocks of knowledge. This means that the rules must address proper combinations of parity solutions which also produce correct combinations of bits in the upper level. Therefore the training of the system must include the proper number and types of functions to facilitate

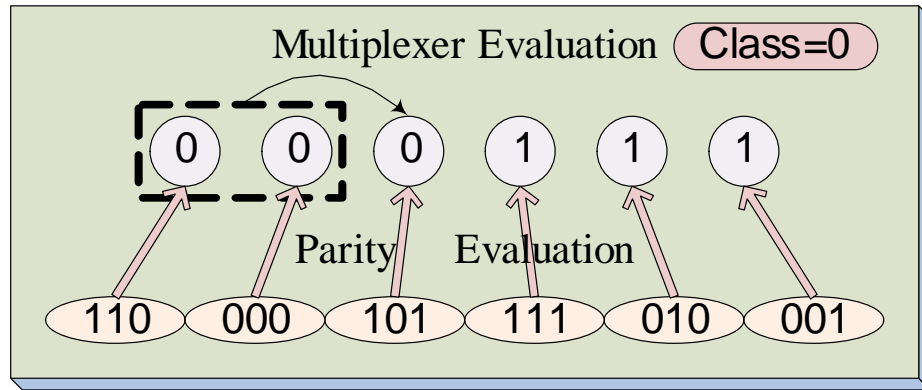


Figure 4.12: Example of the Hidden Multiplexer. The lower level consists of parity problems and the upper level consists of a multiplexer problem.

the correct solution.

The first step is to determine the fundamental functions that should be given to the new system. The more of these that are included, the more domain bias is also included without the capability to learn the connection between the functions and the discovered building blocks. For instance, for the Boolean domains, NAND gates are building blocks through which it is possible to build other gates such as the OR, AND or NOR [3], [68], [86].

The system described here is initially given the NAND rules, to provide a bootstrap function with which to learn the NAND function CFs. NAND is important because it makes it possible to build the other Boolean operators. Although it appears to be inefficient, this step is in preparation for having XCS learn the NAND rules for this step in the future. This will reduce human knowledge and bias from this process. After this step is done, the system is trained to learn the OR function. Once this step is finished, the system will have the NAND and OR rules available for solving problems. There is also a new set of CFs that are linked to the OR function. The system is then trained in a similar manner to learn the AND, XOR and NOR functions.

The purpose of the base training is to have the system build a cache of reusable functions, composed of rule-sets and associated CFs. This cache then becomes relevant building blocks of knowledge for these functions. Then the hidden multiplexer problem is attempted [3].

First, the human experimenter must set the order of the problems to be addressed. This stage consists of three separate branches, see Figure 4.13 for more details. The parity branch continues on to the 2-bit even and odd parity, 3-bit even and odd parity, in that order. When this is completed the system attempts to learn the 18-bit hidden multiplexer. The second branch also consists of learning the boolean operators, the parity problems and then the 6-bit multiplexer problem. When this is accomplished, the proposed system is given the 18-bit hidden multiplexer problem. The last branch consists of learning the boolean operators as mentioned above, then the system is given the 6-bit multiplexer, prior to attempting the 18-bit hidden multiplexer problem.

### 4.3.2 Results

The training regimen provided certain benefits depending on the order of the problems being learned. Figure 4.14 illustrates results for the three separate training branches in the experiments. Including the parity function during the training helps the system learn the hidden multiplexer problem better than otherwise. This was to be expected because the parity problem provides valuable functionality. The type and number of rules produced are also dependent on this factor.

Table 4.5 shows the number of rules for an arbitrarily selected run (No. 8) of these experiments. It is clear that the number of rules and number of CFs produced grow in opposite directions as one reads down the table. What is not clear, however, is the reason for the discrepancy in performance. In other words, one can not readily determine what factor helped the first two training paths to converge with similar numbers of training



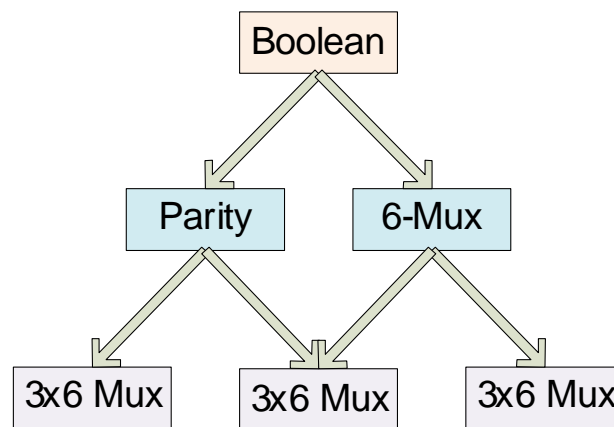


Figure 4.13: Training flow of learned functions for the system. All the paths begin with the Boolean operators and culminate with the 3x6 hidden multiplexer. The first training path includes the Boolean, Parity and 3x6 Mux problems. The second path includes the Boolean, Parity, 6-Mux and 3x6 Mux problems. The third path includes the Boolean, 6-Mux and 3x6 Mux problems.

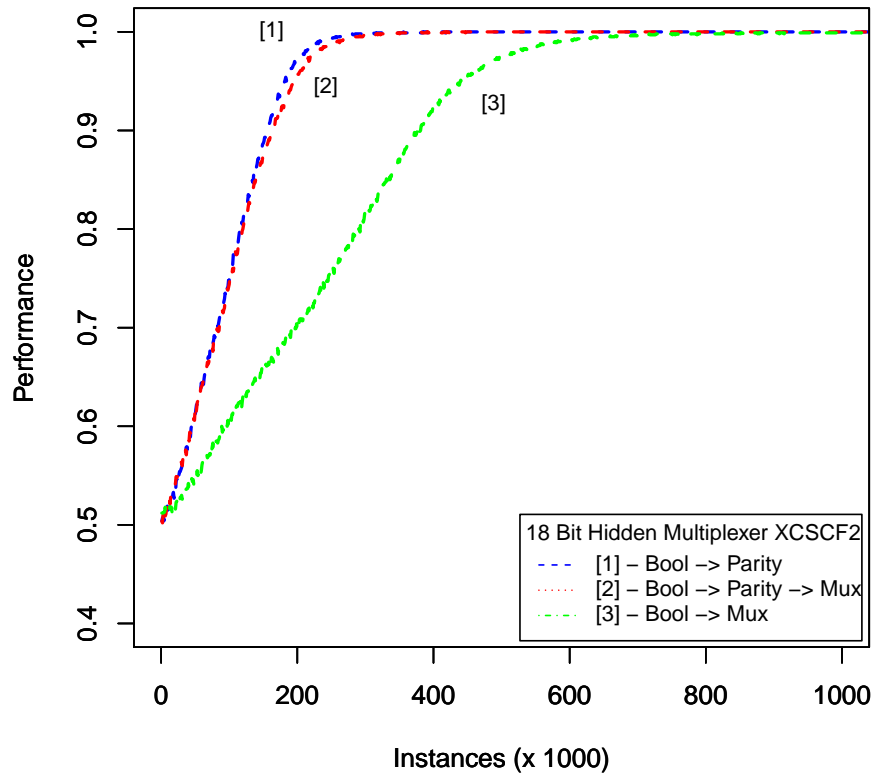


Figure 4.14: 18-bit Hidden Multiplexer problem. The graphs show three different training paths (Boolean, parity, 6mux) for XCSCF<sup>2</sup>.

Table 4.5: Number of rules and CFs produced by XCSCF<sup>2</sup> for three different training paths. These are for run 8.

Training Path	Number of Rules	Number of Code Fragments
Boolean $\rightarrow$ Parity	184,585	14,225
Boolean $\rightarrow$ Parity $\rightarrow$ Mux	169,936	17,164
Boolean $\rightarrow$ Mux	100,558	24,479

instances while the third path needed about 300,000 more instances. The quality of the CFs could be a clue and a sample of these is shown in Table 4.6. The CFs for the three training branches do not demonstrate an obvious difference, with perhaps a difference in CF chain lengths. The chains for the Boolean-Parity-Mux path would be expected to be longer as they have had one more layer of training, however upon a closer inspection of the data produced, no major difference has been noted. This inspection indicates that the critical factor is the Parity function. It is providing useful knowledge blocks in the training paths where it is used. Also, the fact that the training path using Boolean, Parity and Multiplexer functions produces more rules than the path that uses Boolean and Multiplexer functions, is because the Multiplexer does not add as many useful knowledge blocks as the Parity function. In fact, it is causing the system to produce an overabundance of rules, as compared to the path where only Boolean and Parity functions are used. On the other hand, the Parity function makes a noticeable difference in the number of CFs produced, as can be seen on the right hand side column of Table 4.5. The Parity function helps produce a more compact population of CFs.

Figure 4.15 shows the results for the 3x6 hidden multiplexer after having trained the system with the boolean and parity problems. The proposed system performs better than XCS or XCSCFC. Figure 4.16 shows a graph of the results for the training path of the Boolean and Multiplexer problems. The lack of the Parity problems during the training appears to have made a difference in the performance of the proposed system. Fig-

Table 4.6: Final Code Fragments produced by the three different training paths.  $D^*$  represent features while a-z, A-Z are learned functions.

Problem	CFs Produced
Boolean $\rightarrow$ Parity	$J(Y(D5, D10), X(D13, D13, D13))$ $J(D3, D4)$ $J(D14, D12)$ $g(O(D9, D1), H(D13, D4), X(D2, D0, D9))$ $\dots$
Boolean $\rightarrow$ Parity $\rightarrow$ Mux	$c(Y(D3, D4, D9), g(D5, D4, D3), Q(D16, D16, D1, D5))$ $H(J(D16, D11), X(D1, D2, D10))$ $d(d(D10, D9), Q(D15, D17))$ $D16$ $\dots$
Boolean $\rightarrow$ Mux	$D4$ $y(Y(D2, D14), Q(D1, D0))$ $y(D2, Q(D0, D1))$ $d(K(D3, D3), K(D3, D6))$ $\dots$

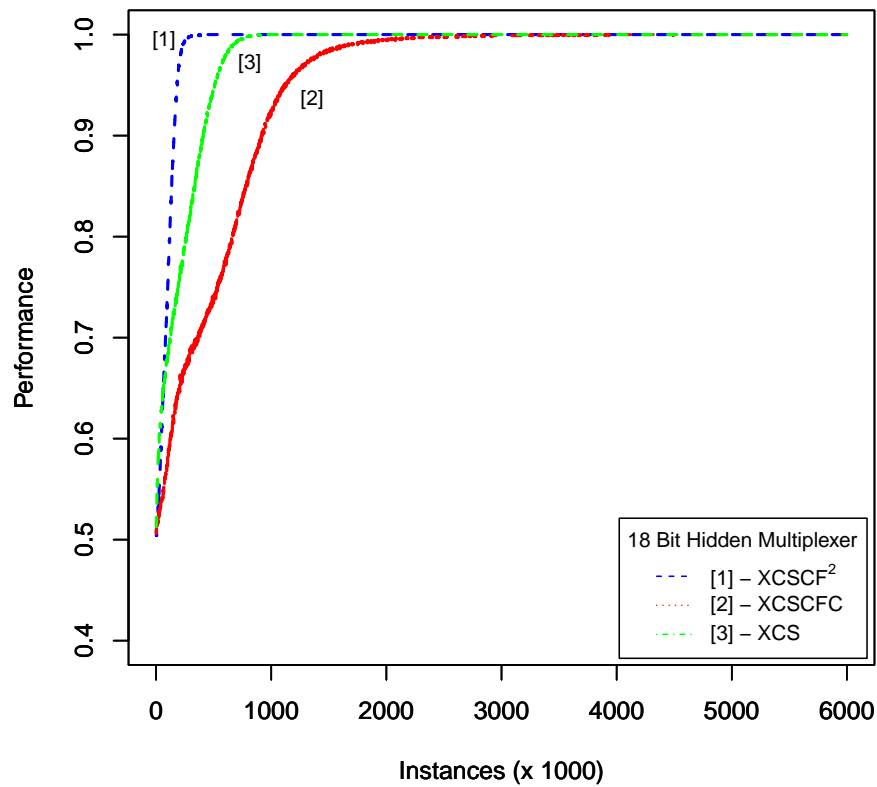


Figure 4.15: 18-bit Hidden Multiplexer problem. The graphs compare XCSCF<sup>2</sup> against two other systems. The training path is (boolean, parity, hidden multiplexer).

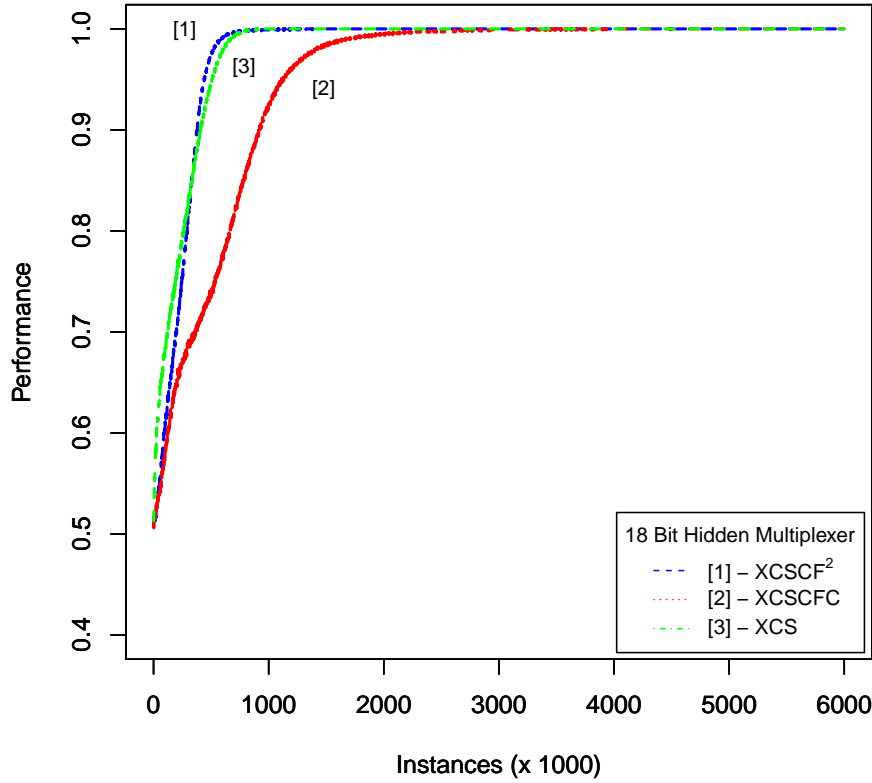


Figure 4.16: 18-bit Hidden Multiplexer problem. (Boolean, mux)

Figure 4.17 shows that the training involving the Boolean, Parity and Multiplexer problems produces results similar to the ones produced by the path with Boolean and Parity problems. This was expected, based on the results pointed out in Table 4.5.

Table 4.7 illustrates the population size and number of training instances for the Boolean experiments. These were only the base problems that would be used later to learn the hidden multiplexer. The number of training instances for the initial nine problems is higher than for the multiplexer problem. The NAND problem needs twice as many classifiers as

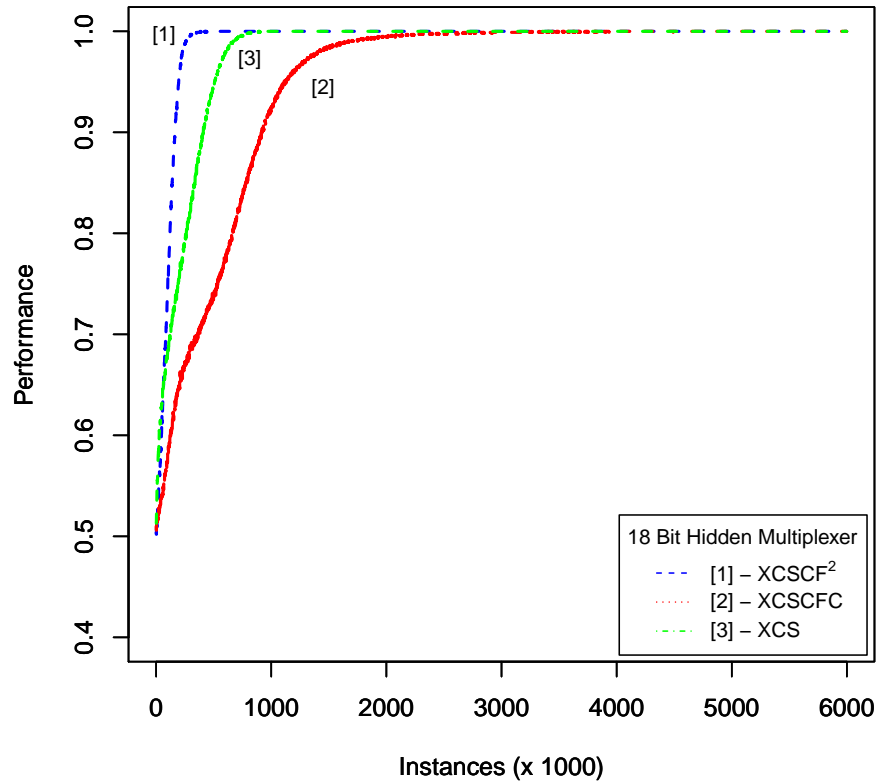


Figure 4.17: 18-bit Hidden Multiplexer problem. (Boolean, parity, mux)

Table 4.7: Number of classifiers and training instances for  $XCSCF^2$ .

Boolean	Classifiers	Instances
NAND	1,000	700,000
OR	2,000	700,000
AND	2,000	700,000
XOR	2,000	800,000
NOR	2,000	900,000
2-Bit Even Parity	2,000	900,000
2-Bit Odd Parity	2,000	1,000,000
3-Bit Even Parity	2,000	1,100,000
3-Bit Odd Parity	2,000	1,200,000
6 Mux	500	500,000

the Mux and 200,000 more instances. This is strange, given the fact that the NAND rules are provided *a priori*. It is hypothesized that this is caused because the system is learning the CFs associated with the NAND rules and there aren't any CFs to reuse at this time. The reason that the system was trained with the NAND Boolean operator in spite of having the rules is because in future versions those rules will be learned by a standard XCS and then will be provided, which will reduce the amount of human knowledge given to the system. The other Boolean operators require more classifiers and training instances, but that can be attributed to the growing cache of learned functionality. If there are too many functions or not enough, it can impact the learning ability of a system. In this case, the system does not have enough useful functions yet. Also, it is logical for the Parity problems to have higher requirements as they can be considered more difficult than the Boolean operators. This initial higher overhead in training instances and population can be attributed to the lack of useful functionality; this is overcome as the system continues to learn and accumulates more building blocks, like during the multiplexer problems.

Table 4.8 shows the classifier and training instance requirements for



Table 4.8: Number of classifiers and training instances for  $XCSCF^2$ , XCSCFC, and XCS for the 3x6 Hidden Multiplexer.

System	Classifiers	Instances
$XCSCF^2$	50,000	6,000,000
XCSCFC	50,000	6,000,000
XCS	100,000	6,000,000

the three systems used while solving the 3x6 hidden multiplexer problem. The training instances for all three systems were the same number. XCS was allotted twice as many individuals as the other two systems for two main reasons: XCS is known to have difficulty solving this problem; it has been suggested by Butz [17] to provide a very large number of classifiers to XCS for this type of problem. According to Butz [17], in the k-parity-k-multiplexer combination, the optimal population is of size:

$$|[O]| = 2(2^{k(k'+1)}) . \quad (4.4)$$

The optimal population size is closely related to the order of difficulty of the problem, taking into account the lower and upper level building blocks. This size is given by the equation:

$$k_d = k(k' + 1) . \quad (4.5)$$

Here k stands for the number of bits in the parity part while k' stands for the number of bits in the multiplexer part [17].

### 4.3.3 Interpretation of Results

One of the main observations during this new work is that the Parity problems are important in finding the correct patterns for the hidden multiplexer. This was demonstrated by the results involving  $XCSCF^2$  and the three different training paths. It is hypothesized that since the structure of

the hidden multiplexer involves a layer of parity problems, the presence of the learned Parity function is important and provides valuable relationships between different problem features. Furthermore, the presence of the multiplexer learned function does not appear to be as important as the Parity. This may be due to the fact that the Boolean problems are enough to enable a solution to the Multiplexer layer. This can be seen in Figures 4.15, 4.16 and 4.17.

XCSCF<sup>2</sup> performed better than XCS and XCSCFC for all the experiments in terms of training instances required. This can be attributed to the strengths of the technique, i.e. reusing CFs at the root and leaf nodes. The performance by XCSCFC and XCS were counter-intuitive, given the somewhat similarity between the proposed system and XCSCFC. It was unexpected, but XCS consistently performed better than XCSCFC, in terms of training instances required. This is due to the overhead of its richer alphabet.

While the 3x6 hidden multiplexer served as a good testbed for the technique, it is not considered a very difficult problem. This is because problems with a size of 18 bits are routinely solved by XCS and similar systems. Therefore more work still remains to be done. It is not obvious how much more the Parity function alone can help in the solution of more difficult problems like the 3x11 hidden multiplexer. Nevertheless, the solution to the 3x6 hidden multiplexer problem by XCSCF<sup>2</sup> indicates the potential for more scalability.

#### **4.3.4 Summary of Reusing Code Fragments to address the Hidden Multiplexer**

Learned knowledge and functionality was successfully reused at the root nodes and leaf nodes of CF trees, respectively. This facilitated opportunities for reducing the search space by providing a tight linking between the learned rule-sets and the learned CFs. Each time a CF was evaluated,

the system searched only among the CFs assigned to the learned function, instead of one large CF repository, as was done before. This demonstrated the feasibility of reusing rule-sets as functions.

Learned knowledge from one domain was successfully transferred to a related domain. The proposed system was initially trained on the Boolean operators and then one of three training paths was chosen. The learned functionality was then used to learn a problem in a related domain, namely the hidden multiplexer.

The graphs are indicative of increased scalability within the tested scope. XCSCF<sup>2</sup> consistently outperformed XCSCFC and XCS, even when it was at a disadvantage; when the Parity problems were not part of the training regimen, performance suffered. This hints at comparative performance in more difficult problems such as the 3x11 hidden multiplexer, however currently the system can not handle variable length CFs, which is a prerequisite for handling the more difficult problems.

Although the system performed well, there is a consistent overhead in the population size and training instances needed to learn the initial functionality, i.e. {NAND, OR, AND, XOR, NOR}. Even the Parity problems proved expensive in comparison to the multiplexer, however these investments in resources resulted in increased performance.

Further work is required to facilitate dynamic length CFs. This is a current limitation that precludes the creation of CFs over a specific threshold. This customization will allow the testing of more difficult problems with an ostensibly larger domain space, e.g. 3x11 hidden multiplexer.

## 4.4 Chapter Summary

The focus of the work was to learn a group of Boolean functions that could be used as a basis for further learning in the same or a related domain. The basis for the learned functionality was to be in the form of rule-sets which would then be used as functions. Part of the novel work also in-

volved determining if the learned functionality could be transferred to related domains to facilitate further learning. Importantly, the effects on the scalability of the proposed technique would figure as an important aspect of the experiments.

Learned functionality in the form of rule-sets was reused at the root nodes, while CFs were reused at the leaf nodes of CF trees. This was achieved by replacing the usual hard-coded functions in the root nodes with learned rule-sets. The newly learned CFs were stored in one repository regardless of the function used to learn them. This builds on previous CF enabled systems which only reused CFs at the leaf nodes. The training included the Boolean operators, i.e. {NAND, OR, AND, XOR, NOR}. From these operators the proposed system learned rule-sets and the associated CFs. The learned rules – used as functions – were utilized to successfully learn more complex problems in the same or a related domain. This was shown by arriving at a solution to the Multiplexer problem by using previously learned CF functionality. The novel system was tested on the 6-20 multiplexer problems, which it was capable of solving. Larger problems caused the system time challenges, however these were not prohibitive.

The important findings obtained in Section 4.1 of this chapter were capitalized upon successfully in the second part of this work. Three different training paths were used to train a CF based system. As in Section 4.2, this system used learned functionality at the leaf and root nodes, with the added benefit of having a more extensive training regimen. This was necessary as the problem being tackled was of a hierarchical nature and hence more complex than the multiplexer problems solved in the previous section.

The benchmark problems used to test the stated goals were suitable as they provided enough difficulty and cross domain requirements to highlight the strengths and weaknesses of the technique. As was anticipated, the new technique provided improved scalability which was evident by

the solution to the 18-bit hidden multiplexer problem. It was also discovered that the Parity problems are important for learning the hidden multiplexer problem.

The problem used to showcase the technique is not considered very difficult for two reasons: first of all, although it is composed of two levels, it is only 18 bits long, longer problems are now routinely solved by comparable techniques; second, the same problem was solved by XCS more efficiently than by XCSCFC, which is considered the state of the art. Therefore, in spite of the successes achieved with this original work, more needs to be done. The new system is not capable of executing beyond the 3x6 hidden multiplexer in its current form. This means that the technique must be updated to address this shortcoming. The system will have to deal with varying lengths of CF trees as the problems grow in complexity. Another very important observation is that the technique, in its current form, has a scalability limit. This limit is linked to the size of the problems, i.e. the number of inputs related to each new function. The next chapter introduces two variations of a compaction technique called Distilled Rules. The techniques were developed with the goal of reducing the search space of problems by providing ternary rule-sets in lieu of long CF chains at the root nodes.



## Chapter 5

# Distilled Rules for CF-based Systems

### 5.1 Introduction

This chapter builds upon the work addressing learned functions composed of rule-sets as well as CFs. In Chapter 4 it was shown that scalability improves by using the technique, and it reduces the search space. Since the overall goal of this thesis is to increase the scalability of LCSs, numerous techniques were explored. In Chapter 4 it was noted that scaling was being adversely affected by computational inefficiencies. It was theorized that if the potentially long chains of CFs emanating from the root nodes could be reduced or changed to a more computable form, it could reduce the time requirements when evaluating said CF chains. This identified the need for a compaction process.

There exist many types of compaction methods for LCSs. One common form of compaction is the subsumption mechanism that is part of XCS. This mechanism takes the form of GA subsumption and action set subsumption, see Chapter 2. A similar compaction technique has been used to extend a CF-based XCS system as described in Chapter 2 and in [39]. This technique is promising, and shares similarities with the proposed Distilled

Rules (DRs) technique. The number of macroclassifiers in low dimensionality problems was reduced, however the technique is not currently scalable [39]. CFs are expressive and compact, which means that they are difficult to compute. A compaction technique that translates CFs into a simpler form could benefit scalability, see Chapter 2 and [39]. This technique is needed because previous compaction techniques have had limited scalability [39]. In the proposed thesis, a CF-based LCS is anticipated to scale to complex problems by using a compaction technique. The evaluation of the CFs as the state of the environment changes is a subtle but very important part of the compaction process, because it economizes computation resources. The results of the subsumption technique identify computation speed as restricting scalability. According to Po-Ming *et al.* [39], the lengthy runs precluded the experimentation with multiplexer problems with a length of 20-bits or more [39]. A similar issue could be faced by the proposed offline DR work; as if performance degrades rapidly with problem scale. Then this could be remedied with an online DRs method.

An alternative technique for compaction performs it as a post-processing step, see Chapter 2 and Wilson [95]. This means that this technique, like that for DRs, is not taxing to the normal execution of XCS. However, the proposed work aims to translate the CFs present in the classifiers [95]. The benefit anticipated from this technique is associated with scalability. The compact and complex CFs will be converted to a simpler alphabet. Another benefit will be that the technique will not burden the normal execution of XCS, rather it will be a post-process after the final population of classifiers has been evolved. However, the disadvantage could rest with the random states that will be used to test each of the classifiers in the final population. It is anticipated that this processing will grow as the problem scales. The hypothesis is that an off-line compaction technique could be used to translate CF-based classifiers into a simpler alphabet. The technique is anticipated to increase scalability.

A different technique for compaction was put forth by Dixon [25].



The technique known as CRA2 is executed during the normal iteration of the training stage. Classifiers that have the highest payoff prediction, numerosity product, are marked 'useful'. When the run is over, all the classifiers not marked 'useful' are removed. According to Dixon [25], this technique can produce a reduced population. However, it is dependent on having at least one dominant classifier in each matchset. Furthermore, this algorithm only works if the classifiers have zero errors and payoffs equal to the environment reward value.

A variant of the XCS technique, known as Supervised Classifier System (UCS), was used to extract optimal rules in an online method; these rules are known as *signatures* [76]. UCS is an accuracy based LCS, like XCS. However, during the explore phase UCS learns directly from the environment state as opposed to receiving a reward. The signature extraction technique seeks to avoid the limitations found in a normal run of UCS. The problem is that the population convergence does not happen quickly and a large number of iterations are required [76]. Also, the end population contains classifiers that are not part of the optimal population. The signature extraction technique automatically detects optimal classifiers as they are discovered by UCS, in addition it terminates the search process as soon as a maximally general solution is found [76]. The signature extraction technique was successfully tested on the multiplexer problem. While testing on a 2-dimensional checkerboard problem, the technique found near optimal decision boundaries. Although the technique was successful during the multiplexer problem, the new research work will be dealing with the translation of a CF-based ruleset into a binary alphabet. Most importantly, the signature technique was implemented using a supervised training model while the new proposed work will utilize a reinforcement learning model. However a number of the intricacies in the signature technique can be readily included in the new proposed work.

This is to be an ancillary process that is anticipated to benefit the overall execution of subsumption and classification by translating the rich CF

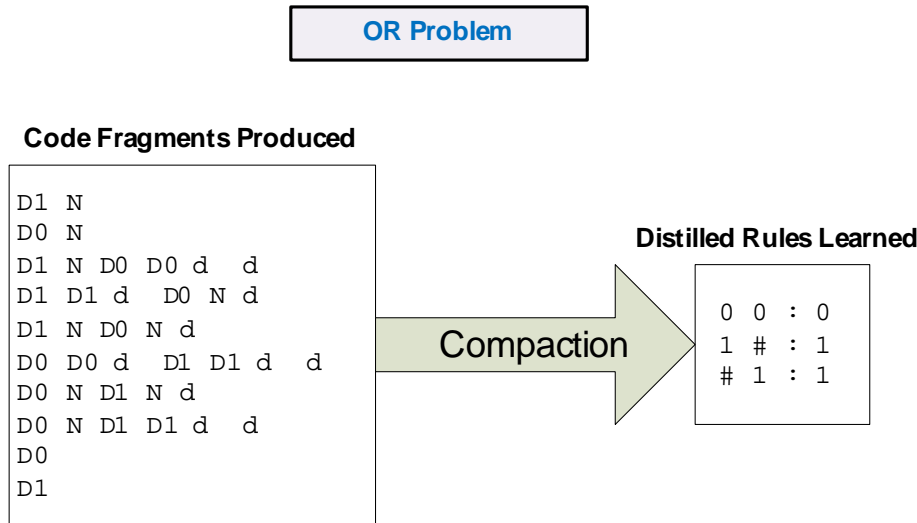


Figure 5.1: OR - Final CFs and their corresponding DRs. D0 and D1 are features from the environment. N and d represent learned functions. The tag 'd' corresponds to the NAND function while 'N' stands for the previously learned NOT function.

alphabet into a simpler ternary alphabet that is faster to process. Therefore, there will be a tight linking between the new Distilled Rules (DRs) created and the CF rules from where they originated, see Figure 5.1. This link is useful as the CFs are reused by the system and they include feature construction and feature selection while the DRs are used instead of evaluating long chains of CFs. The compaction techniques proposed in this chapter are deemed necessary and useful because CF-based classifying systems tend to breed long chains of CFs. Although there exists an initial limit to the CF depth, as the population grows and evolves, it is very common to have CFs that have other CFs at their leaf nodes. If CFs are part of classifiers in learned functions at nodes, then this can add to the length of evaluation of CFs. In the on-line version of DRs, compaction of the population is not intended to be separate and apart from the underlying function of the CF technique.

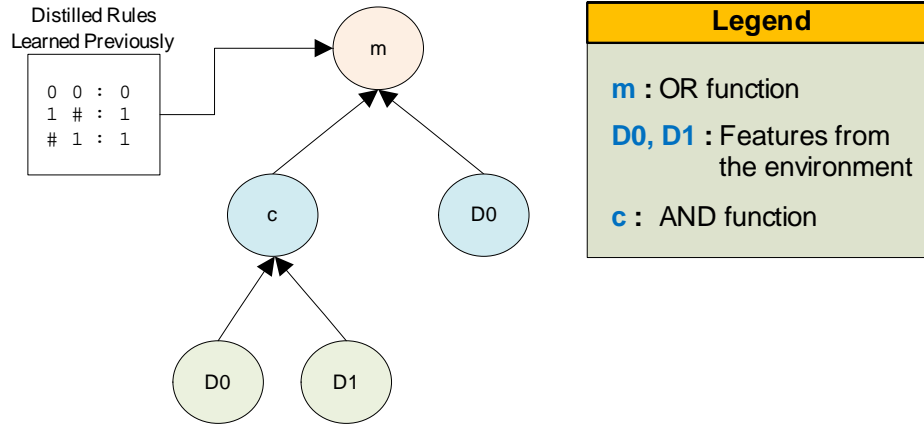


Figure 5.2: CF 43 Sub-Tree - graphical illustration of leaf (bottom) and root (middle and top) nodes. The root node is the learned OR function where the associated DRs are shown linked to it.

Figure 5.2 illustrates a CF produced by one run of the 6-bit Mux problem. The whole sub-tree shows the hierarchy of the previously learned functions, which includes the previously discovered Distilled Rules associated with the OR function. As there are no hard-coded functions in the proposed system, the DRs serve as the ternary definition of the function, in this case OR.

### 5.1.1 Chapter Goals

The proposed work in this chapter aims to produce a replacement population of ternary rules known as Distilled Rules (DRs). The technique is deemed important because CFs tend to grow as the leaf nodes continually reuse previously learned CFs. As these can represent very long chains of CFs, it was necessary to find a way to shortcut the evaluation of CFs, specially during the matchset creation step. The ancillary benefit provided by the DR layer is a circumvention of any hard-coded functions. Meaning that the system learns new functionality without injecting environmental bias via the functions provided by the user. The emphasis for good

learning performance then falls on the choice of problems presented to the system and not so much the training instances or user defined functionality.

The two types of compaction proposed in this chapter build upon a common foundation. By implementing a mutation operator as a post-processing stage, it will be shown that a layer of DRs can be distilled from a final population of CF-based classifiers. Tests on the offline DRs extraction method, see Section 5.2, showed promise in compaction, but at the cost of long running times. The method was used as the basis for the online DRs extraction method, see Section 5.3, where run times could be reduced as the online method executes mostly during the normal XCS stages. There is a DRs reconciliation which takes place after the final population of classifiers has been evolved, however it is a small percentage of the overall online technique.

Objective two of the thesis (Section 1.4) is to identify a compaction technique to simplify the final population of CF rules. The new work to be presented in this chapter, known as XCSCF3, will also expand on the techniques, introduced in Chapter 4, which showed that reusing learned rule-sets at the root nodes of CF sub-trees is beneficial for the performance of the system, as well as for the scalability. The ancillary benefit is anticipated to be a reduction in the length of the CFs leading to the solution of more complex and difficult problems.

## 5.2 Offline Distilled Rules Extraction

### 5.2.1 Method

#### 5.2.1.1 XCSCF3 Training

The proposed technique is based on XCSCFC and utilizes a multistage training regimen. It begins with the NAND function, see Figure 5.3. This is provided for the system as a bootstrap function after having a standard

XCS learn the ternary rules. A standard XCS learns the NAND problem and the ternary rules produced are placed in a file. The system undergoes the customary explore/exploit phases and the original ternary alphabet is used, i.e.  $\{0, 1, \#\}$ . The file containing the learned NAND rules is then imported into the proposed technique, XCSCF3 to be used as a learned NAND function for subsequent problems.

The training then progresses to the Boolean operators  $\{\text{NOT}, \text{OR}, \text{AND}, \text{XOR}, \text{NOR}\}$ , which are learned with the help of the NAND function [3]. The rest of the Boolean operators can be learned by using the NAND function. The training then involves the 2-bit and 3-bit Parity functions, which then leads to the 3-bit multiplexer and the 6-bit multiplexer, see Section 4.2.1. At this point the training diverges into two branches. One branch learns the 11 to 70-bit multiplexers and the other learns the 3x6-bit hidden multiplexer and the 3x11-bit hidden multiplexer. This will help determine if the rules learned are transferable to a related domain.

Throughout the above steps, the system will be accumulating new sets of DRs. Each time a function is learned and the final population of CF-based classifiers is evolved, the DRs translation process takes place. See Algorithm 5 for an overview of the technique. Only the classifiers that are always accurate, correct and with enough experience are selected for translation. These classifiers must also have a prediction error below  $\epsilon_0$ . Each classifier will be evaluated, one non-don'tCare CF at a time. The first step will be to determine which problem features are significant for the current CF. This means that successive problem states will only change the significant features while ignoring the others. Any random problem instance that results in a one (CFs that evaluate to 1), for all the CFs in a classifier, will qualify as a temporary DR. Once the maximum number of random problem instances have been presented to the valid classifiers, the subsumption operator acts on the population of temporary DRs. This consists of comparing all pairs of DRs and subsuming the less general one, if both support the same action. Duplicates are also deleted to reduce the

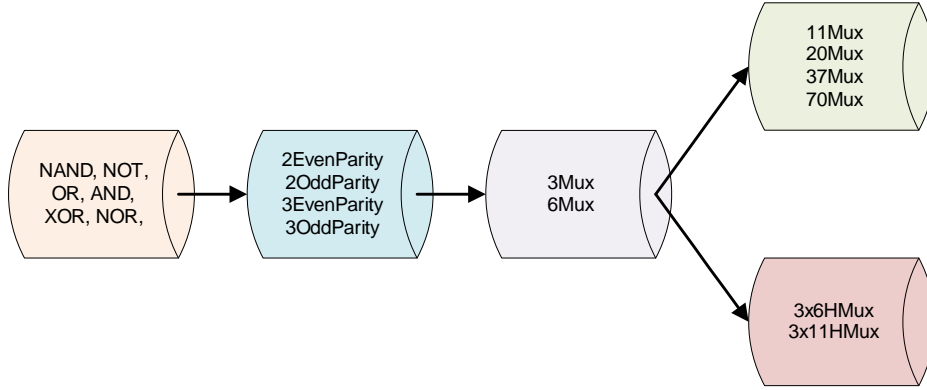


Figure 5.3: Training regimen for XCSCF3. After having learned the Boolean operators, the Parity problems, the 3-bit and the 6-bit multiplexers, there are two training branches to show that the rules learned by the system can be used in a related domain.

number of total DRs produced. The final population of DRs will be added to the network of functions as a brand new DRs function and saved into a file for later retrieval. The system will give it the same tag as for the CF-based rule-set, thus linking the DRs with their CF equivalents.

The new CFs will be learned in the standard way as has been described in the previous chapter. In fact, the proposed system shares many details in common with XCSCFC, specifically the method for feature creation via the CFs [49]. The proposed system will create new CFs for each of the condition bits, while maintaining the original binary alphabet for the action part of the classifiers.

### 5.2.1.2 Benchmark XCSCFC Training

To evaluate the performance of XCSCF3, the XCSCFC technique was chosen as the benchmark for comparison, as it is the state of the art in CF-based systems. Also, since XCSCF3 is an extension of XCSCFC, it will help determine the performance against a closely related technique. The training for XCSCFC was slightly different from that of the proposed system.

**Algorithm 5: Rules Compaction - Offline**


---

**Data:** The final population  $[FP]$  of classifiers  $cl$  with enough experience and always correct.

**Result:** A population of ternary rules translated from  $cl$ .

```

1   $n \leftarrow$  number of  $cl$  in  $[FP]$ 
2  for  $i = 1$  to  $n$  do
3      /* Process all eligible classifiers. */
4       $x \leftarrow$  number of condition bits in  $cl$ 
5      for  $h = 1$  to  $x$  do
6          /* Process each condition bit. */
7           $cf \leftarrow$  code fragment from  $[q]$  indexed at  $cl.condition[h]$ 
8          if  $cf \neq$  'don't care' code fragment then
9              /* Array element will be permuted. */
10              $TempList[currentBit] \leftarrow 0$ 
11          else if  $cf \equiv$  'don't care' code fragment then
12              /* Array element set to constant - will not be permuted */
13               $TempList[currentBit] \leftarrow NOOPERATION$ 
14          end
15      while unprocessed bits do
16          /* Evaluate current classifier using TempList. */
17           $TempList[currentBit] \leftarrow NextPermutation$ 
18          if All code fragments  $\equiv 1$  then
19              /* Current State is a valid candidate DR. */
20              Add  $TempList$  to  $currentRule$ 
21          end
22      end
23   $N \leftarrow 0$  /* DRs Counter */
24  while  $numSubsumed > 0$  AND  $N < MAX\_LOOPS$  do
25      /* Continue until the number of DRs subsumed is 0 or the maximum
26      number of loops is reached. */
27      if  $currentRule$  is duplicate then
28          ignore  $currentRule$ 
29      else if  $currentRule$  is subsumable then
30          subsume  $currentRule$ 
31      else if  $currentRule$  is subsumer then
32          subsume existingRule
33      update  $currentRule$  numerosity
34       $N \leftarrow N + 1$ 
35  end

```

---

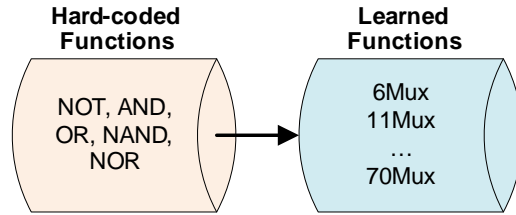


Figure 5.4: Training regimen for XCSCFC for the multiplexer problems. The system is provided with a series of hard-coded functions. Then it is tasked with learning the multiplexer functions sequentially.

This is because XCSCFC was given hard-coded functions, while XCSCF3 will learn rule-sets as functions. The emphasis of this work is to determine the scalability of LCSs, which was tested through the hidden multiplexer problem. Hence effort was put into aligning the training paths for both systems. This alignment is important because it shows that the methods are not overfitted to a single outcome, but can address a variety of problems in a similar domain. Also, unlike the proposed system, XCSCFC utilizes numerous hard-coded functions provided *a priori*. Figure 5.4 illustrates that XCSCFC was provided with five hard-coded Boolean functions. It then uses them to learn the 6-bit multiplexer. Then it uses the hard-coded functions, along with the newly learned CFs, to learn the next multiplexer problem, and so on until it has learned the 70-bit multiplexer. It should be mentioned that for the multiplexer problems this system was not provided with the XOR function, nor the Parity problems <sup>1</sup>. This was in order to adhere as closely as possible to the original implementation of XCSCFC. However, the XOR and Parity problems can provide helpful knowledge blocks when learned, see Section 4.3.3.

Figure 5.5 illustrates the training regimen for the hidden multiplexer problems. The training begins with the even and odd parity problems while relying on a number of hard-coded functions that were provided for the system. It then proceeds to learn the 3-bit and 6-bit multiplexers

<sup>1</sup>This makes a direct comparison with XCSCF3 problematic.



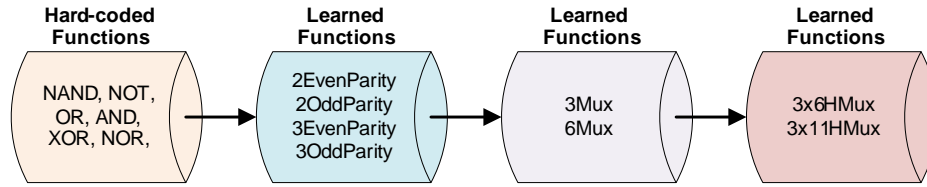


Figure 5.5: Training regimen for XCSCFC for the hidden multiplexer problems. The system is provided with a series of hard-coded Boolean functions. Then it is tasked with learning the Parity functions. Finally it learns the hidden multiplexer problems.

before learning the hidden multiplexer problems.

### 5.2.1.3 Creation of Offline Distilled Rules

After the final population of a problem has been discovered, the system will begin the translation process, see Section 5.2.1.1. During this step, each CF rule is changed into ternary rules using the alphabet  $\{0, 1, \#\}$ . Throughout the translation process, the system eliminates all duplicates to simplify the DR population. One important consideration that the system makes is determining which classifiers are valid for further processing. This is done by comparing their accuracy, experience and prognostication. If the values are acceptable, then the classifiers are processed bit by bit. In this respect, each CF that is included in the condition part will be evaluated against a random set of environment messages. Only non-don'tCare CFs are considered, as it is known that don'tCare CFs always return 1.

The CFs are evaluated using the sequential environment messages produced by DRs method and their result is stored temporarily. If all of the results are 1, i.e. all the CFs match, then the current environment message becomes a potential *candidate* DR. These will be processed further. As the *candidates* are being accumulated, the system also tests if any of them can be subsumed. This process continues until all the CF rules that qualified to be processed have been tested against a predetermined number of envi-

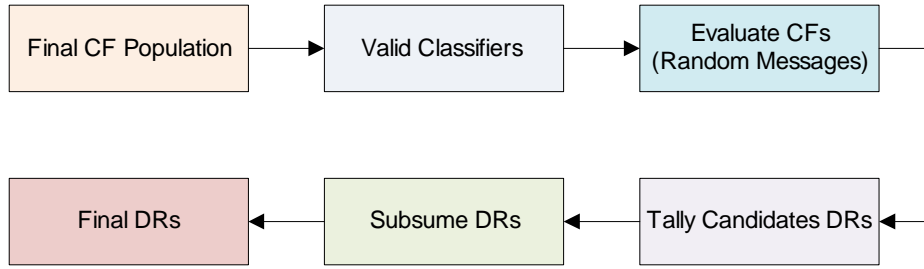


Figure 5.6: Offline Distilled Rules creation process. The technique processes valid classifiers from the final population and finishes with their translation into a ternary alphabet.

ronment messages (the user determines the number of messages). This is a sequential process and ignores any bits considered insignificant to the CF, see Section 5.2.1.1. For example, if a CF in a 6-bit problem were to have D0 and D3 at its leaf nodes, this means that the message bits {D1, D2, D4, D5} do not need to change as the CF is being tested. The first message string would be {0 NOOP NOOP 0 NOOP NOOP} and the last message string will be {1 NOOP NOOP 1 NOOP NOOP} <sup>2</sup>. This means that only a subset of all the possible 6-bit permutations will be used for testing the CF, which reduces the sample space. Once this process is complete, the resulting list of DRs is the translation of the original CFs into a ternary alphabet, see Figure 5.6.

## 5.2.2 Results of Offline Distilled Rules Experiments

The results will be presented completely prior to interpretation so that analogies can be drawn across experiments. The training began with the Boolean operators and proceeded with the multiplexer and hidden multiplexer problems. The results were compared with those obtained from XCS and XCSCFC, as they are comparable systems suitable for the role of

<sup>2</sup>NOOP means that the bit is insignificant and will not be changed during the testing of the CF.

benchmark.

The number of training examples was chosen based on empirical evidence and varied with the problem. It appears on the X-axis on all graphs while the Y-axis represents the performance observed. Performance was measured as the percentage of correct classification during the last 1000 exploit instances, which will help in creating a clear and detailed results plot. All the experiments were run 30 times independently, which is a standard practice for EC experiments. The reward scheme was 1000 for a correct classification and 0 for an incorrect one. Both, GA subsumption and action set subsumption were active and the type of mutation was niche mutation. Both types of subsumption will help in scalability by limiting the number of classifiers present in the final population. Additionally, the systems used two point crossover. This type of mutation is helpful for multilevel types of problems. The success of the algorithm will be measured by using the following methods:

- (1) CF Rules to DRs, i.e. Table 5.2
- (2) Time taken
- (3) Scalability of the technique
- (4) Performance check on other domains to ensure sampling has not introduced errors

The first test will be checking the performance against known experimental results.

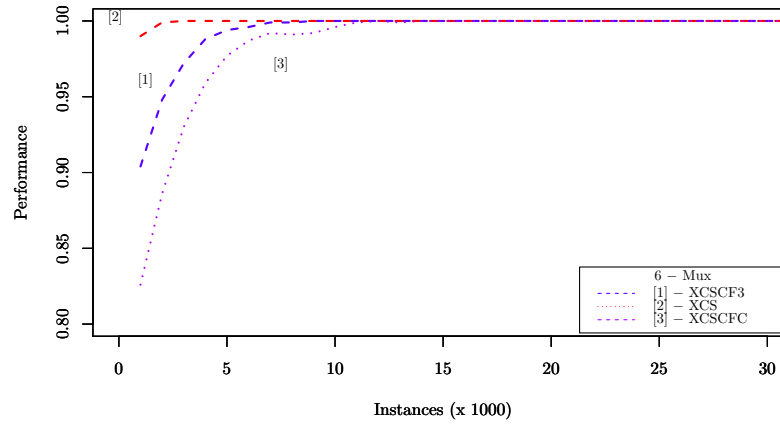
### 5.2.2.1 Multiplexer Problems

The results for the 6-11 bit multiplexer are shown in Figures 5.7(a) and 5.7(b). They illustrate that XCSCF3 has increased performance as the problem scales. It also performs better than XCSCFC during both of these experiments. Initially, the proposed technique under-performs in relation

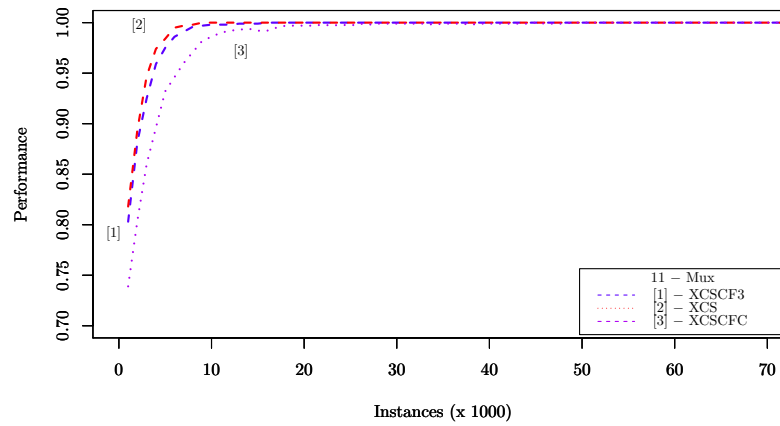
to XCS, however during the 11-bit multiplexer problem, it demonstrates performance similar to XCS. The 6-bit multiplexer problem highlights the benefit of XCS' simple alphabet. However, XCSCF3's combination of CFs and DRs provides increased performance, shown by the 11-bit multiplexer plot. XCSCF3 also performs better than XCSCFC for both problems, showcasing the benefits of combining the rich CFs-based rules with simple DRs.

Figures 5.8(a) and 5.8(b) illustrate the results for the 20 and 37-bit multiplexer problems. Here it is evident that XCSCFC has surpassed the performance of both XCS and XCSCF3 in terms of training instances required to converge. For the 20-bit multiplexer problem, the performance of both XCSCF3 and XCSCFC is very similar, albeit the proposed system lags behind slightly. XCS is having difficulty in finding useful classifiers evidenced by the graph as it evolves a final population. For the 37-bit multiplexer problem, XCSCFC has outperformed the other two systems. However, XCSCF3 demonstrates a higher level of performance compared with XCS. XCS demonstrates periods of inconsistent performance, as evidenced by the graph as it slowly rises to finally converge with about 400 000 training examples.

Figure 5.9 shows the results for the 70-bit multiplexer problem. It indicates that the trend encountered in the previous two multiplexer problems continues. XCSCFC converged with about 500 000 training examples, XCSCF3 required slightly less than 1 500 000 examples, while XCS needed more than 2 000 000 – which is not shown on the figure. This was done to keep the plot as clear as possible. XCSCF3 was incapable of converting the final CF population into DRs in a tractable amount of time. Therefore the system could not process the next problem, i.e. 135-bit multiplexer. Also, XCSCF3 required more iterations than XCSCFC because the DRs are not optimal, i.e. a better mutation operator is needed.

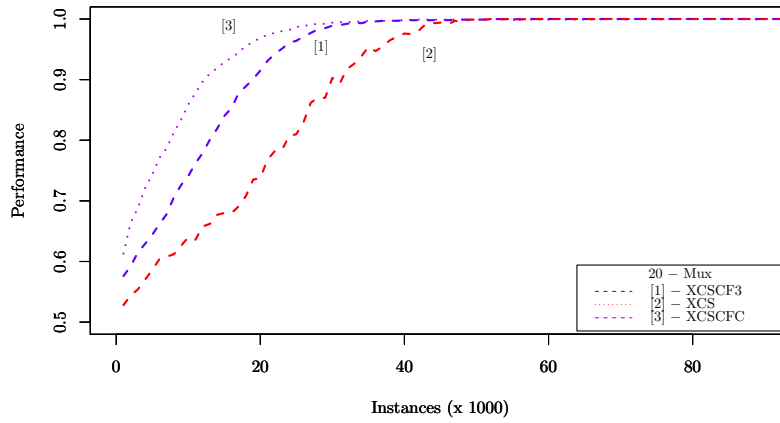


(a) 6-bit multiplexer problem.

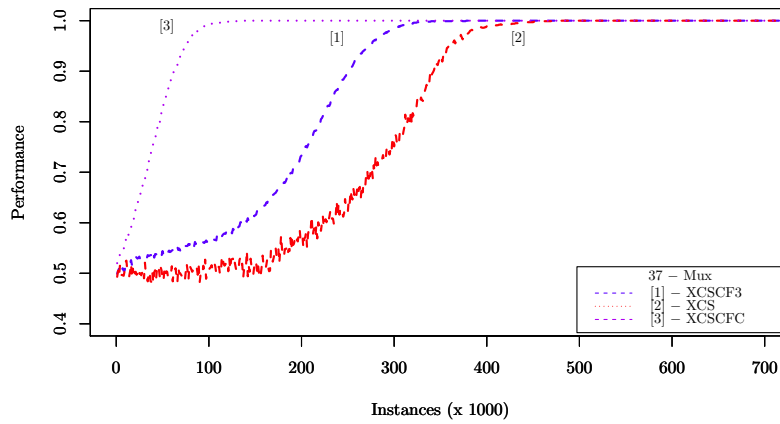


(b) 11-bit multiplexer problem.

Figure 5.7: Results for the 6-bit and 11-bit multiplexer problems using XCSCF3, XCS and XCSCFC. Note: Y axis scale differences for visual clarity.



(a) 20-bit multiplexer problem.



(b) 37-bit multiplexer problem.

Figure 5.8: Results of the 20-bit and 37-bit multiplexer problems using XCSCF3, XCS and XCSCFC.

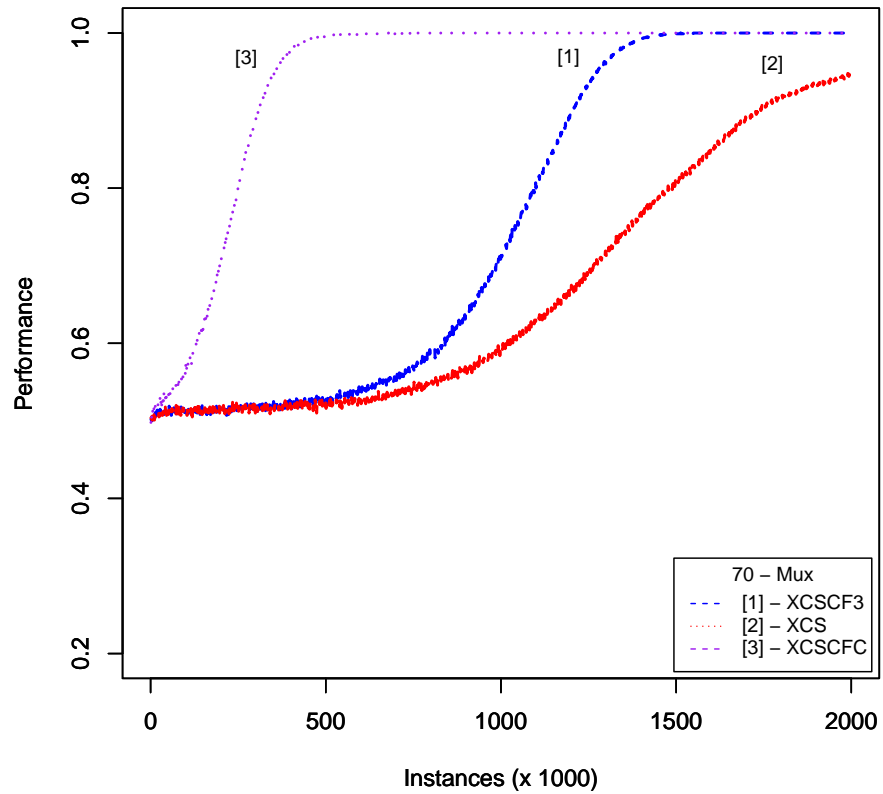


Figure 5.9: Comparison between XCSCF3, XCS and XCSCFC for the 70-bit Multiplexer problem.

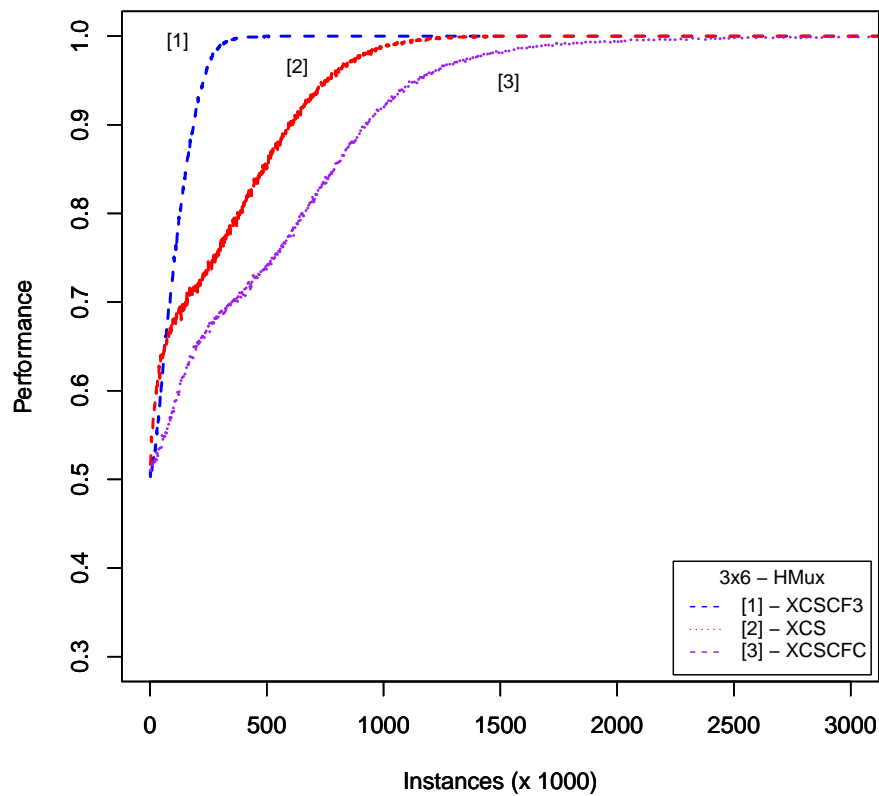


Figure 5.10: Comparison between XCSCF3, XCS and XCSCFC for the 3x6 bit Hidden Multiplexer Problems. The numbers between the brackets are for identifying the curves.



### 5.2.2.2 Hidden Multiplexer Problems

Figure 5.10 shows the results for the 3x6 hidden multiplexer problem. These results are interesting because they are similar to the early experiments for the multiplexer problems, i.e. compare with Figures 5.7(a) and 5.7(b). The aforementioned plots show that XCSCF3 performed better than XCSCFC with respect to the number of training instances needed to converge fully. Additionally, the order of the systems compared with Figure 5.9 is different, where XCSCFC converged with the least number of training instances. This was due to the fact that XCSCF3 was no longer producing an optimal population of DRs, in fact it failed to learn the DRs for the 70-bit multiplexer. On the other hand, XCSCFC had accumulated useful knowledge blocks that helped it learn the 70-bit multiplexer. The proposed system exhibited better performance than the other two benchmark systems. Surprisingly, XCS performed better than XCSCFC for these experiments and this is attributed to XCSCFC's incapability to effectively combine building blocks of knowledge to solve the lower level parity problems, while XCS' simple ternary alphabet gave it an advantage. Figure 5.11 shows the results for the 3x11 hidden multiplexer problem. Here the performance of XCSCFC has improved so that it is now ahead of both the other systems in terms of training instances. In fact, XCS was incapable of learning the problem because its ternary representation is not scalable. XCSCF3 converged with slightly more training instances than XCSCFC.

## 5.2.3 Interpretation of Results

### 5.2.3.1 Multiplexer Problems

All three systems being compared solved the 6-11 bit problems, with XCS demonstrating a slight advantage over the other two systems, in terms of required number of training instances. The ternary condition part of XCS provided an advantage over the complex conditions of the two, CF-based systems. The simple and efficient ternary alphabet was superior to

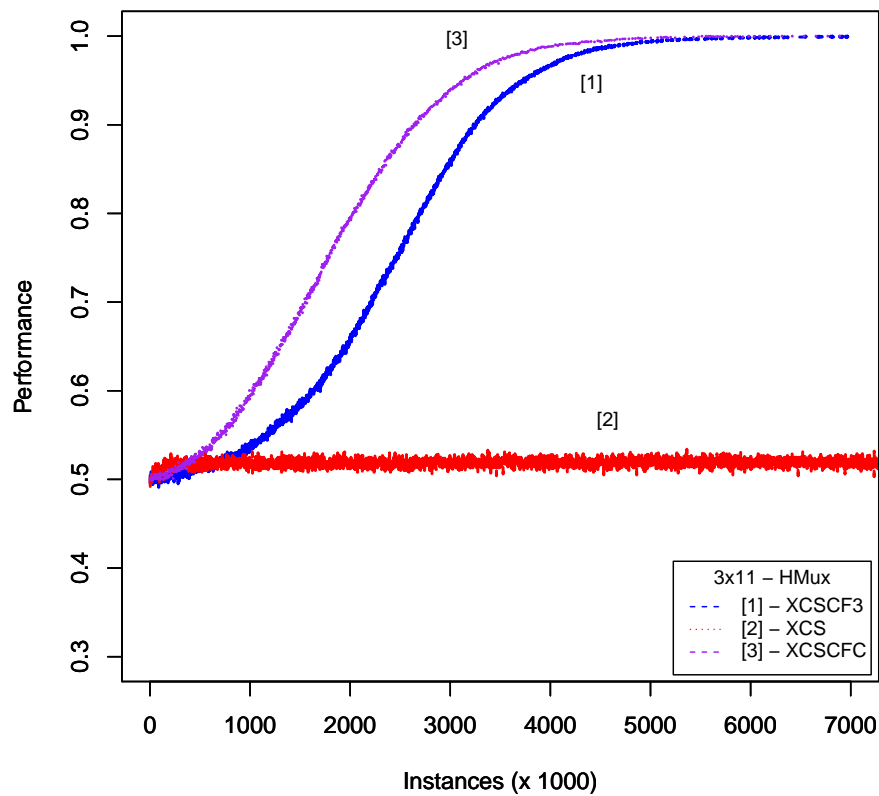


Figure 5.11: Comparison between XCSCF3, XCS and XCSCFC for the 3x11 bit Hidden Multiplexer Problems. The numbers between the brackets are for identifying the curves.

the complex and rich CF expressions. XCSCF3 performed better than XCSCFC for these same problems as well, meaning that the learned functionality has benefited the proposed system while the hard-coded functions of XCSCFC may have hindered it from converging quickly due to having less resources.

When scaling to the 20-bit and 37-bit multiplexer problems, XCSCFC required the least number of iterations of the three systems. For the 20-bit problem, XCSCFC was slightly ahead of XCSCF3 but by the 37-bit problem the advantage had at least tripled, in terms of required iterations. XCSCFC required about 90 000 training examples while XCSCF3 required about 300 000 training examples and XCS required about 400 000 training examples. Another observation of the results is that up to and including the 20-bit multiplexer, all the curves produced by the experiments increased monotonically as they slowly converged on 100% performance. During the 20-bit experiments, XCS produced a rough graph, which is indicative of problems in the learning. It is considered that the rough graphs on the 20-bit and 37-bit experiments are symptomatic of difficulty in learning and possibly the *cover-delete* cycle [19]. During the 37-bit experiments, XCS exhibits more learning difficulty than during the 20-bit experiments. This is because the simple ternary alphabet used in the representation is not as efficient as the CF-based representations for this scale.

The sequential training utilized for the XCSCF3 means that if at any stage a sub-optimal final population of DRs is produced, this will impact the performance during the subsequent problems. This is evidenced by the fact that XCSCF3 is incapable of learning the DRs for the 70-bit multiplexer. Since the DRs learned as a new function are reused by LCSs for future problems, if there are too many specific DRs, this will cause a degradation in performance when evaluating the CFs. This happens because the CF rules are used to generate the DRs and hence there exists an indirect mapping between the DRs and the CFs in reverse. In other words, just like CFs were used to produce DRs, the DRs can be used within CFs dur-

ing the evaluation of potential matchset members. If there are too many DRs (over specific) then they could be used in CFs which do not lead to maximally general conditions that are also accurate.

### 5.2.3.2 Hidden Multiplexer Problems

The hidden multiplexer experiments demonstrated the power of CFs combined with DRs. During the 3x6 hidden multiplexer experiments, see Figure 5.10, the proposed system was able to combine the necessary functions to rapidly converge before the other two benchmarks. Surprisingly, XCS, with its simple ternary representation, was capable of converging with less training examples than XCSCFC. The 3x6 (18-bit) hidden multiplexer is a challenging problem and therefore demonstrated how the mixed representation of XCSCF3 was superior to a ternary and to a CF-based representation. Both of XCS and XCSCFC exhibited an early and rapid rate of learning, visible from the graph, and then both settled into a slightly slower rate of learning as they both converged.

During the 3x11 hidden multiplexer experiments, XCSCFC demonstrated better performance than the proposed system by requiring approximately 400 000 training instances while XCSCF3 required about 450 000, see Figure 5.11. This is indicative that the combination of CFs and DRs in XCSCF3 has lost its advantage over XCSCFC at this scale. For the 6-bit multiplexer experiment, the proposed system produced 11 DRs, see Table 5.1. This number of rules is much smaller than the ideal rules for this problem. It is also lower than the number of rules produced by XCS. For the 3x6 hidden multiplexer experiments, an arbitrarily selected run 8 produced 1 197 DRs while for the 3x11 hidden multiplexer there were 37 166 DRs produced. This increasing trend can be explained in part by the growing complexity of the problems, but also there is an increasing number of sub-optimal DRs being learned as well. This is even before taking into account the difficulty of the hidden multiplexer problem, which naturally requires more DRs than a corresponding multiplexer problem, i.e. with

XCSCF3	Ideal	XCS
0 0 0 # # # : 0	0 0 0 0 0 0 : 0	0 1 # 0 # # : 0
0 0 1 # # # : 1	0 0 1 0 0 0 : 1	0 0 0 # # # : 0
0 1 # 0 # # : 0	...	1 1 # # # 1 : 1
0 1 # 1 # # : 1	0 1 0 0 0 0 : 0	1 0 # # 1 # : 1
★ 0 # 0 0 # # : 0	0 1 0 1 0 0 : 1	1 1 # # # 0 : 0
1 0 # # 0 # : 0	...	0 0 1 # # # : 1
1 0 # # 1 # : 1	1 0 0 0 0 0 : 0	0 1 # 1 # # : 1
★ # 1 <u>0</u> 1 # 1 : 1	1 0 1 0 1 0 : 1	1 0 # # 0 # : 0
1 1 # # # 0 : 0	...	0 # 1 1 # # : 1
1 1 # # # 1 : 1	1 1 1 1 1 0 : 0	# 1 # 0 # 0 : 0
◇ 1 1 <u>1</u> <u>1</u> # 1 : 1	1 1 1 1 1 1 : 1	# 0 0 # 0 # : 0
		1 # # # 0 0 : 0
		# 0 1 # 1 # : 1
		# 1 # 1 # 1 : 1
		1 # # # 1 1 : 0
		# 1 # # # 1 : 0

Table 5.1: Distilled Rules produced by XCSCF3 for the 6-bit Multiplexer problem. ‘\_’ denotes over-specific bits. ‘★’ indicates alternate rules. Alternate rules are one of a number of alternative rules in a population that are equally valid/accurate (but may not be as compact or as intuitive as the ideal rule set). ‘◇’ indicates rules that should be subsumed. The table with the Ideal heading shows the simplest, correct rules. The table with the XCS heading demonstrates the final population of rules produced by XCS.

similar condition length. Therefore, an increasing number of sub-optimal DRs gave rise to even more sub-optimal progeny as the problems scaled.

Table 5.2 illustrates a sample of the DRs produced by the 3x6 hidden multiplexer problem, 1 197 was too many to display. This sample was chosen as it demonstrates the main advantages and disadvantages of the technique. The first rule is an accurate rule which produces the correct action. This rule is evaluated by applying even parity to the first six bits from the left, three bits at a time, which produces the address bits for the

multiplexer part. In this case that will be [0 1]. This translates to an address of '1' in decimal numbers. The first three parity bits are [0 0 1]. The three Parity bits that compose the second address bit are: [0 0 0], which after applying odd Parity gives action '0'. The same process applies to the next four rules which are all accurate as well. The next three sampled rules are interesting in the sense that the data bit is composed of the last three bits, however a number of bits that are not part of the address or the data bit are specific. This means that the breadth of the problem space covered by these rules is diminished, making it necessary to have additional rules to cover the sample space.

The next two rules are interesting because the system determined that half of the address bits should be "don't cares". This raises the question of what is it about this rule that the system found useful. It is equivalent to 0#00##:0 which is an alternate maximally general, accurate rule. According to Kovacs [53], this type of rule overlaps with elements of the optimal population, therefore it competes with them in order to reproduce and is punished for it. The data bits addressed by the possible values of the address bits do provide the correct action. For instance, the first three bits always produce '0' while the next three bits will produce either '0' or '1'. These possible values map to the 7th-12th bits. These always produce the correct action of '0'.

The next to last rule contains extra bits that are specific and do not add to the knowledge of the problem space. In addition, this type of rule makes it more difficult for the agent to learn the problem. As the CFs are evaluated during matching, the extra specific bits make it more likely that a correct rule will be rejected simply because one of the non-relevant bits did not match. It could be argued that this effect will be dampened by the production of maximally general rules in the final population. However, the role of the DRs is to facilitate a faster evaluation of the CFs, therefore any DRs that are too specific hinder learning.

The last rule displays the same structure as some mentioned above, as

Table 5.2: Distilled Rules produced by XCSCF3 for the 3x6 Hidden Multiplexer problem. ‘ $\_$ ’ denotes over-specific bits. ‘ $\triangle$ ’ indicates accurate, maximally general rules. ‘ $\star$ ’ indicates alternate rules that are accurate and overly specific.

XCSCF3																			
$\triangle$	0	0	1	0	0	0	#	#	#	0	1	0	#	#	#	#	#	#	: 0
$\triangle$	0	0	1	0	0	0	#	#	#	0	0	1	#	#	#	#	#	#	: 0
$\triangle$	0	0	1	0	0	0	#	#	#	1	0	0	#	#	#	#	#	#	: 0
$\triangle$	1	1	1	0	0	0	#	#	#	0	0	0	#	#	#	#	#	#	: 1
$\triangle$	1	1	1	0	0	0	#	#	#	0	1	1	#	#	#	#	#	#	: 1
...																			
	1	0	1	0	0	0	#	<u>1</u>	#	#	#	#	<u>1</u>	<u>0</u>	<u>0</u>	0	1	0	: 0
	0	0	0	#	1	0	#	<u>1</u>	#	#	#	#	<u>1</u>	<u>0</u>	<u>0</u>	0	1	0	: 0
	1	0	1	1	0	1	#	<u>1</u>	#	#	#	#	<u>1</u>	<u>0</u>	<u>0</u>	0	1	0	: 0
...																			
	0	1	0	#	#	#	1	0	0	0	1	0	#	#	#	#	#	#	: 0
	0	1	0	#	#	#	1	0	0	1	1	1	#	#	#	#	#	#	: 0
...																			
$\star$	1	0	1	1	1	1	#	#	#	0	1	0	1	0	0	0	#	#	: 0
...																			
	1	1	1	#	#	0	#	#	#	0	0	0	#	#	#	#	1	#	: 1

it contains don’tCares in the address bits. This rule produces the correct action of ‘1’ when the address bits are  $\{0, 1\}$ . However, when the address bits are  $\{0, 0\}$ , this addresses a don’tCare, i.e. the correct action only part of the time. This promotes the creation of incorrect rules during the matching stage. This in turn will degrade performance.

Table 5.3 illustrates a random sample of the results for the 3x11 hidden multiplexer experiments. As has been stated above, the rules sampled here are not optimal. However, there is a useful pattern visible within the population. It is apparent that the system has learned that it is the

Table 5.3: Distilled Rules produced by XCSCF3 for the 3x11 Hidden Multiplexer problem.

XCSCF3																																				
0	1	1	0	1	1	0	1	0	1	0	#	#	1	1	0	0	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	0	#	:	1
0	1	1	0	1	1	0	1	0	1	0	#	#	1	0	1	0	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	0	#	:	1
0	1	1	0	1	1	0	1	0	1	0	#	1	#	0	0	1	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	0	#	:	1
0	1	1	0	1	1	0	1	0	1	0	#	1	#	1	1	1	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	0	#	:	1
0	1	1	0	1	1	0	1	0	1	0	#	#	0	1	0	0	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	1	#	:	1
0	1	1	0	1	1	0	1	0	1	0	#	#	0	0	1	0	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	1	#	:	1
0	1	1	0	1	1	0	1	0	1	0	#	0	#	0	0	1	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	1	#	:	1
0	1	1	0	1	1	0	1	0	1	0	#	0	#	1	1	1	#	#	#	#	#	#	#	#	#	#	#	#	1	0	1	#	1	#	:	1
0	0	1	1	1	0	0	0	0	0	0	#	#	#	#	#	#	#	1	0	1	#	#	0	#	#	#	#	#	1	#	#	#	#	:	1	
0	0	1	1	1	0	0	0	0	0	1	#	#	#	#	#	#	#	1	0	1	#	#	0	#	#	#	#	#	#	#	#	#	#	:	1	
1	#	1	#	0	1	1	0	0	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	1	1	0	#	#	#	:	1
#	#	1	1	0	1	#	0	1	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	1	1	0	#	#	#	:	1
...																																				

rules with specific bits towards the left side that are more useful. Also, along with this pattern, it is apparent that the rules have specific values corresponding to the multiplexer data bit as well as other bits around it. This means that there is a level of learning taking place which could lead to an optimal translation of the CF based rule-set. This is due to the reinforcement learning mechanism which favors those rules which consistently forecast the reward to be obtained.

The problem with increasing numbers of sub-optimal DRs has been thought to have a dual nature. The first one, as discussed above, has to do with the propagation of sub-optimal descendants of the current population of DRs. Since it has been shown that the DRs share a mapping to their CF progenitors. The other cause has to do with the fact that the proposed system does not have a grading system to discern which CF-based rules are producing positive results and will ultimately help produce optimal DRs. This issue is beyond the scope of this work, but is important to note as it provides an opportunity for improvement.



### 5.2.4 Summary of Offline Distilled Rules Extraction

The offline DRs algorithm was shown to provide benefits for shortening the potentially long chains of CFs that can form in LCSs. The technique consists of an offline process that utilizes a mutation operator to translate non don'tCare CFs into ternary rules called DRs. The simpler expressions encompassed by this type of rules means that there is a shortcut to evaluating long CFs.

XCSCF3 exhibited success with the early problems. There was also good performance by XCS. However, as the problems scaled, XCSCFC demonstrated a superior capability than XCS for scaling to more difficult problems. Certain challenges were discovered in the proposed technique, XCSCF3, in that sub-optimal populations of DRs were produced as the problems became more difficult. This had to do with the mutation operator being used.

The benefits exhibited by the technique confirm that further scaling could be realized with adjustments in the training and the algorithm. At this stage the main restrictive factor remains the post-processing that gives rise to the DR population. It is hypothesized that an online compaction technique could provide increased scalability.

## 5.3 Online Distilled Rules Extraction

The first attempt at compacting CFs to DRs in an offline manner (in this chapter) identified several points of interest. It was possible to produce accurate DRs for small problems that were equally effective, while being more efficient. The technique also exhibited a number of limitations; first and foremost were the increasingly prohibitive running times required to produce the new DRs. Also, as the problem size increased, the quality of the DRs produced was related to the number of environmental messages produced by the mutation operator. This means that as the problem scales,

the running times will grow ever larger.

The aim of the novel work in this section is to introduce an online process to improve the efficiency of the DR creation process, without reducing the effectiveness of the CF approach. This two-step process of translating CF-based rules to ternary rules is considered more advantageous to either single step process of creating just CF rules or ternary rules. The ternary rules are compact and efficient in operation, but do not scale well as they do not represent the patterns present in a domain concisely, i.e. they often require multiple rules to be found and interact to describe a search space. The CF rules often produce a small number of rules that express patterns precisely, but can take much longer computational time during the match process, because long chains of rules need to be evaluated. The idea of Distilled Rules is to transform expressive rules into fast/efficient rules to speed up operation and enable practical scaling to large domains.

It is anticipated that by streamlining the process that produces the DRs, it will produce a more optimal population of rules (less rules). This is anticipated to reduce the length of CF chains as the problem scales and the learned functionality is transferred to the new problem(s).

### 5.3.1 Method

CFs can produce accurate and general rules. Combined with easy interpretation for humans, they can produce a solution that spans numerous layers of logic. This complexity can impose a high toll on the scalability of a system; this makes some problems intractable. This new compaction technique aims to improve on the previous work on Distilled Rules. The previous work, see Section 5.2, was only capable of extracting DRs up to the 37-bit multiplexer and 3x11 hidden multiplexer problems, while the proposed system is designed not only to solve the aforementioned problems but also to extract the corresponding DRs from the final population of classifiers for the 70-bit multiplexer.

### 5.3.1.1 Proposed System

The aim here is to shorten the length of the CF chains. There exist other methods for creating compacted ternary rule-sets, but this new work aims to use CFs to generate the final compact rules by directly translating each CF rule into multiple ternary rules. The final DRs will be linked to the uncompact CFs, which can be reused in future problems as they encode the knowledge of relationships between the features.

The boot-strap function of NAND needs to be created from a standard XCS system. The rules-set produced is then imported by the proposed system to serve as a seed function. From this seed function, other Boolean operators are learned, such as NOT, OR, AND, and so forth [3].

The proposed system will adhere to the normal XCS method formulation, i.e., create a matchset, choose a valid action, create an action set, execute the action and update the action set based on the reward returned by the environment. In conjunction with the above steps, the system will construct features composed of CFs, following the method of XCSCFC [49]. It will create new CFs for each of the condition bits but will retain the binary alphabet,  $\{0, 1\}$ , for the action part of the classifiers. During the explore/exploit cycle, each of these two stages will play a critical role in the creation and maintenance of the Distilled Rules Network. The overall process is depicted in Figure 5.12.

The Distilled Rules Network refers to the observation that distilled rules are not created in isolation. As they are created from rules containing learned functions, these functions also contain rules (learned rule-sets). These functions are thus built from other functions and so on, which forms a network showing the relationship between functions, e.g. for Boolean functions: NAND creates NOT creates OR creates AND and so forth. Algorithm 6 shows the DRs Network creation process.

During the *explore* phase, if the number of training examples has reached the preset level, then the reward is checked to ascertain if it was the maximum payoff value discovered so far (often the reward is either

**Algorithm 6:** Creation of DRs Network - Online**Data:** Actionset  $[A]$ .**Result:** A Network of Distilled Rules (DRs).

---

```

1  CONST_DR_SAMPLE_BEGIN  $\leftarrow$  0.5 (range 0.5 to 0.95)
2  if Explore then
3      if Iteration  $\geq$  CONST_DR_SAMPLE_BEGIN * MaxProblems And
        reward  $\geq$  maxPayoff then
4          forall classifiers in actionset do
5              if predictionError < epsilon_0 And experience > 1/beta And
                prediction = maxPayoff then
6                  forall condition bits do
7                      if  $\neq$  dontCare then
8                          getCFConditionBits()
9                      end
10                     add the new temporary DR to the DRs Network
11                 end
12             end
13         end
14     else if Exploit then
15         if Iteration  $\geq$  CONST_DR_SAMPLE_BEGIN * MaxProblems then
16             classifierId  $\leftarrow$  next Id from DeleteList
17             Delete classifierId from Network
18             Delete classifierId from DeleteList
19         end
20     else if Classifier being deleted (discovery) then
21         if predictionError < epsilon_0 And experience > 1/beta And
                prediction = 1000 then
22             if Iteration  $\geq$  CONST_DR_SAMPLE_BEGIN * MaxProblems
                    then
23                 Add classifierId to DeleteList
24             end
25         end
26 end

```

---

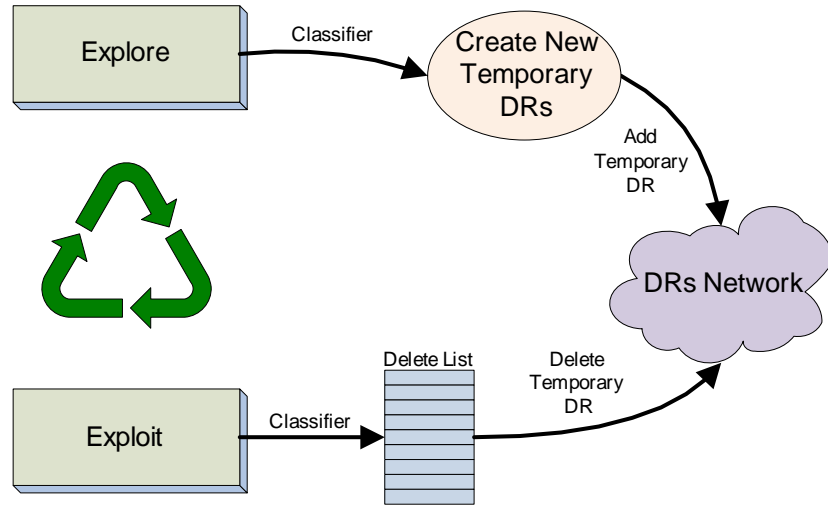


Figure 5.12: Overview of the process which creates and maintains the network of temporary distilled rules. This occurs during the explore/exploit cycles.

0 or 1000 for incorrect or correct classification respectively). If this is the case, then all the classifiers that meet a certain criteria are processed, i.e. have error below a threshold and sufficient experience to be considered accurate.

The processing consists of checking each of the non-don'tCare CFs that compose the condition. Note that there are multiple CFs that represent a don'tCare for the conditions where only the seeded/pre-known don'tCare CFs are checked in this method. In other words, some CFs will evaluate to '1' regardless of their input, just like a normal don'tCare CF. Any evolved, complex trees, that are effectively don'tCares are simply processed as a standard tree, but at additional computational cost. Any terminal bits from the environment message are noted. The next step is to place a '#' character in any of the positions where a CF is a don'tCare. Ones and zeroes 1, 0 are added by evaluating the CF representing the current condition bit and placing its result in the corresponding location of the new DR. At this time there is a temporary DR that is ready to be added to the growing

temporary DRs Network.

If the current phase is *exploit* and the number of training examples has reached the preset threshold, then the current classifiers are checked against a Delete List. This is the list of classifiers that have been deleted by the GA during the normal process of discovering new useful classifiers. Each current classifier that is on the list is deleted from the DRs Network.

If the system deletes any classifiers during the discovery phase, the valid classifier ids are added to the Delete List. This only happens if the number of training instances has reached the pre-determined threshold and if the prediction error, experience and prediction meet the preset criteria.

Once the final population of classifiers has been created and simplified, the post-processing of the temporary DRs in the Network is conducted. In this technique there is less post-processing involved as compared with the off-line DR technique [5]. The reason for this is that a large portion of the processing now takes place during the explore and exploit phases. The first task is to reconcile the temporary DRs attached to each current classifier that was saved throughout the life of the run, this process is described in Algorithm 7. Essentially, the aim of this step is to identify unique DRs, while discarding duplicates.

Another major part of creating the final DRs is to have a genetic operator determine if any of the raw DRs can be combined. This is shown in Algorithm 8. The process centers on examining each unique DR identified so far (these are termed Final DR) and comparing it with each subsequent DR in the list. If the actions match, then each condition bit of the DRs is compared. If there is a corresponding mismatch, then the more general rule will absorb the less general one. During the post-processing state, duplicates are searched for and deleted, as this helps eliminate wasteful comparisons.

**Algorithm 7:** Rules Compaction - Reconcile Network DRs

**Data:** The final population  $[FP]$  of classifiers  $cl$  with enough experience and always correct.

**Result:** A population of ternary rules translated from each  $cl$ .

```

1 forall  $cl$  in  $[FP]$  do
2    $cid \leftarrow cl.Id$ 
3   if  $cid$  in DRs Network then
4     forall DRs attached to  $cid$  do
5       if DR Not duplicate then
6         Store Interim DR
7       end
8     end
9   end
10 end
11 forall Stored Interim DRs (SIDRs) do
12   forall Final DRs (FDRs) do
13     if  $SIDR.action \equiv FDR.action$  then
14       forall condition bits do
15          $MismatchCount \leftarrow$  Number of mismatches
16       end
17       if  $MismatchCount \equiv 1$  then
18         if incoming mismatch is don'tCare then
19           Subsumes existing bit
20         else if existing mismatch is don'tCare then
21           Subsumes incoming bit
22         else
23           new bit is don'tCare
24         end
25       end
26     end
27   end
28   if SIDR not processed then
29     Add SIDR to Final DRs List
30   end
31 end

```

---

**Algorithm 8:** Rules Compaction - Subsumption with more general DRs
 

---

**Data:** A population of Final Distilled Rules (*FDRs*).

**Result:** A population of general Distilled Rules.

```

1 forall Final DRs (FDRs) do
2   if Final DR not subsumed then
3     forall Subsequent Final DRs (SFDRs) do
4       if Subsequent Final DR (SFDR) not subsumed then
5         if FDR.action  $\equiv$  SFDR.action then
6           forall condition bits do
7             if FDR.bit  $\neq$  SFDR.bit then
8               Mismatch  $\leftarrow$  True
9               break
10            end
11            if Mismatch then
12              if FDR more general then
13                SFDR is subsumed
14              else if SFDR is more general then
15                FDR is subsumed
16              end
17            end
18          end
19        end
20      end
21    end
22  end
23 end

```

---



### 5.3.1.2 Usage of Distilled Rules

The distilled rules produced by the system at the end of each run will be reused by the system when solving any subsequent, more difficult tasks, i.e. a Boolean operator or a larger problem. Since the proposed system does not avail itself of hard-coded functions, the DRs will serve as the functions that will be used instead. Each time a CF, forming part of a classifier condition, has to be compared with the message string, e.g. when forming a matchset, the CF sub-tree will be traversed and the appropriate functions at the root nodes will receive their inputs from the terminals at the leaf nodes. These inputs are compared with the list of DRs linked to the aforementioned function. Where all inputs match a particular rule, the linked action becomes the output and potential input for any higher levels in the chain of CFs.

## 5.3.2 Results of the Online Distilled Rules Experiments

### 5.3.2.1 Experimental Setup

The experiments were run 30 times, each having an independent random seed. The stopping criteria was when the programs were exposed to the preset number of training examples. The proposed system was compared with XCSCFC and with XCS. The experiments were single step with the following settings: Payoff 1,000; the learning rate  $\beta = 0.2$ ; the probability of applying crossover  $\chi = 0.8$ ; the probability of using a don't care symbol when covering  $P_{don'tCare} = 0.33$ .

### 5.3.2.2 Experimental Tests

The number of rules produced by the system for the 70-bit multiplexer was suboptimal, in spite of having learned the problem. This appears to be as a consequence of having set the threshold value for processing temporary DRs too high at 95% of total training instances. This was necessary

as the execution times of a problem of this magnitude were anticipated to be prohibitive. The necessary minimum number of general rules for representing a search space of  $2^{70}$  variations is 128. This takes into account the first 6 bits for the address and one more bit for the data. The rest of the bits can all be don'tCares. The quality of the rules produced is mixed as a number of them contained don'tCares where the address bits should be. This produced numerous possible data bits. There are also rules with specific address bits, which is what would be expected, and a number of other specific bits interspersed throughout the rest of the condition. This is undesirable as this type of rule tends to cover less of the problem space than the optimal version, which tends to be more general.

The results for the 70-bit multiplexer problem are illustrated by Figure 5.13. The proposed system performed better than XCS, as was anticipated, but worse than XCSCFC in terms of training instances required to converge. The standard deviation of the proposed system is also visibly larger than XCSCFC but it appears that XCS had a slightly larger deviation.

The 135-bit multiplexer proved to be a far more difficult problem for the proposed system, see Figure 5.14. Although it was provided with similar settings as were given to XCSCFC, it was just beginning to learn when the stopping criteria was reached. This is due to the caliber of the DRs produced from the 70-bit multiplexer problem. It is theorized that the proposed system could learn the 135-bit multiplexer, however it would require far more resources than it was currently given, this also highlights the weakness of the technique in terms of computing resources when the DRs being reused are less than optimal. XCS was unable to learn the problem at all, which was anticipated given the complexity of the problem.

Figure 5.15 illustrates the results for the 3x6 hidden multiplexer. The final results are not surprising in the sense that they are very similar to the results for the offline technique. The online technique was anticipated to be at least as efficient as those for the offline technique. The fact that the

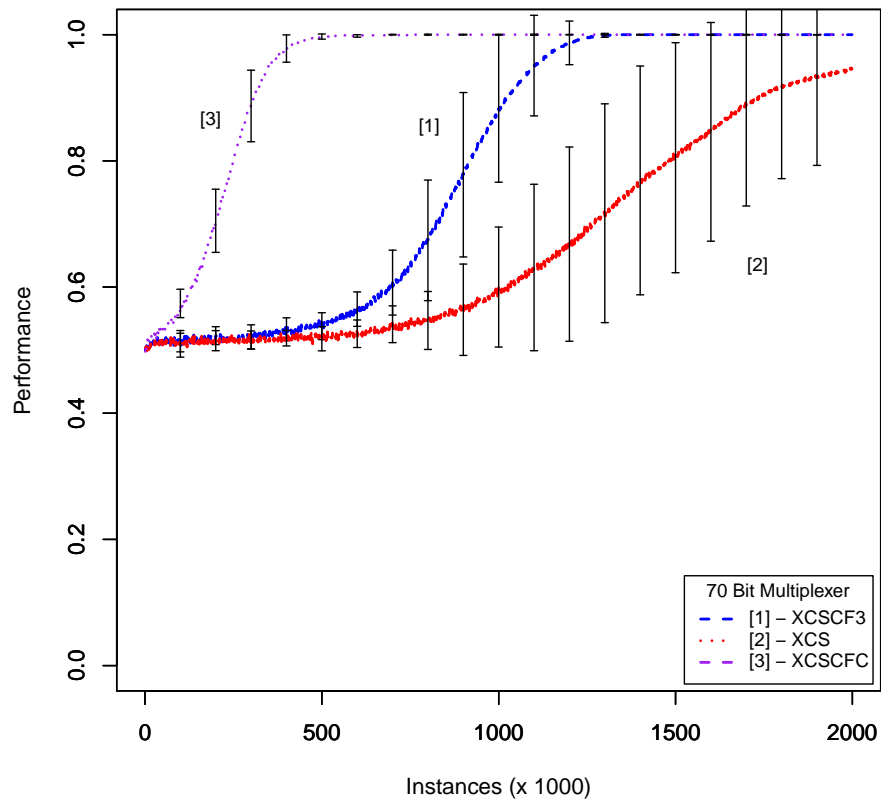


Figure 5.13: Comparison between XCSCF3, XCS and XCSCFC for the 70-bit Hidden Multiplexer Problems using the online DR technique. The numbers between the brackets are for identifying the curves.

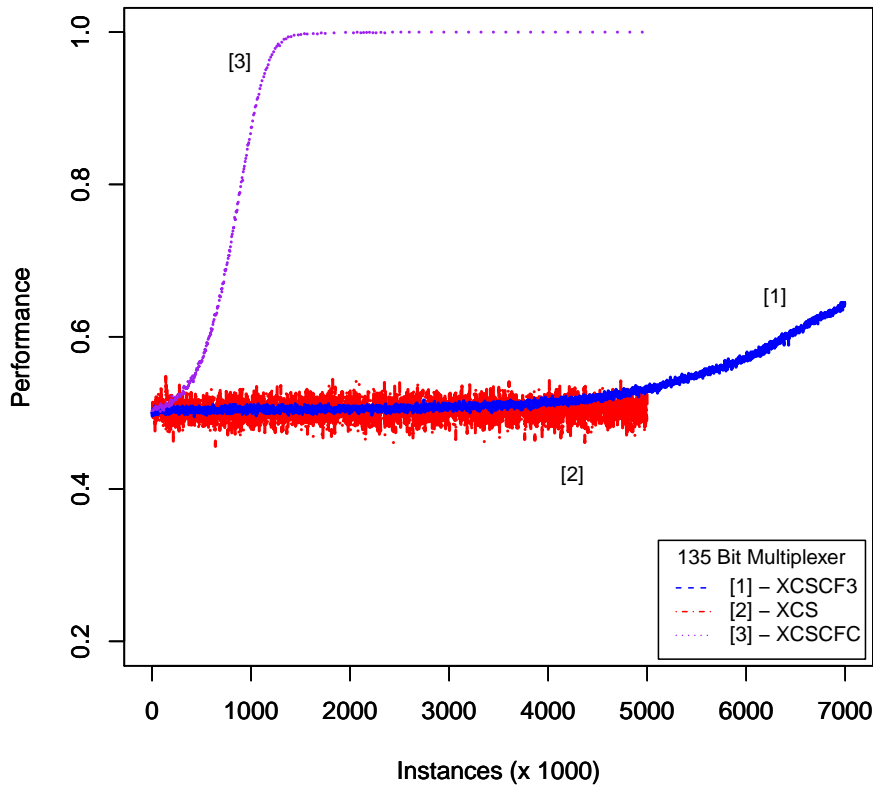


Figure 5.14: Comparison between XCSCF3, XCS and XCSCFC for the 135-bit Hidden Multiplexer Problems using the online DR technique. The numbers between the brackets are for identifying the curves.

online technique produced less DRs on average, supports this assertion.

The results for the 3x11 hidden multiplexer are illustrated in Figure 5.16. Here the performance of all the systems is similar to that for the offline experiments. XCSCFC performed the best, followed by XCSCF3. XCS was not able to learn the problem, as was anticipated.

### 5.3.3 Interpretation of Results

The multiplexer and hidden multiplexer domains provided ample opportunity for the proposed technique to demonstrate the advantages that it exercises over the offline technique. Obviously, there are structural differences between the offline and online techniques that preclude a comparison of like to like, however, the results are encouraging. The reason is that although both systems are different structurally, the fact that their performance was measured against two commonly known systems, i.e. XCS and XCSCFC, is enough to draw fair conclusions.

First of all, the caliber of rules produced by the system was of a better quality, simply because there were fewer of them. For the 3x6 hidden multiplexer problem, there were 812 DRs produced on average by the online system, while the offline system produced an average of 1 319. This implies that the final population of rules covers more of the problem space, on average, than the rules for the offline version. Also, the fact that the new technique is capable of learning the DRs for the 70-bit multiplexer, something that the offline version was incapable of doing, highlights the benefits of the new technique. Of course, there is the new technique's embryonic learning of the 135-bit multiplexer. This is something that the offline DRs technique was not even close to furnishing.

Although the new technique, XCSCF3, has been successful in some ways, it still lacks the performance of XCSCFC. Specifically, by the time the system attempts the 70-bit multiplexer, it is quite clear that it requires at least two times more training examples than XCSCFC. This is not surpris-

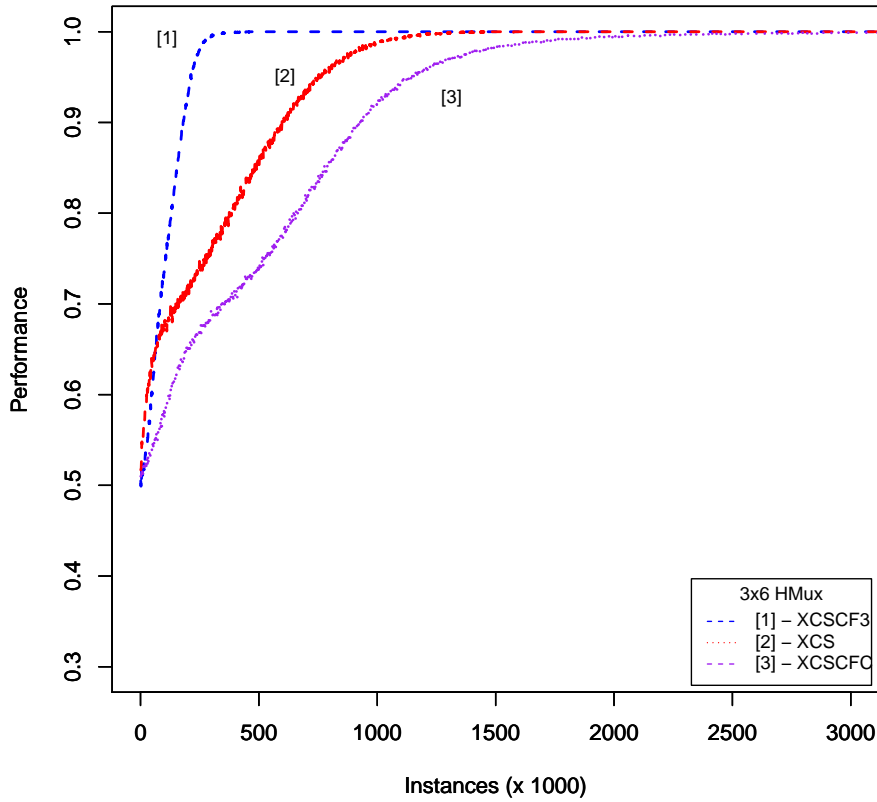


Figure 5.15: Comparison between XCSCF3, XCS and XCSCFC for the 3x6 bit Hidden Multiplexer Problems using the online DR technique. The numbers between the brackets are for identifying the curves.

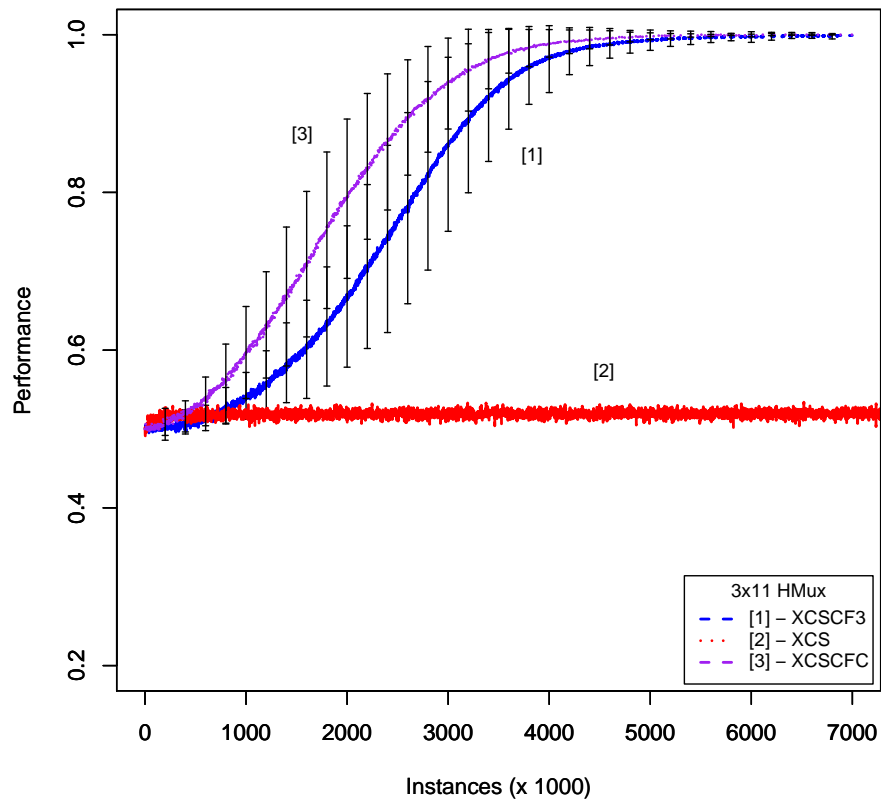


Figure 5.16: Comparison between XCSCF3, XCS and XCSCFC for the 3x11 bit Hidden Multiplexer Problems using the online DR technique. The numbers between the brackets are for identifying the curves.

ing, as the former is not using hard-coded functions but distilled ternary rules instead. That is to say, when evaluating the long chains of CFs, at the function root nodes, the proposed system is using ternary rules. Depending on the usefulness of the rules available, this evaluation could become very time consuming. However, this is not enough to explain the disparity in performance between XCSCF3 and XCSCFC. There must be a property in the DR techniques (both offline and online) which could explain this phenomena.

### 5.3.4 Summary of Online Distilled Rules Extraction

The proposed work was shown to provide improvement in performance while creating a layer of DRs from CF based rules. The technique consists of an online process that gets executed during the explore/exploit stages of the run as well as a post-process stage which produces the final population of CFs. The shortcut afforded by the ternary alphabet  $\{0, 1, \#\}$ , means that long chains of CFs can now be replaced by simple comparisons between the environment situation and the rules pertaining to the CF root nodes being evaluated. The well known limitations of using a ternary alphabet for the representation of the condition part are circumvented by this technique by creating tight linking between the DRs and the CF based rules. The rich and powerful presentation afforded by the CFs, combined with the simplicity of the DRs makes for a useful combination in the problem domains tackled here.

Most of the DRs related processing takes place while the system is learning new rules and testing them. As the new match sets are created, and the action sets are updated, new temporary DRs are created from the well-performing classifiers and stored for later processing. During the exploit phase, the technique deletes any classifier ids that may be in the *delete list*. During the Discovery phase, any classifiers that are deleted by the GA are added to the *delete list*. Once the final population is evolved, the post-



processing takes place. The online version is much less intensive than the offline version described previously in this chapter. It consists of reconciling the temporary DRs into a population that has been expunged of duplicates and that has had a mutation operator applied – this acts like a subsumption mechanism.

XCSCF3 exhibited success with the early problems. There was also some surprisingly good performance by XCS. However, as the problems scaled, XCSCFC demonstrated a superior capability to scale to these more difficult problems. Certain challenges were discovered in the proposed technique in that sub-optimal populations of DRs were produced as the problems became more difficult. This has to do with the mutation operator being used and the inability to address all of the huge number of possible environment messages (for the 3x11 hidden multiplexer).

The benefits exhibited by the new technique confirm that further scaling could be realized with adjustments in the training and the algorithm. At this stage the main restrictive factor remains the post-processing that gives rise to the DR population, making it the primary candidate for customization.

## 5.4 Chapter Summary

In this chapter, two versions of a novel compaction technique were proposed. The reasoning behind a compaction technique for CF based Learning Classifier Systems is twofold: first, although CFs provide various benefits in terms of scalability and power of expression, they can be computationally intensive; second, the number of CF based rules produced as a solution can be very large. Distilled Rules (DRs) provide an alternative to complex CF based rules. They utilize a ternary alphabet which is simple to process and does not contain complex expressions.

The first version of the compaction technique is mostly an offline process that takes place after the final population of classifiers has been pro-

duced by the system. It consists of providing the system with a pre-set number of situations. The classifiers containing some specific amount of experience, accuracy and error are evaluated using the current situation. All the non don'tCare CFs in these classifiers are evaluated and those that are selected are then processed further. The population of DRs is accumulated and checked for redundancy as the process advances. At a later stage, the growing number of DRs is subjected to a mutation operator which functions as a subsumption mechanism. In other words, the temporary DRs are compared in round robin fashion to determine if any rules share a common action. If this is the case, then it is determined if there is at most one corresponding bit that is different between the two rules. If this is true, then the more general rule subsumes the other one. This technique provided certain benefits towards the solution of progressively more difficult problems in the Boolean domain. Ultimately, structural limitations precluded the system from continuing to learn. A more efficient method was required. The next version of the DRs technique is mostly and online process.

The online DRs technique shares similarities with the offline version in that the main objective remains the same. The translation process from CF-based rules to a ternary DR is also similar but with a few caveats. Whereas in the offline compaction version the system was presented with a pre-set number of situations, in this version, the translation process sees the same situations presented to the agent during the explore and exploit phases. The explore phase is when new temporary DRs are created from useful classifiers. They are created via the same process as in the offline version. Then they are added to the temporary DRs network where they remain throughout the run, unless their parent classifier is subsumed. If a classifier is deleted and it meets a criteria, then its id is added to the Delete List. During the exploit phase, the Delete List is checked and one of the classifier ids is selected. All temporary DRs in the network matching this classifier id are deleted. Eventually, when the final population of classi-

fiers has been evolved, the surviving DRs are further processed to produce a population of DRs that can be optimal and duplicate free.

The online version of the DRs introduced additional benefits and facilitated the translation of the DRs for the 70-bit multiplexer, something which had not been possible with the offline version. In spite of having afforded a number of benefits, the online version still has a scalability limit that prevents it from fully learning more difficult problems, e.g. the 135-bit multiplexer. This leaves an opening for further improvement in search of increased scalability, which will be further developed in the next chapter.



## Chapter 6

# Layered/Transfer Learning for CF-based Systems

### 6.1 Introduction

The developments in CF-based LCSs have been shown to increase scalability when learning numerous problem domains (Chapter 4). For instance, it has been shown that reusing learned functionality at the root nodes as well as the leaf nodes of CF sub-trees has its benefits [3]. Also discussed, was the related technique Distilled Rules, that aims to create a compacted layer of ternary rules in order to quicken the evaluation of CF chains (Chapter 5). However, creating a general solution that is universally scalable for certain types of problems is still beyond their capability, e.g. the multiplexer problem. This can be because of the increasingly long chains of CFs, which eventually prevent learning, or because the problem is simply too complex in its sample space for the technique to scale up beyond a certain limit [49]. Alternative traditional EC approaches can not assist, as a number of them throw away learned information between runs. For these reasons, it is necessary to consider the CF technique from a point of view different than traditional EC learning.

Since the unifying thread of this work is scalable learning, it is suitable

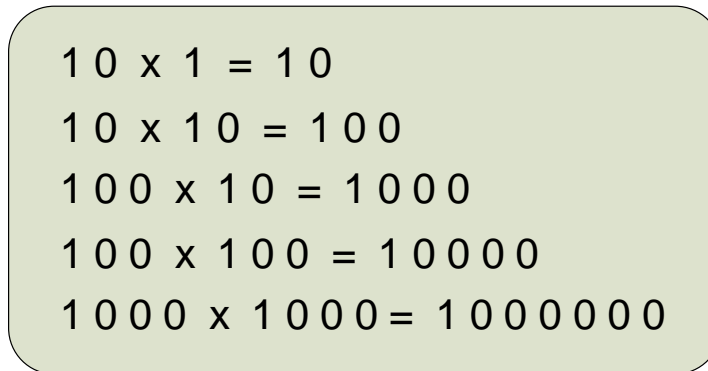

$$\begin{array}{l} 10 \times 1 = 10 \\ 10 \times 10 = 100 \\ 100 \times 10 = 1000 \\ 100 \times 100 = 10000 \\ 1000 \times 1000 = 1000000 \end{array}$$

Figure 6.1: Human method for multiplication shortcut. A person could realize that to multiply by multiples of ten, one only needs to place a one next to the total number of zeroes involved in the multiplication.

to explore how a human approaches certain difficult problems to make learning tractable. During many humans' formative years of education, one of the basic skills taught is the *times* table. Undoubtedly this form of computation seemed daunting to many of the children attempting to master the lesson, however as time passed, specific ancillary skills would have been discovered. For example, the realization that multiplying increasingly larger base ten numbers, that are divisible by 10, is a simple matter of adding zeroes to obtain the answer. Figure 6.1 illustrates the common pattern that emerges, mainly that the answer is given by placing a one followed by the total number of zeros in the numbers involved. A problem in the multiplication domain has been solved by identifying simpler building blocks in the simpler, underlying domain of addition. Granted, this is a trivial example, but it serves to illustrate the drive for the work proposed here, which is to design an agent with the capability to discover the important underlying functionality with associated building blocks that will help it scale to very large problems in the same or a related domain.

One of the research directions currently active in Computer Science is

the decomposition of problems into manageable pieces by software agents [83]. For example, if an agent were tasked with taking part in a ‘football game’, what would be the necessary stages to transform this seemingly intractable and unrecognizable problem into a series of steps, leading to an agent that is capable of taking part in such a game [83]? This hypothetical problem and others like it become more manageable if separated into constituent parts. This implies human involvement, at least in the selection of the said parts. The implication is that by providing the right set of functionality, a difficult problem could be learned in a series of steps.

It is anticipated that enabling a software agent to ‘reason like a human’ as it attempts to solve a problem will provide benefits. The premise is that humans are imbued with natural abilities that help them solve certain types of problems, among which is the concept of the number line [72], as well as humans’ tendency to focus and excel at finding patterns, i.e. regularities. Mimicking a number line would provide an agent with the capability to compare values to determine greater vs smaller, for example. This is a critical step in the solution to problems involving position and values, such as the multiplexer problem. It has been said by Wiener that ‘the world contains certain patterns that are repeated in different types of systems’ [35] , [91]. It is a subset of useful common patterns that this work seeks to learn and reuse. By accruing functionality that contains these regularities, it is anticipated that the resulting functions will form fundamental steps of solutions to larger and more complex problems, thus making it possible for an agent to form solutions to larger, more complex problems than previously possible.

It is hypothesized here that humans scale to complex problems by considering the underlying patterns in such problems that are generated from previous, smaller scale problems, thus transferring knowledge and skills from these problems. This learned knowledge could be from different domain(s) as some of the functionality that is needed may not exist within the problem itself. It is anticipated that when this approach is adopted into

evolutionary computation approaches, it will lead to advances in scalability of those techniques.

The field of Developmental Learning contains an idea known as the Threshold Concept (TC) [26]. This idea conveys the fact that in human learning there exist certain pieces of knowledge that are transformative in advocating the learning of a task without which the learner can not progress [64]. These concepts need to be learned in a particular order, thus providing the student with viable progress towards learning more difficult ideas at a faster pace than otherwise. For instance, humans are routinely taught mathematics in a certain progression; arithmetic is taught before trigonometry and these two are taught before calculus. The empirical evidence indicates that this sequence will be more effective in fostering the learning of progressively more difficult mathematics [26]. In effect, TCs provide a scaffolding upon which principles can be attached that otherwise would be too difficult to learn.

The base framework used for this chapter is CF-based XCS. XCS has consistently demonstrated good performance and is easy to update, compared with other classification techniques. When combined with CFs, XCS has been capable of increased scalability, helping it solve previously intractable problems. Thus, using the CF-based XCS Learning Classifier System provides a common basis on which to gauge any benefits in scalability. The LCS technique has been in use for at least 40 years. They were first introduced by Holland [34]; they were originally cognitive systems designed to evolve a set of rules. LCSs were inspired by the principles of stimulus-response in cognitive psychology [13], [17], [34], [36], [73].

LCSs morphed from being platforms to study cognition and have become powerful classification techniques [62]. An important property of LCSs is their capability to subdivide the problem into niches that can then be solved efficiently [37]. This is made possible by integrating generality in the rules produced. This pressure towards generality means that one classifier could be a solution to more than one problem instance. In other



words, LCSs can divide the problem space into niches, thereby creating smaller ‘problems’ from the whole. This helps in keeping the solution compact, which in turn reduces the search space. In addition, LCSs are a good technique for learning difficult concepts in stages. Depending on the representation used, it is possible for an LCS to learn a series of smaller pieces of a larger, more complex problem.

Despite these advantages, certain problems are difficult to solve, e.g. the multiplexer problem. This is due to fundamental problem characteristics such as epistasis (the importance of certain bits depends on the values of other bits) and heterogeneity (when individual solutions or groups of individuals are independently predictive of the same phenotype). These phenomena were studied by Urbanowicz et al. [87] who introduced the method of attribute tracking for characterizing heterogeneity and interaction in Michigan style LCSs applied to supervised learning problems. This approach is useful and potentially influential, however it considers a complete supervised instance, whereas the proposed work seeks to use reinforcement learning. The technique also seeks to use the ‘divide and conquer’ approach. In other words, the work dealing with attribute tracking considers a single problem while CFs transfer knowledge/functionality between successive problems.

The ‘divide and conquer’ approach is a well known principle in human learning and is based on the notion that the two hemispheres of the human brain can process complex tasks in a parallel fashion. There is also communication that takes place between the hemispheres while performing this type of processing. It has been determined that while the human brain functions in this manner, it exhibits better performance than if the entirety of the tasks were given to only one hemisphere. This is true as long as all the data necessary to solve the problem is given at the same time [10]. In the proposed work, it is intended to go beyond what the Michigan Style LCS currently offers, in terms of niching a complex problem into smaller parts and providing general classifiers that solve those

smaller parts. What is needed is a technique akin to the interhemispheric processing described in [10]. In other words, the proposed technique must be capable of using different parts of its learned functionality to solve the complete problem. In this manner it will resemble the brain while still remaining under the EC umbrella, the different pieces of a problem will be solved by the appropriate functionality.

Although EC techniques have facilitated progress in the field of machine learning, some of them have a fundamental weakness. Each time a solution is produced, the techniques tend to ‘jettison’ any learned knowledge and must start from a blank slate when tasked with a new problem. Reusing learned knowledge has been shown to increase scalability and could provide benefits that decrease the search space of the problem by encoding relationships between environment features [49]. In addition, one of the major goals of this chapter is to discover groups of functions that will map to numerous, heterogeneous structures of problems. This will aid in evolving a compact and optimal set of classifiers at each of the proposed sub-problems [71].

This situation is due to the limitations of CFs; there are too many functions required to learn in one layer. This applies to the proposed work in that it is not possible to learn the target problem as a single functional step, since there are numerous functions that must be combined to achieve a solution; the rigors of CF construction – every new CF has an initial height limit of two – preclude any initial overly rich/complex CF sub-trees. Also, in Layered Learning (LL), the task is decomposed in a bottom-up manner, just like the decomposition anticipated here. Learning occurs separately at each level and the output of each layer feeds into the next layer. Importantly, in LL the decomposition of the task is not automated. This is just like the proposed work; the target problem will be decomposed by a human into a number of sub-problems. However, it will be up to the agent to learn to combine the available functionality in a constructive manner. This means that the agent will be guided by the limitations imposed on

each function: input data types, output data types and numbers of arguments. This will be further complicated as some functions will be required to learn knowledge in different domains, i.e. boolean, integers, real numbers. This will enable knowledge to be transferred from one domain to the next.

The methods outlined a technique for Boolean problems. This technique will be tested on the multiplexer problem due to the following properties: epistasis, high non-linearity and ubiquitousness in research. As such, this type of problem requires learned functionality from many different domains, the target is the Boolean domain. For example, the log and power base two functions will be very important for the solution but they do not appear in the Boolean domain. Some functionality will be provided *a priori*, other functions will be learned online, e.g. the ValueAt learned function will return a Boolean value.

In addition to this, it is important for the technique to learn with a minimal set of training examples. This is because as the problems scale up, they will become intractable at some point. It should also be capable of combining the functionality in new and divergent ways, thereby discovering new solutions [57]. Therefore relevant aspects of the concepts of LL and TL will be combined with the proposed LCS-based technique to facilitate better learning and scalability than previously possible. From LL the technique will use human interaction for breaking the problem into constituent parts. From TL the technique will include the capability of learning knowledge from one domain and then reusing it in a related domain. This combination of relevant aspects will help promote learning of a difficult, monolithic problem by breaking it into sub-problems, each one feeding the next. Another anticipated benefit of the proposed technique is that the number of rules evolved will be small, due to their maximal generality and richness in composite functionality. The maximal generality of the rules is guaranteed due to the fact that XCS strives to achieve rules of this nature. Also, since the CF-based action will contain a rich

combination of functions, it is plausible that one action contains enough functionality to cover the entire action map. This seems to indicate that the representation of the problem plays a large role in enabling a technique to express the necessary patterns which will scale to more difficult problems in a domain.

### 6.1.1 Experiments Chosen

The experiments consist of the multiplexer. This problem was chosen because of its difficulty. It contains epistasis; some bits determine the importance of other bits. The problem is also highly non-linear and it has a long track record in research. In the multiplexer problems, the number of address bits is related to the length of the message string and grows along with the length. The search space of the problem is also extensive enough to show the benefits of the proposed work. For example, for the 135-bit multiplexer the sample space consists of  $2^{135}$  combinations, which is immensely beyond enumerated search [54].

Past work has shown that this type of problem presents a challenge for LCSs such as XCS. Even when combined with CFs, learning happens until the CF chains become excessively long. A number of techniques were developed with this limitation in mind. These techniques deal with compression as well as alternative ways for using CFs, such as learned rule-sets at the root nodes [4], [5]. In other words, the representation played a large role in the aforementioned studies.

One representation (an alphabet for the condition/action encoding) for solving Mux problems is by using Disjunctive Normal Form (DNF). However DNF presents some limitations: as the problem grows, the DNF grows very large as well. The 6-bit multiplexer can be solved but is verbose, see equation 6.1 (the primes indicate negation and 'X' the features (bits) in a multiplexer string) [95]:

00	0000	:	0
00	1000	:	1
01	0000	:	0
01	0100	:	1
10	0000	:	0
10	0010	:	1
11	0000	:	0
11	0001	:	1

Figure 6.2: Humans are capable of recognizing patterns. Therefore they could determine that there exists a relationship between the address bits and the data bit of a multiplexer problem.

$$F6 = X0'X1'X2 + X0'X1X3 + X0X1'X4 + X0X1X5 \quad (6.1)$$

A human would approach the problem differently. They would realize that there is an innate relationship between the address bits and the data bit, even if presented with just a binary input string and no prior knowledge of the underlying problem structure, see Figure 6.2. It is hypothesized that humans scale to complex problems by considering the underlying patterns in such problems that are generated from previous, smaller scale problems, thus transferring knowledge and skills from these precursors. This learned knowledge might be from different domain(s), as some of the functionality that is required may not exist within the problem itself. It is anticipated that when this approach is adapted into EC approaches, it will lead to advances in scalability of these techniques.

### 6.1.2 Chapter Goals

The aim of this chapter is to adapt the principle of threshold concepts, combined with TL and LL, into LCSs to produce general solutions to large scale problems in a complex domain and related domain; this will be demonstrated through the often used benchmark Multiplexer problem. The Multiplexer problem is one that lends itself for research because it is difficult, highly non-linear and has epistasis, e.g. the importance of the data bits is dependent on the address bits.

A multiplexer can be thought of as a logic circuit where a certain number of bits provide the address of the output bit. Assuming  $L$  is the length of the input, then the equation:

$$L = k + 2^k \quad (6.2)$$

defines the relationship between the length of the input and the number of address bits required. This is a fundamental concept which the technique will need to learn. Following this, the technique will also need to use the number of address bits ( $k$ ) to extract them from the bit string. Part of the functionality will need to be capable of converting the address bits into a real number such as the position of the data bit. Importantly, in order to compute formula 6.2, the agent would need to be apprised of the *power base two* function as well as *real number addition*. These functions are not part of the Boolean domain, but humans would have already learned these functions. They would have been taught during their normal education as prescribed by their human instructors. However the concept of the mental number line is something that humans are born already knowing [72]. Therefore humans can readily include them in their reasoning for this domain.

This chapter considers the following research objectives:

- \* **Develop** methods such that learned knowledge and learned functionality can be reused for Layered Learning of Threshold Concepts.

- \* **Determine** the necessary axioms of knowledge, functions and skills needed for any system from which to commence learning the target problem.
- \* **Demonstrate** the efficacy of the introduced methods in two complex domains, i.e. Mux, Hidden Mux.
- \* **Determine** whether the technique can function if the layers are missing or are insufficient.

In the following section a more detailed description of the technique will be presented. This will include a description of the problem domain as well as the methodology for separating the different parts of the target problem. The results of the experiments will be described and their interpretation will provide further insight into the intricacies of the proposed work.

## 6.2 Solving the n-bit Multiplexer Problem Set

### 6.2.1 Method

This work disrupts the standard learning paradigms in EC that seek to address problems in a single stage evolution by aligning it with LL and TC. Here the goal is to learn capabilities using a bottom-up approach, coupled with transformative concepts that foster learning. Via this approach, it may be possible to learn functions and parts of functions conducive to learning an entire problem. The problem will be decomposed into a number of sub-problems with each layer feeding its solution into the subsequent layer. This is necessary as learning the entire problem in one training session is intractable due to the rigors of CF creation [49]. Each sub-problem in and of itself will be a critical component for the system to fully learn the problem, but more importantly, they will be administered to the system in a prescribed manner. In other words, the method here is

to specify the order of problems/domains (together with robust parameter values) while allowing the system to automatically adjust the terminal set through feature construction and selection, and ultimately develop the function set. This is different to the self-contained stages of LL, and closer to TL in this respect, as part-solutions (CFs) can be propagated. Similarly, complete solutions (learned functions) can form part-solutions for future problems. This is analogous to a school teacher determining the order of threshold concepts for a student in a curriculum [64]. The system can use learned rule-sets as functions along with the associated building blocks, i.e. CFs, that capture any associated patterns; this is an advantage over pre-specifying functionality.

The anticipated benefits actualized by this technique will be mainly due to the fact that it will not introduce domain bias, rather it will allow the system to construct features based on the available functionality. The fact that a human will break the problem into a series of steps, each with its own axiomatic requirements, will ensure that the technique does not include domain bias. The system will have just enough axioms to foster further learning of the sub-problems, which will add to the functionality available to the system. This functionality will include that provided by the human user *a priori* (axioms), as well as that which is learned along the way (learned functions). This means that some of the learned functions will contain regularities from domains different from the problem being currently learned. The learned functions will normally contain arguments and a return value while the skills will simply act upon the current system state and may or may not provide a return value. For example, the system will learn the ValueAt function, and this in turn will be made up of functionality from previous steps. These steps will contain functions related to the Boolean domain as well as the domain of real numbers.

This method modifies the intrinsic problem from finding an overarching 'single' solution that covers all instances or features of a problem to finding the structure (links) of sub-problems that construct the overall so-



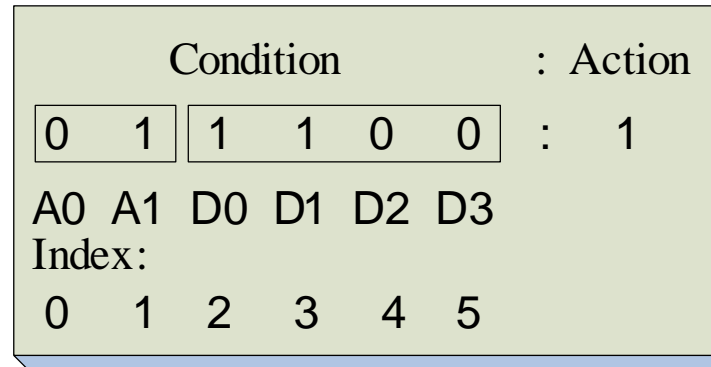


Figure 6.3: 6-Bit multiplexer showing the address bits and the data bits of the condition, this distinction is not provided to the learning system.

lution. Learning the underlying patterns that describe the domain is anticipated to be more compact and reusable as they do not grow as the domain scales (unlike individual solutions that can grow impractically large as the problem grows, e.g. DNF solutions to the multiplexer problem).

#### 6.2.1.1 n-bit Multiplexer Problem

The multiplexer problem can be deconstructed into a series of steps – layers – that the system is to learn. An example of a 6-bit multiplexer is depicted in Figure 6.3. Determining the number of address bits  $k$  requires using the *log* function, as illustrated in equation 6.3, in this example  $k$  is 2. Then  $k$  bits must be extracted from the string of bits to produce the two address bits. The next step is to convert the address bits into decimal form; this requires knowledge of the *power base 2* function as well as elementary *looping*, *addition* and *subtraction* functions. Depending on the approach to this step, *multiplication* may also be required. The two address bits translate to 1 in real form, as shown in Figure 6.3, i.e. A0 and A1. The real number points to the data bit D1 that contains the value to be returned. The index begins at 0 and proceeds from the left towards the right, as shown in Figure 6.3.

One of the underlying reasons for choosing the multiplexer problem domain for the proposed work is that humans can solve this kind of problem by naturally combining functions from other related domains along with functionality from the Boolean domain. For instance, a human can reason that there is a relationship between the address bits and the data bits without being given any knowledge about the problem beforehand. The agent is anticipated to formulate a similar process whereby useful information can be reused. Humans are also able to reason that a sub-set of functions in their ‘experiential toolbox’ may be appropriate for solving the problem.

The experiential toolbox is the whole of learned functionality for the agent. These functions include from primary and secondary education *multiplication, addition, power*, and the Log function. The agent here must build-up its own similar toolbox of functions and associated pieces of knowledge (CFs). A computer program could select any of these functions and potentially many more, but it could not intuit which are appropriate to the problem, and which are not. Therefore, the agent will need guidance in its learning so that it may have enough cross-domain functions to successfully solve the current problem. It will need to be capable of learning with more functions than necessary as the exact number of useful functions may not be known *a priori*, but at this stage of paradigm development is not expected to be able to adjust to fewer functions than necessary.

Besides functions, the experiential toolbox will also contain skills. These are capabilities that the agent will have learned or will have been given beforehand. Unlike functions, which contain arguments and have return values, skills manipulate the current state of the system and may or may not produce an output; one example is the looping skill [23], [55]. This looping skill is important because it will be utilized during successive operations such as converting a sequence of bits into an integer. These skills once learned by the agent and combined with the functions, provide

the seeds from which to discover useful knowledge. For example, a human understands all the operations required for counting  $k$  number of bits, starting from the left of the input string. Then the human would have to conceptualize how to convert the address bits to a real, which requires the ability to multiply. It is important to note that extracting  $k$  from the problem length  $L$  is a difficult task. It can be accomplished through a trial and error iterative approach or a formula can be used. In this case the formula:

$$k = \lfloor \log_2 L \rfloor \quad (6.3)$$

provides the functionality for determining the number of  $k$  address bits by using the length of the input. In this case the person would need familiarity with the log base 2 function as well as the floor function. A human could determine the address bits with increasing difficulty, due to the fact that humans are able to compute things mentally up to a limit, but a software system would have to learn this functionality before even attempting to solve the n-bit multiplexer.

### 6.2.1.2 Proposed System

The proposed system, termed XCSCF\* – this moniker is appropriate because the technique has evolved from a number of earlier CF-based techniques – consists of various components, see Figure 6.4. Since different types of actions are expected, e.g., Binary, Integer, Real, it is proposed that the functions be created by XCSCFA based systems, although any rule production system can also be used, e.g. XCS, XCSCFC, etc. This will facilitate the use of real and integer values for the action as well as enabling it to represent complex functionality. The proposed solution will reuse learned functionality at the terminal nodes. This means that at the terminal nodes there will be CFs as well as environment features. These will also appear at the root nodes of the CFs. In essence, the root nodes will contain learned functions, since this has been shown to be beneficial for scaling. XCSR would not be helpful here because on a number of the

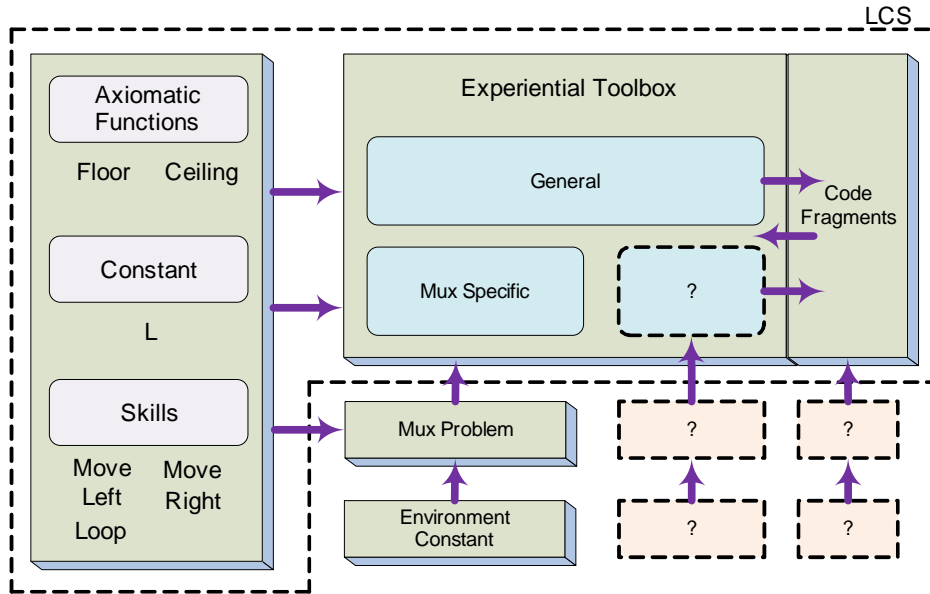


Figure 6.4: Training encompasses different types of functions, skills and axioms. The experiential toolbox will contain general as well as multiplexer specific learned functionality. The question marks indicate CFs from the previous (as well as potential to add from the next) domain and functionality learned from it.

steps, the permitted actions are not a number but a string. Moreover, XCSR with Computed Continuous Actions would present unnecessary complications to this work because the conditions of the classifiers do not require continuous values [44].

The inputs and outputs of the overall system consist of a Mux instance (bit string) and its integer length  $L$ , which is known to standard techniques addressing the Mux problem, and an output of binary type at the very end. Base functionality will have to be provided for the system; we term these Axioms, e.g. *log*. For example, it is anticipated that basic functions like addition, multiplication, subtraction, division, natural log, power base 2 will have to be provided to the systems to bootstrap learning of the target problem. Future work will involve obtaining a minimal set of axioms.

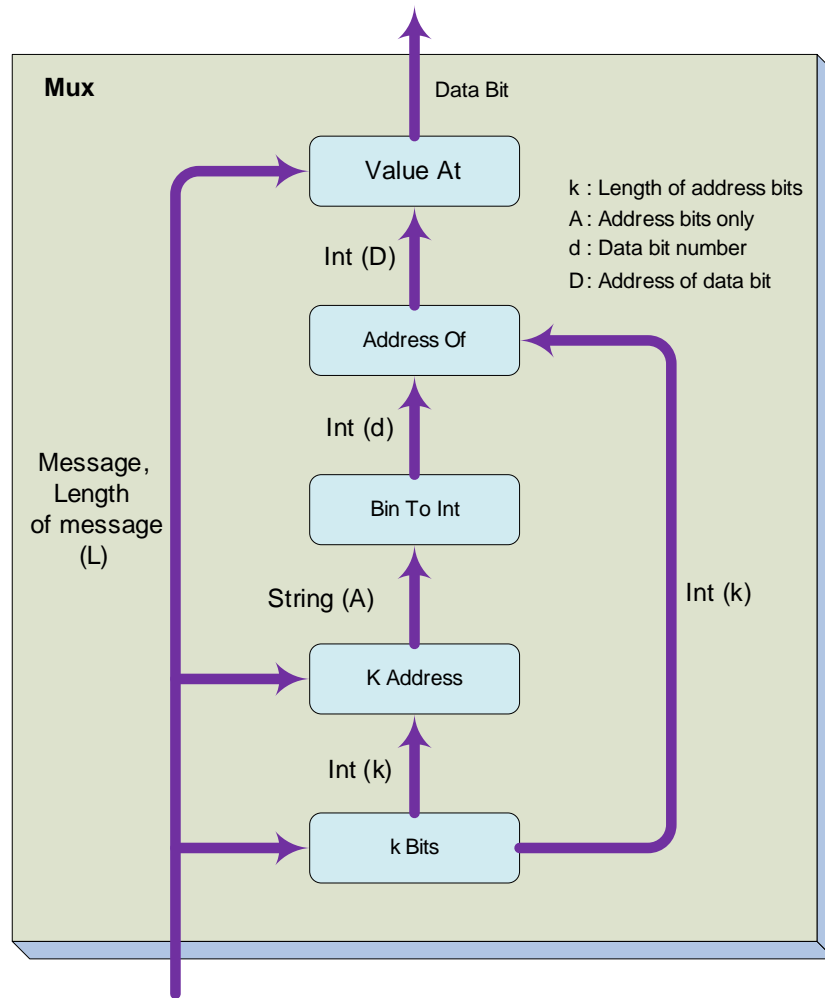


Figure 6.5: Multiplexer training flow, as considered by humans - the message is utilized at three steps, while the  $k$  bits are used in two steps during learning. The training regimen uses a layered approach. Each subsequent step is composed of knowledge blocks from the previous steps.

### 6.2.1.3 Individual Detailed Components

The selected regime for separating the Mux problem into subordinate problems has five parts. Five constituent parts was found to be the minimum needed for a CF-based system to learn the necessary functionality, including the heterogeneous data types and disparate domains involved. Each subsequent part builds upon the rules learned from the previous step as well as from the axioms provided. Figure 6.4 illustrates the relationships between the axioms, skills and learned functionality and their CF representation. The figure also depicts how the type of problem faced can feed domain specific functionality into the experiential toolbox of the system. This is shown by the arrow flowing from the Mux problem towards the Experiential Toolbox.

At each step, the new system has access to the environmental message, constants and hard-coded functions, as well as the learned CFs and CF functions. To bootstrap the learned CF functions the NAND boolean operator will be learned in a standard XCS and the resulting rules will be available to the *n-Bits* system to fabricate a bootstrap for the learning process. Table 6.1 shows a listing of all the axioms and skills made available to the system along with their system generated tags (used to interpret results) and their input/output data types. Table 6.2 shows a listing of the constant(s) provided to the novel system. Table 6.3 shows a listing of the functions to be learned, note that these were furnished in order, as a curricula. However, a system could be developed to address all problems in parallel, such that as each problem is solved, it becomes available (CFs and function) to the remaining problems. The different types of parameters for the sub-problems are also listed in Table 6.3. The multiplexer is listed in the table but it is not a sub-problem – it is the task on which the final rules will be tested in this section. More specifically, the technique will learn the fundamental sub-problems, then the 6-bit mux problem and those rules will be used to test on the more difficult multiplexer problems.

It is important to consider that the work presented here does not seek

Table 6.1: Functionality Provided (Hard-coded functions)

Function	Tag	Input	Output
Floor	[	Float	Integer
Ceiling	]	Float	Integer
Log	{	Float	Float
BinaryString	\$	Integer	String
Power 2 Loop	@	String	Variant
Add	+	Variant	Integer
Subtract	-	Float	Float
Multiply	*	Float	Integer
Divide	/	Float	Integer
ValueAt	=	Integer	Binary

Table 6.2: Constant(s) Provided

Constant	Tag	Input	Output
LEN	L	NA	Variant: any type

Table 6.3: Input and Output Types for all the sub-problems and for the Mux problem. The data type 'Variant' can accept all other types.

Sub-Problem	Input Type	Output Type
KBits	Integer	Integer
KBitString	Integer	String
Pow2Loop	String	Variant
AddressOf	Variant	Integer
ValueAt	Integer	Binary
Mux	Integer	Binary

to provide a specific blueprint for a system to follow and ultimately arrive at the solution to a given multiplexer problem only. Rather, the emphasis here is to facilitate learning in a series of steps, where in this case the

learned functionality could potentially help a system to arrive at a general solution of *any* Multiplexer problem, i.e. any  $n$ -bit multiplexer problem where  $n$  represents the varying length of the multiplexer string of bits. Furthermore, it is important for the system to learn to combine the different learned functions in a way conducive to learning; a way that will produce a general solution. The training data will also originate in not just one type of multiplexer problem, but will be compiled from many different types of sub-problems, see Figure 6.5.

The number of subordinate problems can always be increased (until the foundation axioms are reached, see Table 6.1) in the future, e.g. learning basic functions such as an adder or a multiplier via Boolean functions or even learning the log function via training data.

The five steps are labeled: `kBits`, `kBitString`, `Bin2Int`, `AddressOf` and `ValueAt`. They will be detailed sequentially in the following sections.

#### 6.2.1.4 Sub-Problem - `kBits`

The first step is to determine the number of  $k$  address bits that will contribute to the solution for the  $n$ -bit multiplexer. The constant `LEN` (also termed  $L$ , see equation 6.2) furnishes the system with the length of the environment message. It is an environment constant that is set before the run. The training data-set to be used consists of instances of possible Mux lengths and the corresponding number of address bits, see Table 6.4.

#### 6.2.1.5 Sub-Problem - `kBitString`

This part extracts the first  $k$  bits from a given input string. The data-set will be random bit strings, e.g. length 6, and a fixed  $k$  length where the action is the first  $k$  bits, see Table 6.5. The training data will not include more heterogeneous lengths because this is deemed not necessary. The series of sub-problems will ultimately produce the correct combination of functions, axioms and constants, which is anticipated to scale to any length



Table 6.4: Training data for the KBits sub-problem. It offers a mapping from a possible multiplexer length to the corresponding number of address bits.

Message	Action
3	1
6	2
11	3
20	4
37	5
70	6
135	7
264	8
521	9
1034	10
2059	11

multiplexer problem. By restricting the length of the training data to one length, the size of  $k$  is also being restricted. This may have unexpected consequences if the final rules are used in a related domain to solve other problems.

#### 6.2.1.6 Sub-Problem - Bin2Int

The third sub-problem deals with converting a binary number to an integer. This is crucial because the system needs this information to determine the position of the data bit. However, this is not a trivial task as the system would need to be cognizant of many functions that a human would potentially already have in their experiential toolbox. This includes the concept of a number line, the ability to determine relationships between bit patterns and their translation into real numbers. The data-set will be random strings of length 6, with the action being the equivalent integer number of

Table 6.5: Training data for the kBitString sub-problem.

Bit String	Action
0 0 0 0 0 0	0 0
0 0 0 0 0 1	0 0
0 0 0 0 1 0	0 0
...	...
0 1 1 1 1 1	0 1
1 0 1 1 1 1	1 0
1 1 1 1 1 1	1 1

the first two bits, see Table 6.6. The training data is limited to a length of 6 as it was deemed sufficient for the system to learn to combine the proper sequence of functions, axioms and skills. In addition, the first two bits are considered significant for this sub-problem. However, since the technique uses LL, a full multiplexer problem must be part of the training data as the requisite functions that compute the earlier sub-problem rely on this.

Table 6.6: Training data for the Bin2Int sub-problem.

Bit String	Action
0 0 0 0 0 0	0
0 0 0 0 0 1	0
0 0 0 0 1 0	0
...	...
0 1 1 1 1 1	1
1 0 1 1 1 1	2
1 1 1 1 1 1	3

**6.2.1.7 Sub-Problem - AddressOf**

This functionality determines the location of the data bit from an input string and known address, which is a more difficult problem than learning the summation of address length and decoded length. The data-set will be random strings (length 6) and decoded address with the integer action, see Table 6.7. The first two bits of the training data are important for this sub-problem, however the full 6 bits of a multiplexer problem are required for the chained functionality to compute the sub-problems correctly. Also, the technique appears to be adding 2 to the real translation of the first two bits. However, upon closer examination of the final rules, there may be a more complex combination taking place.

Table 6.7: Training data for the AddressOf sub-problem.

Bit String	Action
0 0 0 0 0 0	2
0 0 0 0 0 1	2
0 0 0 0 1 0	2
...	...
0 1 1 1 1 1	3
1 0 1 1 1 1	4
1 1 1 1 1 1	5

**6.2.1.8 Sub-Problem - ValueAt**

The functionality to be learned is to return the bit referenced from a bit string. The system is trained using a data-set of bit strings of known length (again length 6) with a reference integer and corresponding output bit, see Table 6.8. These sub-problems will be as simple as possible in the multiplexer domain for Layered Learning to investigate scalable and reusable

learning. The alternative data-sets are more complex as they are for general learning to support transfer learning for general problem solving (not just multiplexer solving). These sets could include Sine, Cosine, Tangent and many other functions.

Table 6.8: Training data for the ValueAt sub-problem.

Position	Bit String	Action	Reward
0	0 0 0 0 0 0	0	1000
5	0 0 0 0 0 1	1	1000
4	0 0 0 0 1 0	1	1000
...	...	...	...
0	0 1 1 1 1 1	0	1000
1	1 0 1 1 1 1	0	1000
2	1 1 1 1 1 1	1	1000

### 6.2.2 Hidden Multiplexer

In order to demonstrate the technique's capability for transferring learned knowledge into a related domain, it will be tasked with the hidden multiplexer problem. The hidden multiplexer was deemed appropriate for this work because it presents a very difficult problem with a complex and large search space [17], see Figure 6.6, where all bits are relevant to any instance rather than to a small percentage. It also exhibits a high degree of epistasis and non-linearity. The multi-level nature of the problem means that the system must form useful blocks of knowledge to solve the lower level. Once the Parity problems (lower level) are solved, then the system must combine the newly learned blocks of knowledge to solve the multiplexer problem in the upper level, thus solving the complete problem.

The results are anticipated to be very similar to XCSSMA. The main reason is that the proposed system will be producing general solutions to

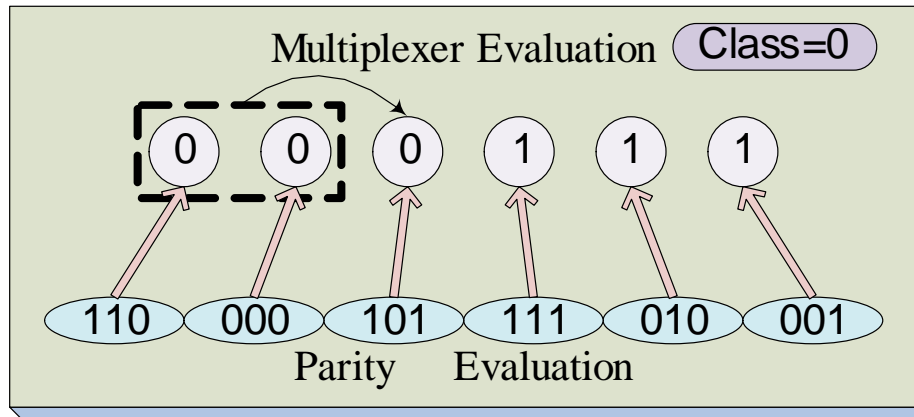


Figure 6.6: 3x6 bit Hidden Multiplexer showing the lower, Parity level and the upper, multiplexer level. In this case the multiplexer address bits are assumed to begin from the left side. The agent is only presented with the lower level bit string during training and testing.

the multiplexer problem. XCSSMA was successful in producing general solutions to several Boolean problems [46]. It has been discovered in previous work that including the Parity learned function promotes increased scalability [4].

The resulting rules from the ValueAt problem will be used to solve a hidden multiplexer problem. This will show whether the rules are effective in a related domain. Furthermore, certain axiomatic functions, like those that compose the Parity problem, have not been included.

## 6.2.3 Results of the n-bit Multiplexer Experiments

### 6.2.3.1 Experimental Setup

The experiments were executed 30 times with each having an independent random seed. The stopping criterion was when the agent completed the number of training instances allocated, which was chosen based on preliminary empirical tests on the convergence of systems. The proposed

systems were compared with XCSCFC and XCS. The settings for the experiments are common to the LCS field [49]. They were as follows: Pay-off 1000; the learning rate  $\beta = 0.1 - 0.2$ ; the Probability of applying crossover to an offspring  $\chi = 0.8$ ; the probability of using a don'tCare symbol when covering  $P\_don'tCare = 0.33 - 0.95$ , this range indicates that as the problems grew in complexity the system faced growing difficulty in finding a solution; the experience required for a classifier to be a subsumer  $\Theta_{sub} = 20$ ; the initial fitness value when generating a new classifier  $F_I = 0.01$ ; the fraction of classifiers participating in a tournament from an action set 0.4. In addition, error threshold  $\epsilon_0$  has been set to 10.0. This setting became necessary as the problems increased in complexity. The population size varied with the sub-problem, see Table 6.9. The KBits sub-problem required a larger population because it could not count on any previously learned knowledge.

Table 6.9: Population size for the fundamental sub-problem experiments.

Sub-problem	Population Size
KBits	2000
KBitString	500
Pow2Loop	500
AddressOf	1000
ValueAt	500

### 6.2.3.2 Experimental Tests

Figures 6.7(a) - 6.7(d), illustrate the training in the KBits through AddressOf sub-problems. The KBits sub-problem training was straightforward. This is reflected by Figure 6.7(a). Since the training data was limited to a list of the possible mappings of a multiplexer problem length to the correct number of address bits, the system quickly determined the appropriate

combination of axioms which would produce the maximum reward. This is reflected by the very small number of training instances required to converge fully. An example of the rules produced by the KBits sub-problem is:  $L \{ \lfloor \cdot \rfloor$ . Here  $L$  stands for the problem length constant, ' $\{$ ' stands for the Log axiom and ' $\lfloor$ ' stands for the Floor axiom. This rule computes the number of  $k$  bits by first using the constant  $L$  as an input to the Log axiom and then applies the Floor axiom to the result. The reason that the sub-problem required a larger number of training individuals than the others is because the number of possible valid actions was 11, which was not a simple binary classification because there were 11 possible actions for each classifier.

The KBitString sub-problem training is illustrated in Figure 6.7(b). Here the training data-set was the same, but the maximum population was smaller. This is the reason that the system began its convergence at a lower percentage (0.98) than the previous sub-problem. Also, the layered learning approach used here means that the function learned for the previous problem could now be used by the KBitString sub-problem to create a solution. This implies that the constant  $L$ , the Log and Floor functions would form part of the final rules determined to produce the maximum reward. This is because the CF rules produced by the previous sub-problem would contain the aforementioned functionality. Furthermore, this dictates a more complex evaluation of the CFs forming part of any covering classifiers for the current sub-problem.

The next sub-problem, Bin2Int, used all the functionality learned in the previous two sub-problems. The training data is at least as complex as in the previous sub-problems as it encompasses them, however the functionality has one additional layer of complexity. Not only did this solution require the axioms and/or functions learned for the KBits sub-problem and require the functionality learned for the KBitString sub-problem, it also needed the final rules produced by this sub-problem. The latter showed that the looping skill is very important to the solutions, see Table 6.10. The

looping skill appears in all the final rules. The rule with the highest numerosity and fitness prefers the constant  $L$  at the leaf nodes in one of its branches, while on the other branch it accepts the previously learned KBitString function,  $m$ . This same function takes code fragment with ID 7 (the system gives this CF the moniker of CF\_7) as its input. This is a redundancy of the system, since this code fragment contains the same functionality as learned function  $m$ . Also, in the third rule of Table 6.10, the looping skill is included twice, something that does not happen in the other three final rules.

Table 6.10: Final rules for the Bin2Int sub-problem (Experiment 8, arbitrarily picked as it is representative of the 30 experiments).  $L$ ,  $m$  and  $\$$  represent the length constant, learned KBitString function and the BinaryString axiom respectively.

Condition	Action	Numerosity	Fitness
# # # # # # CF_7 m CF_7 m @		36	0.25
# # # # # # L CF_7 m @		27	0.19
# # # # # # L L @ CF_7 m @		54	0.37
# # # # # # L \$ CF_7 m @		28	0.19

The AddressOf sub-problem produced a plot with a longer curve, compared to the previous three mentioned above. This problem required that the system use more training instances than all the previous three sub-problems. This is understandable as there was another sub-problem added to the evaluation process for the CFs present in the classifier. In light of this, it is also understandable that this sub-problem produced more rules than any of the previous sub-problems, see Table 6.11. An examination of the table reveals that the Addition axiom is considered very important by the system. It has determined that adding the results of the KBits function and the Bin2Int function will yield the correct location of the data bit within the message string. This is plausible as the former will yield the



number of address bits, while the latter will provide the additional number of bits to add to get to the correct location of the data bit. Furthermore, no one rule is overwhelmingly fitter than any other. Also, the numerosity of the rules falls between 7 and 14, inclusive. This is not a great difference and does not indicate anything interesting with the exception that as the numerosity of a rule increases so does its fitness.

Table 6.11: Final rules for the AddressOf sub-problem (Experiment 8).

Condition	Action	Numerosity	Fitness
# # # # # # N CF_9 CF_8 c +		14	0.08
# # # # # # CF_10 CF_9 c N +		9	0.05
# # # # # # N CF_10 CF_8 c +		8	0.05
# # # # # # CF_8 CF_10 c N +		9	0.05
# # # # # # N CF_10 CF_11 c +		7	0.04
# # # # # # N CF_10 CF_9 c +		7	0.04
# # # # # # CF_9 CF_8 c N +		10	0.06
# # # # # # CF_7 m N +		13	0.07
# # # # # # N CF_8 CF_9 c +		10	0.06
# # # # # # N CF_11 CF_8 c +		9	0.05
# # # # # # CF_11 CF_10 c N +		10	0.06
# # # # # # N CF_11 CF_9 c +		12	0.07
# # # # # # N CF_9 CF_9 c +		11	0.06
# # # # # # CF_8 CF_8 c N +		9	0.05
# # # # # # CF_9 CF_10 c N +		7	0.04
# # # # # # N CF_10 CF_10 c +		7	0.04
# # # # # # N CF_7 m +		9	0.05
# # # # # # N CF_9 CF_10 c +		8	0.04
# # # # # # CF_8 CF_11 c N +		8	0.04

Figure 6.8 shows that the training was successful for the ValueAt sub-problem. The plot indicates that the system converged with a small num-

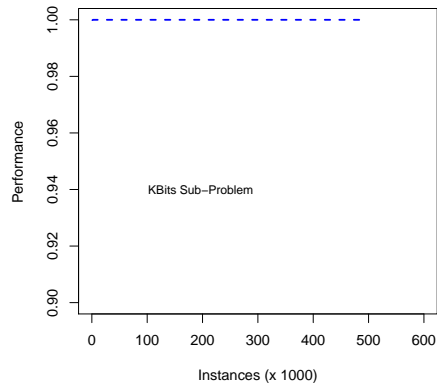
ber of training instances, compared to the AddressOf sub-problem. The fact that the system produced just two rules indicates that the rules are highly general, which is confirmed as the condition part of both rules is composed of don'tCares only, see Table 6.12. The numerosity for both rules was higher than in any of the other sub-problems, also the higher numerosity tends to map to a higher fitness.

Table 6.12: Final rules for the ValueAt sub-problem (Experiment 8).

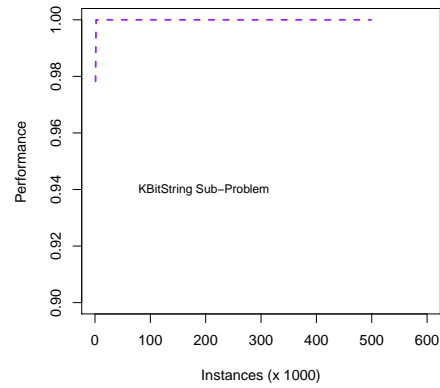
Condition	Action	Numerosity	Fitness
# # # # # # CF_28 CF_14 M =		70	0.68
# # # # # # CF_28 CF_20 M =		34	0.32

The number of rules and CFs generated by the fundamental parts, i.e. kBits, kBitString, Bin2Int, AddressOf and ValueAt was small. In certain cases, the number of rules produced was one, as was the case with run 8 of the kBits sub-problem. This is plausible as the rule is general and will work for any length input string. The same trend continued with the more difficult problems as well. The condition was composed of don'tCares, which may appear counter-intuitive as LCSs partition the search space, and is typically observed in the ternary XCS applied to multiplexer problems. However this was anticipated as the rules could be applicable to all instances of the sub-problem, therefore they could be maximally general.

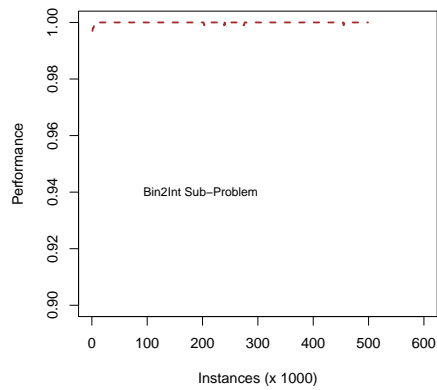
All the sub-problems used the standard XCS training and testing cycles, i.e. explore and exploit. However, for the more difficult problems, the final rules learned by the fundamental steps were used with only the testing phase available, i.e. exploit. The problems presented to the system grew progressively large. This size was large enough to prevent the system from solving them simply by enumerating possible solutions. The LL approach culminated in training the ValueAt function, which is the last step in solving the multiplexer problem. Therefore, only the testing phase was needed for evaluating the LL approach on the multiplexer data-sets.



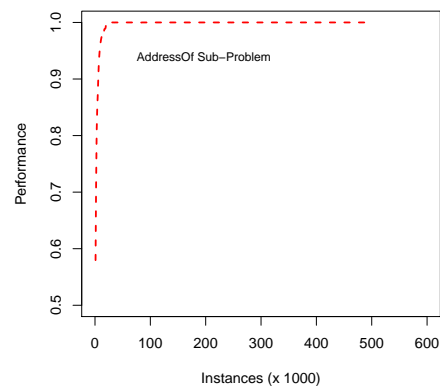
(a) Kbits



(b) KbitString



(c) Bin2Int



(d) AddressOf

Figure 6.7: Results of the Kbits, KbitString, Bin2Int and AddressOf sub-problems using XCSCF\*. There was training and testing involved. Each sub-problem fed its output (CFs and learned function) to the next sub-problem.

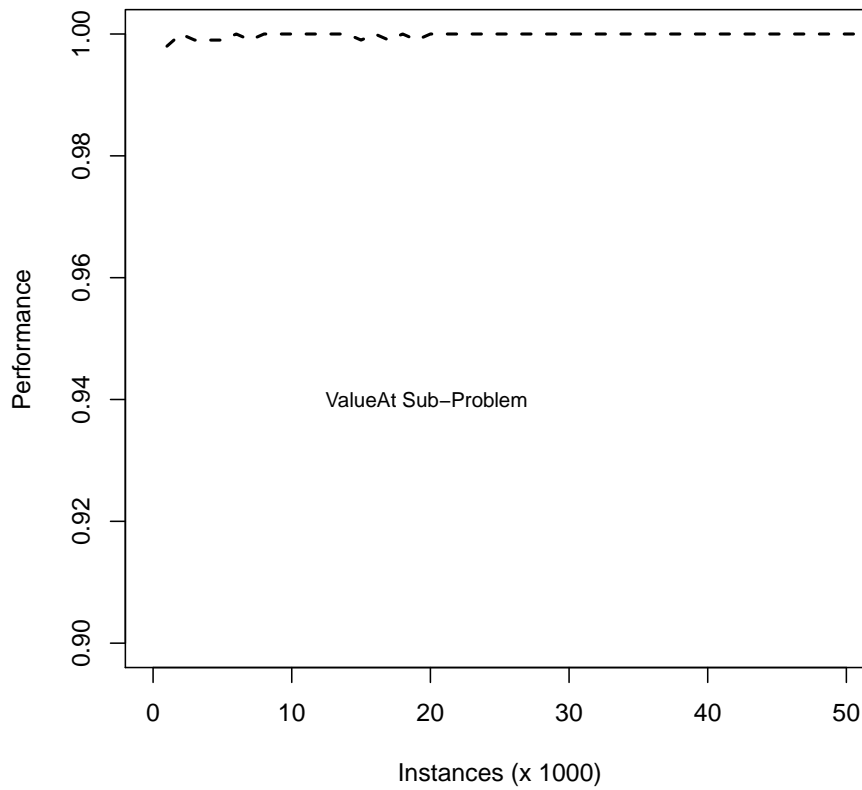


Figure 6.8: Result for the ValueAt sub-problem. These results with XCSCF\* involved training and testing. The system converged to 100%.

Figure 6.9 shows that only the proposed system XCSCF\* and XCSCFC were able to solve the 135-bit Multiplexer. Additionally, the proposed system performed better than the state of the art XCSCFC, albeit with a higher standard deviation at a given number of instances. The Wilcoxon signed rank test comparing XCSCF\* with XCSCFC after two million iterations shows no evident difference between both techniques. Further tests were conducted for the 264-bit and 521-bit multiplexer problems, see Figure 6.10. The rules produced by XCSCF\* using LL for the 6-bit multiplexer were reused in these experiments. This shows scaling by relearning, because with each multiplexer problem the system has to relearn new rules, but it is the capturing of the underlying patterns without retraining at each successive problem as it scales that is the aim of this work. In other words, it is anticipated that the final rules produced by the fundamental steps to the layered learning can capture the necessary regularities present in all multiplexer problems. This way, no further learning would be necessary, only testing in order to find a solution to a problem.

Tests were conducted on the final rules produced by the 6-bit multiplexer to determine if they were general enough to solve more complex problems. Figure 6.11 shows that the rules produced by the 6-bit multiplexer were able to solve the 264-bit, 521-bit and 1034-bit multiplexer problems. Figure 6.12 illustrates the solutions for the 2059, 4108 and 8205-bit Multiplexer problems<sup>1</sup>. Note also that the training data for the KBits sub-problem only included up to  $2059 \rightarrow 11$ , meaning that the system generalized beyond its input data. All the graphs are similar in the sense that the performance begins at 100% and remains at that level throughout the tests. The system used to test the generality of the rules was a version of XCSCFA with certain modifications; there was only the exploit phase and no covering was allowed. The generality of the solution is built upon the power of expression imbued in Code Fragments (CFs). Since many types

---

<sup>1</sup>Note that  $2^{8205}$  is a vast number, meaning that testing a million instances is a fractionally small sub-sample, but will identify many deficiencies.

of actions (e.g. integer, string) were involved during the training process, CFs provided a viable way to express the action part of the rules evolved and their complex functionality.

### 6.2.3.3 Sub-Trees Generated by the 6-bit Multiplexer

The solution tree for a 6-bit Multiplexer is shown in Figure 6.13. The figure illustrates the compactness of the CF rules. One can also observe that the system has determined that the ValueAt function is the one that produces the final value for the multiplexer. This final output is determined by computing the entire chain of CFs which culminates in the ValueAt function.

The fully expanded tree is shown in Figure 6.14. It is evident that the chains of CFs produced are very long and are composed of functional blocks that are repeated in different branches. The learned function  $M$  is expanded in Figure 6.14 (top). In this figure it is clear that the function is composed of many other learned functions that are repeated. For instance, the KBits function ( $N$ ) is present in the right branch of the root node containing the  $+$  function.  $N$  is also present at the bottom of both branches of the root node containing the  $c$  function. The  $N$  function also appears at the bottom of CF 9, which forms the right child of the  $c$  function. Keeping in mind that this portion of the image reflects only the root node containing the  $M$  function, it is clear that CF 10 contains learned function  $m$  – otherwise known as the KBitString function – and CF 7, which are then repeated in CF 28. This CF forms the left child of the root node containing function  $M$ .

Figure 6.14 (bottom) shows CF 28 and CF 16. These are the expanded child nodes of learned function  $M$ . There are some redundancies in the makeup of CF 28. For instance, learned function  $m$  and CF 7 both have the same structure. They are composed of the Bit String axiom combined with function  $N$ . This same structure appears three times in CF 16. The repeated CFs mentioned above return a string representation of the  $k$  address bits calculated by learned function  $N$ . Ultimately, CF 16 performs an addition

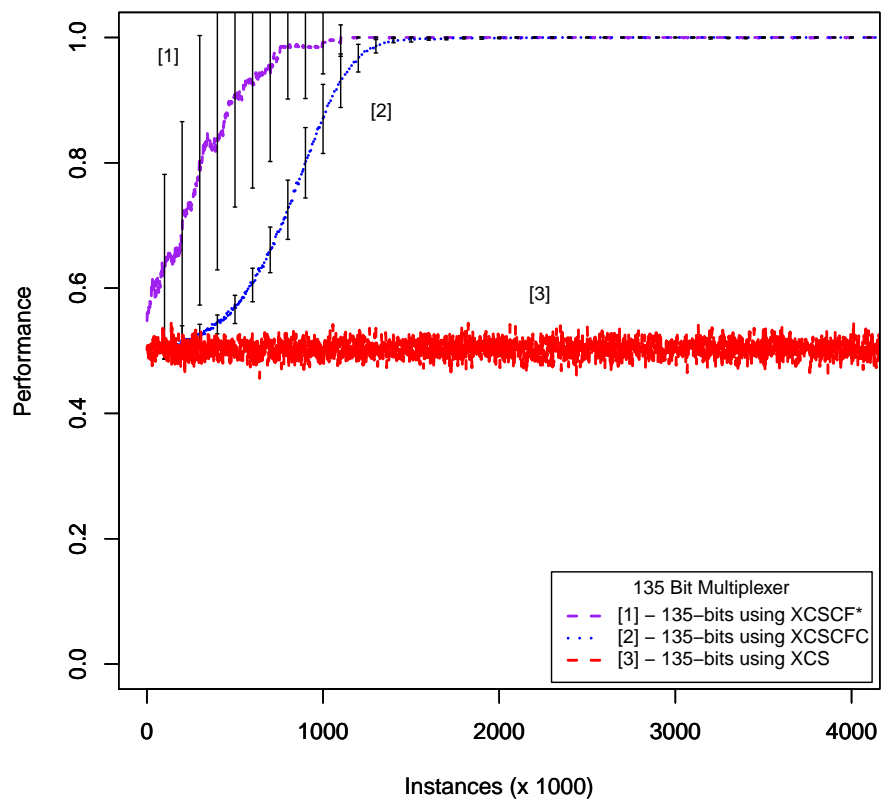


Figure 6.9: 135-bit Multiplexer Solution. Note: Wilcoxon signed rank test comparing XCSCF\* with XCSCFC shows no evident difference between both techniques at two million iterations.

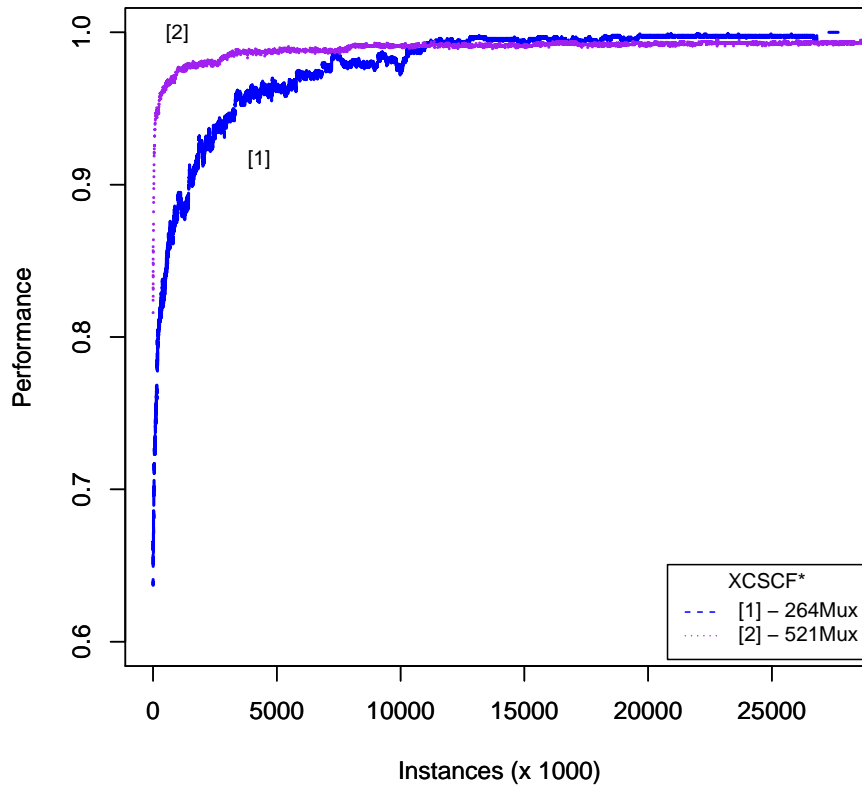


Figure 6.10: 264-bit and 521-bit Multiplexer solution, utilizing both training and testing. The rules, CFs and functions produced by the 135-bit multiplexer problem were transferred to the explore and exploit phases to learn the 264-bit problem. Then, the rules, CFs and functions learned by the 264-bit problem were used to solve the 521-bit problem.



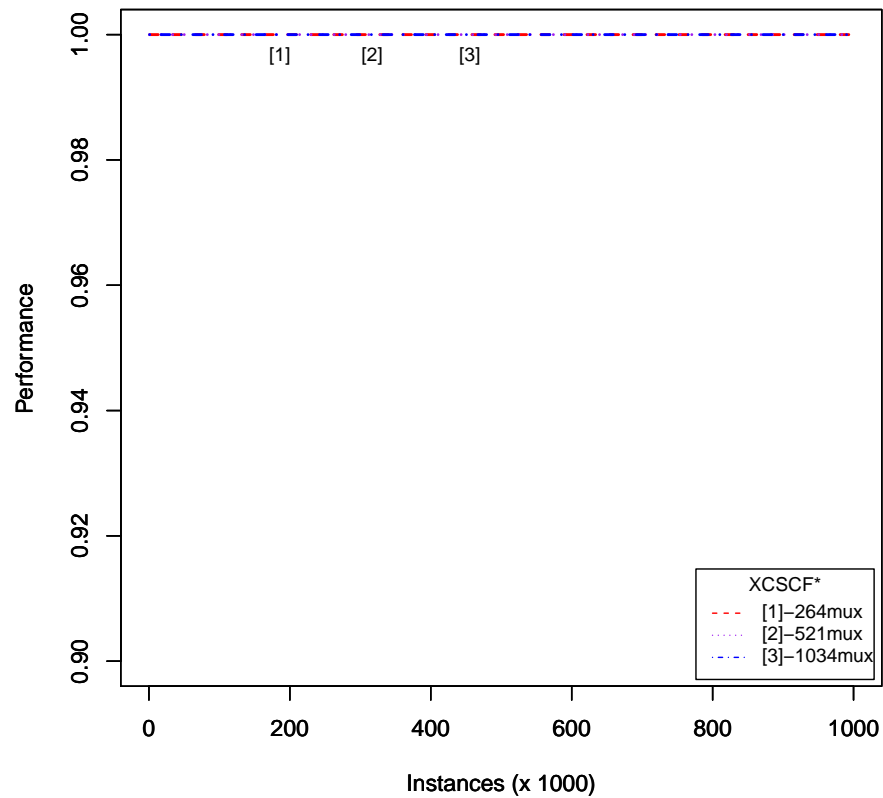


Figure 6.11: 264, 521 and 1034-bit Multiplexer solution, using XCSCF\*. The rules produced by the 6-bit multiplexer were used to solve these problems. Only the exploit phase was activated, the explore phase was deactivated therefore no additional training took place.

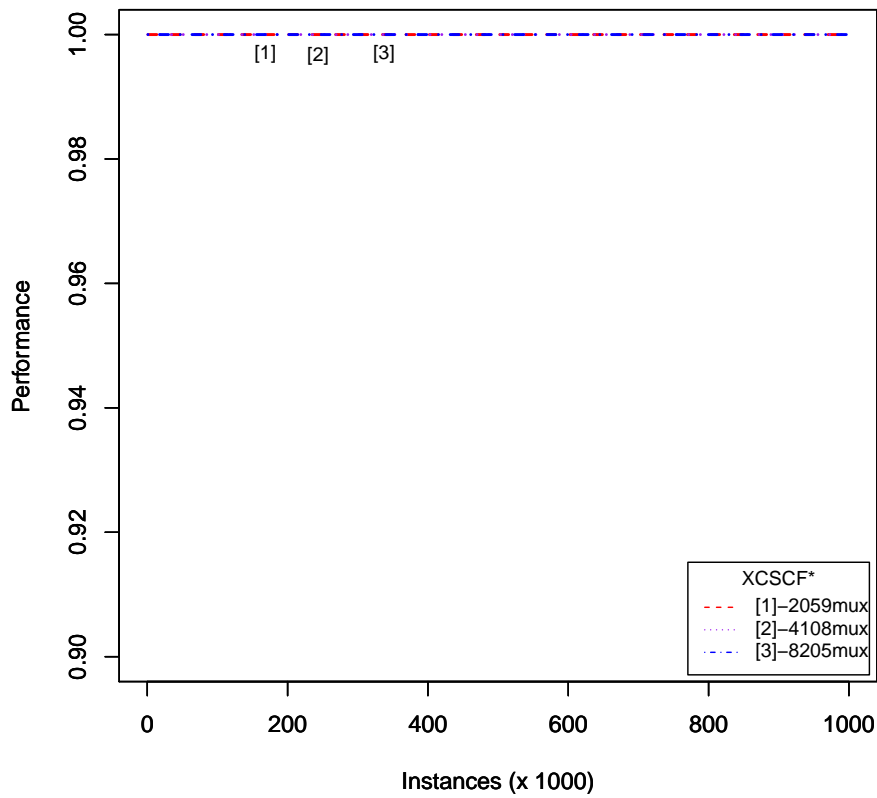


Figure 6.12: 2059, 4108 and 8205-bit Multiplexer solution. These results with XCSCF\* did not involve any training, just the test phase. The rules produced by the 6-bit multiplexer were used to solve these problems. Only the exploit phase was activated, the explore phase was deactivated.

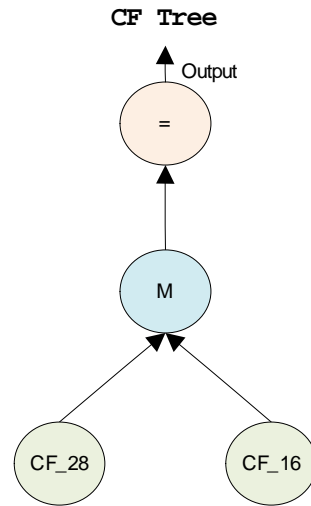


Figure 6.13: 6-bit Multiplexer Solution Rule. Where '=' is the ValueAt function, 'M' is the AddressOf function and CF\_16, CF\_28 are Code Fragments.

of its child branches, but in doing this, it calculates CF functionality that will be used elsewhere, such as that mentioned above. Learned function  $c$  and both of its child nodes contain  $m$  at the bottom of their right branches.

A comparison of the computer generated and human-inspired solutions of the multiplexer is in order here. Figure 6.14 shows a solution of the multiplexer problem that is very complex, large and seemingly convoluted. On the other hand, Figure 6.5 – the human hypothesized version of the multiplexer – is simpler and has fewer layers. The original human inspired model uses layered learning in a straight forward way, each layer feeds its input into the subsequent layer. However in the computer generated model, the learning is more haphazard. Some functionality is learned multiple times and then gets reused further down the CF chains. Also, some functionality appears in several places in a random way.

The systemic redundancies described above presented an opportunity for the novel system to store these computed values for later usage. It also shows that while the final rules for the difficult problems can appear

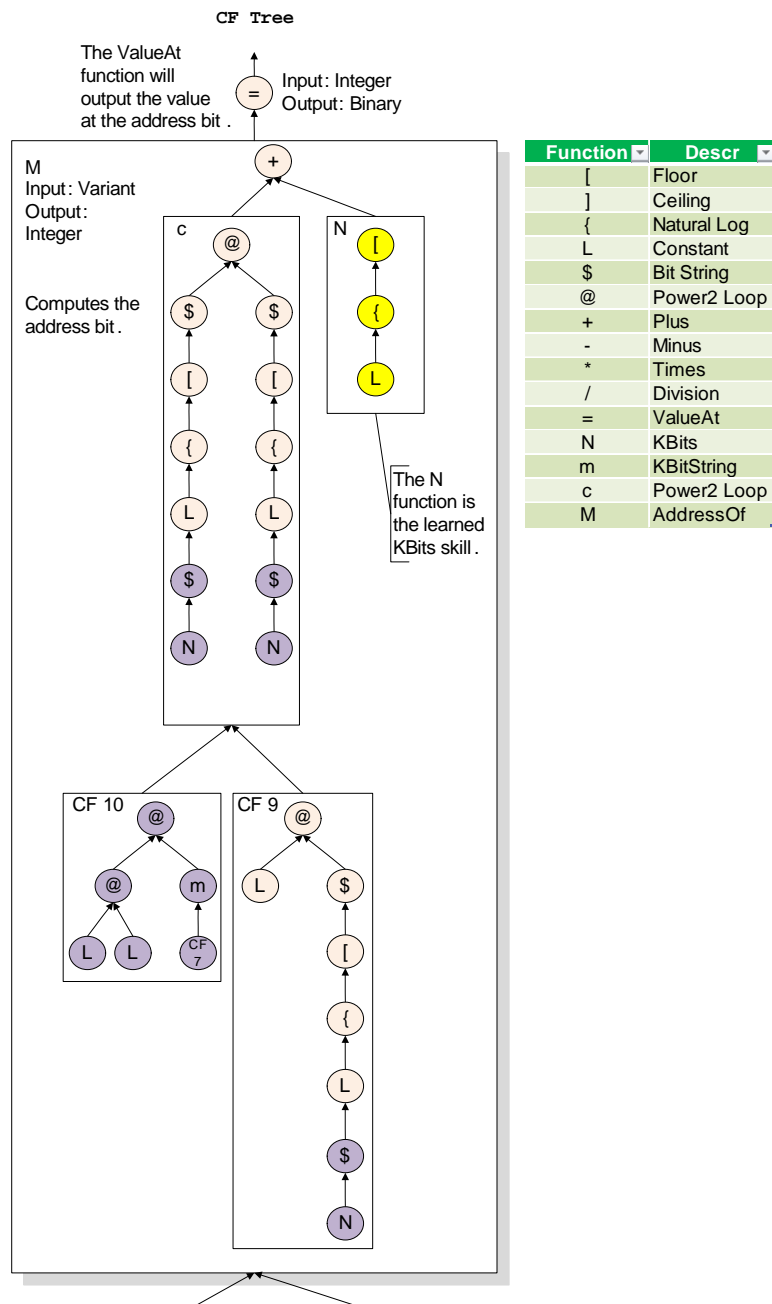
simple at first (Figure 6.13), under careful scrutiny the underlying functionality is extremely complex and is composed of the necessary combinations of axioms, skills and learned functions that were determined by the system to provide a viable solution to the problem. More importantly, as each layer of learned functionality is peeled back, it reveals the regularities which are present and that form the necessary structure of a general solution for the problem. The same structure can make it difficult to interpret the rules, specifically the apparently redundant CF chains present. In addition, the sub-trees are not straightforward and can expand several layers larger than they appear at first.

#### 6.2.3.4 Rules Generated by the 6-bit Multiplexer

The rules produced by the 6-bit multiplexer solution are illustrated by Table 6.13. The rules are maximally general, which is caused by the scope of the functions accumulated by the experiential toolbox. The combined set of functions is sufficient to calculate the correct data bit for any 6-bit multiplexer problem. This is because the ValueAt function combines all the necessary functionality, from all previous domains, to arrive at the correct value. The action part of the rules combined a number of CFs along with their parent functions. CF 16 was found to be helpful by two of the rules, see Table 6.13, however there does not appear to be any particularly useful relationship between the solution and this lone CF, i.e. CF 16. It is similar in structure to the other CFs in the final rule-set. All the rules consider the '+' axiom very important and combine it with other CFs, previously learned functions and other axioms.

#### 6.2.3.5 CF Rules Tested from the 135-bit to the 521-bit Mux problems

The rules produced by the 6-bit multiplexer problem were tested on much larger multiplexer problems. These experiments included the testing and training phases. The system required increasingly larger settings for the



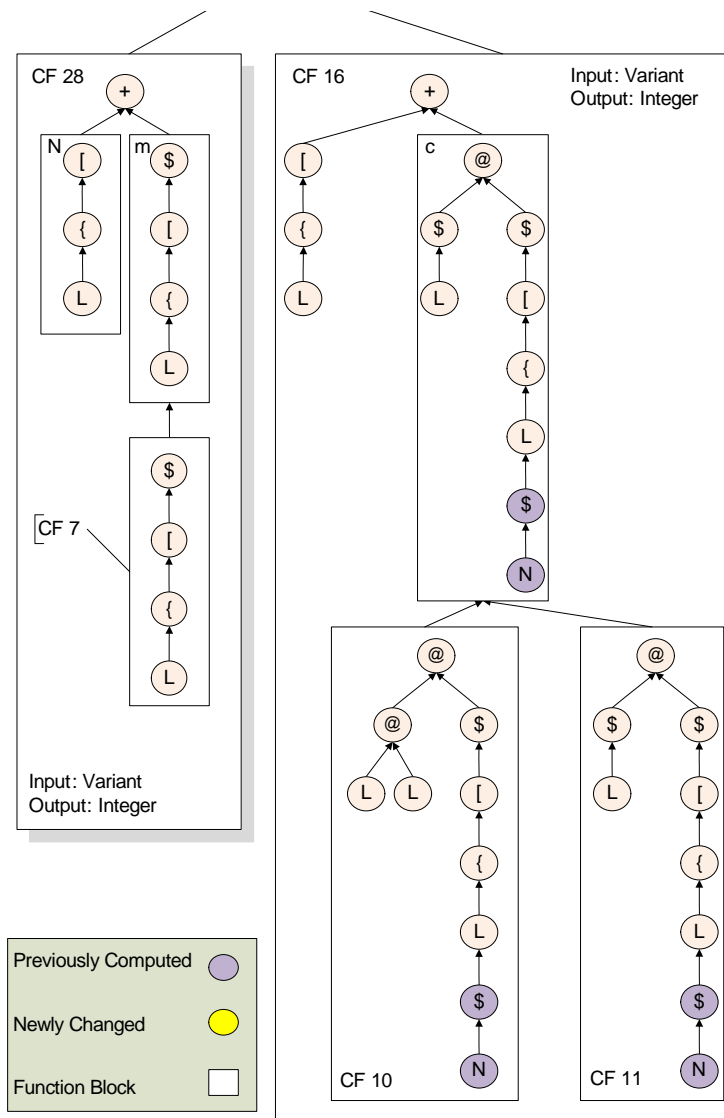


Figure 6.14: 6-bit Mux Solution Tree: ‘Previously Computed’ relates to CFs that were computed in a previous solution, ‘Newly Changed’ refers to the latest CF to have been computed, ‘Function Block’ refers to CFs forming a function. Note that this diagram spans two pages.

Table 6.13: Final rules learned while solving the 6-bit Multiplexer. The condition parts are composed of don'tCares, meaning that the rules are useful for solving any 6-bit multiplexer. Each rule is a complete solution to the problem.

Condition	Action
# # # # # #	CF_14 CF_16 M b
# # # # # #	CF_18 CF_26 M b
# # # # # #	CF_30 CF_16 M b
# # # # # #	CF_31 b

variable *p\_don'tCare*, which could be attributed to the increasing complexity of the problem and the curse of dimensionality, see Table 6.14. XCSCF\* was unable to learn in this manner past the 521-bit multiplexer. During the 264-bit problem the system fully converged after first showing a period of difficulty in learning. For the 521-bit multiplexer problem the system demonstrated a better rate of learning than during the 264-bit problem, however although it approached 100%, it never fully converged. The final rules produced by the 135, 264 and 521-bit problems for run number 8 (out of 30) are in Table 6.15. The rules are similar in that they are all maximally general with an action part composed of CFs. Importantly, all three problems produced just one rule each. This is symptomatic of the compactness and expressive richness of the rules. It is not considered that the rules are subject to over-fitting because subsequent experiments showed that the rules were capable of solving very difficult multiplexer problems well beyond the input data. Also, it is not considered that the rules suffer from over-generality due to the fact that the rules were anticipated to be maximally general. This would mean that the rules could be applied to most if not all the possible instances of multiplexer problems for a specified length.

Table 6.14: Settings for p\_don'tCare for the 135-521 bit multiplexer problems. The training and testing phases were involved.

Problem	p_don'tCare Setting
135-bit	0.90
264-bit	0.95
521-bit	0.99

Table 6.15: Final rules learned while solving the 135, 264 and 521-bit Multiplexer for run number 8. The condition part is composed of don'tCares, meaning that the rule is useful for solving any size multiplexer. The letter b stands for learned ValueAt function. It is used by all three problems.

Problem	Condition	Action
135-bit multiplexer	# # ... # #	CF_161 b
264-bit multiplexer	# # ... # #	CF_273 CF_273 M b
521-bit multiplexer	# # ... # #	CF_547 b

#### 6.2.3.6 CF Rules Tested from the 264-bit to the 8205-bit Mux problems

An important aspect of these experiments was to determine whether learning could be accomplished without re-training. For that reason, a series of tests were executed using the rules produced by the 6-bit multiplexer. The Table 6.16 illustrates the rules produced by the system while testing the series of 264-8205 bit Multiplexer problems. The solution to the 6-bit multiplexer produced only four rules, see Table 6.13, and these were used to conduct the tests. This table confirms that the test population consisted of only one rule, per run, per problem. The test samples consisted of one million problem instances. The ascending CF ids are a feature of the system, which renames each set of reused CFs depending on the size of the Multiplexer being learned at the time. The CFs illustrated in the table are the same four from the 6-bit solution, they are just renamed by each



new problem.

The rules from the 6-bit multiplexer can be reused by a much larger problem like the 8205-bit multiplexer principally for two reasons. The number of arguments linked to the functions in the action part of the classifier is independent of the length of the problem. Also, all the rules are maximally general, and since they are composed of don'tCares, it means that the condition part does not have any bit position limitations. Therefore all the rules can be applied to much larger multiplexer problems and will still yield the correct result.

### 6.2.3.7 Hidden Multiplexer

Figure 6.15 illustrates the results for the 18 bit hidden multiplexer problem. All three systems were able to solve the problem. XCSCF\* converged within about 1 100 000 instances; slightly ahead of XCS, while XCSCFC converged within about 2 million training instances. Although this problem may not appear difficult in comparison to the 135-bit Mux – the 3x6 hidden multiplexer is only 18-bits long – for example, it does serve to showcase the usefulness of the rules produced by the system. The reason is that although the 3x6 hidden multiplexer has a smaller sample space, it does have a larger number of relevant bits in each instance. In this case the learned information was transferred to a related domain with success. This is evident as the same functionality that solved the ValueAt sub-problem was useful for solving the 3x6 hidden multiplexer problem. This implies that functionality from non-Boolean domain, e.g. Log, Power, was re-used to solve a hierarchical Boolean problem. The nascent capability of the rules – run 8 produced 160 specific rules compared to the low number of rules needed to solve the multiplexer problems – for solving the hidden multiplexer looks more positive when certain details are considered. For instance, XCSCFC had been trained with Boolean operators as well as the odd and even parity problems. Although XCSCF\* uses the NAND function as a boot-strap, it does not occur in the actual feature

Table 6.16: Final Code Fragment Actions from maximally general rules produced by 264-8205 Multiplexer tests (No training was involved).

Problem	CFs Produced
264Mux	CF_272 CF_274 M b
	CF_276 CF_284 M b
	CF_288 CF_274 M b
	CF_289 b
521Mux	CF_529 CF_531 M b
	CF_533 CF_541 M b
	CF_545 CF_531 M b
	CF_546 b
1034Mux	CF_1042 CF_1044 M b
	CF_1046 CF_1054 M b
	CF_1058 CF_1044 M b
	CF_1059 b
2059Mux	CF_2067 CF_2069 M b
	CF_2071 CF_2079 M b
	CF_2083 CF_2069 M b
	CF_2084 b
4108Mux	CF_4116 CF_4118 M b
	CF_4120 CF_4128 M b
	CF_4132 CF_4118 M b
	CF_4133 b
8205Mux	CF_8213 CF_8215 M b
	CF_8217 CF_8225 M b
	CF_8229 CF_8215 M b
	CF_8230 b

construction that takes place during the explore phase. This was a design decision as NAND is not directly used in any of the fundamental steps. Therefore XCSCF\* was able to transfer knowledge blocks successfully in spite of not having been trained with a number of functions that naturally benefited XCSCFC, i.e. boolean operators and parity problems.

The rules produced by the system are illustrated in Table 6.17. Unlike the rules produced by the fundamental stages, i.e. (KBits, KBitString, Bin2Int, AddressOf, ValueAt), or the 6-bit multiplexer, these exhibit a higher number of specific alleles (non-don'tCare features) in the condition part. The action part retains its usual configuration of the known learned functions and their required inputs. This might be indicative of the difficulty in solving the Parity problem, coupled with the hierarchical nature of the hidden multiplexer.

Table 6.17: Final rules learned while solving the 18-bit Hidden Multiplexer (Even Parity). The function denoted by the letter b represents the learned ValueAt function. It returns the value of the data bit addressed by the hidden multiplexer.

Condition	Action
1 1 1 1 1 1 1 # 0 1 # # # 1 # 1 #	CF_38 CF_39 M b
0 0 0 1 1 1 # # # # # 1 0 1 # # #	CF_30 CF_41 M b
1 1 1 0 0 1 # 1 0 # 1 # # # # # 1	CF_25 CF_33 M b
0 0 0 1 0 1 # # # # # # # # 1 1 0	CF_41 CF_41 M b
0 1 0 1 1 0 # # # 0 0 1 # # # # #	CF_27 CF_37 M b
0 0 0 1 0 1 # # # # # # # # 1 1 0	CF_33 CF_38 M b
0 0 0 1 0 1 # # # # # # # # 1 0 1	CF_35 CF_30 M b
0 0 0 1 1 1 # # # # # 0 0 0 # # #	CF_34 CF_30 M b
...	...

The results for the 3x6 hidden multiplexer were reused for learning the 3x11 hidden multiplexer using training and testing. Figure 6.16 shows

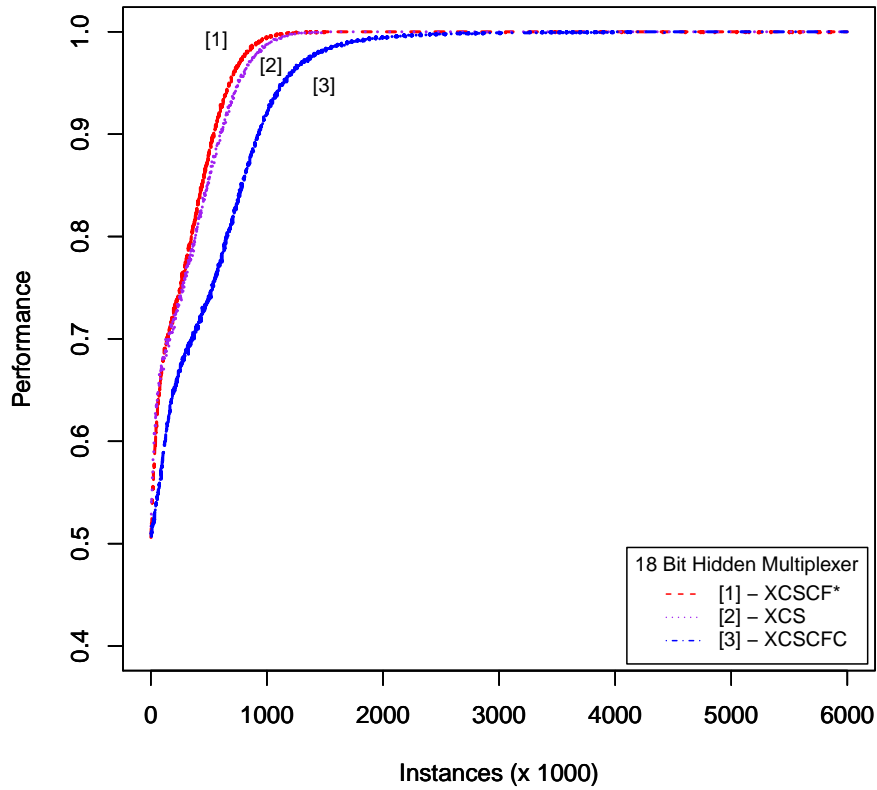


Figure 6.15: 18-bit Hidden Mux Solution: The rules produced by the ValueAt sub-problem were reused in the related Hidden Multiplexer domain.

the results of these experiments. The proposed system was not capable of learning the problem. Also, XCS failed to learn the problem as well. On the other hand, XCSCFC converged with about 4 000 000 training examples. It appears that the rules produced by the ValueAt and the 3x6 hidden multiplexer problems were not useful towards solving this larger problem.

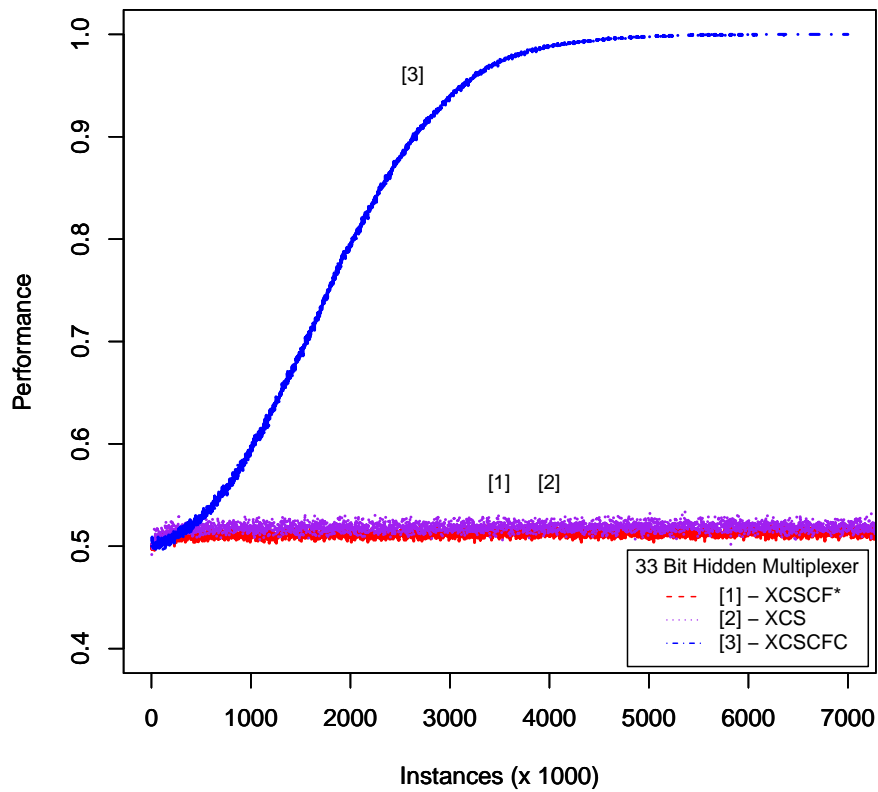


Figure 6.16: 33-bit Hidden Mux Solution: The rules produced up to and including the 3x6 hidden multiplexer were reused here. There was training and testing involved.

The results for another set of experiments for the 3x11 hidden multi-

plexer can be seen in Figure 6.17. These experiments reused all the rules produced up to and including the ValueAt sub-problem. The system was incapable of learning the problem. Again, this is indicative of missing functionality for solving the problem. The most obvious choice is the Parity function as this was present in previous work that was able to solve the 3x11 hidden multiplexer.

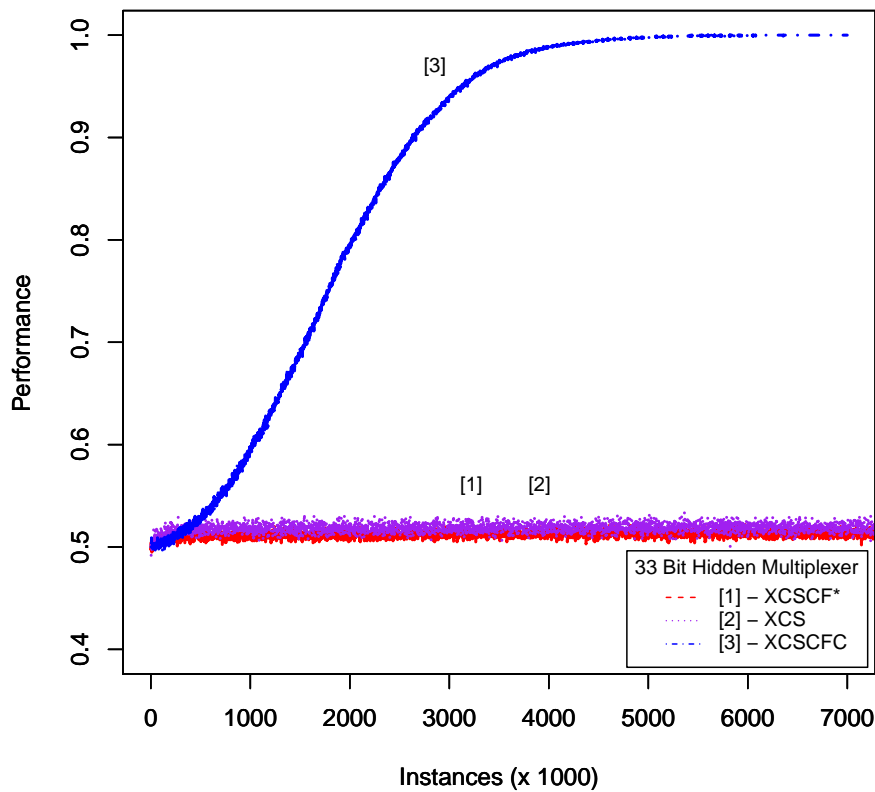


Figure 6.17: 33-bit Hidden Mux Solution: The rules produced up to and including the ValueAt stage were reused here. There was training and testing involved.

### 6.2.4 Interpretation of Results

Throughout this work, the manner in which humans approach tasks has been a recurrent theme. In fact, it forms an essential part of this work [64]. Furthermore, it is considered that the reason this approach can solve problems to a much larger scale than previously, is that human knowledge separated the problem into appropriate, simpler sub-problems [57]. However, it is still a difficult task to learn each sub-problem in such a way that learned knowledge/functionality can be transferred. Furthermore, the need to learn to combine these blocks effectively increases the difficulty.

The way that humans select sub-problems is similar to that of humans selecting function sets in standard EC approaches. In these, too few or inappropriate selection prevents effective learning, while selecting too many unnecessary components inhibits training [55]. Analyzing results for robustness and checking for the inclusion of redundant or irrelevant axioms yielded an interesting finding. In these experiments the ceiling function was available, but never used by the final solutions.

It is evident that XCSCF\* has benefited from the transfer of learned information from each of the sub-problems. This can be observed in the rules produced by the fundamental sub-problems; they contain functionality learned in all the layers. Although a step by step plan was not given to the system, it was still capable of combining the available functions in a productive way, see Figure 6.14. This property of the new system appears to be similar to deriving a set of Threshold Concepts, where significant learning towards the final target problem only advances once the proper chain of functionality is formed and evaluated. This is shown in the structure of the rules produced, where the functions have been combined by the system in order to produce maximally general rules that can scale to any size  $n$  multiplexer problem, see Table 6.16. However, it is concluded that the final structure of the CFs is a product of the reinforcement mechanism of XCS. This is shown in Table 6.12, here the system has found two rules

which provide a solution, and each rule has an amount of fitness. However, both rules have a similar structure in that the ValueAt function is the topmost function with function M as its child. The two leaf nodes consist of CFs. The rules providing the maximum reward get to exist longer and participate in breeding.

The proposed system, XCSCF\*, was capable of solving the 135 and the 264-bit multiplexer problems following the normal training and testing phases for XCS. However, for the 521-bit multiplexer problem it never fully converged. The solutions to the aforementioned problems demonstrated that knowledge transfer benefited the system in learning the problems; the rules demonstrate compounding usage of previously learned functionality at each subsequent layer. The non-monotonic shape of the performance graphs illustrates that the system faced periods of difficulty in learning, primarily during the 264-bit problem. However it eventually converged fully. Surprisingly, the system performed better against the larger 521-bit problem, in terms of fewer training instances to reach maximum performance. Also, the curve was smoother than the one for the 264-bit problem, meaning that it learned more quickly and in a more consistent progression. It is hypothesized that the increasingly larger value for the p\_don'tCare property was instrumental in enabling the solution to these problems which was different in the 264-bit multiplexer (0.95) and the 521-bit multiplexer (0.99).

It is not clear why XCSCF\* would fail to learn past the 521-bit Multiplexer (it never fully converged for this problem). None of the experiments for larger problems were able to learn. It is suggested that the problem has to do with the training phase. It is possible that as the problems scale, the training phase is faced with a large pool of classifiers to use in a matchset and then in an actionset, many of these are of the type with low fitness during the early stages of learning. Since the training part picks a random action winner, it is conceivable that many of these action winners will provide the incorrect answer and therefore delay learning of the problem. As



the problems become more difficult, this situation is exacerbated to the point that the system stops learning altogether. This presents a credible reason because the same rules were able to solve much larger problems, albeit without using training.

XCSCF\* also performed well with the 3x6 hidden multiplexer problem, see Figure 6.15. Also, further evidence of knowledge transfer was observed with the solution of this problem. This was an unexpected accomplishment, given the complex structure of the hidden multiplexer [17] and the novel representation. This difficulty is demonstrated by the larger number of rules required to solve the problem, and specially, by the relatively more specific rules in comparison with the 6-bit multiplexer final rules-set.

Another item of interest is that the majority of the rules contain environment features (specific bits) towards the left side of the condition while the don'tCares tend to appear in the middle and right side of the condition, corresponding to the address part of the higher level multiplexer problems. This is obviously due to the influence that genetic transfer exerts on the creation and evolution of the population. XCS-based systems exert a pressure leading to maximally general rules. This generality can be achieved by reusing useful rules to create new children which share some of their parents' genetic makeup. Those classifiers that tend to solve the layers correctly, and hence receive the highest reward, tend to remain long enough to breed. This has led to the system identifying the first six bits, as very important, which would have been learned while solving the multiplexer part of the problem. That is, consider the first six bits as eventually yielding the two address bits for the 6-bit Mux that is in the upper level. The rest of the condition could be composed mostly of don'tCares as only one more upper level bit is required in order to produce the correct action for the entire problem.

When XCSCF\* was tasked with the 3x11 hidden multiplexer problem, it was not capable of solving it. Two sets of experiments were run, one set

involved all the rules produced by the 3x6 hidden multiplexer, while the other involved all the rules produced by the ValueAt sub-problem. It is considered that the experiential toolbox is missing certain very important axioms which could be useful in solving the more difficult hidden multiplexer problems. One of the obvious axioms is the Parity operator. This could be learned by the system in one or two stages, primarily by learning to tell the difference between different types of bits. Arguably this could be done in a similar fashion as was done for the multiplexer solution; the problem could be broken apart by a human into its main constituent parts. Then it could just be added to the experiential toolbox, ready to be reused as needed.

#### 6.2.4.1 Additional Observations on the $n$ -bit Multiplexer

One of the more interesting questions regarding the results is the technique's usage of leaf nodes. It is clear that learned function  $N$  (returns  $k$ , the number of address bits) appears in most of the branches of the CF sub-trees. It appears either on its own or as part of a larger chain of CFs. It also appears in the guise of CF 6, but this CF does not appear in any other chains for the simple fact that it has the exact same configuration as function  $N$ . This means that it has constant  $L$  as its leaf node. Furthermore, since the technique associates a learned function with the CFs learned, and since the learned function  $N$  has the same structure as CF 6, it follows that CF 6 is redundant information. The important item of information here is that the very last node of the function is constant  $L$ . This is critical because it informs the CF machinery of the length of the current problem. With this information, the rest of the CF evaluations proceed as prescribed by the connecting functions, axioms or skills. The fact that the solutions contain constant  $L$  as a seminal CF member, has implications for virtually all of the rest of the relationships. It means that for the final rules to provide the correct answer to a particular multiplexer problem, constant  $L$  should be included in the chain of CFs. This is evidenced by the fact that the

ubiquitous  $N$  function contains constant  $L$  in its makeup.

Another interesting observation is that unlike the execution of XCSCFC, where the terminal nodes have a direct impact on the final answer returned by the CF sub-tree, it was discovered that for XCSCF\* this does not appear to be the case. In XCSCFC information in the terminal nodes is manipulated up the branches (processed by the pre-coded functions located at the nodes) until the final output is produced. A full trace of the 6-bit multiplexer was conducted for one of the solutions for XCSCF\*, see Figure 6.14. It was identified that repeating patterns of learned knowledge are located throughout the chain of CFs. This indicates that it was helpful to store the computed values of said functional blocks for later usage. It was also observed that for the general solution, the two main branches: CF\_28 and CF\_16 feed their outputs into the function  $M$  but these outputs are not used directly when  $M$  is evaluated. The reason is that the learned function  $M$  relies on its own set of rules to compute its output, which disregards the inputs from its two child branches. Function  $M$  is the AddressOf function which was trained on inputs of a bitstring and past knowledge of  $L$  and  $k$ . The training sets were implicitly based on the multiplexer rather than on a general address of a boolean problem, hence the evolved function did not require explicit inputs. This idiosyncrasy of the technique was addressed above, and it was mainly stated that all branches of learned function  $M$  begin with constant  $L$ , therefore no further inputs are required or expected due to the layered learning. Even if both sub-branches were to be ignored, the correct answer would still be provided by  $M$  as it has this CF knowledge in its learned functions.

The question that arises about the redundancy in the CF chains is where it could be useful in other types of CF based LCSs. The answer is not so simple. It is hypothesized that because the technique is based on XCSCFA, it promotes a compartmentalization of the major functional blocks, see Figure 6.14. For example, inside of the function  $M$ , all of the functional blocks get evaluated as if they were one single function. The

result gets passed up to the function '=' (this is just the internal tag the system gives the ValueAt function) which then returns the action for the problem. However, the results from the two children of  $M$  get computed but never get sent to any of the internal functions of  $M$ , not even to function '='. It is hypothesized that the reason this phenomenon occurs is simply because all of the CF chains inside of function  $M$  eventually end in the constant  $L$ . The training provided the system with a choice of forming CF sub-tree leaf nodes composed of the constant  $L$ , a previously learned CF or an environment feature. In the rules, the constant  $L$  became an atomic property of the CF chain, a value which could not be decomposed into any other value; it could only be replaced with the value for the current problem [100]. In essence, the constant  $L$  provided a starting point, which the system used to eventually compute the final answer by moving up the CF chain. For example, function  $N$ , when expanded, starts the evaluation with  $L$  and proceeds to the Log and Floor axioms. Essentially, using the constant  $L$  'transforms' the  $N$  function to one without the need for any other inputs. This was not anticipated when the system was being put together, however, the effect has provided a subtle benefit; the children branches have become partially redundant. In spite of this redundancy, the child branches are important because they capture relationships between the learned functionality, axioms and skills which lend themselves to compact solutions.

### 6.3 Chapter Summary

The work in this chapter demonstrates how the Layered Learning (LL) approach has been beneficial in facilitating increased scalability. A LL approach was used to facilitate the learning of the  $n$ -bit multiplexer problem. The problem was broken into five main sub-problems which were then solved in stages. Each subsequent stage was built upon the foundations provided by the previous sub-problems. This technique is also aligned

with the Threshold Concept in that each of the sub-problems was critically important in facilitating the learning of the next sub-problem. But more importantly, all of the sub-problems together were transformative and led to a scalable, general solution to the  $n$ -bit multiplexer.

The generality of the solution is built upon the power of expression imbued in Code Fragments (CFs). Since many types of actions were involved during the training process, CFs provided a viable way to express the action part of the rules evolved. For this reason, the sub-problems were composed of XCSCFA-based systems. The condition part used a ternary alphabet, i.e.  $\{0, 1, \#\}$ , however the final rules were composed of don'tCares, making them maximally general. Of course any rules-based classifier system could have been used instead of the one used by the solution (XCSCFA). However, the candidates would have required the capability of expressing complex actions, since different types of actions are needed to find the solutions. This is why CFs were chosen for the representation of the action part in this work.

The ValueAt sub-problem evolved a series of general rules which were then used to solve the 6-bit multiplexer problem. The rules produced here were then used to solve the 264-8205 bit multiplexer problems. These experiments did not involve any training, just testing, in order to fit the 6-bit input to the new, larger domain. The results were notable as the performance demonstrated a rapid convergence rate in terms of test samples. More importantly, the astronomically huge size of the problem domain was sufficient to illustrate the benefits of the technique. Being that no other LCS or EC-based system has been capable of solving the magnitudes of problems presented here by using inductive learning, this qualifies as a step forward in the field of Evolutionary Computation.

The technique was also tested against the hidden multiplexer and here the results were mixed. Initially the rules were useful in solving the 3x6 hidden multiplexer, however, the 3x11 hidden multiplexer was too difficult. It is hypothesized that necessary functionality is missing from the

experiential toolbox. It is estimated that at the very least, a learned function or an axiom reflecting the Parity boolean function is needed. This is anticipated to advance the learning process and to help the system in combining the necessary building blocks needed to solve the hierarchical problem.

Another finding involved the structure of the final rules. A typical rule from the 6-bit multiplexer problem solution was fully traced and it was discovered that not all branches cooperate towards finding the final answer at the very top of the CF chain. For example, the function  $M$  was not directly dependent on the inputs provided by its two child branches. It relied entirely on its own set of rules and CFs. This is due to the fact that there exist certain redundancies in the functionality of the CF chains, that is, certain rules and their CFs appear in different functional blocks. This makes it feasible for a function like  $M$  to compute a final answer based solely on its own associated knowledge. It was also discovered that the constant  $L$  plays a critical role in this idiosyncrasy of the technique.

Overall, XCSCF\* has been successful in evolving maximally general, scalable rules which were used to solve a number of very difficult multiplexer problems. This has expanded the scalability of XCSCFA-based systems to multiplexer problems of any size, something that was out of reach for LCS based systems until now. However, there is opportunity for improvement in terms of the hidden multiplexer problem. The technique demonstrated a limited capability for transferring learned information, and more work is required for overcoming the more difficult problems.

# Chapter 7

## Conclusions and Future Work

The overall goal of this thesis was to improve the scalability of learning classifier systems. Emphasis has been placed on XCS systems extended with Code Fragments. In this novel work, CFs were customized to reuse learned functionality at the root nodes of the CF sub-trees. In addition, hard-coded functions, routinely used in EC techniques, were replaced with learned rule-sets. These learned rule-sets contributed to the reduction of the search space in problems such as the multiplexer and hidden multiplexer. The results from the newly developed systems were compared with the existing related systems.

The rest of this chapter presents the achieved objectives, principal conclusions from each of the contribution chapters, and the future work that flows from this original research work.

### 7.1 Achieved Objectives

The following research objectives have been fulfilled by this original work to achieve the overall research goal.

1. It was shown that re-using CF-based building blocks at the root nodes of CF sub-trees increases scalability. The new system known

as  $XCSCF^2$  was trained with several problems in the Boolean domain. The new system then demonstrated increased performance in terms of required training instances to converge fully. This performance was compared to related systems like XCS and XCSCFC.

2. A novel compaction technique was designed and implemented. This technique, known as XCSCF3, showcased the benefits of transforming a population of CF-based rules to another composed of ternary rules. CFs are powerfully expressive and can describe a problem domain in a compact and rich manner. Ternary rules are easy to interpret and quick to compute. However, they lack expressivity and are also subject to limitations in the location of the bits as well as the number of bits per classifier rule. Essentially, both sets of rules are equivalent, in the sense that the ternary rules were derived from the richly compact CF rules. XCSCF3 was shown to perform well in comparison with other related systems. It was able to solve up to the 70-bit multiplexer and the 3x11 hidden multiplexer. On the other hand, the technique is not scalable beyond the 70-bit multiplexer. Additionally, although XCSCF3 was capable of solving the 3x11 hidden multiplexer, it required more training instances than XCSCFC.
3. It was shown that Layered Learning, combined with Transfer Learning can provide benefits in scaling for a CF based LCS by solving an intractable problem. The problem known as the multiplexer was partitioned by a human into five constituent parts. Then the novel technique known as XCSCF\* was trained with these sub-parts, each sub-problem re-used the learned building blocks from the previous one. After the elemental training was complete, the system was trained with a 6-bit multiplexer problem. The rules produced at this stage were tested against other much larger multiplexer problems. The rules were shown to be scalable and optimally general for each of the tests. Although the rules were scalable to problems with a sam-



ple space of astronomically large sizes, it was noted that a sample size of 1 000 000 testing samples is large enough to discover many deficiencies.

The above mentioned achievements form a major portion of this thesis. They encompass the original research objectives, however additional findings were discovered. A full tracing of the rules evolved by XCSCF\* for a 6-bit multiplexer revealed that the technique produced several redundancies. These redundancies provided the system with the opportunity to store these computed values for future usage. It was also confirmed that the encoding used by CFs is rich and compact. A simple two level solution to a simple multiplexer problem has the tendency to expand into CF sub-trees with many more levels. This also confirms the benefit of human intervention in partitioning the multiplexer problem into sub-problems. Prior to designing XCSCF\*, it was known that the standard CF technique was incapable of learning all the necessary functionality in one layer. This is the reason that the multiplexer problem required partitioning.

## 7.2 Main Conclusions

This section presents the main conclusions and highlights from the three major contribution chapters (Chapter 4 to Chapter 6).

### 7.2.1 Reusing CFs in the Root Nodes of CF Sub-trees

The original CF sub-tree formulation was extended to accommodate learned information at the root nodes of the sub-trees. The CFs also were changed to use learned rule-sets at the root nodes instead of the usual hard-coded functions. These rule-sets were learned from functions originating in the Boolean domain, they included {NAND, OR, AND, XOR, NOR}. The system was then given problems in the multiplexer domain to solve.

The new technique known as  $XCSCF^2$  showed that it is possible to learn building blocks of functionality from basic boolean rules. In addition, it was shown that knowledge learned in the boolean logic domain could be used to solve problems in a different domain such as the multiplexer. This transfer of knowledge, learned in one domain, to a related domain, facilitated the solution to problems in the new domain.

The  $XCSCF^2$  successfully solved the 6-20 bit multiplexer problems with better than expected results. In other words, its training instance requirements were minimal compared to the other systems being compared. However, it was slower when it came to the more difficult problems, but not prohibitively so.

The  $XCSCF^2$  offered opportunity for further scalability by linking a set of Function rule-sets to their respective CFs. This extension to the CF technique is something that was explored in Chapter 4.

### 7.2.2 Develop a New Compaction Technique

A new CF compaction technique was introduced in Chapter 5. This technique, known as  $XCSCF^3$ , converted the final population of CF-based rules into a ternary rules-set. The techniques developed were of two kinds, offline and online compaction techniques. These will be described further.

The offline compaction technique was a post-processing translation that operated on the final population of rules produced by the system. The process took place after the final set of classifiers had been evolved. Only the classifiers meeting a pre-set criteria of experience, fitness and accuracy were considered. The translation consisted mainly of presenting the CF rules with a pre-set number of problem instances to afford a translation of the CFs into rules using a ternary alphabet.

The offline compaction technique demonstrated that it is possible to translate CF rules into a ternary format. Additionally, These rules can be easier to compute than the normal CF rules. This was shown by the so-

lutions to problems including the 70-bit multiplexer and the 3x11 hidden multiplexer.

The offline compaction technique provided new benefits to the field of LCS, as shown by the experimental data, see Figure 4.7. Said experiments demonstrate that the offline compaction technique is beneficial in terms of training instances required to solve a problem. However the technique contains a number limitations. First, the fact that it is a post-process means that it adds to the execution time required for a normal run. In addition, the process is very lengthy and requires a large number of training instances to produce a ternary rules-set from the CF population. This is complicated by the fact that the problem instances presented to the algorithm scale with the problem. This implies that the technique will stop creating DRs at some point. This is because it is infeasible to enumerate all the possible instances of large problems, such as the 70-bit multiplexer.

The online compaction technique was a mixture of in-process and post-process. Like its offline sibling, its overall goal was of converting CF-based rules into ternary rules. The major difference between both techniques is that the online compaction made use of the problem instances normally presented to the learning agent. Also, it divides its processing between the explore and exploit stages. However, there is some post-processing after the final population of classifiers has been evolved, but this is restricted to removing useless ternary rules that are no longer needed or that are duplicates.

The online compaction technique was capable of producing ternary rules in a quicker manner than the offline version. This was due to the simplicity of the process which made use of the normal explore and exploit phases of XCS. Also, the compaction technique was presented with the same training/testing instances as the learning agent. This resulted in substantial time savings, culminating in the translation of the 70-bit multiplexer rules. This was something that the offline technique was incapable of producing.

In spite of providing reduced time requirements for the translation of CF rules, the online technique still reached a limit in the translation of rules. For example, for the 135-bit multiplexer, the online technique was beginning to learn the problem but was not able to do this in a reasonable amount of time. Furthermore, it was observed that the ternary rules produced for the 70-bit multiplexer were not optimal. These substandard rules then made it difficult for the system to learn the 135-bit multiplexer. It is plausible that by adjusting the parameter controlling when the Distillation of rules begins, more optimal rules could be evolved.

### **7.2.3 Develop a Layered Learning Methodology for a CF System**

Previous work has shown that CF-based learning classifier systems can demonstrate better scalability than standard XCS systems using the original ternary representation for the rules. Although CFs provide many benefits, they also contain limitations. Since the CF sub-trees can reuse other CFs at the leaf nodes, the chains of CFs can grow intractably long. This compounding effect reaches a point where learning is no longer possible for the system. Even using different techniques such as learned rule-sets at the root nodes is not scalable to the very large problems, those beyond the 135-bit multiplexer. XCSCFC successfully solved the 135-bit multiplexer, but it was not capable of solving the more difficult problems such as the 264-bit multiplexer and beyond.

The multiplexer is a difficult problem because it has epistasis and is highly non-linear. This means that the importance of certain bits depends on other bits in the message string. Also, this problem is popular in research and therefore is routinely used as a benchmark. The multiplexer has been troublesome for some CF-based systems like XCSCFC due to its properties. It is not possible to discover regularities that will ultimately lead to a scalable and general solution of the multiplexer problem.

Since a general solution to the multiplexer problem involves functionality from different domains, e.g. real, Boolean and so forth, it was hypothesized that a human could partition the problem into constituent sub-problems. These sub-problems would then be used to train the system known as XCSCF\* in a series of steps. Once the fundamental training were complete, the system would be ready to learn the multiplexer problem itself.

XCSCF\* produced a set of general rules which contained all the functionality learned from all the previous sub-problems. These rules were tested against a number of increasingly difficult multiplexer problems, all of which the technique was able to solve with the aforementioned rules. The outcome of the experiments indicated that CFs were instrumental in this success. The CFs, along with a Layered Learning approach, led to a set of rules that are scalable to astronomically large multiplexer problems.

## 7.3 Future Work

### 7.3.1 Distilled Rules

XCS systems based on CFs (described in Chapter 4 onwards) play an important role in the novel work described here. Consequently, it is clear that the re-usability of previously learned rules is of paramount importance to this work [3], [4], [49]. However, it has emerged that there are various ways in which said rules and CFs can be reused. In the new work presented in Chapter 5, new classifiers were imbued with new CFs for each condition bit. These CFs were created on-the-fly from a random choice of previously learned functions and their assigned CFs. The leaf nodes of the new CFs could contain either an environment feature or one of the CFs which had been previously linked to its parent function.

It is theorized that by reusing previously learned CF-based rules as a template for the new classifiers, it is possible to reduce the search space,

when tackling the subsequent, more difficult problems. For example, if the system were facing the 11-bit multiplexer, the new classifier conditions would get seeded with entire rules from the 6-bit multiplexer function. The remaining condition bits would follow the current process. That is, functions would be chosen at random and the leaf nodes would be assigned linked CFs, as described in Chapter 5. The reasoning behind this is as follows: since the final rule population for the previous problem was found to be helpful, then it must also present benefits to the more difficult current problem.

Another opportunity for improvement is the fact that the classifiers do not implement any CF fitness mechanism, besides the original XCS implementation, i.e. fitness, accuracy, experience, etc. However these metrics operate at the classifier level. It could be worthwhile to implement an additional fitness mechanism at the function level. In other words, functions whose classifiers do not do well during the test phase would receive a negative reward or some other type of censure. The functions that do well would receive a positive score and its CFs would also be impacted positively. This could be extended in granularity so that instead of the functions being impacted, only the CFs linked to the function would get a score.

There is another way in which the proposed technique could improve its performance. This involves visualizing the long chains of CFs as monolithic objects. That is to say, a CF sub-tree can be thought of as an object that accepts a number of inputs – the leaf nodes – and produces an output at the first root node, i.e. the one at the very top of the CF sub-tree. The leaf nodes could constitute additional CFs, but this is not important as only the output produced by the topmost CF is the one that produces the value of the input for the monolithic object. It is then possible to construct a truth table. As the CFs are evaluated, this truth table can be stored in temporary memory and utilized whenever the same CF is encountered again in the same run. Figure 7.1 depicts the concept described here.

This technique is anticipated to produce an additional shortcut when evaluating the CFs. At first, the process would proceed as described in [3], with the additional step of adding the current inputs to the truth table along with its value. The number of possible combinations of inputs and corresponding values in this case would be 16. In a future time step, when the list of possible values would be more complete, the system could start searching the list and reusing any of the values that had already been calculated. This is anticipated to save computing time by freeing the system from recalculating long chains of CFs. Of course, there is an initial investment of resources as the CF values list starts to be filled. However, this initial investment in time should be compensated once the list is complete. It is not clear at this time the full impact of searching the list of stored inputs and the corresponding values. However by keeping the number of inputs for the monolithic object small, the linear search may be kept at a tractable amount of time. This would make it more efficient and would preclude the system from dealing with very large numbers of parameters. However, using this last technique alone would deprive the system of the CFs evolved for the functions with more than the maximum number of parameters allowed.

The seemingly natural combination of the template method and monolithic object method mentioned above could deliver increased scalability.

### 7.3.2 n-Bit Multiplexer

The current system is serial, which has the following consequences: training is lengthy, useful rules are discovered in layers, the discovered CFs are compact. A system is possible where each sub-problem is addressed in parallel with the resulting functionality and building blocks becoming available in a shared location. Simple (low-level) problems would complete first, enabling higher-order functions to be solved consecutively. The links and order of solved problems would contain interesting meta-

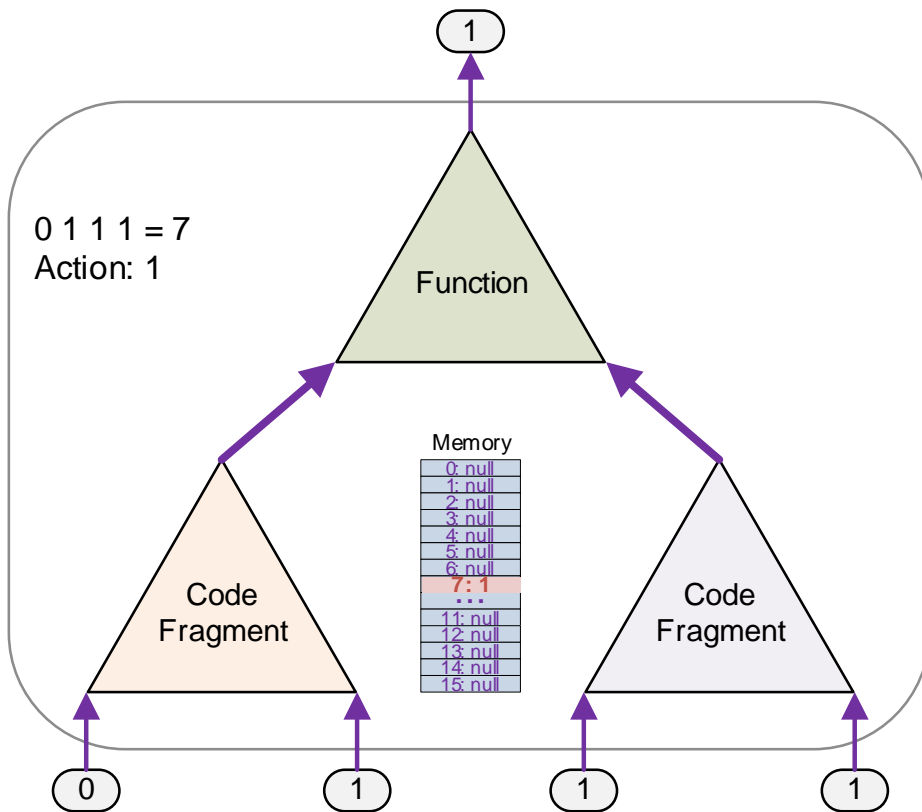


Figure 7.1: CFs viewed as monolithic objects. The entire assembly can be conceptualized as one large object with temporary memory to catalog all the possible combinations of inputs to outputs. The number of inputs should be limited so as not to make this method unwieldy.



knowledge, a form of learning curricula.

Some of the practical candidate problems to be solved next are: Count Ones, Even Parity, Odd Parity. With these solutions being part of the functional repository of the system.



# Bibliography

- [1] AHLUWALIA, M., AND BULL, L. A genetic programming-based classifier system. In *Proceedings of GECCO '99* (Orlando, Florida, USA, 13-17 July 1999), W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., vol. 1, Morgan Kaufmann, pp. 11–18.
- [2] AHLUWALIA, M., AND BULL, L. Coevolving functions in genetic programming. *Journal of Systems Architecture* 47, 7 (2001), 573–585.
- [3] ALVAREZ, I., BROWNE, W., AND ZHANG, M. Reusing learned functionality in XCS: code fragments with constructed functionality and constructed features. In *Genetic and Evolutionary Computation Conference GECCO '14 Companion* (2014), ACM, pp. 969–976.
- [4] ALVAREZ, I., BROWNE, W., AND ZHANG, M. Reusing learned functionality to address complex boolean functions. In *10th International Conference on Simulated Evolution and Learning SEAL* (2014), Springer, pp. 383–394.
- [5] ALVAREZ, I. M., BROWNE, W. N., AND ZHANG, M. Compaction for code fragment based learning classifier systems. In *Artificial Life and Computational Intelligence - Second Australasian Conference, ACALCI 2016, Canberra, ACT, Australia, February 2-5, 2016, Proceedings* (2016), T. Ray, R. A. Sarker, and X. Li, Eds., vol. 9592 of *Lecture Notes in Computer Science*, Springer, pp. 41–53.

- [6] ALVAREZ, I. M., BROWNE, W. N., AND ZHANG, M. Compaction for code fragment based learning classifier systems - redux. In *IEEE Congress on Evolutionary Computation, CEC 2016, Vancouver, BC, Canada, July 24-29, 2016* (2016), pp. 2217–2224.
- [7] ALVAREZ, I. M., BROWNE, W. N., AND ZHANG, M. Human-inspired scaling in learning classifier systems: Case study on the n-bit multiplexer problem set. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016* (2016), pp. 429–436.
- [8] AMIR HAERI, M., EBADZADEH, M. M., AND FOLINO, G. Improving gp generalization: a variance-based layered learning approach. *Genetic Programming and Evolvable Machines* 16, 1 (2015), 27–55.
- [9] BAJURNOW, A., AND CIESIELSKI, V. Layered learning for evolving goal scoring behavior in soccer players. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation* (Portland, Oregon, 20-23 June 2004), IEEE Press, pp. 1828–1835.
- [10] BANICH, M. T., AND BELGER, A. Interhemispheric interaction: How do the hemispheres divide and conquer a task? *Cortex* 26, 1 (1990), 77–94.
- [11] BANTHAF, W., NORDIN, P., KELLER, R. E., AND FRANCONI, F. D. *Genetic Programming : An Introduction*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.
- [12] BOOKER, L. B., GOLDBERG, D. E., AND HOLLAND, J. H. Classifier systems and genetic algorithms. *Artificial Intelligence* 40 (1989), 235–282.
- [13] BULL, L. A brief history of learning classifier systems: from CS-1 to XCS and its variants. *Evolutionary Intelligence* 8, 2-3 (2015), 55–70.

- [14] BULL, L., AND AHLUWALIA, M. Coevolving functions in genetic programming: Classification using k-nearest-neighbor. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)* 2 (July 1999), 947–952.
- [15] BULL, L., AND KOVACS, T., Eds. *Foundations of Learning Classifier Systems*, vol. 183 of *Studies in Fuzziness and Soft Computing*. Springer, 2005.
- [16] BURLING-CLARIDGE, F., IQBAL, M., AND ZHANG, M. Evolutionary algorithms for classification of mammographic densities using local binary patterns and statistical features. In *Proceedings of 2016 IEEE Congress on Evolutionary Computation (CEC 2016)* (Vancouver, 24-29 July 2016), Y.-S. Ong, Ed., IEEE Press.
- [17] BUTZ, M. V. *Rule-Based Evolutionary Online Learning Systems: A Principal Approach to LCS Analysis and Design*. Springer, Berlin, Germany, 2006.
- [18] BUTZ, M. V., KOVACS, T., LANZI, P. L., AND WILSON, S. W. How XCS evolves accurate classifiers. In *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference (2001)*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., Morgan Kaufmann, pp. 927–934.
- [19] BUTZ, M. V., KOVACS, T., LANZI, P. L., AND WILSON, S. W. Toward a theory of generalization and learning in XCS. *IEEE-EC* 8 (Feb. 2004), 28–46.
- [20] BUTZ, M. V., LANZI, P. L., AND WILSON, S. W. Hyper-ellipsoidal conditions in xcs: rotation, linear approximation, and solution structure. In *Proc. genetic and evolutionary computation conference (GECCO 2006)* (2006), ACM, pp. 1457–1464.

- [21] BUTZ, M. V., PELIKAN, M., LLORÀ, X., AND GOLDBERG, D. E. Extracted global structure makes local building block processing effective in XCS. In *Genetic and evolutionary computation conference, GECCO 2005* (2005), H. G. Beyer and U. M. O'Reilly, Eds., ACM, pp. 655–662.
- [22] BUTZ, M. V., AND WILSON, S. W. An algorithmic description of XCS. *Soft Computing* 6 (2002), 144–153.
- [23] CHEN, G., AND ZHANG, M. Evolving while-loop structures in genetic programming for factorial and ant problems. In *AI 2005: Advances in Artificial Intelligence*, S. Zhang and R. Jarvis, Eds., vol. 3809 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 1079–1085.
- [24] DARWIN, C. *The Illustrated Origin of Species - Abridged and Introduced by Richard E. Leakey*. Faber and Faber Ltd, London, England, 1979.
- [25] DIXON, W. *An Investigation of the XCS Classifier System in Data Mining*. The University of Reading, Reading, United Kingdom, 2004.
- [26] FALKNER, N. J. G., VIVIAN, R. J., AND FALKNER, K. E. Computer science education: The first threshold concept. In *LaTiCE* (2013), IEEE Computer Society, pp. 39–46.
- [27] FENG, L., ONG, Y.-S., TAN, A.-H., AND TSANG, I. W. Memes as building blocks: a case study on evolutionary optimization + transfer learning for routing problems. *Memetic Computing* 7, 3 (2015), 159–180.
- [28] FERREIRA, C. Automatically defined functions in gene expression programming. In *Genetic Systems Programming: Theory and Experiences*, N. Nedjah, A. Abraham, and L. de Macedo Mourelle, Eds., vol. 13 of *Studies in Computational Intelligence*. Springer, Germany, 2006, pp. 21–56.

- [29] FESTINGER, L. *A Theory of Cognitive Dissonance*. Stanford University Press, Stanford, California, 1957.
- [30] GOLDBERG, D. The genetic algorithm approach: Why, how, and what next. In *Adaptive and Learning Systems.*, K. Narendra, Ed. Plenum, 1986.
- [31] GOLDBERG, D. E. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems* 3 (1989), 493–530.
- [32] GOLDBERG, D. E., AND HOLLAND, J. H. Genetic algorithms and machine learning. *MACHLEARN: Machine Learning* 3 (1988).
- [33] HOANG, T.-H., MCKAY, R. I., ESSAM, D., AND HOAI, N. X. On synergistic interactions between evolution, development and layered learning. *IEEE Transactions on Evolutionary Computation* 15, 3 (June 2011), 287–312.
- [34] HOLLAND, J. Adaptation. In *Progress in Theoretical Biology* (New York, 1976), R. Rosen and F. M. Snell, Eds., vol. 4, Academic Press, pp. 263–293.
- [35] HOLLAND, J. Escaping brittleness. In *Machine Learning*, R. Michalski, J. Carbonell, and T. Mitchell, Eds., vol. 2. Morgan Kaufmann, 1986.
- [36] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The University of Michigan Press, Ann Arbor, 1975.
- [37] HOLLAND, J. H., BOOKER, L. B., COLOMBETTI, M., DORIGO, M., GOLDBERG, D. E., FORREST, S., RIOLO, R. L., SMITH, R. E., LANZI, P. L., STOLZMANN, W., AND WILSON, S. W. What is a learning classifier system? In *IWLCS* (1999), P. L. Lanzi, W. Stolzmann, and

- S. W. Wilson, Eds., vol. 1813 of *Lecture Notes in Computer Science*, Springer, pp. 3–32.
- [38] HOLMES, LANZI, STOLZMANN, AND WILSON. Learning classifier systems: New models, successful applications. *IPL: Information Processing Letters* 82 (2002).
- [39] HSUAN-TA, L., PO-MING, L., AND TZU-CHIEN, H. The subsumption mechanism for XCS using code fragmented conditions. In *Proceedings Companion of the Genetic and Evolutionary Computation Conference* (2013), pp. 1275–1282.
- [40] HUELSBERGEN, L. Finding general solutions to the parity problem by evolving machine-language representations. In *Conference on Genetic Programming* (July 1998), pp. 158–166.
- [41] IOANNIDES, C., AND BROWNE, W. Investigating scaling of an abstracted LCS utilising ternary and S-expression alphabets. In *Learning Classifier Systems. 10th and 11th International Workshops (2006-2007)*, J. Bacardit, E. Bernadó-Mansilla, M. Butz, T. Kovacs, X. Llorà, and K. Takadama, Eds., vol. 4998/2008 of *Lecture Notes in Computer Science*. Springer, 2008, pp. 46–56.
- [42] IQBAL, M., BROWNE, W. N., AND ZHANG, M. Evolving optimum populations with xcs classifier systems. *Soft Computing* 2013, 17 (September 2012), 503–518.
- [43] IQBAL, M., BROWNE, W. N., AND ZHANG, M. Extracting and using building blocks of knowledge in learning classifier systems. In *GECCO* (2012), T. Soule and J. H. Moore, Eds., ACM, pp. 863–870.
- [44] IQBAL, M., BROWNE, W. N., AND ZHANG, M. XCSR with computed continuous action. *Proceedings of the Australasian Joint Conference on Artificial Intelligence* (2012), 350–361.



- [45] IQBAL, M., BROWNE, W. N., AND ZHANG, M. Comparison of two methods for computing action values in XCS with code-fragment actions. *GECCO'13 Companion* (July 2013), 1235–1242.
- [46] IQBAL, M., BROWNE, W. N., AND ZHANG, M. Extending learning classifier system with cyclic graphs for scalability on complex, large-scale boolean problems. *GECCO Proceedings of the 15th annual conference on Genetic and evolutionary computation* (2013), 1045–1052.
- [47] IQBAL, M., BROWNE, W. N., AND ZHANG, M. Learning complex, overlapping and niche imbalance boolean problems using XCS-based classifier systems. *Evolutionary Intelligence* 6, 2 (2013), 73–91.
- [48] IQBAL, M., BROWNE, W. N., AND ZHANG, M. Learning overlapping natured and niche imbalance boolean problems using XCS classifier systems. *Proceedings of the IEEE Congress on Evolutionary Computation* (2013), 1818–1825.
- [49] IQBAL, M., BROWNE, W. N., AND ZHANG, M. Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems. *IEEE Transactions on Evolutionary Computation* PP, 99 (September 2013), 1–16.
- [50] KOVACS, T. XCS classifier system reliably evolves accurate, complete, and minimal representations for boolean functions. In *Soft Computing in Engineering Design and Manufacturing* (1997), Roy, Chawdhry, and Pant, Eds., Springer-Verlag, London, pp. 59–68. <ftp://ftp.cs.bham.ac.uk/pub/authors/T.Kovacs/index.html>.
- [51] KOVACS, T. Strength or accuracy? A comparison of two approaches to fitness calculation in learning classifier systems. In *2nd International Workshop on Learning Classifier Systems* (Orlando, Florida, USA, 13 July 1999), P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds., pp. 258–265.

- [52] KOVACS, T. Genetics-based machine learning. In *Handbook of Natural Computation*, G. Rozenberg, T. Back, and J. N. Kok, Eds. Springer, 2012, pp. 937–986.
- [53] KOVACS, T. M. D. *A Comparison of Strength and Accuracy-Based Fitness in Learning Classifier Systems*. The University of Birmingham, Birmingham, United Kingdom, 2002.
- [54] KOZA, J. R. A hierarchical approach to learning the boolean multiplexer function. In *Foundations of Genetic Algorithms* (San Mateo, 1991), G. J. E. Rawlins, Ed., Morgan Kaufmann, pp. 171–192.
- [55] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- [56] KOZA, J. R. Hierarchical automatic function definition in genetic programming. In *Foundations of Genetic Algorithms 2* (Vail, Colorado, USA, 24–29 July 1992), L. D. Whitley, Ed., Morgan Kaufmann, pp. 297–318.
- [57] LAKE, B. M., SALAKHUTDINOV, R., AND TENENBAUM, J. B. Human-level concept learning through probabilistic program induction. *Science* 350, 6266 (Dec. 2015), 1332–1338.
- [58] LANZI, P., AND PERRUCCI, A. Extending the representation of classifier conditions part II : From messy coding to s-expressions. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999) 1* (July 1999), 345–352.
- [59] LANZI, P. L. Extending the representation of classifier conditions part I: From binary to messy coding. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999) 1* (1999), 337–344.
- [60] LANZI, P. L. XCS with stack-based genetic programming. In *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*

- (Canberra, 8-12 Dec. 2003), R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, Eds., IEEE Press, pp. 1186–1191.
- [61] LANZI, P. L. Learning classifier systems: Then and now. *Evol. Intel.* 1 (2008), 63–82.
- [62] LANZI, P. L., AND RIOLO, R. L. A roadmap to the last decade of learning classifier system research (from 1989 to 1999). In *Learning Classifier Systems. From Foundations to Applications* (Berlin, 2000), P. L. Lanzi, W. Stolzmann, and S. W. Wilson, Eds., vol. 1813 of *LNAI*, Springer-Verlag, pp. 33–62.
- [63] LIEPINS, G. E., AND HILLIARD, M. R. Genetic algorithms: Foundations and applications. *Annals of Operations Research* 21, 1 (1989), 31–57.
- [64] MEYER, J. H., AND LAND, R., Eds. *Overcoming barriers to student understanding: Threshold concepts and troublesome knowledge*. Routledge, New York, NY, 2006.
- [65] MONTANA, D. J. Strongly typed genetic programming. *Evolutionary Computation* 3, 2 (1995), 199–230.
- [66] MOSER, A., ZIMMERMANN, L., DICKERSON, K., GRENNELL, A., BARR, R., AND GERHARDSTEIN, P. They can interact, but can they learn? toddlers transfer learning from touchscreens and television. *Journal of Experimental Child Psychology* 137 (2015), 137 – 155.
- [67] NAKATA, M., LANZI, P. L., AND TAKADAMA, K. Simple compact genetic algorithm for XCS. In *2013 IEEE Conference on Evolutionary Computation* (Cancun, Mexico, June 20-23 2013), L. G. de la Fraga, Ed., vol. 1, pp. 1718–1723.

- [68] NISAN, N., AND SCHOCKEN, S. *The Elements of Computing Systems : Building a Modern Computer from First Principles*. MIT Press, Cambridge, Massachusetts, 2008.
- [69] PAN, S. J., AND 0001, Q. Y. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng* 22, 10 (2010), 1345–1359.
- [70] POLI, R., LANGDON, W. B., AND MCPHEE, N. F. *A Field Guide to Genetic Programming*. Riccardo Poli, William B. Langdon, Nicholas Freitag McPhee, 2008.
- [71] PRICE, C. J., AND FRISTON, K. J. Functional ontologies for cognition: The systematic definition of structure and function. *Cognitive Neuropsychology* 22, 3-4 (2007), 262–275.
- [72] RUGANI, R., PRIFTIS, G. V. K., AND REGOLIN, L. Number-space mapping in the newborn chick resembles humansFL mental number line. *Science* 347, 6221 (2015), 534–536.
- [73] SCHAFFER, J. D. Learning multiclass pattern discrimination. In *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications (ICGA85)* (July 1985), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates: Pittsburgh, PA, pp. 74–79.
- [74] SETTE, S., AND BOULLART, L. Genetic programming: principles and applications. *Engineering Applications of Artificial Intelligence* 14, 6 (2001), 727 – 736.
- [75] SETTE, S., WYNS, B., AND BOULLART, L. Comparing learning classifier systems and genetic programming: a case study. *Engineering Applications of Artificial Intelligence* 17, 2 (2004), 199 – 204. Intelligent Control and Signal Processing.
- [76] SHAFI, K., ABBASS, H. A., AND ZHU, W. Real time signature extraction from a supervised classifier system. In *Proceedings of the 2007 Congress on Evolutionary Computation* (2007).

- [77] SHAHBAZI, H., JAMSHIDI, K., MONADJEMI, A. H., AND ESLAMI, H. Biologically inspired layered learning in humanoid robots. *Knowledge-Based Systems* 57 (2014), 8 – 27.
- [78] SIDDIQUE, A., IQBAL, M., AND BROWNE, W. N. A comprehensive strategy for mammography image classification using learning classifier systems. In *Proceedings of 2016 IEEE Congress on Evolutionary Computation (CEC 2016)* (Vancouver, 24-29 July 2016), Y.-S. Ong, Ed., IEEE Press.
- [79] SIGAUD, O., AND WILSON, S. W. Learning classifier systems: a survey. *Soft Computing* 11, 11 (2007), 1065–1078.
- [80] SMART, W., AND ZHANG, M. Empirical analysis of schemata in genetic programming using maximal schemata and msg. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)* (June 2008), pp. 2983–2990.
- [81] SMART, W. R., AND ZHANG, M. Classification strategies for image classification in genetic programming. In *Proceeding of Image and Vision Computing NZ International Conference* (Palmerston North, New Zealand, Nov. 2003), D. Bailey, Ed., Massey University, pp. 402–407.
- [82] SMITH, R. E., AND CRIBBS III, H. B. Is a learning classifier system a type of neural network? *Evolutionary Computation* 2, 1 (1994), 19–36.
- [83] STONE, P., AND VELOSO, M. Layered learning. In *ECML (2000)*, R. L. de Mántaras and E. Plaza, Eds., vol. 1810 of *Lecture Notes in Computer Science*, Springer, pp. 369–381.
- [84] TAYLOR, M. E., AND STONE, P. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research* 10 (2009), 1633–1685.
- [85] TAYLOR, M. E., AND STONE, P. An introduction to intertask transfer for reinforcement learning. *AI Magazine* 32, 1 (2011), 15–34.

- [86] TOCCI, R. J., WIDMER, N. S., AND MOSS, G. L. *Digital Systems : Principles and Applications*. Prentice Hall, Upper Saddle River, New Jersey, 2011.
- [87] URBANOWICZ, R. J., GRANIZO-MACKENZIE, A., AND MOORE, J. H. Instance-linked attribute tracking and feedback for michigan-style supervised learning classifier systems. In *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012* (2012), T. Soule and J. H. Moore, Eds., ACM, pp. 927–934.
- [88] URBANOWICZ, R. J., AND MOORE, J. H. Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications* 2009 (2009). Article ID 736398.
- [89] WHITESON, S. *Evolutionary computation for reinforcement learning*. Springer, 2012.
- [90] WHITLEY, D. A genetic algorithm tutorial. *Statistics and Computing* 4 (1994), 65–85.
- [91] WIENER, N. *Cybernetics: or Control and Communication in the Animal and the Machine*. The MIT Press, Cambridge, Massachusetts, 2006.
- [92] WILSON, S. Compact rulesets from XCSI. In *Advances in Learning Classifier Systems 2002*, P. Lanzi, W. Stolzmann, and S. Wilson, Eds. Springer-Verlag, 2002, pp. 196–208.
- [93] WILSON, S. W. Quasi-darwinian learning in a classifier system. In *Proceedings of the Fourth International Workshop on Machine Learning* (1987), Morgan Kaufmann, pp. 59–65.
- [94] WILSON, S. W. ZCS: A zeroth level classifier system. *Evolutionary Computation* 2, 1 (1994), 1–18.
- [95] WILSON, S. W. Classifier fitness based on accuracy. *Evolutionary Computation* 3, 2 (1995), 149–175.

- [96] WILSON, S. W. Generalization in the XCS classifier system. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (1998), J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riololo, Eds., Morgan Kaufmann, pp. 665–674. <http://prediction-dynamics.com/>.
- [97] WILSON, S. W. Classifiers that approximate functions. *Natural Computing*, 1 (2002), 211–233.
- [98] WILSON, S. W. Classifier systems and the animat problem, Oct. 05 2013.
- [99] WILSON, S. W., AND GOLDBERG, D. E. A critical review of classifier systems. In *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA89)* (George Mason University, June 1989), J. D. Schaffer, Ed., Morgan Kaufmann, pp. 244–255. <http://prediction-dynamics.com/>.
- [100] WITTGENSTEIN, W. *Tractatus Logico-Philosophicus*. Routledge and Kegan Paul, London, 1962.
- [101] WONG, P., AND ZHANG, M. Algebraic simplification of GP programs during evolution. In *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006* (2006), M. Cattolico, Ed., ACM, pp. 927–934.
- [102] WONG, P., AND ZHANG, M. Effects of program simplification on simple building blocks in genetic programming. In *2007 IEEE Congress on Evolutionary Computation* (Sept 2007), pp. 1570–1577.
- [103] WONG, P., AND ZHANG, M. SCHEME: Caching subtrees in genetic programming. In *IEEE Congress on Evolutionary Computation* (2008), IEEE, pp. 2678–2685.

- [104] ZHANG, M., AND WONG, P. Explicitly simplifying evolved genetic programs during evolution. *International Journal of Computational Intelligence and Applications* 7, 2 (2008), 201–232.
- [105] ZHANG, M., AND WONG, P. Genetic programming for medical classification: a program simplification approach. *Genetic Programming and Evolvable Machines* 9, 3 (2008), 229–255.
- [106] ZHAO, P., HOI, S. C., WANG, J., AND LI, B. Online transfer learning. *Artificial Intelligence* 216 (2014), 76 – 102.