Improving the Scalability of
XCS-Based Learning
Classifier Systems

by

Muhammad Iqbal

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
2014
Abstract

Using evolutionary intelligence and machine learning techniques, a broad range of intelligent machines have been designed to perform different tasks. An intelligent machine learns by perceiving its environmental status and taking an action that maximizes its chances of success.

Human beings have the ability to apply knowledge learned from a smaller problem to more complex, large-scale problems of the same or a related domain, but currently the vast majority of evolutionary machine learning techniques lack this ability. This lack of ability to apply the already learned knowledge of a domain results in consuming more than the necessary resources and time to solve complex, large-scale problems of the domain. As the problem increases in size, it becomes difficult and even sometimes impractical (if not impossible) to solve due to the needed resources and time. Therefore, in order to scale in a problem domain, a system is needed that has the ability to reuse the learned knowledge of the domain and/or encapsulate the underlying patterns in the domain.

To extract and reuse building blocks of knowledge or to encapsulate the underlying patterns in a problem domain, a rich encoding is needed, but the search space could then expand undesirably and cause bloat, e.g. as in some forms of genetic programming (GP). Learning classifier systems (LCSs) are a well-structured evolutionary computation based learning technique that have pressures to implicitly avoid bloat, such as fitness sharing through niche based reproduction.

The proposed thesis is that an LCS can scale to complex problems in a domain by reusing the learnt knowledge from simpler problems of the domain and/or encapsulating the underlying patterns in the domain. Wilson’s XCS is used to implement and test the proposed systems, which is
a well-tested, online learning and accuracy based LCS model. To extract
the reusable building blocks of knowledge, GP-tree like, code-fragments
are introduced, which are more than simply another representation (e.g.
ternary or real-valued alphabets). This thesis is extended to capture the
underlying patterns in a problem using a cyclic representation. Hard prob-
lems are experimented to test the newly developed scalable systems and
compare them with benchmark techniques.

Specifically, this work develops four systems to improve the scalability
of XCS-based classifier systems. (1) Building blocks of knowledge are ex-
tracted from smaller problems of a Boolean domain and reused in learning
more complex, large-scale problems in the domain, for the first time. By
utilizing the learnt knowledge from small-scale problems, the developed
XCSCFC (i.e. XCS with Code-Fragment Conditions) system readily solves
problems of a scale that existing LCS and GP approaches cannot, e.g. the
135-bit MUX problem. (2) The introduction of the code fragments in classi-
fier actions in XCSCFA (i.e. XCS with Code-Fragment Actions) enables the
rich representation of GP, which when couples with the divide and con-
quer approach of LCS, to successfully solve various complex, overlapping
and niche imbalance Boolean problems that are difficult to solve using nu-
meric action based XCS. (3) The underlying patterns in a problem domain
are encapsulated in classifier rules encoded by a cyclic representation. The
developed XCSSMA system produces general solutions of any scale $n$ for
a number of important Boolean problems, for the first time in the field of
LCS, e.g. parity problems. (4) Optimal solutions for various real-valued
problems are evolved by extending the existing real-valued XCSR system
with code-fragment actions to XCSRCFA. Exploiting the combined power
of GP and LCS techniques, XCSRCFA successfully learns various continuous
action and function approximation problems that are difficult to learn
using the base techniques.

This research work has shown that LCSs can scale to complex, large-
scale problems through reusing learnt knowledge. The messy nature, dis-
association of message to condition order, masking, feature construction, and reuse of extracted knowledge add additional abilities to the XCS family of LCSs. The ability to use rich encoding in antecedent GP-like code-fragments or consequent cyclic representation leads to the evolution of accurate, maximally general and compact solutions in learning various complex Boolean as well as real-valued problems. Effectively exploiting the combined power of GP and LCS techniques, various continuous action and function approximation problems are solved in a simple and straightforward manner.

The analysis of the evolved rules reveals, for the first time in XCS, that no matter how specific or general the initial classifiers are, all the optimal classifiers are converged through the mechanism ‘be specific then generalize’ near the final stages of evolution. Also that standard XCS does not use all available information or all available genetic operators to evolve optimal rules, whereas the developed code-fragment action based systems effectively use figure and ground information during the training process.

This work has created a platform to explore the reuse of learnt functionality, not just terminal knowledge as present, which is needed to replicate human capabilities.
List of Publications


Acknowledgments

All praise unto Allah, Lord of all the worlds, Who created us devoid of any knowledge and endowed us with hearing, vision, and intellect so that we might be thankful to Him.

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 to read many paper drafts written by me throughout the PhD duration. Their corrections and suggestions contributed to improve the readability and quality of the papers. I am specifically grateful to them to read my thesis within a very tight schedule, but not compromising on the quality of the thesis. They were available for any question or discussion at any time during the whole PhD, especially Will Browne being next to the lab was frequently interrupted by me. Will Browne never objected, even if he was busy with his own important tasks. Thanks 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 the Victoria University of Wellington for awarding me the Victoria PhD Scholarship and to Mengjie Zhang for supporting my PhD scholarship application. I suffered twice from financial crisis during my PhD. I acknowledge the financial support from Student Academic Services at the Victoria University of Wellington, International Student Ministries of New Zealand, Federation of Islamic Associations of New
Zealand, Higher Education Commission of Pakistan, Ambreen Khan-Evans, Will Browne, and my elder sister.

I acknowledge the sacrifice of my family, back in Pakistan, to sending me abroad for PhD studies even though I was the main earning person financially supporting my family. I also acknowledge the sacrifice of my wife and apologies to my two little sons (Wajeeh and Huzaifah) to not be able to give them their deserved time and affections especially during the thesis writing period.

I am thankful to various reviewers of my papers for their constructive criticism, suggestions and appreciation of my work submitted to various conferences and journals. I am specifically thankful to Stewart Wilson for his explicit interest in my developed systems and suggesting new directions to utilize and improve my ideas in order to further contribute to the field of evolutionary machine learning.
Dedication

To my grandfather “Qunber”.
Contents

List of Figures xi
List of Tables xvii
List of Algorithms xxi

1 Introduction 1
   1.1 Scope .................................................. 1
   1.2 Motivation ............................................. 2
   1.3 Thesis Statement .................................... 3
   1.4 Research Goals .................................... 4
   1.5 Major Contributions .............................. 5
   1.6 Organization of Thesis ......................... 8

2 Background 11
   2.1 Machine Learning .................................... 11
   2.2 Transfer Learning .................................. 12
   2.3 Evolutionary Computation .......................... 13
     2.3.1 Genetic Algorithms .......................... 13
     2.3.2 Genetic Programming ......................... 15
   2.4 Learning Classifier Systems ...................... 18
     2.4.1 XCS ............................................. 21
     2.4.2 Real Valued XCS (XCSR) .................... 24
     2.4.3 Various Rich Encoding Schemes ............... 25
CONTENTS

2.5 Chapter Summary .............................................. 29

3 Experimental Design ............................................ 31
  3.1 Research Methodology ......................................... 31
    3.1.1 Reusing Building Blocks of Extracted Knowledge .... 32
    3.1.2 Increasing Generalization Ability of Classifier Rules .... 33
    3.1.3 Evolvability of Classifier Rules ......................... 34
  3.2 The Problem Domains ........................................ 35
    3.2.1 Boolean Problems ........................................ 35
    3.2.2 Continuous Action Problems ............................ 39
    3.2.3 Real-Valued Function Approximation Problems ........... 41
  3.3 Experimental Setup .......................................... 42
  3.4 Evaluation ................................................ 43

4 Reusing Building Blocks of Extracted Knowledge to Solve Complex, Large-Scale Boolean Problems 45
  4.1 Introduction ................................................ 45
  4.2 XCS with Code-Fragment Conditions
    (A Simple Approach) ........................................ 46
      4.2.1 Method ................................................ 46
      4.2.2 Results .............................................. 56
      4.2.3 Interpretation of Results ............................ 59
      4.2.4 Summary of Simple Approach to XCSCFC ............... 65
  4.3 XCS with Code-Fragment Conditions
    (An Improved Approach) ....................................... 66
      4.3.1 Method ................................................ 66
      4.3.2 Results .............................................. 71
      4.3.3 Interpretation of Results ............................ 84
  4.4 Chapter Summary ............................................ 91

5 Learning Complex, Overlapping and Niche Imbalance Boolean Problems Using XCS-Based Classifier Systems 93
CONTENTS

5.1 Introduction .................................................. 93

5.2 XCS with Code-Fragment Actions ............................ 95
  5.2.1 Code-Fragment Action Value ............................. 96
  5.2.2 Covering Operation ....................................... 97
  5.2.3 Rule Discovery Operation ................................. 97
  5.2.4 Comparing Two Code-Fragment Actions ................. 99
  5.2.5 Subsumption Deletion ................................... 99

5.3 Results ....................................................... 100
  5.3.1 The Multiplexer Problem Domain ......................... 102
  5.3.2 The Majority-on Problem Domain ......................... 103
  5.3.3 The Count Ones Problem Domain ......................... 103
  5.3.4 The Design Verification Problem Domain ................. 104
  5.3.5 The Carry Problem Domain ............................... 105
  5.3.6 The Even-Parity Problem Domain ......................... 105

5.4 Analysis of Evolved Classifiers ............................. 107
  5.4.1 The Multiplexer Problem Domain ......................... 107
  5.4.2 The Majority-On Problem Domain ......................... 109
  5.4.3 The Even-Parity Problem Domain ......................... 111

5.5 XCS’s Bias against Overlapping Rules ....................... 113

5.6 Action Inconsistency and Redundancy in XCSCFA ............... 116

5.7 Optimum Solutions in the Multiplexer Problem Domain ...... 120
  5.7.1 Condensation ........................................... 122
  5.7.2 Specialization of Address Bits .......................... 126
  5.7.3 Consistency of a Classifier’s Action Value .............. 128

5.8 Chapter Summary ............................................. 132

6 Extending Learning Classifier Systems with Cyclic Graphs for Scalability on Complex, Large-Scale Boolean Problems 135

6.1 Introduction .................................................. 135

6.2 Finite State Machines ....................................... 137

6.3 XCS with Finite-State-Machine Actions ....................... 138
CONTENTS

6.3.1 State-Machine Action Value . . . . . . . . . . . . . . . 140
6.3.2 Covering Operation . . . . . . . . . . . . . . . . . . . . . 140
6.3.3 Rule Discovery Operation . . . . . . . . . . . . . . . . 141
6.3.4 Comparing Two State-Machine Actions . . . . . . . . 143
6.3.5 Subsumption Deletion . . . . . . . . . . . . . . . . . . . 144
6.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
6.4.1 The Even-Parity Problem Domain . . . . . . . . . . . . 146
6.4.2 The Majority-on Problem Domain . . . . . . . . . . . . 147
6.4.3 The Count Ones Problem Domain . . . . . . . . . . . . 149
6.4.4 The Design Verification Problem Domain . . . . . . . . 151
6.4.5 The Carry Problem Domain . . . . . . . . . . . . . . . . 151
6.4.6 The Multiplexer Problem Domain . . . . . . . . . . . . 154
6.5 Effect of Redundant States in XCSSMA . . . . . . . . . . . . 156
6.6 Further Discussions . . . . . . . . . . . . . . . . . . . . . . . 160
6.7 Chapter Summary . . . . . . . . . . . . . . . . . . . . . . . . . 162

7 Computing Continuous Actions in Learning Classifier Systems 163
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
7.2 XCSR with Code-Fragment Actions (XCSRCFA) . . . . . . . . 164
7.2.1 Code-Fragment Action Value . . . . . . . . . . . . . . . 165
7.2.2 Match Set Creation . . . . . . . . . . . . . . . . . . . . . 165
7.2.3 Rule Discovery Operation . . . . . . . . . . . . . . . . . 166
7.2.4 Comparing Two Classifier Rules . . . . . . . . . . . . . 166
7.2.5 Subsumption Deletion . . . . . . . . . . . . . . . . . . . 166
7.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167
7.3.1 The Frog Problems . . . . . . . . . . . . . . . . . . . . . 167
7.3.2 Function Approximation Problems . . . . . . . . . . . . 173
7.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
7.5 Chapter Summary . . . . . . . . . . . . . . . . . . . . . . . . . 182
CONTENTS

8 Evolvability of Classifier Rules in XCS-Based Classifier Systems 185
  8.1 Introduction ......................................................... 185
    8.1.1 Goals ......................................................... 189
    8.1.2 Organization ............................................... 191
  8.2 Parent-Trees ....................................................... 191
  8.3 Experimental Design ............................................. 196
    8.3.1 The Problem Domains ........................................ 196
    8.3.2 Experimental Setup ......................................... 198
  8.4 Evolution and Analysis of Classifier Rules ..................... 200
    8.4.1 Classification Performance .................................... 200
    8.4.2 The Convergence of Optimal Classifiers .................... 203
    8.4.3 The Occurrence of Optimal Classifiers ..................... 204
    8.4.4 Further Discussions ......................................... 210
  8.5 Chapter Summary .................................................. 211

9 Conclusions and Future Work ...................................... 213
  9.1 Achieved Objectives ............................................... 213
  9.2 Main Conclusions ................................................ 215
    9.2.1 Reusing Building Blocks of Extracted Knowledge .......... 215
    9.2.2 Increasing Generalization Ability of Classifier Rules .... 217
    9.2.3 Evolvability of Classifier Rules ............................ 220
  9.3 Future Work ........................................................ 221
  9.4 Closing Remarks .................................................. 223
**List of Figures**

2.1 A sample GP generated tree program. . . . . . . . . . . . . . 16
2.2 Schematic depiction of a learning classifier system. . . . . . 19

4.1 Classifier population using code-fragment conditions. . . . . 47
4.2 An example of a code fragment. . . . . . . . . . . . . . . . . . 48
4.3 A code fragment used as ‘don’t care’ symbol in the condition of a classifier rule. . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.4 A sample population of code fragments in the 20-bit multiplexer problem. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.5 Results of the multiplexer problems using XCS and XCSCFC. 58
4.6 Results of the even-parity problems using XCS and XCSCFC. 60
4.7 A code-fragment mask. . . . . . . . . . . . . . . . . . . . . . . . 61
4.8 A classifier rule complementing a mask code-fragment. . . . 63
4.9 A classifier from the final population of the 11-bit MUX. . . 65
4.10 A population of classifiers using code-fragment conditions. . 67
4.11 A sample of code fragments in the 20-bit multiplexer problem. 72
4.12 Results of the multiplexer problems using XCS and XCSCFC. 74
4.13 Results of the majority-on problems using XCS and XCSCFC. 75
4.14 Results of the carry problems using XCS and XCSCFC. . . . 77
4.15 Results of the even-parity problems using XCS and XCSCFC. 79
4.16 Results of the even-parity problems obtained using XCSCFC with the function set \{AND, OR, NAND, NOR\}. . . . . . . . . . . . . . . . . 82
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.17</td>
<td>Results of the majority-on problems obtained using XCSCFC with the function set { AND, OR, NOT }</td>
<td>83</td>
</tr>
<tr>
<td>4.18</td>
<td>Results of the even-parity problems obtained using XCSCFC with the function set { AND, OR, NOT, XOR }</td>
<td>84</td>
</tr>
<tr>
<td>4.19</td>
<td>A classifier rule from the final rule base obtained for a typical run of the 20-bit multiplexer problem</td>
<td>85</td>
</tr>
<tr>
<td>4.20</td>
<td>A classifier rule from the final rule base obtained for a typical run of the 4-bit even-parity problem</td>
<td>88</td>
</tr>
<tr>
<td>4.21</td>
<td>The performance of XCSCFC, using different number of code fragments in the condition of a classifier rule, for the 6-bit multiplexer problem</td>
<td>91</td>
</tr>
<tr>
<td>5.1</td>
<td>Classifier population using code-fragment actions</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>A classifier with code-fragment action</td>
<td>96</td>
</tr>
<tr>
<td>5.3</td>
<td>Rule discovery operation in XCSCFA</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>Results of multiplexer problems</td>
<td>102</td>
</tr>
<tr>
<td>5.5</td>
<td>Results of the 7-bit majority-on problem</td>
<td>103</td>
</tr>
<tr>
<td>5.6</td>
<td>Results of the 7-bit count ones problem</td>
<td>104</td>
</tr>
<tr>
<td>5.7</td>
<td>Results of the DV1 problem</td>
<td>105</td>
</tr>
<tr>
<td>5.8</td>
<td>Results of the 4+4 bit carry problem</td>
<td>106</td>
</tr>
<tr>
<td>5.9</td>
<td>Results of the 7-bit even-parity problem</td>
<td>106</td>
</tr>
<tr>
<td>5.10</td>
<td>Results of overlapping and niche imbalance problems using XCS</td>
<td>115</td>
</tr>
<tr>
<td>5.11</td>
<td>Result of the 20-bit multiplexer problem using XCS</td>
<td>116</td>
</tr>
<tr>
<td>5.12</td>
<td>Effect of action inconsistency and redundancy in the XCSCFA approach</td>
<td>119</td>
</tr>
<tr>
<td>5.13</td>
<td>Population size in terms of macro-classifiers in learning different problems using XCS and XCSCFA systems</td>
<td>121</td>
</tr>
<tr>
<td>5.14</td>
<td>The numerosity and fitness of classifiers in final population for a typical run in learning the 6-, 11-, 20-, and 37-bit multiplexer problems using XCSCFA</td>
<td>124</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.15</td>
<td>The numerosity and fitness of classifiers in final population for a typical run in learning the 37-bit multiplexer problem using standard XCS with static binary actions.</td>
</tr>
<tr>
<td>5.16</td>
<td>A classifier rule with code-fragment action where the action is dependent on a ‘#’ bit in the condition.</td>
</tr>
<tr>
<td>6.1</td>
<td>A Moore state machine with three states.</td>
</tr>
<tr>
<td>6.2</td>
<td>A Moore state machine with a deactivated state.</td>
</tr>
<tr>
<td>6.3</td>
<td>A classifier rule with state-machine action.</td>
</tr>
<tr>
<td>6.4</td>
<td>Results of the 24-bit even-parity problem.</td>
</tr>
<tr>
<td>6.5</td>
<td>A sample classifier rule from final solution of the 24-bit even-parity problem in XCSSMA.</td>
</tr>
<tr>
<td>6.6</td>
<td>A genotype evolved using SMCGP for the even-parity problem domain.</td>
</tr>
<tr>
<td>6.7</td>
<td>Results of the 7-bit majority-on problem.</td>
</tr>
<tr>
<td>6.8</td>
<td>A sample classifier rule from final solution of the 7-bit majority-on problem in XCSSMA.</td>
</tr>
<tr>
<td>6.9</td>
<td>Results of the 7-bit count ones problem.</td>
</tr>
<tr>
<td>6.10</td>
<td>A sample classifier rule from final solution of the 7-bit count ones problem in XCSSMA.</td>
</tr>
<tr>
<td>6.11</td>
<td>Results of the DV1 problem.</td>
</tr>
<tr>
<td>6.12</td>
<td>A sample classifier rule from final solution of the DV1 problem in XCSSMA.</td>
</tr>
<tr>
<td>6.13</td>
<td>Results of the 6+6 bit carry problem.</td>
</tr>
<tr>
<td>6.14</td>
<td>A sample classifier rule from final solution of the 6+6 bit carry problem in XCSSMA.</td>
</tr>
<tr>
<td>6.15</td>
<td>A genotype evolved using SMCGP for the carry problem domain.</td>
</tr>
<tr>
<td>6.16</td>
<td>Results of the 20-bit multiplexer problem.</td>
</tr>
<tr>
<td>6.17</td>
<td>A sample classifier rule from final solution of the 20-bit multiplexer problem in XCSSMA.</td>
</tr>
<tr>
<td>6.18</td>
<td>Effect of redundant states in XCSSMA.</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

7.1 A sample code-fragment action. ................. 165
7.2 Results in learning the frog1 problem using XCSRCFA. ... 168
7.3 Best action $a^*$ in learning the frog1 problem using XCSRCFA. 170
7.4 Results in learning the frog2 problem using XCSRCFA. ... 172
7.5 Best action $a^*$ in learning the frog2 problem using XCSRCFA. 173
7.6 A sample classifier rule from final solution of the frog2 problem in XCSRCFA. .............................................. 174
7.7 Approximating a continuous function with continuous derivative using XCSRCFA. ........................................ 175
7.8 A sample classifier rule from the final solution obtained in approximating a continuous function with continuous derivative using XCSRCFA. ........................................ 176
7.9 Approximating a continuous function with discontinuous derivative using XCSRCFA. ................................. 177
7.10 Two sample classifier rules from the final solution obtained in approximating a continuous function with discontinuous derivative using XCSRCFA. ........................................ 178
7.11 Approximating a discontinuous function using XCSRCFA. ... 179
7.12 Two sample classifier rules from the final solution obtained in approximating a discontinuous function using XCSRCFA. 180
7.13 A sample classifier rule from final solution of the $x+y^2$ problem in XCSRCFA. .............................................. 180

8.1 The search space of a 6-bit binary problem represented in two-dimensional grid form. ......................... 188
8.2 Different mechanisms to evolve a classifier rule in an XCS-based system. ........................................ 190
8.3 A sample parent-tree for the classifier rule ‘01#0## : 0’ generated in learning the 6-bit multiplexer problem using XCS. 193
8.4 A sample parent-tree, in 2-dimensional form, for the classifier rule ‘01#0## : a’ generated in learning the 6-bit multiplexer problem using XCS. .............................. 195
LIST OF FIGURES

8.5  A sample parent-tree, in 2-dimensional form, for the classifier rule ‘001### : a’ generated in learning the 6-bit multiplexer problem using XCSCFA. 197
8.6  The classification performance of XCS in learning the 11-bit multiplexer problem and the 5-bit count ones problem. 201
8.7  The classification performance of XCSCFA in learning the 11-bit multiplexer problem and the 5-bit count ones problem. 202
8.8  The convergence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS. 204
8.9  The convergence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCSCFA. 205
8.10 The average depth of the parent-trees obtained in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS with different values of $P_\#$. 206
8.11 The average depth of the parent-trees obtained in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCSCFA with different values of $P_\#$. 206
8.12 The first occurrence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS. 207
8.13 The first occurrence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCSCFA. 209
List of Tables

3.1 The optimum ternary encoded rule set for the DV1 problem. 38
3.2 The optimum ternary encoded rule set for the 3+3 bit carry problem. ............................................. 39
3.3 Properties of different Boolean problem domains. .............. 39
4.1 The top 10 highly ranked code fragments. ......................... 61
4.2 Truth table for a code-fragment mask. ............................. 62
4.3 The low-length code fragments. .................................... 64
4.4 The code fragments using address-bits. ............................. 64
4.5 The Wilcoxon signed rank test for performance comparison in the majority-on problem domain. ....................... 75
4.6 The Wilcoxon signed rank test for performance comparison in the carry problem domain. ............................. 76
4.7 The Wilcoxon signed rank test for performance comparison in the even-parity problem domain. ....................... 78
4.8 Performance of different GP systems for the even-parity problems. ....................................................... 81
4.9 Performance of different GP systems for the majority-on problems. ......................................................... 82
4.10 Truth table for the code fragment ‘L1_29 D1 d D4 D2 | d’ .... 86
4.11 Truth table for the code fragment ‘L1_12 D0 d ∼’ .............. 86
4.12 Truth table for the code fragment ‘L2_3 D18 & ∼’ .............. 87
4.13 Truth table for the code fragment ‘L2_4’ ......................... 89


<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.14</td>
<td>Truth table for the code fragment ‘D2 D2 &amp; L1,4’</td>
<td>90</td>
</tr>
<tr>
<td>4.15</td>
<td>Truth table for the code fragment ‘D2 L1,7 d’</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>A sample of classifiers from a final solution obtained using XCS</td>
<td>108</td>
</tr>
<tr>
<td>5.2</td>
<td>A sample of classifiers from a final solution obtained using XCSCFA,</td>
<td>109</td>
</tr>
<tr>
<td>5.3</td>
<td>A sample of classifiers from a final solution obtained using XCSCFA,</td>
<td>109</td>
</tr>
<tr>
<td>5.4</td>
<td>A sample of classifiers from a final solution obtained using XCS in learning the 7-bit majority-on problem.</td>
<td>111</td>
</tr>
<tr>
<td>5.5</td>
<td>A sample of classifiers from a final solution obtained using XCSCFA,</td>
<td>112</td>
</tr>
<tr>
<td>5.6</td>
<td>A sample of classifiers from a final solution obtained using XCSCFA,</td>
<td>113</td>
</tr>
<tr>
<td>5.7</td>
<td>A sample of classifiers from a final solution obtained using XCSCFA,</td>
<td>114</td>
</tr>
<tr>
<td>5.8</td>
<td>Effect of redundancy in the XCSCFA approach.</td>
<td>117</td>
</tr>
<tr>
<td>5.9</td>
<td>The population size in the final solutions obtained using XCS and XCSCFA systems for different problems.</td>
<td>122</td>
</tr>
<tr>
<td>5.10</td>
<td>Final compact population of maximally general and accurate classifiers obtained in a typical run for the 6-bit multiplexer problem using XCSCFA,</td>
<td>125</td>
</tr>
<tr>
<td>5.11</td>
<td>Final compact population of maximally general and accurate classifiers obtained in a typical run for the 11-bit multiplexer problem using XCSCFA,</td>
<td>126</td>
</tr>
<tr>
<td>5.12</td>
<td>Four sample classifier rules for the 6-bit multiplexer problem, demonstrating the specialization of address bits.</td>
<td>128</td>
</tr>
<tr>
<td>5.13</td>
<td>Consistency of classifiers in terms of action values when using different patterns as the condition.</td>
<td>131</td>
</tr>
</tbody>
</table>
6.1 The description of a state machine encoded by the string ‘21110 – 00102 – 11001’. 140
6.2 The population size in the final solutions obtained using XCS and XCSSMA for different problems. 159
7.1 A sample of classifier rules, obtained in a typical run for the frog1 problem, using XCSRCFA. 171
8.1 The ternary encoded rule set for the 11-bit multiplexer problem. 198
8.2 The ternary encoded rule set for the 5-bit count ones problem. 199
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XCSCFC&lt;sub&gt;simple&lt;/sub&gt;: Does_Match</td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>XCSCFC&lt;sub&gt;simple&lt;/sub&gt;: Covering_Operation</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>XCSCFC&lt;sub&gt;simple&lt;/sub&gt;: Crossover_Operation</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>XCSCFC&lt;sub&gt;simple&lt;/sub&gt;: Mutation_Operation</td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>XCSCFC&lt;sub&gt;simple&lt;/sub&gt;: Is_More_General</td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>XCSCFC&lt;sub&gt;simple&lt;/sub&gt;: Are_Equal_Classifiers</td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>XCSCFC&lt;sub&gt;efficient&lt;/sub&gt;: Covering_Operation</td>
<td></td>
<td>68</td>
</tr>
<tr>
<td>8</td>
<td>XCSCFC&lt;sub&gt;efficient&lt;/sub&gt;: Crossover_Operation</td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>XCSCFC&lt;sub&gt;efficient&lt;/sub&gt;: Mutation_Operation</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>10</td>
<td>XCSCFA: Covering_Operation</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td>11</td>
<td>XCSCFA: Are_Equal_Actions</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>12</td>
<td>XCSSMA: Covering_Operation</td>
<td></td>
<td>142</td>
</tr>
<tr>
<td>13</td>
<td>XCSSMA: Crossover_Operation</td>
<td></td>
<td>143</td>
</tr>
<tr>
<td>14</td>
<td>XCSSMA: Mutation_Operation</td>
<td></td>
<td>144</td>
</tr>
<tr>
<td>15</td>
<td>XCSSMA: Are_Equal_Actions</td>
<td></td>
<td>145</td>
</tr>
</tbody>
</table>
LIST OF ALGORITHMS
Chapter 1

Introduction

1.1 Scope

Using artificial intelligence and machine learning techniques, a broad range of intelligent machines have been designed to perform different tasks [86, 91, 99]. An intelligent machine learns by perceiving its environmental status and taking an action that maximizes its chances of success.

Human beings have the ability to apply knowledge learned from a smaller problem to more complex, large-scale problems of the same or a related domain, but currently the vast majority of evolutionary machine learning techniques lack this ability. This lack of ability to apply the already learned knowledge of a domain results in consuming more than the necessary resources and time to solve complex, large-scale problems of the domain. As the problem increases in size, it becomes difficult and even sometimes impractical (if not impossible) to solve due to the needed resources and time. Therefore, in order to scale in a problem domain, a system is needed that has the ability to reuse the learned knowledge of the domain and/or encapsulate the underlying patterns in the domain [109].
1.2 Motivation

In order to scale in a problem domain, reusable building blocks of knowledge must be extracted and/or the system should be able to capture the underlying patterns in the domain and produce a generalized solution for any scale \( n \) in the domain. To extract and reuse building blocks of knowledge or to encapsulate the underlying patterns in a problem domain, a rich encoding is needed, but the search space could then expand undesirably and cause bloat, e.g. as in some forms of genetic programming (GP). Learning classifier systems (LCSs) are a well structured evolutionary computation based learning technique that have pressures to implicitly avoid bloat, such as fitness sharing through niche based reproduction [26].

Traditionally, an LCS represents a rule-based agent that incorporates evolutionary computing and machine learning to solve a given task by interacting with an unknown environment [18, 49]. After observing the current state of the environment, the agent performs an action, and the environment provides a reward. The goal of an LCS is to evolve a set of classifier rules, of the form “if condition then action”, that collectively solve the problem. It is to be noted that in an LCS cooperation is used alongside the common evolutionary computation pressure of competition. The generalization property in LCS allows a single rule to cover more than one environmental state provided that the action-reward mapping is similar. Traditionally, generalization in an LCS is achieved by using a special ‘don’t care’ symbol (#) in classifier conditions, which matches any value of a specified attribute in the vector describing the environmental state. Commonly, the action in a classifier rule is represented by a numeric constant. LCS can be applied to a wide range of problems including classification, data mining, function approximation, control, modeling and optimization problems [10, 17, 22, 101]. LCSs have been shown to be robust to small amounts of noise and are often more robust than most machine learning techniques with increasing amounts of noise [22].
The LCS technique can scale in various problem domains, but has to relearn from the start each time to solve a problem [28]. Further, increased dimensionality of the problem, resulting in increased search space, demands large memory space and leads to much longer training times, and eventually restricts LCS to a limit in problem size. By explicitly feeding the domain knowledge to an LCS, scalability can be achieved but it adds bias and restricts use in multiple domains [56].

1.3 Thesis Statement

The proposed thesis is that an LCS can scale to complex problems in a domain by reusing the learnt knowledge from simpler problems of the domain and/or encapsulating the underlying patterns in the domain.

This thesis has three challenges: first to identify and extract the building blocks of knowledge from a problem domain, second to reuse the extracted knowledge to solve complex, large-scale problems of the domain, and third to capture the underlying patterns in the domain.

Wilson’s XCS [28], a well-tested online accuracy-based LCS model, is to be used to implement and test the proposed systems. In XCS the rule discovery operation is applied to a subset of the population, i.e. an action set, instead of the whole population to conserve similar building blocks of information. These features of XCS make it possible to form a complete and accurate mapping from inputs and actions to payoff predictions. The complete map produced by an XCS-based classifier system provides more building blocks, for a given problem, than any other LCS model [13], which motivated its suitability for this research work. If a learning system is unable to produce a complete and accurate solution, then the extracted building blocks lack important knowledge and so may not be suitable candidates to be used to scale the system.
1.4 Research Goals

The overall goal of this thesis is to improve the scalability of learning classifier systems. This goal is divided into the following two subgoals.

1. To reuse the learnt knowledge from smaller problems in a domain in order to solve complex, large-scale problems in the domain. To achieve this goal the following research objective has been established:

(a) Developing an XCS-based learning classifier system, which identifies and extracts building blocks of knowledge from smaller problems of a Boolean domain and reuses the extracted knowledge in learning complex, large-scale problems in the same domain.

By utilizing this layered learning approach, it is hoped that eventually a problem will be solved in the domain, which had not previously been solved using the base techniques. To extract and reuse the building blocks of knowledge, the traditional ternary alphabet based classifier conditions in XCS will be replaced by code-fragment conditions. A code-fragment is an expression tree similar to a tree generated in GP.

2. To increase the generalization ability of classifier rules in the system in order to capture the underlying patterns in a problem domain. To achieve this goal the following research objectives have been established:

(a) Extending XCS classifier system with code-fragment actions in order to evolve more generalized rules than a numeric action based XCS in learning Boolean problems.
1.5. MAJOR CONTRIBUTIONS

(b) Extending XCS with cyclic representations to further increase the generalization ability of the system in learning Boolean problems.

(c) Extending real-valued XCS (known as XCSR) with code-fragment actions in order to solve real-valued, continuous action and function approximation problems.

A classifier system that has the ability to produce more generalized classifier rules will require a smaller number of rules in the final solution for a problem, therefore it may solve problems to a scale beyond the base techniques.

The developed systems will be evaluated using various complex problems used in the literature and the results will be compared with the existing related systems.

In addition to improving the scalability of learning classifier systems, this thesis also aims at understanding of where optimal rules come from in order to appreciate the role of building blocks, and operators on building blocks. To achieve this goal, a concept of parent-trees will be introduced to investigate and analyze the evolution of optimal classifiers produced in the final solutions.

1.5 Major Contributions

This thesis led to the following major contributions to the field of evolutionary machine learning in general and specifically to the field of XCS-based learning classifier systems.

1. Building blocks of knowledge were extracted from smaller problems of a Boolean domain to reuse the extracted knowledge in learning more complex, large-scale problems in the domain, for the first time, by using code-fragment based classifier conditions. By utilizing this layered learning approach, the developed XCSCFC system readily
solved problems of a scale that existing classifier system and genetic programming approaches cannot, e.g. the 135-bit MUX problem.

Parts of this contribution have been published in:


2. The generalization ability of classifier rules was increased beyond a numeric action based XCS by using a code-fragment based action in a classifier rule. The introduction of the code fragments in classifier actions enabled the rich representation of GP, which coupled with the divide and conquer approach of LCS successfully solved various complex, overlapping and niche imbalance Boolean problems that are difficult to be solved using numeric action based XCS.

Parts of this contribution have been published in:


1.5. **MAJOR CONTRIBUTIONS**


3. The generalization ability of classifier rules was further increased by encapsulating repeated patterns in the domain using a cyclic representation to encode a classifier’s action. The developed system produced general solutions of any scale $n$ for a number of important Boolean problems by encapsulating the underlying patterns in the domain, e.g. parity problems.

Parts of this contribution have been published/submitted in:


4. After the successful application of the developed code-fragment based systems in learning various complex, large-scale Boolean problems, the code-fragment action based approach was adopted to extend the existing real-valued XCSR system to XCSRCFA, which produced the
optimum solutions for various real-valued, function approximation, and continuous action problems.

Part of this contribution has been published in:


5. A detailed investigation and analysis of the evolution of classifier rules in XCS-based classifier systems is provided in this work. This analysis revealed that the introduced code-fragments are not simply another representation as the associated methods fundamentally change the way XCS functions, e.g. enabling correct and incorrect classifiers to breed directly.

Part of this contribution has been submitted in:


### 1.6 Organization of Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents the literature review of related works. Chapter 3 describes the research methodology to be used in this work to achieve the overall goal. This chapter also introduces problem domains and experimental setup to be used for testing and evaluating the developed systems. Chapter 4 to Chapter 8 present major contributions to fulfill the established research objectives. Chapter 9 concludes this work.

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 the research methodology adopted in this work, to achieve the overall goal, and briefly describes the systems designed and implemented following this research methodology. The details of each implemented system is provided in a separate contribution chapter, from Chapter 4 to Chapter 8. This chapter also provides details of the problem domains experimented here and the experimental setup used for testing and evaluating the developed systems. Further, this chapter 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 to learn more complex, large-scale problems in the domain. By utilizing this layered learning approach, the resulting system readily solves problems of a scale that existing classifier system and genetic programming approaches cannot, e.g. the 135-bit MUX problem.

In Chapter 5, the generalization ability of classifier rules is increased beyond a numeric action based XCS by using a GP-tree like code-fragment action in a classifier rule. The introduction of the code fragments in classifier actions enabled the rich representation of GP, which coupled with the divide and conquer approach of LCS to successfully solve various complex, overlapping and niche imbalance Boolean problems that were previously difficult to be solve using numeric action based XCS.

In Chapter 6, the generalization ability of classifier rules is further increased by encapsulating repeated patterns in the domain using a cyclic representation to encode a classifier’s action. The developed system produced general solutions of any scale \( n \) for a number of important Boolean problems by encapsulating the underlying patterns in the domain, e.g. parity problems.

After the successful application of the developed code-fragment based systems in learning various complex, large-scale Boolean problems, in Chapter 7 the code-fragment action based approach is adopted to learn
real-valued problems. The developed system produced optimal solutions for various continuous action and function approximation problems.

Chapter 8 provides a detailed investigation and analysis of the evolution of classifier rules in XCS-based classifier systems. This analysis revealed that the introduced code-fragments are not simply another representation as the associated methods fundamentally change the way XCS functions, e.g. enabling correct and incorrect classifiers to breed directly.

Chapter 9 presents the achieved objectives, main conclusions from each contribution chapter, and the future work that stems from this research work.
Chapter 2

Background

This chapter provides an overview of machine learning and evolutionary computation. After introducing these fields, learning classifier systems are presented in detail. This chapter also describes various encoding schemes that have been used by the LCS community to represent classifier rules. A rich encoding scheme is considered the first step to scalable learning [56]. Very little work has been conducted into scaling of evolutionary machine learning techniques, but the related fields of lifelong learning and transfer learning will be considered.

2.1 Machine Learning

Machine learning is a field of artificial intelligence that is concerned with the study and application of a range of algorithms and techniques capable of learning through experience in their environments [3, 86]. According to Tom Mitchell [86]:

“A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E”.
CHAPTER 2. BACKGROUND

Machine learning techniques are usually categorized into three types: supervised learning, unsupervised learning, and reinforcement learning techniques [99]. In supervised learning, the desired outputs for a problem are known in advance, and the goal is to learn a function that maps inputs to the outputs. In unsupervised learning, there are no correct answers for the learner to explicitly learn from. It attempts to find hidden structure in the given input examples to generate a model. This model is then used to predict output for unseen data examples in future. In reinforcement learning, the learning agent (learner) interacts with its environment by producing actions on sensing the environmental status [108]. These actions affect the state of the environment, and the environment then provides a feedback in the form of a scalar reward (or punishment) depending upon the utility of the action. The goal of the agent is to learn to act in a way that maximizes the future rewards it receives (or minimizes the punishments) over its lifetime.

2.2 Transfer Learning

Transfer learning is a process to transfer knowledge learned in one or more source tasks to a related but more complex unseen target task, in an effort to facilitate learning in the target task [110]. The source and target tasks may be from the same or different problem domains [92].

Layered learning is a subclass of transfer learning, formally introduced by Stone and Veloso [107] as an extension of earlier work by de Garis [32] and Asada et al. [7], where the task to be learned is decomposed into a hierarchy of subtask layers. At each layer a subtask is learned separately, commonly in sequence, by applying a suitable machine learning algorithm that is usually chosen manually according to the subtask characteristics. The knowledge learned at lower layers is used to learn the subtask at the next higher layer. Layered learning mostly applies to complex tasks for which: 1) direct learning is not tractable, and 2) a bottom-up hierar-
Evolutionary computation is a population-based computing paradigm [65] where each individual represents a potential solution or a part of the solution to the problem at hand. The population is evolved by applying, with certain probability, the genetic operations of reproduction, elitism, crossover and mutation on the selected individuals based upon their utility for the task being solved.

In the following subsections, two of the most common evolutionary techniques, namely genetic algorithms and genetic programming, are briefly described as they are directly related to the work presented in this thesis.

2.3.1 Genetic Algorithms

The discovery component of an LCS [28] is commonly implemented using a GA. An LCS seeks to evolve a population of co-operative rules, where each individual rule is optimized using the GA.

GAs are an evolutionary computational technique where each individual member of the population is usually represented by a bitstring of fixed length, and represents a potential problem solution [36]. The evolutionary process in GAs has been described using the concept of schema. A schema is a similarity template for describing a set of finite-length strings defined over a finite alphabet [48]. For example, if the alphabet is \{0, 1, *\} then the schema “10**1” is describing all strings of length five that start with symbols ‘10’ and end with symbol ‘1’, such as ‘10001’, ‘10011’, ‘10101’, and
‘10111’. It is to be noted that here ‘*’ is treated as a ‘don’t care’ symbol that can be either ‘0’ or ‘1’. The distance between first and last specific string positions in a schema $H$ is called its defining length, denoted by $\delta(H)$, and the number of specific positions in it is called its order, denoted by $o(H)$, e.g. the defining length of the schema “10**1” is 4 and the order is 3.

Goldberg hypothesized that higher performance individuals are actually generated as a result of the combination of short-length, low-order and high-performance schemata [36]. These schemata are called the building blocks of the system. These building blocks are likely to be selected and combined via crossover to produce longer and fitter individuals in a GA. These building blocks are also relatively less affected by mutation. The assumption by Goldberg that this is the way a GA works, is termed the building block hypothesis.

For a population of individuals represented by fixed-length strings, the genetic operators sometimes cannot process the building blocks effectively as a random crossover point may lie within a building block. To avoid this disruption of partial solutions by the genetic operators, a probability distribution based approach, known as Estimation of Distribution Algorithm (EDA), was developed [88]. In the various forms of EDAs, the crossover and mutation operators are replaced by generating new offspring according to the probability distribution of the selected individuals [93]. Santana et al. [100] and Pelikan et al. [94] have incorporated transfer learning in the field of EDAs to transfer information between optimization problems. This shows that useful information can be extracted from fitter individuals within a population.

A sample from a schema such as “1**0” is implicitly sampling from many other schemata too, like “10*0”, “11*0”, “1*00”, and “1*10”. It has been estimated that in any generation of a population of $n$ individuals, the number of schemata being processed by the GA is proportional to $n^3$ [36]. This inherently parallel processing of a large quantity of schemata is known as implicit parallelism.
2.3. **EVOLUTIONARY COMPUTATION**

The building block hypothesis has been criticized due to its weak theoretical foundations [5, 14, 20], but the schema theory still remains a popular tool to explain the power of evolutionary machine learning techniques [33, 95, 96].

### 2.3.2 Genetic Programming

GP is an evolutionary approach to generating computer programs for solving a given problem automatically [70], and uses a much richer alphabet than GA to encode the solution, i.e. more expressive symbols that can express functions as well as numbers. A GP-like alphabet to describe the problem is used in the LCSs developed here, so the GP technique is described to aid understanding.

In GP each individual is a computer program, commonly represented by a tree, that when executed generates the potential solution. The task to be solved is represented by a primitive set of operations, known as the *function set*, and a set of operands, known as the *terminal set*. The internal nodes of the tree are functions and leaves are the terminals.

To generate a computer program for regression and classification using GP, a set of (input, output) pairs is needed, along with sets of functions and terminals, for training the candidate solutions. GP attempts to construct a computer program that maps each of the (input, output) pairs correctly. For example, if the (input, output) pairs set is \{(0, 1), (1, 3), (2, 7), (3, 13)\} and \{+, −, ∗, /\} and \{x, 1\} are the function set and the terminal set respectively, then the optimal corresponding GP generated program is as shown in Figure 2.1. This GP tree is equivalent to the output expression \((x ∗ x) + (x + 1)\), where \(x\) is the input. In this thesis, a GP tree is often represented in the postfix form, e.g. the postfix form of the tree shown in Figure 2.1 is “xx ∗ x1 + +”.

A tree-GP computer program may contain unnecessary bloating terms and non-optimum expressions. These problems are usually addressed by
Figure 2.1: A GP generated tree program to map the set of (input, output) pairs \{\{(0, 1), (1, 3), (2, 7), (3, 13)\}\).

limiting maximal allowed depth for an individual tree and/or using a fitness measure that punishes excess sized individuals [83]. The other ways to control bloat in GP include simplifying individual programs using algebraic and numerical simplification methods [66, 127], or using specific bloat control operators [2].

A GP system produces an individual as a ‘single’ solution, rather than a co-operative set of rules as in an LCS. It generally requires supervised learning with the whole training set, rather than online, reinforcement learning as in LCSs.

GP has also been implemented using non-tree representations such as linear GP (LGP) [16] and cartesian GP (CGP) [85]. A number of GP researchers have incorporated and investigated layered learning in GP [38, 47, 64]. CGP and layered learning GP are briefly introduced here as they are closely related to the work presented in this thesis.

**Cartesian Genetic Programming**

Cartesian genetic programming (CGP) is a flexible graph-based version of GP that allows a program to be evolved with more than one output, often using an evolution strategy [85]. In CGP, a program is represented as a directed graph that is encoded in the form of a linear string of integers. The graph-based representation has benefit of implicitly reusing the nodes in
2.3. EVOLUTIONARY COMPUTATION

the graph. In CGP there is a many-to-one genotype to phenotype mapping due to the presence of a large amount of redundancy [84].

Self-Modifying CGP (SMCGP) is a developmental form of CGP, where an evolved individual computer program can be iterated to produce an infinite sequence of computer programs using a set of self-modifying functions [39]. It allows programs to take more inputs and provide further outputs during each iteration so that each of the generated programs solves a particular problem in the domain. Using SMCGP, Harding et al. [40] evolved programs that can provide general solutions to a number of problems including an $n$-bit parity problem and an adder to add two $n$-bit binary numbers. The limitation of SMCGP is that cycles were not allowed\(^1\) in the graphical programs to avoid infinite loops. Therefore, the size of the obtained solutions to Boolean problems increases for every next level problem in the domain, due to the application of self-modifying operators, especially the duplication operator that duplicates a part of the graph in it. So for large-scale Boolean problems, solutions are very long making visualization and interpretation difficult.

Layered Learning Genetic Programming

For complex problems, the standard monolithic GP may not find a solution due to the large search space leading to an intractable problem. In layered learning, the complex target task is decomposed into subtasks and each subtask is learned in a bottom-up hierarchical order [107]. Gustafson and Hsu [38] implemented layered learning in GP to learn the keep-away soccer game which is a multi-agent system problem. The main task was decomposed into two subtasks and the final population in the bottom task layer was used as the initial population for the top task layer. The layered learning GP approach evolved better solutions faster than standard GP.

Jackson and Gibbons [64] applied layered learning in GP to solve Boolean logic problems of the even-parity and the majority-on problem domains,

\(^1\)In principle SMCGP could have cycles, as can CGP.
using a two-layered approach. The solutions of the bottom layer were en-
capsulated as parametrized modules and reused to learn the main task
in the top layer. The layered learning approach outperformed standard
monolithic GP [70] and GP with automatically defined functions (ADFs)
[71], albeit it did not achieve 100% success rate for the higher-order prob-
lems.

Hien et al. [45] investigated layered learning with incremental sam-
ppling in GP. They tested twelve symbolic regression problems and results
were compared with standard GP [70]. The combination of incremental
sampling with layered learning in GP showed improvement in terms of
reducing the training time and complexity of the solutions. Later Hien
and Hoai [44] incorporated parameter setting techniques derived from
progressive sampling to overcome ad-hoc parameter setting issues in the
incremental sampling based layered learning GP.

Hoang et al. [47] investigated interactions between evolution, devel-
opment, and layered learning using tree adjoining grammar guided GP
(TAG3P) [46]. The developed system, called DTAG3P, was tested in sym-
bolic regression problems, Boolean even-parity problems, and ORDERTREE
problems. The layered learning DTAG3P system produced more struc-
tured and scalable solutions to the problems as compared with two single-
short learning GP systems: standard tree-based GP [70] and the pre-existing
TAG3P [46]. This shows the ability of GP to learn problems in a layered
learning fashion.

2.4 Learning Classifier Systems

Traditionally, an LCS represents a rule-based agent that incorporates evo-
lutionary computing and machine learning to solve a given task by inter-
acting with an unknown environment via a set of sensors for input and
a set of effectors for actions [18, 49]. After observing the current state of
the environment, the agent performs an action, and the environment pro-
2.4. LEARNING CLASSIFIER SYSTEMS

vides a reward, as depicted in Figure 2.2. The goal of an LCS is to evolve a set of classifier rules that collectively solve the problem. The evolved solution can be condensed using an appropriate compaction algorithm in order to reduce the number of classifier rules in the final population [67]. The generalization property in LCS allows a single rule to cover more than one environmental state provided that the action-reward mapping is similar. Traditionally, generalization in an LCS is achieved by using a special ‘don’t care’ symbol (#) in classifier conditions, which matches any value of a specified attribute in the vector describing the environmental state. LCS can be applied to a wide range of problems including data mining, control, modeling and optimization problems [10, 17, 101]. LCS have been shown to be robust to small amounts of noise and are often more robust than most machine learning techniques with increasing amounts of noise [22].

Figure 2.2: Schematic depiction of a learning classifier system.

There are two important families of LCSs: the Pittsburgh [102] and
Michigan [15] approaches. Michigan classifier systems can be used for online learning as well as offline learning whereas the Pittsburgh classifier systems can only be applied to offline learning [75]. The Michigan approach is considered in this work, because an online learning system is potentially more scalable as it does not require the whole training set or any sub-sampling of the training set.

In a Michigan-style LCS, the population consists of a single set of cooperative rules, i.e. each individual represents a unique, distinct rule. The goal here is to find the best set of classifier rules that, when applied, gain an optimum result for the problem to be solved. Michigan-style LCSs have two main types of fitness definitions: strength-based, e.g. ZCS [117] and accuracy-based, e.g. XCS [118]. In ZCS the fitness of classifier rules depends on the magnitude of predicted rewards, whereas in XCS the fitness depends on the accuracy of predicted rewards. Due to the strength-based fitness, ZCS proliferates overgeneral classifier rules [68] in certain domains resulting in unsatisfactory performance. XCS is used to implement and test the proposed systems as XCS evolves maximally general and accurate classifiers, attributed to different evolutionary pressures in it [26]. The resulting complete map of states to rewards is considered to contain more building blocks of knowledge than ZCS.

LCS have also been adapted to supervised learning where the environment also returns the ‘correct’ optimal action through the UCS (sUpervised Classifier System) framework [13,90]. UCS is an accuracy-based LCS specifically designed for supervised learning problems. In XCS fitness is computed using a reinforcement learning scheme so it provides a complete mapping of states and actions to rewards, whereas in UCS fitness is calculated from a supervised learning perspective so it evolves a best action map [13]. UCS can only be applied to single-step classification tasks, where supervision is available. However, XCS is more general and can be applied to multi-step problems as well as online reinforcement learning environments [108].
2.4. LEARNING CLASSIFIER SYSTEMS

2.4.1 XCS

XCS is a formulation of LCS that uses accuracy-based fitness to learn the problem by forming a complete mapping of states and actions to rewards. In XCS, the learning agent evolves a population \( [P] \) of classifiers, where each classifier consists of a rule and a set of associated parameters estimating the quality of the rule. Each rule is of the form ‘if \( \text{condition} \) then \( \text{action} \)’, having two parts: a condition and the corresponding action. Commonly, the condition is represented by a fixed-length bitstring defined over the ternary alphabet \( \{0, 1, \#\} \), and the action is represented by a numeric constant. Each classifier has three main parameters: (1) prediction \( p \), an estimate of the payoff expected from the environment if its action is executed; (2) prediction error \( \epsilon \), an estimate of the errors between the predicted payoff and the actually received reward; and (3) fitness \( F \), an estimate of the classifier’s utility. In addition, each classifier keeps an experience parameter \( exp \), which is a count of the number of times it has been updated, and a numerosity parameter \( n \), which is a count of the number of copies of each unique classifier.

The agent has two modes of operation, explore (training) and exploit (application). In the following, XCS operations are concisely described. For a complete description, the interested reader is referred to the original XCS papers by Wilson [118, 119], and to the algorithmic details by Butz and Wilson [28].

In the explore mode, the agent attempts to obtain information about the environment and describes it by creating the decision rules, using the following steps:

- observes the current state of the environment \( s \in S \) where \( S \) is the set of all possible environmental states. The current state \( s \) is usually represented by a fixed-length bitstring defined over the binary alphabet \( \{0, 1\} \).

---

\(^2\)For a detailed review of different types and approaches in LCS refer to [112].
• forms the match set \([M]\) by selecting classifiers from the classifier population \([P]\) that have conditions matching the state \(s\).

• performs covering: for every action \(a_i \in A\) in the set of all possible actions, if \(a_i\) is not represented in \([M]\) then a random classifier is generated with a given generalization probability such that it matches \(s\) and advocates \(a_i\), and added to the set \([M]\) as well as to the population \([P]\). The prediction, prediction error, and fitness of the generated classifier are set to very small initial values.

• forms a system prediction array \(P(a_i)\) for every \(a_i \in A\) that represents the system’s best estimate of the payoff should the action \(a_i\) be performed in the current state \(s\). Commonly, \(P(a_i)\) is a fitness weighted average of the payoff predictions of all classifiers advocating \(a_i\).

• selects an action \(a\) to explore (probabilistically or randomly) and selects all the classifiers in \([M]\) that advocated \(a\) to form the action set \([A]\).

• performs the action \(a\), records the reward \(r\) received from the environment, and uses \(r\) to update the associated parameters of all classifiers in \([A]\).

On receiving the environmental reward \(r\), the parameters of each classifier \(j\) in the action set \([A]\) are updated as follows\(^4\): First of all, the experience \(exp_j\) is increased by one. Then, the prediction error \(\epsilon_j\) is updated: \(\epsilon_j \leftarrow \epsilon_j + \beta(|r - p_j| - \epsilon_j)\) for \(exp_j > 1/\beta\), otherwise \(\epsilon_j \leftarrow [\epsilon_j \ast (exp_j - 1) + |r - p_j|]/exp_j\), where \(\beta(0 \leq \beta \leq 1)\) is the

\(^3\)If the classifier population size grows larger than the specified limit, then one of the classifier rules has to be deleted so that the new rule can be inserted.

\(^4\)Currently only single step problems are under investigation so the parameter updates being described here are for single step problems. For multi-step problems, parameter updates occur in the previous action set \([A]_{-1}\), as described in [118, 119].
learning rate and $p_j$ is the prediction of the classifier $j$. Next, the prediction $p_j$ is adjusted: $p_j \leftarrow p_j + \beta (r - p_j)$ for $exp_j > 1/\beta$, otherwise $p_j \leftarrow [p_j * (exp_j - 1) + r]/exp_j$. After that, the classifier’s accuracy is computed: $k_j = \alpha(\epsilon_j/\epsilon_0)^{-\nu}$ for $\epsilon_j \geq \epsilon_0$, otherwise 1. The parameter $\epsilon_0 (\epsilon_0 > 0)$ determines the threshold error under which a classifier is considered to be accurate, providing robustness to noise. The parameters $\alpha (0 < \alpha < 1)$ and $\nu (\nu > 0)$ are used to handle the degree of decline in accuracy \[25\]. The parameter $\nu$ separates rules of similar fitness to increase the probability for selection of better rules. Then, the relative accuracy $k_j'$ is computed by dividing the accuracy $k_j$ by the total amount of accuracies in the action set. Finally, the fitness $F_j$ is updated according to the classifier’s relative accuracy: $F_j \leftarrow F_j + \beta (k_j' - F_j)$. The relative accuracies based fitness update mechanism implicitly exhibits fitness sharing among the classifiers in an action set. Fitness sharing in conjunction with panmictic deletion allocates resources to niches evenly, i.e. unbalanced classes or complex classes do not get ignored.

- when appropriate, implements rule discovery by applying an evolutionary mechanism (commonly a GA) in the action set $[A]$, to introduce new classifiers to the population.

Additionally, the explore mode may perform subsumption to remove overly specific classifiers by a more general and accurate classifier. There are two subsumption procedures: (a) GA subsumption, and (b) action set subsumption. If GA subsumption is being used and an offspring generated by the GA has the same action as that of the parents, then its parents are examined to see if either of them: (i) has an experience value greater than a threshold, (ii) is accurate, and (iii) is more general than the offspring, i.e. has a set of the matching environmental inputs that is a proper superset of the inputs matched by the offspring. If this test is satisfied then the offspring is discarded and the numerosity of the subsuming parent is
incremented by one. If the offspring is not subsumed by its parents then it can be checked if it is subsumed by other classifiers in the action set.

In action set subsumption, any less general classifiers in an action set $[A]$ are subsumed by the most general subsumer (i.e. accurate and sufficiently experienced) classifier in the set $[A]$. Subsumption deletion is a way of biasing the genetic search towards more general, but still accurate, classifiers [26, 119]. It also effectively reduces the number of classifier rules in the final population [67].

In contrast to the explore mode, in the exploit mode the agent does not attempt to discover new information and simply performs the action with the best predicted payoff. The exploit mode is also used to test learning performance of the agent in the application.

It is to be noted that, due to accuracy-based fitness, XCS keeps a complete map of both correct and incorrect classifiers, i.e. the classifiers advocating consistently correct classification (and hence predicting an accurate, maximum environmental reward of say 1000) as well as the classifiers advocating consistently incorrect classification (and hence predicting an accurate, minimum environmental reward of say 0).

### 2.4.2 Real Valued XCS (XCSR)

The changes to XCS for real-valued inputs were as follows [120, 121]. The classifier condition was changed from a string from \{0, 1, #\} to a concatenation of interval predicates, $\text{int}_i = (l_i, u_i)$, where $l_i$ (“lower”) and $u_i$ (“upper”) are real values. A classifier matches an input message $x$ if each element $x_i$ belongs to the corresponding interval predicate, i.e. $l_i \leq x_i \leq u_i$. When a new covered classifier is created, each interval predicate $\text{int}_i = (l_i, u_i)$ is generated as $l_i = x_i - \text{rand}(r_0)$ and $u_i = x_i + \text{rand}(r_0)$, where $\text{rand}(r_0)$ is a value uniform randomly from $[0, r_0]$ and $r_0$ is a real constant.

The GA works as in XCS. Crossover permutes alleles of two parents between two crossover points. Since an allele is a real value, a new mutation
2.4. LEARNING CLASSIFIER SYSTEMS

operator is introduced. Mutation modifies an allele by adding an amount \( \pm \text{rand}(m_0) \) where \( m_0 \) is a real constant. A classifier \( \text{clfr}1 \) can subsume another classifier \( \text{clfr}2 \) if at least one interval predicate in \( \text{clfr}1 \)'s condition is more general than the corresponding predicate in \( \text{clfr}2 \)'s condition and there is no less general interval predicate in \( \text{clfr}1 \)'s condition than the corresponding predicate in \( \text{clfr}2 \)'s condition.

Covering and mutation operators can generate intervals out of the condition range. If it happens then \( l_i \) and \( u_i \) of an interval predicate \( \text{int}_i \) are brought back to the extremes of the condition range and they are possibly permuted in order to respect the predicate constraint \( l_i \leq u_i \).

XCSR can handle real-valued inputs using interval based conditions, but the possible actions still need to be determined in advance. Yet domains such as robot control require numerical actions, so that neither XCS nor XCSR with their discrete actions can yield high performance in such domains.

2.4.3 Various Rich Encoding Schemes

Various rich encoding schemes have been investigated in LCS to represent high level knowledge in classifier rules. Most of these schemes have been implemented on Wilson’s XCS, which is a well-tested LCS model.

A GP-based rich encoding has been used by Ahluwalia and Bull [1] within a simplified strength-based LCS, named GP-CS [117]. They used binary strings to represent the condition and an S-expression to represent the action of a classifier rule. This GP-based LCS generates filters for feature extraction, rather than performing classification directly. The extracted features are used by the k-nearest neighbor algorithm to perform classification.

Bull and O’Hara [19] developed XCS-based neuro and neuro-fuzzy classifier systems (named X-NCS and X-NFCS), where a condition-action rule was represented by a small neural network; and the action value of
a classifier rule was computed by feedforwarding the environmental state to the neural network. Experimental results indicate that neural network based classifier systems are able to learn single-step as well as multi-step problems [19,52]. Dam et al. [30] implemented a UCS-based classifier system by replacing the typically used numeric action in a classifier rule with a neural network. The developed system, known as NLCS, resulted in better generalization, more compact solutions, and the same or better classification accuracy than a numeric action based UCS. However, in a neural network based system it is difficult to reuse the learnt knowledge as the knowledge is not encapsulated in a reusable form rather it is distributed within the network [105].

Lanzi extended the fixed-length bitstring representation of a classifier condition to a variable-length messy coding in XCSm [73]. A messy coded string may be over- or under-specified, due to its variable-length structure [37]. In the messy coded conditions by Lanzi, environmental inputs were translated into bitstrings, which have no positional linking between bits in a classifier condition and any feature in the environmental input. This messy coding of classifier conditions improved the portability of the behaviors learned between different agents in the Maze4 environment. Then Lanzi and Perrucci [79] enhanced a step further from messy coding to a more complex representation in which LISP s-expressions were used to represent a classifier’s condition in XCSL. XCSL was used to learn the 3-bit multiplexer, the 6-bit multiplexer and the Woods1 problems. In XCSL, the subsumption deletion was not used due to the complexity of determining whether a classifier is more general than another one.

In 2000, Wilson introduced XCSR, a version of XCS taking real-valued inputs [120]. In XCSR, a classifier’s condition is represented as a concatenation of “interval predicates”, \( int_i = (c_i, s_i) \), where \( c_i \) and \( s_i \) are reals. A classifier matches an input \( x \) if and only if \( c_i - s_i \leq x_i < c_i + s_i \), for all \( x_i \). Wilson also introduced an integer-based version of XCS, known as XCSI [121]. In XCSI, a classifier’s condition is represented as a concatenate-
nation of “interval predicates”, \( \text{int}_i = (l_i, u_i) \), where \( l_i \) and \( u_i \) are integers defining the lower and upper bounds of the interval such that \( l_i \leq u_i \). A classifier matches an input \( x \) if and only if \( l_i \leq x_i \leq u_i \), for all \( x_i \). Subsequently, Stone and Bull [106] introduced an ‘unordered bound representation’ to avoid the swapping of endpoints in an interval in classifier conditions, which is otherwise needed in XCSI on the violation of the ordering restriction \( l_i \leq u_i \). Behdad et al. [11] combined XCSR with principal component analysis (PCA) to reduce the computational time and the population size required by XCSR in learning high-dimensional real-valued problems of MiniBooNE, KDD99 Network Intrusion, and Census-Income (KDD) [8]. The important building blocks of knowledge may be lost while performing dimensionality reduction using PCA, so it is not a suitable approach in scalable learning.

Wilson introduced the idea of computed prediction, as a function of the input matched by a classifier’s condition and a weight vector, to learn approximations to functions [122]. In the implemented system, known as XCSF, a classifier’s condition was changed from a ternary alphabet based string to a concatenation of interval based numeric values. Stalph et al. [104] compared the function approximation performance of XCSF with the locally weighted projection regression algorithm (LWPR) [114], which is a statistics-based machine learning technique mostly used for function approximation tasks in robotics. The experimental results show that both methods achieve a suitable performance, but the evolutionary structuring capability of XCSF is more powerful than LWPR’s stochastic gradient descent. In 2005, Lanzi et al. [77] used XCSF for the learning of various Boolean functions. They have shown that XCSF can produce more compact classifier rules as compared to XCS since the use of computed prediction allows more general solutions [78]. Tran et al. [111] implemented XCSF [122] having computed continuous actions, named as XCSFCA, where the action is computed directly as a linear combination of the input state and a vector of action weights. XCSFCA has produced
very good results for the continuous actions frog problems, but could not achieve the 100% performance [111].

Lanzi developed an XCS with stack-based genetic programming [74] where a classifier’s condition was represented by a mathematical expression using reverse polish notation (RPN). A mathematical expression represented in RPN form takes less time to evaluate than the typical infix form. The system did not restrict the generation of syntactically incorrect conditions, therefore the search space was unnecessarily large. Even then, it is reported that the system was able to learn the 6-bit multiplexer, the 11-bit multiplexer and the woodsI problems.

Butz et al. [27] incorporated the EDA mechanism in XCS to identify and process building blocks for solving hierarchical decomposable binary classification problems. They have used extended compact GA (ECGA) and the Bayesian optimization algorithm (BOA) to estimate the probability of distribution. In domains containing building blocks, this approach has shown the benefits of not applying the potentially destructive crossover operation to the fixed-length bitstrings. Therefore, care should be taken while applying the crossover operation to not destruct building blocks.

Charalambos and Browne [56] investigated scaling of an abstracted LCS by implementing classifier conditions as a combination of ternary and S-expression alphabets, and using pre-constructed functions for a specific problem domain. By using domain-relevant functions the scalability of XCS was shown to be improved, but without the domain knowledge the appropriate functions for a problem need to be automatically discovered [56].

Lanzi and Loiacono [76] introduced a version of XCS with computed actions, named XCSCA, to be used for problem domains involving a large number of actions. A classifier’s action was computed using a parametrized function in a supervised fashion. They have shown that XCSCA can evolve accurate and compact representations of binary functions that would be difficult to solve using standard XCS. Then, Loiacono et al. [81] extended
2.5. **CHAPTER SUMMARY**

XCS using support vector machines to compute a classifier’s action, which resulted in reaching optimal performance faster than the original XCS.

In 2008, Wilson implemented a classifier’s condition using gene expression programming (GEP) [124]. GEP based conditions have captured and shown greater insight into the environment’s regularities, albeit the evolution of the system was slower, than traditional methods. The issue with a GEP-based classifier system is that it needs a predetermined threshold to match a classifier’s condition against an environmental input. The match threshold may affect the generalization ability and speed of the system.

2.5 Chapter Summary

Evolutionary machine learning techniques are powerful tools, which autonomously solve various problems such as classification, optimization, data mining, and function approximation [10, 17, 101]. However, the vast majority of these techniques are based on supervised learning, e.g. GA and GP, requiring the whole training set to determine utility of an individual in the population. Therefore, it is hard to learn large-scale problems using supervised learning techniques due to the large search space demanding large memory and long training times. Transfer learning, subsampling, and incremental learning found to be helpful in improving scalability of these techniques [38, 45, 64].

LCSs are a well-structured and flexible evolutionary machine learning paradigm, which can be implemented using either supervised learning or reinforcement learning approach. XCS is a well-tested, online learning and accuracy based LCS model. XCS can scale in problem domains, but has to relearn from the start each time to solve a problem. Further, typically ternary encoded rules in XCS require a large number of rules in the final solution, which is difficult to evolve in learning a large-scale problem due to the large search space demanding much longer training times. By ex-
explicitly feeding the domain knowledge to XCS, scalability can be achieved but it adds bias and restricts use in multiple domains [56].

Various rich encoding schemes have been investigated in XCS to represent high level knowledge in an attempt to improve the generalization, to obtain compact classifier rules, to reach the optimal performance faster, and to generate feature extractors. All of these schemes have their own advantages and limitations, and very little work has been conducted to investigate scalability of XCS. This thesis seeks to improve the scalability of XCS by reusing the learned knowledge from smaller problems in a domain and/or increasing the generalization ability of the system.
Chapter 3

Experimental Design

This chapter provides details of the research methodology, the problem domains experimented here and the experimental setup used for the experimentation. This chapter also clarifies various differences between the evaluation of an LCS and a traditional evolutionary machine learning approach.

3.1 Research Methodology

The overall goal of this thesis is to improve the scalability of learning classifier systems. To achieve this goal the following two research direction are adopted: (1) identify and extract good building blocks of knowledge from smaller problems in a domain and then reuse the extracted knowledge to learn complex, large-scale problems in the domain, (2) increase the generalization ability of classifier rules so that they can capture the underlying patterns in a domain and produce general solutions, solving problems of any scale $n$ in the domain.

In addition to improving the scalability of learning classifier systems, this thesis also aims at understanding of where optimal rules come from in order to appreciate the role of building blocks, and operators on building blocks. To achieve this goal, a concept of parent-trees will be introduced.
to investigate and analyze the evolution of optimal classifiers produced in the final solutions.

The proposed systems extend Wilson’s XCS described in [28], which is an online accuracy-based LCS model. The XCS’s online learning property makes it suitable for this work because it does not require the whole dataset or any subsampling as often required in a supervised learning system to learn a large-scale problem. Firstly, the developed systems will be tested in learning various complex, large-scale Boolean problems. If the proposed techniques successfully solve large-scale Boolean problems, then the ideas will be adopted to design and develop scalable systems to learn real-valued problems.

The rest of this chapter briefly describes the systems to be designed and implemented in this work following the above mentioned research methodologies. The details of each implemented system will be provided in a separate contribution chapter, from Chapter 4 to Chapter 8.

### 3.1.1 Reusing Building Blocks of Extracted Knowledge

In the first proposed system, the typically used ternary alphabet based conditions in an XCS will be replaced by code-fragment based conditions in order to identify, extract and reuse building blocks of knowledge (see Chapter 4). A code fragment is a tree-expression similar to a tree generated in GP (see Section 2.3.2). The fitter building blocks extracted from the learning system trained against smaller problems are to be utilized in learning complex, large-scale problems in the same domain, similar to transferring knowledge in a transfer learning technique (see Section 2.2), in an attempt to develop a scalable classifier system.

The code-fragment conditions can be implemented in two ways: (1) creating and maintaining a separate population of code-fragments and indexing a code-fragment from the population in classifier conditions, and (2) directly replacing a classifier’s condition symbol with a code-fragment.
In the second implementation approach of code-fragment conditions, a separate population of code-fragments is not needed. In both approaches, a code-fragment will be limited to have maximum seven nodes in order to control bloat and save computational time.

### 3.1.2 Increasing Generalization Ability of Classifier Rules

In order to produce a scalable classifier system by evolving more generalized rules than standard XCS, the action in a classifier rule will be replaced by a code-fragment while using the ternary alphabet based classifier conditions (see Chapter 5). The action value in a code-fragment action can be computed in two ways: (1) by loading the terminal symbols in the action tree with the corresponding binary values from the condition in the classifier rule, and (2) by loading the terminal symbols with the corresponding binary values from the environmental input. In the first approach to compute the action value of a classifier rule, a ‘#’ symbol will be randomly treated as 0 or 1 while loading into a terminal symbol in the code-fragment action.

It is expected that the proposed code-fragment based systems will solve problems of a scale that existing classifier systems cannot. However, an anticipated limitation in the above mentioned proposed techniques is the lack of a cyclic representation to encapsulate the underlying repeated patterns in a problem domain. In principle they could have cycles, but it is difficult to avoid infinite loops in the evolved solutions. A finite state machine (FSM) is a cyclic representation, which has the ability to encapsulate repeated patterns in a problem domain and does not become stuck in infinite loops. However, the evolution of FSMs is a hard task due to the combinatorially large number of possible states, connections and interaction. Usually this requires supervised learning to minimize inappropriate FSMs [53,82], which for large-scale problems necessitates subsampling and/or incremental testing. To avoid these constraints, in the third
proposed system, the action in a classifier rule will be replaced by a state-machine while using the ternary alphabet based classifier conditions (see Chapter 6).

If the above mentioned proposed techniques successfully solve large-scale Boolean problems, then the method of learning general rules using code-fragment actions will be adopted to extend the existing real-valued XCSR system [120] in order to tackle continuous learning problems (see Chapter 7). The code-fragment action based XCSR system will be tested in learning function approximation as well as continuous action problems where the action is a continuous function of the input.

3.1.3 Evolvability of Classifier Rules

In addition to developing the above mentioned scalable classifier systems, this work will introduce and implement a method to trace the evolution of classifier rules generated in an XCS-based classifier system. Specifically, the concept of a family tree, termed parent-tree, for each individual classifier rule generated in the system during training is to be introduced, which describes the whole generational process for that classifier (see Chapter 8). Various statistics at each level in the parent-trees are also to be computed to analyze the evolution of rules from different aspects.

In order to trace the family hierarchy of an evolved classifier rule using its parent-tree, the following additional attributes will be maintained for each classifier:

- $id$, which is a unique identification number for each classifier.
- $gen$, which is the generation number of the classifier.
- $pid_1$ and $pid_2$, which are the identification numbers of its parents (if any).
- $d$, which is the depth of its parent-tree.
3.2. **THE PROBLEM DOMAINS**

The developed system will be tested on Boolean problems using two XCS-based systems, i.e. standard XCS and XCS with code-fragment actions as they produce easily interpretable classifier rules.

## 3.2 The Problem Domains

### 3.2.1 Boolean Problems

This work mainly focused on Boolean problems because they have measurable search space and identifiable building blocks. Further, it is easy to analyze and investigate evolved solutions in Boolean problems. However, there is no apparent hurdle to extend and adopt the proposed systems in other problem domains. The Boolean problems from the following domains are to be used in the experimentation: the multiplexer, the majority-on, the count ones, the digital design verification, the carry, and the even-parity.

**The Multiplexer Problem Domain**

A multiplexer is an electronic circuit that accepts input strings of length $n = k + 2^k$, and gives one output [22]. The value encoded by the first $k$, so called, address bits is used to select one of the $2^k$ remaining data bits to be given as output. For example in the 6-bit multiplexer, if the input is 011011 then the output will be 0 as the first two bits 01 represent the index 1 (in base ten), which is the second bit following the address. Multiplexer problems are highly non-linear, multi-modal and have epistasis, i.e. importance of data bits is dependent on address bits.

**The Majority-On Problem Domain**

In majority-on problems [64], the output depends on the number of ones in the input instance. If the number of ones is greater than the number
of zeros, the problem instance is of class one, otherwise class zero. In the majority-on problem domain, the complete solution consists of overlapping classifiers, so is difficult to learn. For example, ‘1##11:1’ and ‘11#1#:1’ are two maximally general and accurate classifiers, but they overlap in the “11*11” subspace.\footnote{Here, * can be 0, 1, or #.}

The Count Ones Problem Domain

Count ones problems are similar to majority-on problems in that the output depends on the number of ones in the input instance [22]. In count ones problems only $k$ bits are relevant in an input instance of length $l$ [22]. If the number of ones in the $k$ relevant positions is greater than half $k$, the problem instance is of class one, otherwise class zero. For example, consider a count ones problem of length $l = 7$ with the first $k = 5$ relevant bits. In this problem, input string 1010110 would be in class one whereas input string 1001011 would be class zero. Similar to majority-on problems, the complete solution for a count ones problem consists of overlapping classifiers.

The Design Verification Problem Domain

Digital design verification is a real world problem domain, where a digital design is verified before manufacturing in order to discover as many bugs in the design as possible. The design verification problem experimented in this work is a 7-bit Boolean example of a simulation-based DV problem, named DV1, originally introduced by Ioannides et al. [55]. A Boolean function can be represented compactly in the Sigma notation by listing each onset row from the truth table of the function [31]. For example, the function $xy + xy$ can be represented in Sigma notation as $\Sigma(1, 3)$. The DV1 problem is denoted by the following Sigma notation: $\Sigma(1, 2, 3, 8, 9, 10, 11, 13, 14, 24, 25, 26, 27, 28, 30, 40, 41, 42, 43, 46, 47, 56, 57, 58, 59, 61, 65, 66, 67$, \footnote{Here, * can be 0, 1, or #.}
3.2. THE PROBLEM DOMAINS

Similar to majority-on and count ones problems, the complete solution for the DV1 problem consists of overlapping classifiers, and in addition it is a niche imbalance problem. For example, ‘#00#0#1 : 1’ and ‘#00#01# : 1’ are two maximally general and accurate classifiers, but they overlap in the “**00**11” subspace. Similarly, ‘#1 1 1 0#1 : 1’ and ‘1####01 : 1’ are two maximally general and accurate classifiers, but the size of the niche covered by the latter is twice that of the former. The optimum ternary encoded solution set for the DV1 problem consists of 30 maximally general and accurate classifiers [55], shown in Table 3.1. It is to be noted that there are different numbers of ‘#’ symbols in the conditions of these overlapping classifiers, which indicate that these classifiers cover different sized niches.

The Carry Problem Domain

In carry problems [22], two binary numbers of the same length are added. If the addition triggers a carry, then the class is one otherwise zero. For example, in case of 3-bit numbers 110 and 101, the class is one, whereas for the numbers 010 and 011 the class is zero. The complete solution in the carry problem domain consists of strongly overlapping classifiers, and in addition it is a niche imbalance problem domain. For example, ‘###00# : 0’ and ‘###000 : 0’ are two maximally general and accurate classifiers, but they overlap in the “***0000” subspace. Similarly, ‘1##1## : 1’ and ‘1#1#11 : 1’ are two maximally general and accurate classifiers, but the size of the niche covered by the former is twice that of the latter. The optimum ternary encoded solution set for the 3+3 bit carry problem consists of 18 maximally general and accurate classifiers, shown in Table 3.2. It is to be noted that there are different numbers of ‘#’ symbols in the conditions of these overlapping classifiers, which indicate that these classifiers cover different sized niches.
Table 3.1: The optimum ternary encoded rule set for the DV1 problem [55].

<table>
<thead>
<tr>
<th>No.</th>
<th>Input</th>
<th>Output</th>
<th>No.</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>###0#00</td>
<td>0</td>
<td>16</td>
<td>1####10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>010#10#</td>
<td>0</td>
<td>17</td>
<td>#00#01#</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0##01##</td>
<td>0</td>
<td>18</td>
<td>#0#10##</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>##10111</td>
<td>0</td>
<td>19</td>
<td>#001#01</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>#00#100</td>
<td>0</td>
<td>20</td>
<td>#0010##</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1###100</td>
<td>0</td>
<td>21</td>
<td>0011##0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>00##111</td>
<td>0</td>
<td>22</td>
<td>#0#1##10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1101111</td>
<td>0</td>
<td>23</td>
<td>0101##1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0#10###</td>
<td>0</td>
<td>24</td>
<td>100###1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>111##00</td>
<td>0</td>
<td>25</td>
<td>0##10##</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>001#1#1</td>
<td>0</td>
<td>26</td>
<td>1#00##1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>011#1#0</td>
<td>0</td>
<td>27</td>
<td>#111##01</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>01#0###</td>
<td>0</td>
<td>28</td>
<td>1#11##1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0#1#111</td>
<td>0</td>
<td>29</td>
<td>1###0##1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>#00#0#1</td>
<td>1</td>
<td>30</td>
<td>1###0#1</td>
<td>1</td>
</tr>
</tbody>
</table>

The Even-Parity Problem Domain

In even-parity problems [22], the output depends on the number of ones in the input instance. If the number of ones is even, the output will be one, and zero otherwise. Using the ternary alphabet based conditions with the static numeric action, no useful generalizations can be made for even-parity problems.

The properties of different Boolean problem domains to be experimented in this work are summarized in Table 3.3.
3.2. **THE PROBLEM DOMAINS**

Table 3.2: The optimum ternary encoded rule set for the 3+3 bit carry problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#0000#</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>###000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>00###0#</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>000###</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0##0##</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0###00</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>00###0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0#0#0#</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>#0#0#0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>#0#00#</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>#000##</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1##1##</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1###1#</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>#1#11#</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>111###</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1#1#11</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>#111#1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>##1111</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: Properties of different Boolean problem domains.

<table>
<thead>
<tr>
<th>Problem Domain</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplexer</td>
<td>multi-modal and epistatic</td>
</tr>
<tr>
<td>Majority-On</td>
<td>overlapping</td>
</tr>
<tr>
<td>Count Ones</td>
<td>overlapping</td>
</tr>
<tr>
<td>Design Verification</td>
<td>overlapping and niche imbalance</td>
</tr>
<tr>
<td>Carry</td>
<td>overlapping and niche imbalance</td>
</tr>
<tr>
<td>Even-Parity</td>
<td>hard to generalize</td>
</tr>
</tbody>
</table>

3.2.2 **Continuous Action Problems**

The continuous action problem to be used for experimentation in this work is the frog problem, introduced by Wilson [123], which is a benchmark in the field of LCS. In the frog problem, a system (frog) senses an object (fly) via a signal that monotonically decreases with the distance between them [123]. The frog should learn to catch the fly in one jump. Let $d (0.0 \leq d \leq 1.0)$ be the frog’s distance from the fly. For simplicity, it is assumed that the frog’s sensory input $x$, falls linearly with distance $d$, as
given in Equation 3.1:

\[ x(d) = 1 - d \] (3.1)

On receiving the signal input \( x \), the frog jumps a certain distance \( a \) as an action. A jump can go beyond or fall short of the fly. Then the frog receives a payoff related to the remaining distance, as given in Equation 3.2:

\[
P(x, a) = \begin{cases} 
  x + a & \text{if } x + a \leq 1 \\
  2 - (x + a) & \text{otherwise}
\end{cases}
\] (3.2)

It is to be noted that the payoff function is continuous and nonlinear – albeit composed of two linear planes. To solve the frog problem, a system must learn to choose, given \( x \), the value of \( a \) corresponding to maximum payoff.

Tran et al. [111] introduced a modified version of the frog problem, which is still continuous and nonlinear but composed of two nonlinear forms, given in Equation 3.3.

\[
P(x, a) = \begin{cases} 
  xe^a & \text{if } a \leq -\ln(x) \\
  x^{-1}e^{-a} & \text{otherwise}
\end{cases}
\] (3.3)

The frog’s sensory input \( x \) in the modified version is given by Equation 3.4.

\[ x(d) = e^{-d} \] (3.4)

For the sake of readability, the original frog problem is called \textit{frog1} and the modified version is called \textit{frog2}. The developed real-valued classifier system will be tested on both versions of the frog problem in order to investigate its capability in learning continuous problems of different difficulty levels.
3.2.3 Real-Valued Function Approximation Problems

It is to be noted that the main aim of developing a real-valued classifier system in this work is to learn continuous action problems instead of developing an approximator. However, due to the generic nature of code-fragment actions the developed system can also be used as an approximator. There are many function approximation problems used in the literature. However, in this work the following real-valued function approximation problems, of different difficulty levels, will be used to test the approximation capability of the developed real-valued classifier system.

**Continuous Functions with Continuous Derivative**

The simplest approximation function to be tested in this work is the parabola given in Equation 3.5. This is a continuous Function with continuous derivative.

\[ y = 1 + x^2 \text{ where } -1 \leq x \leq 1 \]  

(3.5)

**Continuous Functions with Discontinuous Derivative**

The continuous function with discontinuous derivative to be used for approximation in this work is given in Equation 3.6.

\[
y = \begin{cases} 
1 - x & \text{if } -1 \leq x < 0 \\
1 + x^2 & \text{if } 0 \leq x \leq 1 
\end{cases}
\]  

(3.6)

**Discontinuous Functions**

The discontinuous function to be used for approximation in this work is given in Equation 3.7.

\[
y = \begin{cases} 
1 + x^2 & \text{if } -2 \leq x \leq -1 \\
3 - x & \text{if } 1 \leq x \leq 2 
\end{cases}
\]  

(3.7)
The problems given in Equation 3.6 and Equation 3.7 are difficult to be learn by traditional evolutionary machine learning techniques that evolve a single individual as a solution. It is to be noted that the function boundary of the two pieces in Equation 3.7 is separated.

Multi-dimensional Functions

The real-valued approximation problem given in Equation 3.8 is a two-dimensional function and the one given in Equation 3.9 is a four-dimensional approximation function to be used to test the developed system on multi-dimensional real-valued problems.

\[ z = x + y^2 \text{ where } -1 \leq x \leq 1 \text{ and } -1 \leq y \leq 1 \]  
(3.8)

\[ y = a^2 + b^2 + c^2 + d^2 \text{ where } a, b, c, d \in [-1, 1]. \]  
(3.9)

3.3 Experimental Setup

Unless stated otherwise, the following parameter values, commonly used in the literature, are used for the experimentation here, as suggested by Butz in [21], and by Butz and Wilson in [28]: learning rate \( \beta = 0.2 \); fitness fall-off rate \( \alpha = 0.1 \); prediction error threshold \( \epsilon_0 = 10 \); fitness exponent \( \nu = 5 \); threshold for GA application in the action set \( \theta_{GA} = 25 \); two-point crossover with probability \( \chi = 0.8 \); mutation probability \( \mu = 0.04 \); 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’t care’ symbol in covering \( P_{\text{don’tCare}} = 0.33 \); reduction of the prediction error \( \text{predictionErrorReduction} = 0.25 \); reduction of the fitness \( \text{fitnessReduction} = 0.1 \); and the selection method is tournament selection with a tournament size ratio 0.4. Both GA subsumption and action set subsumption are activated.\(^2\) Explore and exploit problem instances are

\(^2\)The benefit, or otherwise, of action set subsumption is explored further in Section 5.5.
alternated. The reward scheme used is 1000 for a correct classification and 0 otherwise. All the experiments have been repeated 30 times with a different seed in each run.

All results obtained in this work are the average of the 30 independent runs. In all graphs presented here, 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 exploit problem instances.

3.4 Evaluation

Usually in a traditional evolutionary machine learning technique, e.g. GA and GP, the learning system evolves a predictor using a training set, which is then evaluated/tested using a separate testing set from the problem dataset. However, an LCS evolves a set of classifier rules to generate a model for the problem using the randomly chosen problem instances from the whole dataset during the learning/training process. In learning small datasets using an LCS approach only the generated model is tested, but in large datasets, e.g. MUX greater than 20, the number of training instances is (much) less than the number of problem instances, so generalization is also tested.

In traditional evolutionary machine learning techniques the whole training set is used to determine the utility/fitness of an individual in each generation whereas in LCSs a single problem instance is used at each learning step/iteration to evaluate/update the fitness of the individuals belonging to the action set. The standard approach, which is adopted here, is to report the classification performance measured as the moving average over the last 1000 exploit trials.
Chapter 4

Reusing Building Blocks of Extracted Knowledge to Solve Complex, Large-Scale Boolean Problems

4.1 Introduction

XCS is a rule-based classifier system in which fitness of a classifier rule depends on the prediction-accuracy of the rule. In the work presented in this chapter, the typically used ternary alphabet based conditions in an XCS will be replaced by code-fragment based conditions in order to extract and reuse building blocks of knowledge. A code fragment is a tree-expression similar to a tree generated in GP (see Section 2.3.2). Code-fragments are easy to use as knowledge extractors than other alternatives such as automatically defined functions (ADFs) [71] and artificial neural networks (ANNs) [32], because ADFs have a predefined structure and fixed number of arguments, and ANNs are black box with knowledge distributed throughout the neural net.
CHAPTER 4. REUSING BLOCKS OF EXTRACTED KNOWLEDGE

The fitter building blocks extracted from the learning system trained against smaller problems will be utilized in learning more complex, large-scale problems in the same domain, similar to transferring knowledge in a transfer learning technique (see Section 2.2), in an attempt to develop a scalable classifier system.

The following sections describe in detail two implementation methods for the proposed approach of XCS with code-fragment conditions. The first is a simple implementation method whereas the second is an improved version of the first implementation. The experimental results in different complex Boolean problem domains are presented and compared with the standard ternary alphabet based XCS and related layered learning GP systems.

4.2 XCS with Code-Fragment Conditions
(A Simple Approach)

4.2.1 Method

In the proposed XCS with code-fragment conditions, called XCSCFC, each code fragment is a binary tree of depth up to two, which was set to limit the tree size. A binary tree of depth two can have a maximum of seven nodes. The function set for the tree is problem dependent such as \{+, -, *, /\} for symbolic regression problems, and \{AND, OR, NAND, NOR \} for binary classification problems. The terminal set is \{D0, D1, D2, ... Dn-1\}, where n is the length of the environmental input state. First of all a random population of, genotypically distinct, code fragments [Q] is generated using the ramped half and half initialization method commonly used in GP, which ensures a degree of syntactic variety [9]. The code-fragments population [Q] is kept static throughout the learning process. Then these code fragments are randomly chosen by the system to generate classifier rules. The populations of classifiers and code fragments are illustrated in
4.2. XCS WITH CODE-FRAGMENT CONDITIONS (A SIMPLE APPROACH)

Figure 4.1. The symbols &, |, d, r, and ~ denote AND, OR, NAND, NOR, and NOT operators respectively. The code fragments are shown in the postfix form.

The XCSCFC system extends standard XCS [28] in the following aspects: 1) the classifier matching procedure, the covering operation, the rule discovery operation, the subsumption deletion mechanism, and the procedure comparing equality of two classifiers are modified; and 2) the extracted domain knowledge is reused in the form of code fragments.

**Classifier Matching**

A classifier rule $cl$ from the population $P$ is said to be matched against a problem instance $s$ from the environment if each code fragment in its condition outputs 1. A code fragment is evaluated by loading the terminal symbols with corresponding binary bits from the observed environmental state $s$. Assuming the problem instance $s$ is 110101, then the code fragment shown in Figure 4.2 will give 1 as the output. This output value was generated by loading D0, D1, D2, and D5 with 1, 1, 0, and 1 respectively.
There is a special code fragment to be used as ‘don’t care’ symbol in the condition of a classifier rule, shown in Figure 4.3. This code fragment always outputs 1.

Although for simplicity there is the same number of code fragments as condition features, e.g. 6 for the 6-bit MUX problem, there is a decoupling between a code fragment and a position within the condition that is, unlike in standard ternary alphabet based XCS, the order of code fragments is unimportant. The number of ‘specific’ code fragments is essentially messy as the system can choose how many ‘don’t care’ code fragments it uses. The classifier matching procedure is described in Algorithm 1. Here \( n \) is the length of condition \( \text{cond} \) in a classifier rule.
4.2. XCS WITH CODE-FRAGMENT CONDITIONS (A SIMPLE APPROACH)

Algorithm 1: XCSCFC\textsubscript{simple}: Does Match

**Data:** The currently observed input state \( s \) and a classifier \( cl \in [P] \)
where \([P]\) is the population of classifiers.

**Result:** If the classifier \( cl \) matches the state \( s \), then this algorithm will return \( true \) otherwise \( false \).

\begin{verbatim}
for i = 1 to n do
    cf ← code fragment from \([Q]\) indexed at \( cl\.cond[i] \)
    if \( cf \neq \text{'don't care' code fragment} \) then
        load terminal symbols in \( cf \) with corresponding binary bits
        from the state \( s \)
        val ← evaluate value of \( cf \)
        if val \( \neq 1 \) then
            return \( false \)
        end
    end
end
return \( true \)
\end{verbatim}

Covering Operation

Covering occurs if an action is missing in the match set \([M]\). In the covering operation, a random classifier is created whose condition matches the current environmental state \( s \) and contains ‘don’t care’ code fragments with probability \( P_{\text{don't Care}} \). All the ‘non-don’t care’ code fragments in this newly created classifier must output 1 against the observed state \( s \). The covering operation is described in Algorithm 2. Here \( n \) is the length of condition \( cond \) in a classifier rule, and \( P_{\text{don't Care}} \) is the probability of ‘don’t care’ code fragment in the condition of the newly created classifier in the covering operation.
Algorithm 2: XCSFC<sub>simple</sub>: Covering.Operation

**Data:** The currently observed input state \( s \) and an action \( a \) missing in the match set \([M]\).

**Result:** A newly created classifier \( cl \) matching the state \( s \) and advocating the action \( a \).

1. initialize classifier \( cl \)
2. initialize condition \( cl.cond \) with length \( n \)
3. for \( i = 1 \) to \( n \) do
   4. if RandomNumber\([0, 1) < P_{\text{don't care}} \) then
      5. \( cl.cond[i] \leftarrow \) index of the 'don't care' code fragment
   6. else
      7. initialize value \( val \) to 0
   8. while \( val \neq 1 \) do
      9. \( cf \leftarrow \) randomly choose a 'non-don't care' code fragment from \([Q]\)
      10. load terminal symbols in \( cf \) with corresponding binary bits from the state \( s \)
      11. \( val \leftarrow \) evaluate value of \( cf \)
   12. \( cl.cond[i] \leftarrow \) index of \( cf \) from \([Q]\)
   13. end
   14. end
15. \( cl.action \leftarrow a \)
16. return \( cl \)

**Rule Discovery Operation**

In the rule discovery operation, two offspring are produced by applying the GA in the action set \([A]\). First of all, two parent classifiers are selected from \([A]\) based on fitness and the offspring are created out of them. Next, the conditions of the offspring are crossed with probability \( \chi \) using a two
point crossover operation. The crossover operation is described in Algorithm 3. Here $n$ is the length of condition $cond$ in a classifier rule.

### Algorithm 3: XCSCFC\textit{simple}: Crossover Operation

**Data:** Two offspring classifiers $cl_1$ and $cl_2$.

**Result:** The crossed over classifiers $cl_1$ and $cl_2$.

1. $x \leftarrow \text{RandomNumber}[0, n]$
2. $y \leftarrow \text{RandomNumber}[0, n]$
3. if $x > y$ then
4.     swap $x$ and $y$
5. end
6. for $i = x$ to $y$ do
7.     swap $cl_1.\text{cond}[i]$ and $cl_2.\text{cond}[i]$
8. end
9. return $cl_1$ and $cl_2$

After that, each code fragment in the conditions of the produced children by crossover is mutated with probability $\mu$, such that both children match the currently observed state $s$. In the mutation operation, the index of a ‘non-don’t care’ code fragment is replaced by the index of the ‘don’t care’ code fragment, and the index of the ‘don’t care’ code fragment is replaced by the index of a randomly selected ‘non-don’t care’ code fragment that outputs 1 against the state $s$. Then, the actions of the children are mutated with probability $\mu$. The mutation operation is described in Algorithm 4. Here $n$ is the length of condition $cond$ in a classifier rule, and $\mu$ is the mutation probability.

The prediction of the offspring is set to the average of the parents’ values whereas the prediction error and the fitness of the offspring are set to the average of the parents’ values multiplied by constants $\text{predictionErrorReduction}$ and $\text{fitnessReduction}$ respectively, as in [21].

\footnote{The XCSCFC system does not depend on any specific type of crossover operation, so the interested researcher can use any type of crossover operation in XCSCFC.}
CHAPTER 4. REUSING BLOCKS OF EXTRACTED KNOWLEDGE

Algorithm 4: XCSCFC simp lee: Mutation_Operation

Data: The currently observed input state $s$ and an offspring classifier $cl$ matching the state $s$.

Result: The mutated classifier $cl$ which still matches the state $s$.

\begin{verbatim}
for $i = 1$ to $n$ do
  if $\text{RandomNumber}[0, 1) < \mu$ then
    if $\text{cl.cond}[i] =$ index of the 'don’t care' code fragment then
      initialize value $val$ to 0
      while $val \neq 1$ do
        $cf$ ← randomly choose a 'non-don’t care' code fragment from $[Q]$
        load terminal symbols in $cf$ with corresponding binary bits from the state $s$
        $val$ ← evaluate value of $cf$
      end
      $\text{cl.cond}[i] \leftarrow$ index of $cf$ from $[Q]$
    else
      $\text{cl.cond}[i] \leftarrow$ index of the 'don’t care' code fragment
    end
  end
  if $\text{RandomNumber}[0, 1) < \mu$ then
    $a \leftarrow \text{cl.action}$
    $\text{cl.action} \leftarrow$ randomly chosen action other than $a$
  end
return $cl$
\end{verbatim}

Subsumption Deletion

Utilizing code fragments for the matching component of XCS removes the implicit linking between the position of a condition bit in a classifier rule
and the corresponding feature in the environmental input. Although this can lead to compaction of a rule, it also places additional pressure on subsumption deletion as the reordering of the same conditions needs to be taken into account. It is to be noted that due to the multiple genotypes to a single phenotype issue caused by using tree-based code fragments in place of ternary symbols in the conditions of classifier rules, subsumption deletion is less likely to occur as compared with standard XCS. Subsumption deletion is still made possible, albeit problematic, by matching the code fragments on a character by character basis. The reason for a syntactic equality comparison of code fragments, instead of a semantic one, is that semantic comparison of two code fragments would require evaluation of the code fragments against each possible value of the terminal symbols in both code fragments. As the terminal symbols can be smaller level code fragments (see Section 4.2.1), for large-scale problems semantic comparison is impractical due to the amount of time needed for evaluation of the code fragments.

A classifier $cl_1$ can subsume another classifier $cl_2$ if both have the same action and $cl_1$ is accurate, sufficiently experienced, and more general than $cl_2$. Classifier $cl_1$ will be more general than classifier $cl_2$ if $cl_1$ has a set of the matching environmental inputs that is a proper superset of the inputs matched by $cl_2$. In XCSCFC, a classifier $cl_1$ is said to be more general than a classifier $cl_2$ if: 1) the number of ‘don’t care’ code fragments in the condition of $cl_1$ is larger than the number of ‘don’t care’ code fragments in the condition of $cl_2$; and 2) each ‘non-don’t care’ code fragment in the condition of $cl_1$ is in the condition of $cl_2$. The algorithm to determine whether a classifier is more general than another one is described in Algorithm 5.

Comparing Equality of Two Classifiers

If a newly created classifier in the rule discovery operation is not subsumed (either by the parents or in the action set) and there is no classifier equal to it in the population, then it will be added to the population. Two
Algorithm 5: XCSFC\textsubscript{simple}: Is\_More\_General

Data: Two classifiers $cl_1$ and $cl_2$.

Result: If classifier $cl_1$ is more general than classifier $cl_2$, then this
procedure will return \textit{true} otherwise \textit{false}.

1. $x \leftarrow$ number of ‘don’t care’ code fragments in $cl_1$
2. $y \leftarrow$ number of ‘don’t care’ code fragments in $cl_2$
3. if $x \leq y$ then
   4. return \textit{false}
   5. end
6. $X \leftarrow$ set of all ‘non-don’t care’ code fragments in $cl_1$
7. $Y \leftarrow$ set of all ‘non-don’t care’ code fragments in $cl_2$
8. if $X \nsubseteq Y$ then
    9. return \textit{false}
   10. end
11. return \textit{true}

Classifiers are considered to be equal if and only if both have the same action and the same code fragments in their conditions. The procedure to compare two classifiers for equality is given in Algorithm 6.

Reusing Extracted Knowledge

The fitter code fragments, i.e. building blocks of information, from a smaller problem, are used to create the population of code fragments in a higher level problem of the same domain. Each code fragment can be considered a module as in modular CGP [116], and each problem level can be considered a subtask layer as in layered learning [107]. In the work presented in this section, code fragments are kept static throughout the learning process whereas in modular CGP the modules are allowed to evolve.

The code fragments from a smaller problem are used as terminals in the code fragments of the next level problem. The probability of a terminal to
Algorithm 6: XCSFC\textsubscript{simple}: Are\_Equal\_Classifiers

**Data:** Two classifiers $cl_1$ and $cl_2$.

**Result:** If classifiers $cl_1$ and $cl_2$ have the same action and the same code fragments in their conditions, then this procedure will return $true$ otherwise $false$.

1. if $cl_1\.action \neq cl_2\.action$ then
2.  return $false$
3. end
4. $x \gets$ number of ‘non-don’t care’ code fragments in $cl_1$
5. $y \gets$ number of ‘non-don’t care’ code fragments in $cl_2$
6. if $x \neq y$ then
7.  return $false$
8. end
9. $X \gets$ set of all ‘non-don’t care’ code fragments in $cl_1$
10. $Y \gets$ set of all ‘non-don’t care’ code fragments in $cl_2$
11. if $X \neq Y$ then
12.  return $false$
13. end
14. return $true$

be a previous level code fragment or a condition bit from the current level problem is set to 0.5 following the ramped half and half approach of initializing a population in GP, which ensures a degree of syntactic variety [9].

An example of the population of code fragments for the 20-bit MUX is shown in Figure 4.4. The code-fragments in the 20-bit MUX contain fitter code fragments from the 11-bit MUX, similarly the code-fragments in the 11-bit MUX contain code fragments from the 6-bit MUX. The code-fragments in the 6-, 11-, and 20-bit MUX problems are named as L1\textsubscript{n}, L2\textsubscript{n}, and L3\textsubscript{n} respectively, where $n = 0, 1, 2 \ldots$ etc.

The fitness of a code fragment $cf$ is determined according to Equation 4.1, where specificity is the number of ‘non-don’t care’ code fragments
CHAPTER 4. REUSING BLOCKS OF EXTRACTED KNOWLEDGE

Figure 4.4: A sample population of code fragments in the 20-bit multiplexer problem. The code fragments in the 6-, 11-, and 20-bit MUX problems are named as L1_n, L2_n, and L3_n respectively, where n = 0, 1, 2 ... etc.

in the classifier \( cl \) containing the code fragment \( cf \).

\[
 cf.fitness = cf.fitness + (cl.specifincity)^{−\nu} \tag{4.1}
\]

Initially each code fragment has fitness value equal to 0. Whenever the fitness of a classifier is updated, then fitness of all the code fragments occurring in condition of the classifier is also updated according to the Equation 4.1. It is to be noted that in Equation 4.1, the fitness of a code fragment does not explicitly depend on the fitness of a classifier. Instead the code fragments in a more general classifier that frequently occur in an action set will have relatively larger fitness values than the others, because in accuracy-based classifier systems an action set is formed by the good classifiers.

4.2.2 Results

In order to test the performance of XCSCFC, the results have been compared with standard XCS on the multiplexer problems and the even-parity
problems. The function set for the code fragments used is \{AND, OR, NAND, NOR, NOT\}, for all the problems experimented here. Each result obtained in this work is the average of the 30 independent runs. In all graphs presented here, 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 exploit problem instances.

The Multiplexer Problem Domain

The performance of standard XCS and XCSCFC in the multiplexer problem domain is shown in Figure 4.5. The number of classifiers used is 500, 1000, 2000, and 5000 for the 6-, 11-, 20-, and 37-bit multiplexer problems respectively, as commonly used in the literature. The number of code fragments used is twice the number of classifiers, chosen empirically. The number of fitter code fragments taken from 6-bit MUX to seed the population of code fragments for 11-bit MUX is 22, from 11-bit MUX to 20-bit MUX is 40, and from 20-bit MUX to 37-bit MUX is 74. The number of reused code fragments was chosen after initial experiments. The number of training examples used is half a million for the 6-, 11-, and 20-bit multiplexer problems and one million for the 37-bit multiplexer problems. Standard XCS was not able to solve the 37-bit MUX problem with \( P_{\text{don't care}} = 0.33 \), so \( P_{\text{don't care}} \) was increased to 0.5 in Figure 4.5(b). Here \( p\# \) denotes the probability of ‘don’t care’ symbol.

XCSCFC needs more training examples than standard XCS to learn the 6-bit and the 11-bit MUX problems, as shown in Figure 4.5(a). For the 20-bit MUX, XCSCFC achieved greater than 90% performance more rapidly than XCS, but to reach 100% performance level it requires more training examples than XCS, see Figure 4.5(a). Standard XCS, with parameter tuning, needs approximately 800k problem instances to solve the 37-bit MUX problem, see Figure 4.5(b). However, XCSCFC takes approximately

\(^2\)In simple problems the conventional parameter values set produces robust performances, but requires adjustment in complex domains, e.g. 37-bit MUX and above.
CHAPTER 4. REUSING BLOCKS OF EXTRACTED KNOWLEDGE

Figure 4.5: Results of the multiplexer problems using XCS and XCSCFC.

(a) for 6-, 11-, and 20-bit problems.

(b) for 37-bit problem.

250k problem instances to solve the 37-bit MUX problem, without parameter tuning. For the 37-bit MUX, XCSCFC, reusing the extracted building blocks of knowledge from smaller problems, has shown improvement in terms of reducing the required input instances from the environment.
4.2. XCS WITH CODE-FRAGMENT CONDITIONS (A SIMPLE APPROACH)

The Even-Parity Problem Domain

The performance of standard XCS and XCSCFC in the even-parity problem domain is shown in Figure 4.6. The number of classifiers used is 200, 300, 400, and 500 for the 2-, 3-, 4-, and 5-bit problems respectively. The number of code fragments used is twice the number of classifiers. The number of fitter code fragments taken from 2-bit problem to seed the population of code fragments for 3-bit is 20, from 3-bit to 4-bit is 40, and from 4-bit to 5-bit is 60, again chosen empirically. Each run is stopped after half a million training examples.

It is observed that XCSCFC needs more training examples than standard XCS to learn the 2-, 3-, and 4-bit even-parity problems, as shown in Figure 4.6. As the problem scales to 5-bit, standard XCS cannot learn the even-parity problem using half a million training examples with the experimental setup given in section 3.3, see Figure 4.6(b), whereas XCSCFC has solved up to 5-bit even-parity problems successfully.

4.2.3 Interpretation of Results

Building Blocks Analysis

The final rule-base of a typical XCSCFC run on the 6-bit MUX problem was analyzed to extract the most used code fragments, see Table 4.1 for the top 10 code fragments out of 208 retained from the initial 2000 created. The system has eliminated the less fit code fragments from the classifiers. It is a good feature of XCSCFC that it can automatically choose the fittest code fragments from a possibly large population of code fragments.

If the code fragments behaved as a standard representation, then the code fragments of length 1 and ‘don’t care’s would be expected in rules, e.g. D0~, D1, D4~ : 0. However, such rules were not present, with rules containing longer code fragments preferred. These longer code fragments often contained the address bits (either D0, D1 or combination). The most common code fragment (rank 1) was the ‘don’t care’ equivalent. The rank
CHAPTER 4. REUSING BLOCKS OF EXTRACTED KNOWLEDGE

(a) for 2- and 3-bit problems.

(b) for 4- and 5-bit problems.

Figure 4.6: Results of the even-parity problems using XCS and XCSCFC.
Table 4.1: The top 10 highly ranked code fragments.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Code Fragment</th>
<th>Frequency</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D0D0</td>
<td>2121</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>D3D0rD1D5d</td>
<td>770</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>D2D1r~</td>
<td>516</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>D5D1&amp;D0~</td>
<td>490</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>D1<del>D5</del></td>
<td>469</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>D1D2rD0</td>
<td></td>
<td>455</td>
</tr>
<tr>
<td>7</td>
<td>D1D3&amp;~</td>
<td>436</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>D4D1rD1</td>
<td></td>
<td>435</td>
</tr>
<tr>
<td>9</td>
<td>D4</td>
<td>317</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>D0~D5D2r</td>
<td></td>
<td>281</td>
</tr>
</tbody>
</table>

2 code fragment was very interesting, (see Figure 4.7 and Table 4.2) as it acts as a specialized ‘don’t care’ rule. It is only not true in two situations: (1) when $D0 = D1 = 0$ so $D3$ and $D5$ are unimportant, and (2) when $D3 = 0$ and $D3$ is addressed. It is to be noted that $D3$ is addressed when $D0 = 0$ and $D1 = 1$. This code fragment effectively acts as a mask, indicating that a rule containing it should not cover this particular situation. This is different to many standard representations that can only indicate which situations they do cover.

![Figure 4.7](image)  
Figure 4.7: A code-fragment mask for not considering states when $D3$ is 0 and $D3$ is addressed.

\[^3\]If $D0 = D1 = 0$ then only $D2$ is important because the address ‘00’ indexes to the data bit $D2$.\]
Table 4.2: Truth table for a code fragment mask for not considering states when D3 is 0 and D3 is addressed.

<table>
<thead>
<tr>
<th>D0</th>
<th>D1</th>
<th>D3</th>
<th>D5</th>
<th>D3D0r</th>
<th>D1D5</th>
<th>D3D0rD1D5d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The field of LCS claims that transparent learning and cooperation among rules is a strength of the technique [22]. Considering the rule shown in Figure 4.8, cooperation is needed amongst both code fragments and other rules, e.g. this rule covers the case when D3 = 0 and is addressed. Although still human readable, the code fragment based rules currently require more interpretation than standard, say ternary encoded, rules.

It is worth considering why the simple code fragments available to XCSCFC were not retained, see Table 4.3. Address bits D0 and D1 were not used often in isolation, presumably due to the need to combine with other data-bit features to be meaningful. Considering this insight, D5 was relatively lowly ranked, but this was likely due to the ‘true’ state being favored when matching, i.e. D0 = D1 = 1 = true ⇒ addresses D5. The negation of D0, D1 and D5 probably did not get retained for similar reasons. Curiously, the system effectively created D0 twice, see rank 19, 34, highlighting the problem with GP like encoding in terms of multiple genotypes to a single phenotype hindering both interpretation and functionality.
4.2. XCS WITH CODE-FRAGMENT CONDITIONS (A SIMPLE APPROACH)

Figure 4.8: A classifier rule complementing a mask code-fragment. Consider that D0=0 and D1=1 to interpret this rule.

Finally, as the address bits function together, the system has autonomously isolated them. Table 4.4 shows all the retained code fragments using just the address bits, which are few and ranked lowly. Address bits in isolation are meaningless, so they need to be combined with the data-bits as in the highly ranked code fragments.

Building Blocks Reuse

A classifier rule from the final rule base obtained for a typical run of the 11-bit multiplexer problem is depicted in Figure 4.9, along with the code fragments being used by the classifier. It is to be noted that only specific code
fragments in the condition are shown, the nine ‘don’t care’ code fragments occurring in the condition are not shown to save space. This is a compact rule, using just two code fragments from the population of 11MUX (i.e. Level 2) code fragments, namely L2.2 and L2.7. These 11MUX code fragments are using two building blocks of knowledge, in the form of code fragments, from 6MUX (i.e. Level 1), namely L1.20 and L1.19.

In XCSCFC, a classifier rule is said to be matched against a problem instance if the computed value of all the code fragments in the classifier’s condition is equal to 1. The two code fragments being used from 6MUX by the classifier rule shown in Figure 4.9, contain important knowledge.
4.2. XCS WITH CODE-FRAGMENT CONDITIONS (A SIMPLE APPROACH)

L1_20 outputs 1 if and only if D0 = 0 and D2 = 1, and L1_19 outputs 1 if and only if D1 = 0 and D4 = 0. This extracted knowledge is effectively used by the 11MUX code fragments. L2_2 outputs 1 if D6 = 1 and the value returned by L1_20 is also 1. Therefore, L2_2 will output 1 when D0 = 0, D2 = 1, and D6 = 1. The L2_7 is an interesting building block. It will output 1 when L1_19 outputs 0, and vice versa. Now L1_19 outputs 1 when both D1, and D4 are equal to 0, therefore L2_7 will output 1 in all cases except D1, and D4 being both equal to 0.

Therefore, the classifier rule ‘L2_2 L2_7 : 1’ will match all the problem instances having features D0 = 0, D2 = 1, D6 = 1, and any pair of values for features D1 and D4 except 00. This classifier is equivalent to two classifiers, represented in ternary alphabet based form: ‘011###1#### : 1’, and ‘001#1#1#### : 1’. The first classifier rule is maximally general and accurate, and the second rule is also accurate.

4.2.4 Summary of Simple Approach to XCSCFC

Building blocks of knowledge were successfully extracted from small-scale problems and reused to learn large-scale problems in the multiplexer and even-parity problem domains. The XCSCFC approach has outperformed standard XCS in terms of reducing the required number of training instances to learn large-scale multiplexer and even-parity problems.
This initial investigation of code fragments in XCS show that the multiple genotypes to a single phenotype issue in feature rich encoding schemes disables the subsumption deletion function resulting in larger population of classifiers and increased execution time as compared with standard XCS. The additional methods and increased search space leads to much longer training times. This is compensated by the identification of code fragments containing useful knowledge, such as the importance of the address bits in the multiplexer problem. Code fragments that linked the address to data bits were preferred over simpler representations as they captured important information. Code fragments in XCS can also autonomously determine which situations their associated rules cover and uniquely do not cover.

The current implementation method of XCS with code-fragment conditions uses static code fragments, extracted from one level to seed a population of code fragments in the next level, in a hierarchical fashion. The order and amount of code fragments to select has been empirically set, so further optimization may be possible. A mechanism is needed to introduce plausibly better code fragments as training progresses, without disrupting existing classifiers, so that the limit on code fragment numbers is removed. An enhanced version of XCSCFC implementation is presented in the next section that copes with some of the limitations in the simple XCSCFC approach.

4.3 XCS with Code-Fragment Conditions
(An Improved Approach)

4.3.1 Method
In the improved implementation of XCSCFC, the condition bit in a classifier is directly replaced with a code fragment instead of addressing a separate population of code-fragments, which is no longer used. There-
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

Therefore, there is no limit on the number of available code fragments, except in the number of rules in the classifiers population. The system is allowed to reuse the extracted code fragments from all previous levels, instead of just one level. The number of code fragments to be reused from a certain level is governed by the unique code fragments in good classifiers, i.e. equal to the number of distinct code fragments in the conditions of accurate and experienced classifiers in the final population with a fitness value greater than the average fitness of the classifier population. A population of classifiers having code-fragment conditions is illustrated in Figure 4.10.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0D0~</td>
<td>D0D5d~</td>
</tr>
<tr>
<td>D0D0~</td>
<td>D0D0~</td>
</tr>
<tr>
<td>D0D0~</td>
<td>D0D5d~</td>
</tr>
<tr>
<td>D3D0rD5D1d&amp;</td>
<td>D0D0~</td>
</tr>
<tr>
<td>D0D1dD0D4d&amp;</td>
<td>D0D0~</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4.10: A population of classifiers using code-fragment conditions. Here ‘&’, ‘|’, ‘d’, ‘∼’, and ‘r’ denote AND, OR, NAND, NOT, and NOR operators respectively. The code-fragment conditions are shown in postfix form.

The improved XCSCFC system enhances the simple XCSCFC approach, described in Section 4.2.1, in the following aspects: the covering operation, the rule discovery operation, and the mechanism to reuse the extracted domain knowledge. From this point onward, XCSCFC refers to the improved implementation method.

**Covering Operation**

Covering occurs if an action is missing in the match set $[M]$. In the covering operation, a random classifier is created whose condition matches the current environmental state $s$ and contains ‘don’t care’ code fragments with probability $P_{don'tCare}$. All the ‘non-don’t care’ code fragments in this newly created classifier must output 1 against the observed state $s$. The
covering operation is described in Algorithm 7 (cf. Algorithm 2 for differences to simple approach). Here \( n \) is the length of condition \( \text{cond} \) in a classifier rule, and \( P_{\text{don't care}} \) is the probability of ‘don’t care’ code fragment in condition of the newly created classifier in the covering operation.

**Algorithm 7: XCSCFC\textsubscript{efficient}: Covering Operation**

**Data:** The currently observed input state \( s \) and an action \( a \) missing in the match set \([M]\).

**Result:** A newly created classifier \( cl \) matching the state \( s \) and advocating the action \( a \).

1. initialize classifier \( cl \)
2. initialize condition \( cl.\text{cond} \) with length \( n \)
3. **for** \( i = 1 \) to \( n \) **do**
   4. if \( \text{RandomNumber}(0, 1) < P_{\text{don't care}} \) then
      5. \( cl.\text{cond}[i] \leftarrow \text{‘don’t care’ code fragment} \)
   6. else
      7. initialize value \( \text{val} \) to 0
   8. **while** \( \text{val} \neq 1 \) **do**
      9. \( cf \leftarrow \text{randomly create code fragment} \)
      10. **load** terminal symbols in \( cf \) with corresponding binary bits from the state \( s \)
      11. \( \text{val} \leftarrow \text{evaluate value of } cf \)
   12. **end**
   13. \( cl.\text{cond}[i] \leftarrow cf \)
   14. **end**
5. \( cl.\text{action} \leftarrow a \)
6. **return** \( cl \)
Rule Discovery Operation

In the rule discovery operation, two offspring are produced by applying the GA in the action set $[A]$. First of all, two parent classifiers are selected from $[A]$ based on fitness and the offspring are created out of them. Next, the conditions of the offspring are crossed with probability $\chi$ using a two-point crossover operation, treating each code fragment as a single allele similar to a bit symbol in ternary alphabet based conditions. The crossover operation is described in Algorithm 8. Here $n$ is the length of condition $\text{cond}$ in a classifier rule.

**Algorithm 8: XCSCF_{efficient}: Crossover Operation**

Data: Two offspring classifiers $cl_1$ and $cl_2$.

Result: The crossed over classifiers $cl_1$ and $cl_2$.

1. $x \leftarrow \text{RandomNumber}[0, n]$
2. $y \leftarrow \text{RandomNumber}[0, n]$
3. if $x > y$ then
   4. swap $x$ and $y$
5. end
6. for $i = x$ to $y$ do
   7. swap $cl_1.\text{cond}[i]$ and $cl_2.\text{cond}[i]$
8. end
9. return $cl_1$ and $cl_2$

After that, each code fragment in the conditions of the crossed over children is mutated with probability $\mu$, such that both children match the currently observed state $s$. In the mutation operation, a ‘non-don’t care’ code fragment is replaced by a ‘don’t care’ code fragment, and a ‘don’t care’ code fragment is replaced by a randomly generated ‘non-don’t care’ code fragment that outputs 1 against the state $s$. Then, the actions of the children are mutated with probability $\mu$. The mutation operation is described in Algorithm 9 (cf. Algorithm 4 for differences to simple ap-
Here $n$ is the length of condition $cond$ in a classifier rule, and $\mu$ is the mutation probability.

---

**Algorithm 9: XCSCFC${}_{\text{efficient}}$: Mutation Operation**

Data: The currently observed input state $s$ and an offspring classifier $cl$ matching the state $s$.

Result: The mutated classifier $cl$ which still matches the state $s$.

1. for $i = 1$ to $n$ do
2.   if $\text{RandomNumber}(0, 1) < \mu$ then
3.     if $cl.cond[i] = \text{‘don’t care’ code fragment}$ then
4.       initialize value $val$ to 0
5.       while $val \neq 1$ do
6.         $cf \leftarrow$ randomly create code fragment
7.         load terminal symbols in $cf$ with corresponding binary bits from the state $s$
8.         $val \leftarrow$ evaluate value of $cf$
9.       $cl.cond[i] \leftarrow cf$
10.      else
11.     $cl.cond[i] \leftarrow \text{‘don’t care’ code fragment}$
12.     end
13.  end
14.  end
15. if $\text{RandomNumber}(0, 1) < \mu$ then
16.   $a \leftarrow cl.action$
17.   $cl.action \leftarrow$ randomly chosen action other than $a$
18. end
19. return $cl$

The prediction of the offspring is set to the average of the parents’ values whereas the prediction error and the fitness of the offspring are set to
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

the average of the parents’ values reduced by constants predictionErrorReduction and fitnessReduction respectively, as in [21].

Reusing Extracted Knowledge

The fitter code fragments, i.e. building blocks of information, from smaller problems, are used to create the code fragments in a higher level problem of the same domain. The code fragments in the conditions of accurate and experienced classifiers in the final population, with a fitness value greater than the average fitness of the classifier population, are taken as the fitter code fragments (cf. Equation 4.1 for differences to simple approach) and reused to learn the higher level complex problems in the domain. The code fragments from smaller problems are used as terminals in the code fragments of a higher level problem. The probability of a terminal to be a code fragment from previous levels or a condition bit from the current level problem is set to 0.5 following the ramped half and half approach of initializing a population in GP [9].

An example of code fragments in the 20-bit MUX problem is shown in Figure 4.11. The code fragments in the 20-bit MUX problem contain fitter code fragments from the 6-bit MUX problem and the 11-bit MUX problem, similarly the code fragments in the 11-bit MUX problem contain code fragments from the 6-bit MUX problem. The code fragments in the 6-, 11-, and 20-bit MUX problems are named as L1,n, L2,n, and L3,n respectively, where n = 0, 1, 2 ... etc.

4.3.2 Results

In order to test the performance of improved XCSCFC, the results have been compared with the base techniques, i.e. standard XCS and related layered learning GP-systems. All results obtained in this work are the average of the 30 independent runs. In all graphs presented here, the X-axis is the number of problem instances used as training examples and the
Figure 4.11: A sample of code fragments in the 20-bit multiplexer problem. The code fragments in the 6-, 11-, and 20-bit multiplexer problems are named as L1_n, L2_n, and L3_n respectively, where n = 0, 1, 2 ... etc.

Y-axis is the classification performance measured as the moving average over the last 1000 exploit problem instances.

Results Comparison with XCS

The Multiplexer Problem Domain  The performance of standard XCS and XCSCFC in the multiplexer problem domain is shown in Figure 4.12. The number of classifiers used, denoted by N, is 500, 1000, 2000, 5000, 10000, and 50000 for the 6-, 11-, 20-, 37-, 70-, and 135-bit multiplexer problems respectively. The number of training examples used is half a million for the 6-, 11-, and 20-bit multiplexer and one million, two million, and five million for the 37-, 70-, and 135-bit multiplexer problems respectively. Standard XCS was not able to solve the 37-bit MUX problem with $P_{don't Care} = 0.33$ and $N = 5000$,\(^4\) so $P_{don't Care}$ was increased to 0.5 in Figure 4.12(d). For the 70-bit and the 135-bit MUX problems, $P_{don't Care}$ is set to 1.0 and $\mu$ is set to 0.01 in standard XCS as commonly used in the literature.

\(^4\)In simple problems the conventional parameter values set produces robust performances, but requires adjustment in complex domains, e.g. 37-bit MUX and above.
The condition length of a classifier rule in XCSCFC is set to $70/2 = 35$, and $135/4 = 33$ for the 70-bit and the 135-bit MUX problems respectively. Standard XCS failed 13 times out of 30 runs to solve the 70-bit MUX problem with $N = 10000$, so $N$ was increased to 20000, Figure 4.12(e). Here $\#p$ and $N$ denote the probability of ‘don’t care’ symbol and the number of classifiers used respectively.

XCSCFC needs more training examples than standard XCS to learn the 6-bit and the 11-bit MUX problems, see Figure 4.12(a) and Figure 4.12(b), but less training examples for the 20-bit MUX problem, as shown in Figure 4.12(c). Standard XCS, with parameter tuning, needs approximately 800k and 3000k problem instances to solve the 37-bit and the 70-bit MUX problems, see Figure 4.12(d) and Figure 4.12(e) respectively. However, XCSCFC takes approximately 200k and 500k problem instances to solve the 37-bit and the 70-bit MUX problems respectively, without parameter tuning. The performance curves for the 70-bit MUX problem using XCSCFC are almost coincident in Figure 4.12(e).

Standard XCS was not able to solve the 135-bit MUX problem, either in the literature or with further parameter tuning conducted here. However, if a stepped reward function is used to guide learning [22] then the state-of-the-art in the field was able to solve the 135-bit MUX problem. XCSCFC, reusing the extracted domain knowledge, successfully solved the standard 135-bit MUX problem taking approximately two million training instances, Figure 4.12(f), without needing a stepped reward. Considering the number of possible instances is $2^{135} \approx 4 \times 10^{40}$, and that XCSCFC takes only $2 \times 10^6$ instances (i.e. sampling only one in $10^{34}$ instances) to be able to solve the problem, this result is remarkable.

The Majority-on Problem Domain The performance of standard XCS and XCSCFC in the majority-on problem domain is shown in Figure 4.13. The number of classifiers used is 500, 1000, and 2000 for the 3-, 5-, and 7-bit majority-on problems respectively. The number of training examples used is half a million.
Figure 4.12: Results of the multiplexer problems using XCS and XCSCFC. The performance curves for the 70-bit MUX problem using XCSCFC are almost coincident in (c).

The complete solution of the majority-on problem domain consists of strongly overlapping classifiers. The overlapping nature of classifiers in the final solution makes it harder to learn the problem. XCSCFC success-
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

Figure 4.13: Results of the majority-on problems using XCS and XCSCFC.

fully learned the 3-, 5-, and 7-bit majority-on problems, whereas standard XCS failed to learn the 5-bit and the 7-bit majority-on problems.

To test statistical significance of XCSCFC compared with standard XCS, the Wilcoxon signed rank test was conducted, see Table 4.5. The values in column two and column three are the average performance values of the last 100 test cases along with the standard deviation. The last column shows the p-value obtained with confidence interval of 95%. The performance improvement of XCSCFC is statistically significant as for both cases the p-value is much less than 0.05.

Table 4.5: The Wilcoxon signed rank test for performance comparison in the majority-on problem domain.

<table>
<thead>
<tr>
<th>Majority-On</th>
<th>XCS</th>
<th>XCSCFC</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-bit</td>
<td>95.17 ± 2.49</td>
<td>100.00 ± 0.00</td>
<td>3.54e−6</td>
</tr>
<tr>
<td>7-bit</td>
<td>94.43 ± 2.65</td>
<td>100.00 ± 0.00</td>
<td>1.66e−6</td>
</tr>
</tbody>
</table>
The Carry Problem Domain  The performance of standard XCS and XCSCFC in the carry problem domain is shown in Figure 4.14. The number of classifiers used is 1000, 2000, 4000, and 6000 for the 2-, 3-, 4-, and 5-bit carry problems respectively. The number of training examples used for the 2-, and 3-bit carry problems is half a million whereas for the 4-, and 5-bit carry problems one million training examples have been used.

The complete solution in the carry problem domain consists of overlapping classifiers, in addition it is a niche imbalance domain, therefore very difficult to learn. Standard XCS was not able to reach consistent 100% performance even for the 2-bit carry problem, see Figure 4.14(a), whereas XCSCFC successfully solved the 2-bit and the 3-bit carry problems. XCSCFC also learned the 4-bit carry problem and in the case of the 5-bit carry problem XCSCFC outperformed standard XCS, albeit not reaching 100% consistent and stabilized performance as shown in Figure 4.14(b).

The results of the Wilcoxon signed rank test conducted to measure the statistical significance of XCSCFC compared with standard XCS are shown in Table 4.6. The values in column two and column three are the average performance values of the last 100 test cases along with the standard deviation. The performance improvement of XCSCFC is statistically significant as for all cases the p-value, obtained with confidence interval of 95%, is far less than 0.05.

Table 4.6: The Wilcoxon signed rank test for performance comparison in the carry problem domain.

<table>
<thead>
<tr>
<th>Carry</th>
<th>XCS</th>
<th>XCSCFC</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-bit</td>
<td>99.17 ± 1.19</td>
<td>100.00 ± 0.00</td>
<td>7.80e⁻³</td>
</tr>
<tr>
<td>3-bit</td>
<td>96.30 ± 2.67</td>
<td>100.00 ± 0.00</td>
<td>1.66e⁻⁶</td>
</tr>
<tr>
<td>4-bit</td>
<td>93.47 ± 2.60</td>
<td>100.00 ± 0.00</td>
<td>1.64e⁻⁶</td>
</tr>
<tr>
<td>5-bit</td>
<td>92.10 ± 3.12</td>
<td>99.87 ± 0.43</td>
<td>1.59e⁻⁶</td>
</tr>
</tbody>
</table>
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

(a) for 2-, and 3-bit problems.

(b) for 4-, and 5-bit problems.

Figure 4.14: Results of the carry problems using XCS and XCSCFC.

The Even-Parity Problem Domain The performance of standard XCS and XCSCFC in the even-parity problem domain is shown in Figure 4.15. The number of classifiers used is 200, 300, 400, 500, 1000, and 2000 for the 2-, 3-, 4-, 5-, 6-, and 7-bit problems respectively. Each run is stopped after half a million training examples.
It is observed that XCSCFC needs more training examples than standard XCS to learn the 2-, 3-, and 4-bit even-parity problems, see Figure 4.15(a). As the problem scales to 6-bit, standard XCS cannot learn the even-parity problem, see Figure 4.15(b), whereas XCSCFC successfully solved up to the 7-bit even-parity problems.

The results of the Wilcoxon signed rank test conducted to measure the statistical significance of XCSCFC with comparison to standard XCS are shown in Table 4.7. The values in column two and column three are the average performance values of the last 100 test cases along with the standard deviation. The performance improvement of XCSCFC is statistically significant as for all the three cases the p-value, obtained with confidence interval of 95%, is far less than 0.05.

Table 4.7: The Wilcoxon signed rank test for performance comparison in the even-parity problem domain.

<table>
<thead>
<tr>
<th>Parity</th>
<th>XCS</th>
<th>XCSCFC</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-bit</td>
<td>93.77 ± 9.79</td>
<td>100.00 ± 0.00</td>
<td>4.88e⁻⁴</td>
</tr>
<tr>
<td>6-bit</td>
<td>79.23 ± 9.26</td>
<td>100.00 ± 0.00</td>
<td>8.07e⁻⁶</td>
</tr>
<tr>
<td>7-bit</td>
<td>76.47 ± 6.18</td>
<td>100.00 ± 0.00</td>
<td>2.49e⁻⁶</td>
</tr>
</tbody>
</table>

If a classifier rule is encoded using the standard ternary alphabet based conditions and the static numeric actions, then the even-parity problem domain does not allow any generalizations. Therefore, each bit must be specific for a classifier rule to be accurate in standard XCS. For small-scale problems, it is relatively easy to learn each bit, so standard XCS successfully learnt up to the 5-bit even-parity problems. As the problem scaled to 6-bit and higher levels, standard XCS was not able to solve them, having typically used XCS parameter settings where probability of ‘don’t care’ symbol and that of mutation was set 0.33 and 0.04 respectively. However, in XCSCFC the number of ‘specific’ code fragments is essentially messy as the system can choose the number of ‘don’t care’ fragments it uses. Also, utilizing code fragments for the matching component of the LCS removes
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

- for 2-, 3-, and 4-bit problems.
- for 5-, 6-, and 7-bit problems.

Figure 4.15: Results of the even-parity problems using XCS and XCSCFC.

the implicit linking between the position of a condition bit in a rule and
the corresponding feature in the problem input. Therefore, the XCSCFC
system, having the ability to generalize, has performed efficiently in the
even-parity problem domain.
For example, consider an experienced, accurate, correct and general classifier rule ‘L1_7 D2 D0D0| : ’1’, taken from the final rule base of the 3-bit (i.e. Level 2) even-parity problem where L1_7 is a Level 1 (i.e. 2-bit even-parity problem) code fragment given by D1D0|D1D0d&. In XC-SCFC, a classifier rule is said to be matched against an environmental input state if the computed value of all the code fragments in the classifier’s condition is equal to 1. Now, L1_7 is equivalent to ‘D1 XOR D0’ that outputs 1 if and only if D0 and D1 have different values and D0D0| is the ‘don’t care’ code fragment that always outputs 1. Therefore, this rule will match an environmental state if D0 and D1 have different values and D2 is equal to 1 (as the second code fragment is just D2 in this rule) in the environmental state. So, this general rule is equivalent to two specific rules: ‘011 : 1’ and ‘101 : 1’. The generalization ability of XCSCFC in the even-parity problems will be further discussed in Section 4.3.3.

Results Comparison with GP Systems

The LCS and GP systems are two different evolutionary techniques that solve a problem in different ways, i.e. LCS is an online reinforcement learning system whereas GP is a supervised learning batch processing approach. The primary aim of the work presented here was not to develop a competitor for the GP systems or other layered learning approaches, and it is not straightforward to compare the proposed system with a GP system. However some attempt at comparison with layered learning GP approaches, based on the total number of training examples, has been made to clarify the benefits of the proposed approach in terms of scalability.

The first comparison is with a layered learning GP system, called LLGP, developed by Jackson and Gibbons [64] using a two-layered approach where the solutions of the bottom layer were encapsulated as parametrized modules and reused to learn the main task in the top layer. They tested the LLGP system on the even-parity problems and the majority-on problems. The 2-bit even-parity problem was used at the bottom layer to solve
the 4-, 5-, and 6-bit even-parity problems having the function set \{AND, OR, NAND, NOR\}. Each experiment was repeated 100 times with maximum 50 generations in each run. The population size used for the 4-bit even-parity problem was 500 and it was increased to 2000 for the 5-, and 6-bit even-parity problems. The layered learning approach outperformed the standard monolithic GP [70] and the GP with ADFs [71], albeit not achieving 100% success rate as shown in Table 4.8.

Table 4.8: Performance of different GP systems, in terms of success rate out of 100 runs, for the even-parity problems [64].

<table>
<thead>
<tr>
<th>Problem</th>
<th>GP</th>
<th>GP with ADFs</th>
<th>LLGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit</td>
<td>14</td>
<td>43</td>
<td>95</td>
</tr>
<tr>
<td>5-bit</td>
<td>0</td>
<td>32</td>
<td>92</td>
</tr>
<tr>
<td>6-bit</td>
<td>0</td>
<td>16</td>
<td>70</td>
</tr>
</tbody>
</table>

To compare XCSCFC with the LLGP system for the even-parity problems, the function set of XCSCFC was changed to \{AND, OR, NAND, NOR\}. The number of classifiers used is 200, 300, 400, 500, and 1000 for the 2-, 3-, 4-, 5-, and 6-bit even-parity problems respectively. The number of training examples used is half a million. The performance of XCSCFC for the 2-bit to 6-bit even-parity problems is shown in Figure 4.16. XCSCFC solved all these problems successfully in each of the 30 conducted experiments.

The second comparison is with the LLGP system for the majority-on problems. In LLGP, the 3-bit majority-on problem was used at the bottom layer to solve the 5-, and 7-bit majority-on problems having the function set \{AND, OR, NOT\}. Each experiment was repeated 100 times with maximum 50 generations in each run. The population size used for the 5-bit majority-on problem was 500 and it was increased to 1000 for the 7-bit majority-on problem. The LLGP system outperformed the standard monolithic GP [70] and the GP with ADFs [71], albeit not achieving 100% success rate for the 7-bit majority-on problem as shown in Table 4.9.
Figure 4.16: Results of the even-parity problems obtained using XCSCFC with the function set \{AND, OR, NAND, NOR\}.

Table 4.9: Performance of different GP systems, in terms of success rate out of 100 runs, for the majority-on problems [64].

<table>
<thead>
<tr>
<th>Problem</th>
<th>GP</th>
<th>GP with ADFs</th>
<th>LLGP</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-bit</td>
<td>62</td>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>7-bit</td>
<td>18</td>
<td>not attempted</td>
<td>90</td>
</tr>
</tbody>
</table>

To compare XCSCFC with LLGP for the majority-on problems, the function set of XCSCFC was changed to \{AND, OR, NOT\}. The number of classifiers used is 500, 1000, and 2000 for the 3-, 5-, and 7-bit majority-on problems respectively. The number of training examples used is half a million. The performance of XCSCFC for the 3-, 5-, and 7-bit majority-on problems is shown in Figure 4.17. XCSCFC solved successfully all these problems in each of the 30 conducted experiments.

The third comparison is with the DTAG3P system developed by Hoang et al. [47]. They tested the DTAG3P system on the 8-bit even-parity problem, in a layered learning fashion, using the function set \{AND, OR, NOT, XOR\}, the population size \(max_{pop} = 250\), and the number of maximum generations at each problem level \(max_{gen} = 101\). Although DTAG3P out-
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

Figure 4.17: Results of the majority-on problems obtained using XCSCFC with the function set \{AND, OR, NOT\}.

performed the two single-short learning GP systems, the standard tree-GP system [70] and the TAG3P system [46], it could not achieve 100% success rate for the 8-bit even-parity problem. The reported success rates are 6.67%, 10%, and 86.67% for the standard tree-GP, the TAG3P, and the DTAG3P systems, respectively [47].

To compare XCSCFC with DTAG3P, the function set of XCSCFC was changed to \{AND, OR, NOT, XOR\}. The number of classifiers used is 200, 300, 400, 500, 1000, 1500, and 2000 for the 2-, 3-, 4-, 5-, 6-, 7-, and 8-bit even-parity problems respectively. The number of training examples used is half a million. The performance of XCSCFC for the 2-bit to 8-bit even-parity problems is shown in Figure 4.18. XCSCFC solved successfully all these problems in each of the 30 conducted experiments.

Poli and Page [97] have developed a monolithic GP system by using smooth uniform crossover, sub-machine code GP, and distributed demes to solve higher-order even-parity problems. It is reported that the 12-, 13-, 15-, 17-, 20- and 22-bit even-parity problems were solved successfully, but they have used all the 16 Boolean operators of two variables [89] in
CHAPTER 4. REUSING BLOCKS OF EXTRACTED KNOWLEDGE

Figure 4.18: Results of the even-parity problems obtained using XCSCFC with the function set \{AND, OR, NOT, XOR\}.

the function set. As XCSCFC is a hierarchical approach where length of the nested tree structures grows with every next level problem in the domain, resulting in increased search space, it is impractical to use the 16 Boolean functions and apply XCSCFC from 2-bit to 22-bit even-parity problems. Therefore, we adopted the code-fragment approach in the action of a classifier rule, see Chapter 5, and then further extended it using cyclic graphs, see Chapter 6, to encapsulate repeated patterns in the problem and evolved solutions for any \(n\)-bit even-parity problem.

4.3.3 Interpretation of Results

The XCSCFC system has solved up to and including the 135-bit multiplexer problems by extracting and reusing the building blocks of domain knowledge. The reuse of extracted knowledge has shown generalization ability in the even-parity domain problems that is not possible using the standard ternary alphabet based representation. The following subsection describes in detail the reuse of the extracted knowledge in the multiplexer and even-parity problem domains. This is followed by a discussion of messy code-fragment conditions.
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH) 85

Reuse of Extracted Knowledge

A classifier rule from the final rule base of the 20-bit multiplexer problem is depicted in Figure 4.19, along with the code fragments being used by the classifier. Here \( A \) and \( p \) represent action and prediction of the classifier, respectively. It is to be noted that only specific code fragments in the condition are shown, the 16 ‘don’t care’ code fragments occurring in the condition are not shown to save space. This is a compact rule, using just four code fragments. These code fragments in the 20-bit MUX are using three building blocks of knowledge, in the form of code fragments, from the 6-bit MUX (i.e. Level 1), namely \( L_{129}, L_{112} \) and \( L_{121} \), and one from the 11-bit MUX (i.e. Level 2), namely \( L_{23} \) that is further using a code fragment from the 6-bit MUX, namely \( L_{16} \).

<table>
<thead>
<tr>
<th>Condition</th>
<th>( A )</th>
<th>( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_{129} D_1 d D_4 D_2</td>
<td>d )</td>
<td>( L_{112} D_0 d \sim )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiplexer</th>
<th>Code Fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Expression</td>
</tr>
<tr>
<td>( 11 \text{MUX} ) (Level 2)</td>
<td>( L_{23} )</td>
</tr>
<tr>
<td></td>
<td>( L_{16} D_9</td>
</tr>
<tr>
<td>( 6 \text{MUX} ) (Level 1)</td>
<td>( L_{16} )</td>
</tr>
<tr>
<td></td>
<td>( D_0 )</td>
</tr>
<tr>
<td>( L_{112} )</td>
<td>( D_3 D_1 &amp; \sim )</td>
</tr>
<tr>
<td>( L_{121} )</td>
<td>( D_2 )</td>
</tr>
<tr>
<td>( L_{129} )</td>
<td>( D_1 D_2 &amp; \sim )</td>
</tr>
</tbody>
</table>

Figure 4.19: A classifier rule from the final rule base obtained for a typical run of the 20-bit multiplexer problem.

In XCSCFC, a classifier rule is said to be matched against a problem instance if the computed values of all the code fragments in the classifier’s
condition are equal to 1. The fourth code fragment ‘L1.21’ in the classifier rule shown in Figure 4.19 is just \( D2 \), therefore, \( D2 \) must be 1 in the environmental instance to be matched by this classifier.

The first code fragment ‘L1.29 D1 d D4 D2 | d’ is using three environmental features, i.e. \( D1 \), \( D2 \), and \( D4 \). Now, \( D2 \) must be 1 if the environmental instance is to be matched by this classifier, so the code fragment ‘L1.29 D1 d D4 D2 | d’ will output 1 if and only if the value of the feature \( D1 \) is 1, as shown in Table 4.10. Note: In all the tables shown in this thesis “Sr. No.” denotes a row number in the table.

### Table 4.10: Truth table for the code fragment ‘L1.29 D1 d D4 D2 | d’, where ‘L1.29’ is ‘D1 D2’.

| Sr. No. | D1 | D2 | D4 | L1.29 | L1.29D1d | L1.29D1dD4D2|d |
|---------|----|----|----|-------|-----------|----------------|
| 1       | 0  | 1  | 0  | 1     | 1         | 1              | 0      |
| 2       | 0  | 1  | 1  | 1     | 1         | 1              | 0      |
| 3       | 1  | 1  | 0  | 1     | 0         | 1              | 1      |
| 4       | 1  | 1  | 1  | 1     | 0         | 1              | 1      |

The second code fragment ‘L1.12 D0 d ~’ is using three environmental features, i.e. \( D0 \), \( D1 \), and \( D3 \). Now, \( D1 \) must be 1 if the environmental instance is to be matched by this classifier, so the code fragment ‘L1.12 D0 d ~’ will output 1 if and only if \( D0 = 1 \) and \( D3 = 0 \), as shown in Table 4.11.

### Table 4.11: Truth table for the code fragment ‘L1.12 D0 d ~’, where ‘L1.12’ is ‘D3 D1 & ~’.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>D0</th>
<th>D1</th>
<th>D3</th>
<th>L1.12</th>
<th>L1.12D0d</th>
<th>L1.12D0d~</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The third code fragment ‘L2.3 D18 & ~’ = ‘D0 D9 | D1 & D18 & ~’ is using four environmental features, i.e. \( D0 \), \( D1 \), \( D9 \), and \( D18 \). Now, \( D0 \)
4.3. XCS WITH CODE FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

and $D_1$ must be 1 if the environmental instance is to be matched by this classifier, so the code fragment ‘$L_2.3 D_18 \& \sim$’ will output 1 if and only if $D_{18} = 0$, as shown in Table 4.12.

Table 4.12: Truth table for the code fragment ‘$L_2.3 D_{18} \& \sim$’, where ‘$L_2.3$’ is ‘$L_1.6 D_9 | D_1 \&$’ and ‘$L_1.6$’ is ‘$D_0$’.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$D_9$</th>
<th>$D_{18}$</th>
<th>$X = D_0 D_9$</th>
<th>$Y = XD_1 &amp;$</th>
<th>$Z = YD_{18} &amp;$</th>
<th>$Z \sim$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Therefore, the classifier rule ‘$L_1.29 D_1 d D_4 D_2 | d L_1.12 D_0 d \sim L_2.3 D_{18} \sim L_1.21: 1$’ will match all the problem instances having features $D_0 = 1$, $D_1 = 1$, $D_2 = 1$, $D_3 = 0$, and $D_{18} = 0$. This classifier is maximally general and accurate, being equivalent to the classifier ‘1110########## ####0# : 0’ represented in ternary alphabet based form.

To illustrate the generalization ability of XCSFC in the even-parity problems, a classifier rule from the final rule base of the 4-bit even-parity problem, depicted in Figure 4.20, is analyzed. Here $A$ and $p$ represent action and prediction of the classifier, respectively. It is to be noted that only specific code fragments in the condition are shown, the two ‘don’t care’ code fragments occurring in the condition are not shown to save space. These code fragments in the 4-bit even-parity problem (i.e. 4EP) are using a code fragment from the 3EP (i.e. Level 2), namely $L_2.4$ that is further using a code fragment from the 2EP (i.e. Level 1), namely $L_1.7$.

The first code fragment ‘$D_3 \sim$’ in the classifier rule shown in Figure 4.20 is just negation of $D_3$, therefore, $D_3$ must be 0 in the environmental instance to be matched by this classifier.

The second code fragment ‘$L_2.4’ = ‘D_2 L_1.7 | \sim = ‘D_2 D_1 D_0 | D_1 D_0 d & | \sim’ uses three environmental features, i.e. $D_0$, $D_1$, and $D_2$. The code
Figure 4.20: A classifier rule from the final rule base obtained for a typical run of the 4-bit even-parity problem.

fragment ‘L1_7’ is equivalent to ‘D1 XOR D0’ that outputs 1 if and only if $D_0$ and $D_1$ have different values, so the code fragment ‘L2_4’ will output 1 if and only if $D_2 = 0$ and $D_0 = D_1$, as shown in Table 4.13. The ability to consider the features’ property, such as $D_0 = D_1$, is not expressible in the ternary alphabet based representation. Therefore, the classifier rule ‘D3~ L2_4: 1’ will match all the problem instances having features $D_0 = D_1$, $D_2 = 0$, and $D_3 = 0$. This general classifier is equivalent to two specific classifiers ‘0000 : 1’ and ‘1100 : 1’.

Consider another general and interesting classifier rule ‘D2D2&L1_4| D0D0~| D2L1_7d : 0’, taken from the final rule base of the 3-bit even-parity problem. In this classifier rule ‘L1_4’ and ‘L1_7’ are the code fragments from the 2-bit even-parity problem given by ‘D0D0rD1D1&r’ and ‘D1D0|D1D0d&’ respectively. The code fragment ‘L1_4’ outputs 1 if and only if $D_0$ is 1 and $D_1$ is 0. To determine the subset of environmental instances being matched by this rule, consider the truth tables for the code fragments ‘D2D2&L1_4’ and ‘D2L1_7d’ shown in Table 4.14 and Table 4.15.
4.3. XCS WITH CODE-FRAGMENT CONDITIONS (AN IMPROVED APPROACH)

Table 4.13: Truth table for the code fragment ‘L2.4’, where ‘L2.4’ is ‘D2 L1.7 | ~’ and ‘L1.7’ is ‘D1 D0 | D1 D0 d &’ = ‘D1 XOR D0’.

| Sr. No. | D0 | D1 | D2 | L1.7 | D2L1.7 | D2L1.7|~
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

respectively. This rule will match against an environmental instance if the output values for both code fragments ‘D2D2&L1.4|’ and ‘D2L1.7d’ are equal to 1. Therefore, the instances numbered 2, 5, and 8 in Table 4.14 and Table 4.15 constitute the matching subset of environmental instances for this rule. So, this general rule is equivalent to three specific rules: ‘001 : 0’, ‘100 : 0’, and ‘111 : 0’.

Messy Code-Fragment Conditions

In XCSCFC, there is no linking between the position of a condition bit in a classifier rule and the corresponding feature in the environmental input state. Therefore, it is not necessary to use the same number of code fragments in a classifier’s condition as the number of problem features. For example, different numbers of code fragments can be used to learn the 6-bit multiplexer problem as shown in Figure 4.21. I used up to six code-fragments to learn the 6-bit multiplexer problem as it has six attributes. It is to be noted that more than six code-fragments can be used to solve

---

5 It is to be noted that the code fragment ‘D0D0~’ in the classifier being analyzed here is the ‘don’t care’ code fragment.
Table 4.14: Truth table for the code fragment ‘D2 D2 & L1\_4 ’, where ‘L1\_4’ outputs 1 if and only if D0 is 1 and D1 is 0.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D2D2&amp;</th>
<th>L1_4</th>
<th>D2D2&amp;L1_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.15: Truth table for the code fragment ‘D2 L1\_7 d’, where ‘L1\_7’ outputs 1 if and only if D0 and D1 have different values.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>L1_7</th>
<th>D2L1_7d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

the 6-bit multiplexer problem, but there is no obvious benefit of using the number of code-fragments more than the problem size.

It is observed that the 6-bit multiplexer problem can be solved using different numbers of code fragments in a classifier’s condition, but to solve it effectively a minimum of three code fragments should be used. If more than the minimum required code fragments are used, then the performance is found to be more robust (faster) for the 6-bit multiplexer problem.
problem, but if too many code fragments are used it will increase the computational time. The minimum number of code fragments needed in any problem in a domain is not optimized currently.

4.4 Chapter Summary

Building blocks of knowledge were successfully extracted from small-scale problems and reused to learn more complex, large-scale problems in the domain. For example, in the 135-bit multiplexer problem, where the number of possible instances is $2^{135} \approx 4 \times 10^{40}$, XCSCFC takes only $2 \times 10^6$ instances (i.e. sampling only one in $10^{34}$ instances) to successfully solve the problem.

The XCSCFC system, using a GP-like rich encoding scheme, has shown the generalization ability in the even-parity domain problems that is not expressible using the standard ternary alphabet-based representation.

XCSCFC readily solves problems of a scale that existing classifier system and genetic programming approaches cannot, e.g. the 135-bit MUX
problem. However, the results obtained cannot be proved to be general due to the large tree structures.

Although new code fragments were introduced in the system by the mutation operation, in the current implementation of XCSCFC the genetic operations of crossover and mutation were not performed at the code-fragment levels. By evolving the code-fragments in the training process, the performance and/or scalability of XCSCFC may be improved, but this hierarchical approach will eventually hit a limit in the problem size because length of the nested tree structures grows with every next level problem in the domain, resulting in increased search space and demanding more space and computational time. The next chapter introduces code-fragments in the action of a classifier rule in an attempt to produce scalable classifier system by evolving more generalized rules compared with standard XCS.
Chapter 5

Learning Complex, Overlapping and Niche Imbalance Boolean Problems Using XCS-Based Classifier Systems

5.1 Introduction

Previously code-fragment based XCS were introduced, which were more than simply a representation (e.g. ternary to real alphabets). The messy nature, disassociation of message to condition order, masking, feature construction, and reuse of extracted knowledge added additional abilities to the XCS family of LCSs. Hard problems were sought to test the new code-fragment based approaches. Recently, Ioannides et al. [55] experimentally showed that XCS was not able to learn digital design verification (DV) problems, which are Boolean problems where the complete solution set consists of overlapping classifiers. In addition, they are niche imbalance problems, i.e. the niches covered by the classifiers in the complete solution have different sizes (see Section 3.2). Ioannides et al. [54] suggested two
techniques to improve XCS performance in DV problems, but could not reach 100% performance level.

In Chapter 4, a hierarchical approach was developed, using code-fragment conditions in XCS (XCSCFC), which rapidly solved problems of a scale that existing LCS and GP techniques cannot, e.g. the 135-bit multiplexer problem. XCSCFC is specially beneficial if the target problem can be divided into a hierarchy of sub-problems and each of them is solvable in a bottom-up fashion. However, if the hierarchy of sub-problems is too long, then XCSCFC will become impractical, due to the needed computational time, and eventually hit a limit in problem size.

In this chapter, the action of a classifier in XCS is encoded using the code-fragment scheme, in an attempt to develop a scalable system, called XCSCFA, by evolving more generalized rules than standard XCS. The aim of the work presented in this chapter is to investigate the standard XCS and the code-fragment based XCSCFA systems in learning different complex Boolean problems, especially overlapping and niche imbalance problems. The experimental results indicate that to learn overlapping and niche imbalance problems using XCS, it is beneficial to either deactivate action set subsumption or use a relatively high subsumption threshold and a small error threshold. The XCSCFA approach successfully solved the tested overlapping and niche imbalance problems without parameter tuning, because of the inconsistent actions and especially the redundancy provided by the code-fragment actions. The major contribution of the work presented in this chapter is overcoming the identified problem in the wide-spread XCS technique.

The rest of the chapter is organised as follows. In Section 5.2 the development of XCSCFA is detailed. In Section 5.3 experimental results are presented and compared with standard XCS as the most appropriate benchmark algorithm. Section 5.4 is an analysis of the evolved classifier rules. In Section 5.5 the capabilities of standard XCS in learning overlapping and niche imbalance problems are further investigated using different exper-
5.2. XCS WITH CODE-FRAGMENT ACTIONS

Experimental settings. Section 5.6 describes the effect of action inconsistency and redundancy in the XCSCF A approach. Section 5.7 is the explanation how XCSCF A evolved optimum solutions for the multiplexer domain. The ending section concludes the work presented in this chapter and outlines the work to be presented in the next chapter.

5.2 XCS with Code-Fragment Actions

In XCSCF A, the typically used numeric action is replaced by a GP-tree like code fragment [59]. Each code fragment is a binary tree and to limit the tree size a code fragment can have a maximum of seven nodes. The function set for the tree is problem dependent such as \{AND, OR, NOT \ldots\} for binary classification problems and \{+, -, \ast, /\ldots\} for symbolic regression problems. The terminal set is \{D0, D1, D2, … Dn-1\} where \(n\) is the length of an environmental input message. A population of classifiers having code-fragment actions is illustrated in Figure 5.1. The symbols & , | , ∼, d, and r denote AND, OR, NOT, NAND, and NOR operators respectively. The code-fragment trees are shown in postfix form.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>D1</td>
<td>D2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>#</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.1: Classifier population using code-fragment actions.

The XCSCF A approach extends standard XCS [28] in the following aspects: the action value, the covering operation, the rule discovery opera-
CHAPTER 5. LEARNING COMPLEX BOOLEAN PROBLEMS

There are two ways to compute the action value of a classifier in XCSCFA: (1) by loading the terminal symbols in the action tree with the corresponding binary values from the condition in the classifier rule, and (2) by loading the terminal symbols with the corresponding binary values from the environmental input. For sake of readability the two implementations of XCSCFA are named XCSCFA$_c$ (i.e. classifier based) and XCSCFA$_e$ (i.e. environment based). In XCSCFA$_c$ the action value is independent of the input, as it is in standard XCS, whereas in XCSCFA$_e$ the action value depends upon the input matched by the classifier condition.

To compute the action value of a classifier rule in XCSCFA$_c$, a ‘don’t care’ symbol in the condition of the classifier will be randomly treated as 0 or 1. For example, consider the classifier rule shown in Figure 5.2 and the environmental input message ‘110010’. In the condition of this classifier rule, D2 is a ‘#’ symbol. To compute the action value of this classifier rule in XCSCFA$_c$, D2 will be loaded with 0 or 1 randomly. Now, if the binary value taken for D2 is 0 then the action will be 0, otherwise 1. In XCSCFA$_e$, the computed action value of this classifier rule for the environmental input message ‘110010’ will be 0.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>D1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5.2: A classifier with code-fragment action.
It is to be noted that the action value of a classifier in XCS$^A_c$ may vary, even for the same environmental instance, at different times during the training process; unlike standard XCS. A classifier rule in XCS$^A_c$ can have different action values for different environmental instances, but for the same instance it will have the same action value every time, unlike XCS$^A_c$ where this can vary.

### 5.2.2 Covering Operation

Covering occurs if an action $a$ is missing in the match set $[M]$. In the covering operation, a random classifier is created whose condition matches the current environmental state $s$ and contains ‘#’ symbols with probability $P_#$. The code-fragment action tree is randomly generated until its output is $a$. The covering operation is described in Algorithm 10. Here $n$ is the length of condition $cond$ in a classifier rule, and $P_#$ is the probability of the ‘don’t care’ symbol ‘#’ in condition of the newly created classifier in the covering operation.

### 5.2.3 Rule Discovery Operation

In the rule discovery operation, two offspring are created using the following steps. First of all, two parent classifiers are selected from the action set $[A]$ based on fitness and the offspring are created from them. Next, the conditions and action trees of the offspring are crossed with probability $\chi$ by applying GA- and GP-based crossover operations respectively.

Then each symbol in the conditions of the crossed over children is mutated with probability $\mu$, such that both children match the currently observed state $s$. Then, the action trees of the children are mutated with probability $\mu$, using GP-based mutation, by replacing a subtree of the action with a randomly generated subtree of depth up to 1.

The prediction and prediction error of the offspring are set to the average of the parents’ values whereas the fitness of the offspring is set to the
Algorithm 10: XCSCF A: Covering Operation

**Data:** The currently observed input state \( s \) and an action \( a \) missing in the match set \([M]\).

**Result:** A newly created classifier \( cl \) matching the state \( s \) and advocating the action \( a \).

1. initialize classifier \( cl \)
2. initialize condition \( cl\.cond \) with length \( n \)
3. for \( i = 1 \) to \( n \) do
4. if \( \text{RandomNumber}[0, 1) < P_\# \) then
5. \( cl\.cond[i] \leftarrow \# \)
6. else
7. \( cl\.cond[i] \leftarrow s[i] \)
8. end
9. end
10. repeat
11. \( cf \leftarrow \) randomly create code fragment
12. \( val \leftarrow \) evaluate value of \( cf \)
13. until \( val = a \)
14. \( cl\.action \leftarrow cf \)
15. return \( cl \)

average of the parents’ values multiplied by the constant \( \text{fitnessReduction} \), as suggested by Butz and Wilson in [28].

For example, consider the rule discovery operation graphically summarized in Figure 5.3. Figure 5.3(a) shows the action set containing three classifiers with action value 1, formed from the classifier population \([P]\) shown in Figure 5.1, against the environmental input \( s = 001010 \). First of all, two parent classifiers are selected, Figure 5.3(b), from the action set based on fitness \( F \), and two children are created from them, Figure 5.3(c). Then, Figure 5.3(d), the conditions of the reproduced children are crossed over by applying two-point GA-crossover operation at the two marked
points, and the action trees of the children are crossed over by applying GP-crossover operation. The fitness value of crossed over children is set to the average of parents’ fitness values multiplied by 0.1, as suggested by Butz and Wilson in [28]. After that, Figure 5.3(e), the conditions of the crossed over children are mutated by applying GA-mutation such that both children match the currently observed state $s = 001010$, and the action trees of the children are mutated by applying GP-mutation. The final children generated in the rule discovery operation are shown in Figure 5.3(f).

### 5.2.4 Comparing Two Code-Fragment Actions

If a newly created classifier in the rule discovery operation is not subsumed (either by the parents or in the action set) and there is no classifier equal to it in the population, then it will be added to the population. Two classifiers are considered to be equal if and only if both have the same conditions and the genotypically same code fragment in their actions. The code fragment **genotype** is its formal expression as seen in the classifier action, and the **phenotype** is the value that the action computes with a given input. The procedure to compare two code-fragment actions for equality is given in Algorithm 11.

### 5.2.5 Subsumption Deletion

A classifier $cl_1$ can subsume another classifier $cl_2$ if both have the same action and $cl_1$ is accurate, sufficiently experienced, and more general than $cl_2$ [28].

It is to be noted that due to the multiple genotypes to a single phenotype mapping of code-fragment actions in classifier rules, subsumption deletion is less likely to occur in the XCSCFA systems. Subsumption deletion is still made possible by matching the code-fragment actions on a character by character basis.
Figure 5.3: Rule discovery by applying GA- and GP-based operations in an action set.

### 5.3 Results

In order to test the performance of the XCSCFA approach, results have been compared with standard XCS on the six problem domains, i.e., the
Algorithm 11: XCSFA: Are EQUAL_Actions

**Data**: Two code-fragment actions $cf_1$ and $cf_2$.

**Result**: If the actions $cf_1$ and $cf_2$ are genotypically the same, then this algorithm will return `true` otherwise `false`.

```plaintext
1 $m_1 \leftarrow $ Length of code-fragment action $cf_1$
2 $m_2 \leftarrow $ Length of code-fragment action $cf_2$
3 if $m_1 \neq m_2$ then
4         return `false`
5 end
6 for $i = 1$ to $m_1$ do
7       if $cf_1[i] \neq cf_2[i]$ then
8           return `false`
9       end
10 end
11 return `true`
```

multiplexer, the majority-on, the count ones, the digital design verification, the carry, and the even-parity problems. The function set for the code fragments used is \{$\text{AND, OR, NOT, NAND, NOR}$\}, denoted by \{$&, |, \sim, d, r$\}, for all the problem domains experimented in this work. The number of training examples used is two million for all the experiments. Each result reported in this work is the average of 30 independent runs.

In all graphs presented here, the X-axis is the number of problem instances used as training examples, the Y-axis is the classification performance measured as the moving average over the last 1000 exploit problem instances, and the error bars show the standard deviation in the 30 runs. All graphs in this chapter are in color for better readability.
5.3.1 The Multiplexer Problem Domain

The performance of standard XCS and the XCSCFA methods in learning multiplexer problems is shown in Figure 5.4. The number of classifiers used is 500, 1000, 2000, and 6000 for the 6-, 11-, 20-, and 37-bit multiplexer problems respectively. The value of $P_\#$ was increased to 0.5 for the 37-bit multiplexer problem, as both the standard XCS and XCSCFA methods were not able to solve it using $P_\# = 0.33$.

(a) for the 6-bit multiplexer problem. 
(b) for the 11-bit multiplexer problem. 
(c) for the 20-bit multiplexer problem. 
(d) for the 37-bit multiplexer problem.

Figure 5.4: Results of multiplexer problems.

The multiplexer is a niche balanced problem domain and there exists a complete solution that does not contain any overlapping classifier rules. Hence, the standard XCS effectively solved the 6-, 11-, 20-, and 37-bit multiplexer problems. The XCSCFA methods used more training examples, to reach a similar performance level, due to the increased search space.
5.3. RESULTS

5.3.2 The Majority-on Problem Domain

The majority-on problem used in this work is of length \( l = 7 \). The number of classifiers used is \( N = 3000 \). The performance of standard XCS and the XCSCFA methods in learning the 7-bit majority-on problem is shown in Figure 5.5. It is observed that standard XCS reached approximately 94% performance, but could not completely solve the 7-bit majority-on problem, whereas the XCSCFA methods solved it using approximately 20,000 training examples.

![Figure 5.5: Results of the 7-bit majority-on problem.](image)

5.3.3 The Count Ones Problem Domain

The count ones problem used in this work is of length \( l = 20 \) with the first \( k = 7 \) relevant bits. The number of classifiers used is \( N = 4000 \). The performance of standard XCS and the XCSCFA methods is shown in Figure 5.6. It is observed that standard XCS reached approximately 95% performance, but could not completely solve the 7-bit count ones problem, whereas the XCSCFA methods solved it using approximately 50,000 training examples.
5.3.4 The Design Verification Problem Domain

The performance of standard XCS and the XCSCFA methods in learning the DV1 problem is shown in Figure 5.7. The number of classifiers used is \( N = 3000 \). It is observed that standard XCS reached approximately 97% performance level, but could not completely solve the DV1 problem, whereas the XCSCFA methods successfully solved it. Ioannides et al. [54] improved the performance of XCS in learning the DV1 problem to 99.76%, by modifying the standard fitness update procedure and using an individually computed learning rate for each classifier\(^1\), but could not completely solve it.

\(^{1}\)They also used a larger value for action set subsumption threshold, i.e. 100 instead of the commonly used value of 20.
5.3. RESULTS

5.3.5 The Carry Problem Domain

The carry problem used in this work is the 4+4 bit carry problem. The number of classifiers used is \( N = 4000 \). The performance of standard XCS and the XCSCFA methods is shown in Figure 5.8. It is observed that standard XCS reached approximately 93% performance level, but could not completely solve the 4+4 bit carry problem, whereas the XCSCFA methods successfully solved it.

5.3.6 The Even-Parity Problem Domain

The performance of standard XCS and the XCSCFA methods in learning the 7-bit even-parity problem is shown in Figure 5.9. The number of classifiers used is \( N = 2000 \). It is observed that standard XCS and XCSCFA\(_c\) could not learn the 7-bit even-parity problem, whereas XCSCFA\(_c\) successfully solved it.

The even-parity problem domain does not allow generalizations if the standard ternary alphabet based encoding scheme is used with static nu-
Figure 5.8: Results of the 4+4 bit carry problem. The performance curves using the XCSCFA methods are coincident.

Figure 5.9: Results of the 7-bit even-parity problem.

Numerical action. So each bit must be specific for a rule to be accurate, requiring \(2^n\) such rules for the 7-bit even-parity problem. The standard XCS and XCSCFA\(_c\) methods were not able to evolve enough accurate rules us-
5.4 Analysis of Evolved Classifiers

In this section, the evolved rules in different problem domains are analyzed. It is to be noted that from the overlapping and niche imbalance problems only the 7-bit majority-on problem is chosen for rules analysis, and the other problems follow a similar pattern.

5.4.1 The Multiplexer Problem Domain

Every optimal classifier rule in a multiplexer problem, encoded in ternary representation, contains \( k + 1 \) specific bits in the condition (i.e. \( k \) address bits and the corresponding data bit), see Section 3.2, e.g. ‘10##1# : 1’ is an optimal rule for the 6-bit MUX. The solutions obtained using XCS contain both optimal and sub-optimal classifier rules, but often no distinct separation of optimal and sub-optimal classifiers, see Table 5.1. To obtain the desired optimum rule set in binary action based XCS, extensive processing is needed [67], e.g. condensation or compaction algorithm. The XCSCFA\( _e \) system has the advantage of producing the optimal classifiers separated from the sub-optimal ones with respect to the numerosity values, see Table 5.2, that were easily converted to the optimum solutions. The detail is described in Section 5.7.

The XCSCFA\( _e \) method produced more general classifiers than the other two methods containing only \( k \) specific bits in conditions, see Table 5.3. These may appear overgenerals as they have more ‘#’ symbols than the optimal ternary representation classifiers. However, they are perfectly accurate for all the matching environmental instances due to the referencing

---

\(^2\)These methods need larger population size to solve even-parity problems.
Table 5.1: A sample of classifiers from a final solution obtained using XCS in learning the 20-bit multiplexer problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Action</th>
<th>(n)</th>
<th>(F)</th>
<th>(\epsilon)</th>
<th>(p)</th>
<th>(exp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1011###########1####</td>
<td>0</td>
<td>26</td>
<td>0.90</td>
<td>0</td>
<td>0</td>
<td>2871</td>
</tr>
<tr>
<td>2</td>
<td>1001###########1#####</td>
<td>1</td>
<td>24</td>
<td>0.84</td>
<td>0</td>
<td>1000</td>
<td>23914</td>
</tr>
<tr>
<td>3</td>
<td>0011#####0###########</td>
<td>0</td>
<td>22</td>
<td>0.77</td>
<td>0</td>
<td>1000</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>0100#####0##############</td>
<td>1</td>
<td>21</td>
<td>0.73</td>
<td>0</td>
<td>0</td>
<td>3487</td>
</tr>
<tr>
<td>5</td>
<td>0001#1####################</td>
<td>1</td>
<td>20</td>
<td>0.80</td>
<td>0</td>
<td>1000</td>
<td>22556</td>
</tr>
<tr>
<td>6</td>
<td>1010##############1#######</td>
<td>0</td>
<td>19</td>
<td>0.77</td>
<td>0</td>
<td>0</td>
<td>5510</td>
</tr>
<tr>
<td>7</td>
<td>1000##############0##########</td>
<td>0</td>
<td>18</td>
<td>0.60</td>
<td>0</td>
<td>1000</td>
<td>24508</td>
</tr>
<tr>
<td>8</td>
<td>1100###############00###</td>
<td>1</td>
<td>17</td>
<td>0.62</td>
<td>0</td>
<td>0</td>
<td>15977</td>
</tr>
<tr>
<td>9</td>
<td>1010##############00#####</td>
<td>0</td>
<td>16</td>
<td>0.69</td>
<td>0</td>
<td>1000</td>
<td>10427</td>
</tr>
<tr>
<td>10</td>
<td>1110###############0#</td>
<td>0</td>
<td>15</td>
<td>0.61</td>
<td>0</td>
<td>1000</td>
<td>4056</td>
</tr>
<tr>
<td>11</td>
<td>0100##############1##</td>
<td>1</td>
<td>14</td>
<td>0.65</td>
<td>0</td>
<td>1000</td>
<td>3810</td>
</tr>
<tr>
<td>12</td>
<td>0000###0#####0##########</td>
<td>1</td>
<td>13</td>
<td>0.46</td>
<td>0</td>
<td>0</td>
<td>7861</td>
</tr>
<tr>
<td>13</td>
<td>0001#00#####0##############</td>
<td>1</td>
<td>13</td>
<td>0.51</td>
<td>0</td>
<td>0</td>
<td>8299</td>
</tr>
<tr>
<td>14</td>
<td>0011#####00#####0##########</td>
<td>0</td>
<td>10</td>
<td>0.31</td>
<td>0</td>
<td>1000</td>
<td>2094</td>
</tr>
<tr>
<td>15</td>
<td>#100#####00#####0####</td>
<td>1</td>
<td>8</td>
<td>0.23</td>
<td>0</td>
<td>0</td>
<td>1452</td>
</tr>
</tbody>
</table>

of specific bits in the code-fragment action. For example, the classifier number eight ‘0010#00000000000000’ in Table 5.3 does not contain any specific data bit in the condition, but the addressed data bit is in the action, i.e. D6. Now, the action value of this classifier is 1 if the environmental bit D6 is 1 and 0 otherwise. Therefore, it is correct for all the matching instances. The optimal classifiers produced in XCSCFA\(_e\) were not clearly separated from the sub-optimal ones; unlike XCSCFA\(_e\).

The XCSCFA systems have larger final populations than XCS, see Section 5.6 for more details, due to the multiple genotypes to a single phenotype mapping in the code-fragment actions. As there are multiple classifiers in the XCSCFA systems for a corresponding single classifier in XCS, the fitness values of the final classifiers are smaller in the former systems.
Table 5.2: A sample of classifiers from a final solution obtained using XCSF A in learning the 20-bit multiplexer problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Code-Fragment Action</th>
<th>( n )</th>
<th>( F )</th>
<th>( \epsilon )</th>
<th>( p )</th>
<th>( \text{exp} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1001#............#</td>
<td>D3 D8 r D8 D8 r</td>
<td>11</td>
<td>0.34</td>
<td>0</td>
<td>0</td>
<td>2651</td>
</tr>
<tr>
<td>2</td>
<td>0101#............#</td>
<td>D3 D3</td>
<td>D3 D3</td>
<td>11</td>
<td>0.32</td>
<td>1000</td>
<td>4177</td>
</tr>
<tr>
<td>3</td>
<td>1100#............#</td>
<td>D3 D3 D7 &amp; &amp;</td>
<td>10</td>
<td>0.28</td>
<td>0</td>
<td>0</td>
<td>1672</td>
</tr>
<tr>
<td>4</td>
<td>0110#............#</td>
<td>D1 D1 D1 D1 r r</td>
<td>10</td>
<td>0.32</td>
<td>1000</td>
<td>5100</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0101#............#</td>
<td>D3 D11</td>
<td>D3 D11</td>
<td>9</td>
<td>0.26</td>
<td>1000</td>
<td>1068</td>
</tr>
<tr>
<td>6</td>
<td>1001#............#</td>
<td>D3 D3 r D3 D3 r</td>
<td>9</td>
<td>0.28</td>
<td>0</td>
<td>0</td>
<td>1745</td>
</tr>
<tr>
<td>7</td>
<td>0000#............#</td>
<td>D1</td>
<td>9</td>
<td>0.29</td>
<td>1000</td>
<td>4338</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1011#............#</td>
<td>D3 D3 r</td>
<td>8</td>
<td>0.29</td>
<td>1000</td>
<td>1300</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1001#............#</td>
<td>D3 D5</td>
<td>D3 D5</td>
<td>8</td>
<td>0.36</td>
<td>0</td>
<td>1577</td>
</tr>
<tr>
<td>10</td>
<td>1000#............#</td>
<td>D1 D0 &amp; D1 D0 &amp; d</td>
<td>8</td>
<td>0.24</td>
<td>0</td>
<td>1961</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0010#............#</td>
<td>D3 D6 d D6 d &amp;</td>
<td>8</td>
<td>0.20</td>
<td>0</td>
<td>2238</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>00001#............#</td>
<td>D4 D4</td>
<td>D4 D4</td>
<td>8</td>
<td>0.22</td>
<td>1000</td>
<td>2668</td>
</tr>
<tr>
<td>13</td>
<td>0010#............#</td>
<td>D3 D3 d D3 D3 d &amp;</td>
<td>8</td>
<td>0.25</td>
<td>0</td>
<td>2912</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1000#............#</td>
<td>D3 D7 &amp; D3 D7 d &amp;</td>
<td>8</td>
<td>0.26</td>
<td>1000</td>
<td>4169</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0101#............#</td>
<td>D0 ~ D3 D3 &amp; &amp;</td>
<td>7</td>
<td>0.30</td>
<td>0</td>
<td>849</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: A sample of classifiers from a final solution obtained using XCSF A in learning the 20-bit multiplexer problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Code-Fragment Action</th>
<th>( n )</th>
<th>( F )</th>
<th>( \epsilon )</th>
<th>( p )</th>
<th>( \text{exp} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0100#............#</td>
<td>D8 D8 D8 D3</td>
<td>10</td>
<td>0.18</td>
<td>0</td>
<td>0</td>
<td>26791</td>
</tr>
<tr>
<td>2</td>
<td>0101#............#</td>
<td>D9 D9 r</td>
<td>9</td>
<td>0.16</td>
<td>0</td>
<td>0</td>
<td>52880</td>
</tr>
<tr>
<td>3</td>
<td>1011#............#</td>
<td>D3 D15 d D15 ~ &amp;</td>
<td>8</td>
<td>0.12</td>
<td>0</td>
<td>0</td>
<td>2274</td>
</tr>
<tr>
<td>4</td>
<td>1001#............#</td>
<td>D0 D13 &amp; &amp;</td>
<td>7</td>
<td>0.13</td>
<td>0</td>
<td>0</td>
<td>7726</td>
</tr>
<tr>
<td>5</td>
<td>1001#............#</td>
<td>D13 ~</td>
<td>6</td>
<td>0.12</td>
<td>0</td>
<td>0</td>
<td>635</td>
</tr>
<tr>
<td>6</td>
<td>0001#............#</td>
<td>D5 D5</td>
<td>D5 D5</td>
<td>6</td>
<td>0.10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1110#............#</td>
<td>D18 D18</td>
<td>r D2 &amp;</td>
<td>5</td>
<td>0.07</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0010#............#</td>
<td>D6 ~ ~</td>
<td>5</td>
<td>0.08</td>
<td>1000</td>
<td>2604</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0100#............#</td>
<td>D3 D8 D8</td>
<td>r</td>
<td>4</td>
<td>0.08</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0001#............#</td>
<td>D1 D5</td>
<td>3</td>
<td>0.02</td>
<td>1000</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0011#............#</td>
<td>D7 ~ D7 D7 r r</td>
<td>3</td>
<td>0.07</td>
<td>0</td>
<td>0</td>
<td>143</td>
</tr>
<tr>
<td>12</td>
<td>1001#............#</td>
<td>D13 D2 D13</td>
<td>&amp;</td>
<td>3</td>
<td>0.05</td>
<td>1000</td>
<td>70</td>
</tr>
<tr>
<td>13</td>
<td>0111#............#</td>
<td>D11 D11</td>
<td>D3 &amp; &amp;</td>
<td>3</td>
<td>0.05</td>
<td>1000</td>
<td>227</td>
</tr>
<tr>
<td>14</td>
<td>1011#............#</td>
<td>D3 D15 D15 d d</td>
<td>3</td>
<td>0.05</td>
<td>1000</td>
<td>333</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1010#............#</td>
<td>D14 ~</td>
<td>3</td>
<td>0.07</td>
<td>0</td>
<td>0</td>
<td>685</td>
</tr>
</tbody>
</table>

5.4.2 The Majority-On Problem Domain

We believe that majority-on problems are hard to learn because the complete solution for a majority-on problem consists of overlapping classifiers.
It is observed that standard XCS produced overgeneral classifiers in the 7-bit majority-on problem, see Table 5.4. For example, the 10th classifier rule ‘1###1#1 : 1’ is an overgeneral classifier which matches 16 environmental instances. It is a “dangerous” classifier because it is correct for all the matching instances except ‘1000101’, so likely to be considered accurate ($\epsilon < \epsilon_0$) and therefore, it is highly likely that it will subsume the specific classifier ‘1000101 : 1’ in the training process, resulting in decreased chances for production of the desired classifier ‘1000101 : 0’.

It is observed that almost all the classifiers in the final solutions obtained using XCS for the 7-bit majority-on problem were overgeneral. This indicates that the overgeneral classifiers having $\epsilon < \epsilon_0$, such as the 2nd, 10th and 15th classifiers in Table 5.4, would have subsumed the otherwise needed accurate classifiers, resulting in the poor performance of the system.

The XCSCFA systems managed to avoid overgeneral classifier rules in the final solutions, see Table 5.5 and Table 5.6. The XCSCFA $_e$ method produced more general (and still accurate) classifiers than the other two methods, similar to the multiplexer domain. For example, the 15th classifier in Table 5.6 ‘0##1#11 : D4 D1 | D2 D1 | ’ matches eight environmental instances: ‘0001011’, ‘0001111’, ‘0011011’, ‘0011111’, ‘0101011’, ‘0101111’, ‘0111011’, and ‘0111111’. Now, the action value of this classifier is 1 if any of the environmental bits D1, D2, or D4 is 1. Therefore, it is correct for all the matching instances. It is to be noted that the classifier number eight in Table 5.6 is an erroneous classifier, therefore, it has relatively low fitness value. The reason for it being in the final solution is that it is newly created (cf. low experience value), otherwise it would have been deleted in the training process.
Table 5.4: A sample of classifiers from a final solution obtained using XCS in learning the 7-bit majority-on problem.

| No. | Condition | Action | \(|n| \) | \(F\) | \(\epsilon\) | \(p\) | \(exp\) |
|-----|-----------|--------|--------|------|--------|------|--------|
| 1   | ####11#   | 0      | 83     | 0.01 | 166.85 | 82.28| 194    |
| 2   | #0#0##0   | 1      | 73     | 0.51 | 3.71   | 0.45 | 1508   |
| 3   | ##00###   | 0      | 67     | 0.06 | 139.84 | 964.81| 360    |
| 4   | 0###0##0   | 1      | 61     | 0.21 | 73.40  | 18.67| 830    |
| 5   | ###1##1    | 1      | 58     | 0.10 | 238.67 | 792.05| 921    |
| 6   | #1##1##0   | 0      | 57     | 0.01 | 16.67  | 2.60 | 974    |
| 7   | #0#0#0#0  | 1      | 55     | 0.08 | 284.91 | 233.67| 537    |
| 8   | ###000##   | 1      | 54     | 0.21 | 168.94 | 82.68 | 216    |
| 9   | ##0##0##0  | 0      | 53     | 0.06 | 470.20 | 681.82| 201    |
| 10  | 1###1###   | 1      | 50     | 0.44 | 6.82   | 689.14| 499    |
| 11  | #111###    | 0      | 48     | 0.09 | 301.47 | 203.09| 504    |
| 12  | ###00##0   | 0      | 47     | 0.01 | 296.17 | 279.33| 447    |
| 13  | #1##1##1   | 0      | 47     | 0.00 | 89.27  | 27.08 | 392    |
| 14  | 0##0#0#0   | 1      | 47     | 0.16 | 183.70 | 72.53 | 598    |
| 15  | #00##0##0  | 1      | 47     | 0.56 | 2.03   | 0.23 | 1019   |

5.4.3 The Even-Parity Problem Domain

In standard XCS with numeric action, it is not possible for a classifier rule to have a ‘#' symbol in the condition and still be accurate for an even parity problem. However, it is interesting to note that XCSCFA, was able to produce accurate general classifier rules having a ‘#' symbol in the condition for the 7-bit even parity problem, see Table 5.7. For example, the third classifier rule ‘111000# : D6’ matches two environmental instances, ‘1110000’ and ‘1110001’; and the action value is the binary value of the last symbol in the environmental instance, i.e. D6. Therefore the advocated action for the problem instance ‘1110000’ is 0 and for ‘1110001’ is 1, and both are accurate. It is to be noted that in the 7-bit even parity problem, the
Table 5.5: A sample of classifiers from a final solution obtained using \( \text{XCSCFA}_c \) in learning the 7-bit majority-on problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Code-Fragment Action</th>
<th>( n )</th>
<th>( F )</th>
<th>( \epsilon )</th>
<th>( p )</th>
<th>( \text{exp} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>110#000</td>
<td>D0</td>
<td>12</td>
<td>0.52</td>
<td>0</td>
<td>0</td>
<td>201</td>
</tr>
<tr>
<td>2</td>
<td>110#000</td>
<td>D0 D2 &amp;</td>
<td>11</td>
<td>0.37</td>
<td>0</td>
<td>1000</td>
<td>69</td>
</tr>
<tr>
<td>3</td>
<td>1001011</td>
<td>D5 D5 D6 &amp;</td>
<td></td>
<td>10</td>
<td>0.39</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>0101010</td>
<td>D0 D3 &amp; D0 D3 &amp; r</td>
<td>10</td>
<td>0.46</td>
<td>0</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>#000001</td>
<td>D6 D3 r</td>
<td>9</td>
<td>0.25</td>
<td>0</td>
<td>1000</td>
<td>74</td>
</tr>
<tr>
<td>6</td>
<td>1100111</td>
<td>D6 D5 d D5 r</td>
<td>9</td>
<td>0.34</td>
<td>0</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>7</td>
<td>0110110</td>
<td>D4 D2 d D6 D0 &amp;</td>
<td></td>
<td>9</td>
<td>0.39</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0010011</td>
<td>D2 D5 &amp; D6 &amp;</td>
<td>9</td>
<td>0.46</td>
<td>0</td>
<td>0</td>
<td>399</td>
</tr>
<tr>
<td>9</td>
<td>111110</td>
<td>D6 D6 r</td>
<td>8</td>
<td>0.12</td>
<td>0</td>
<td>1000</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>1111111</td>
<td>D1 D2 r</td>
<td>8</td>
<td>0.33</td>
<td>0</td>
<td>0</td>
<td>157</td>
</tr>
<tr>
<td>11</td>
<td>111001</td>
<td>D2 ~ D2 ~</td>
<td></td>
<td>8</td>
<td>0.34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0011110</td>
<td>D1 D6</td>
<td>D1 D6</td>
<td></td>
<td>8</td>
<td>0.29</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0011011</td>
<td>D5 ~</td>
<td>8</td>
<td>0.33</td>
<td>0</td>
<td>0</td>
<td>106</td>
</tr>
<tr>
<td>14</td>
<td>111001</td>
<td>D2 D3</td>
<td>D5 &amp;</td>
<td></td>
<td>8</td>
<td>0.32</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>011000</td>
<td>D3 D2 d D3 &amp;</td>
<td>8</td>
<td>0.26</td>
<td>0</td>
<td>1000</td>
<td>238</td>
</tr>
</tbody>
</table>

Classifiers obtained using \( \text{XCSCFA}_c \) have sufficiently higher fitness values than other domains, because it is relatively difficult to produce multiple classifiers covering the same niche. This is unlikely in other problem domains such as the multiplexer and majority-on problems.

As a result of the generalized classifiers in \( \text{XCSCFA}_c \), the number of classifiers required in the final solution set reduced to half of the number of specific classifiers needed otherwise and it successfully solved the 7-bit even-parity problem. However, standard XCS as well as \( \text{XCSCFA}_c \) could not produce accurate general classifier rules in the 7-bit even-parity problem, and failed to solve it with this setup. It is to be noted that there is just one '#' symbol in the condition of a general classifier produced in \( \text{XCSCFA}_c \) for the 7-bit even-parity problem. In \( \text{XCSCFA}_c \) the
### 5.5. XCS’S BIAS AGAINST OVERLAPPING RULES

It has been reported that XCS is biased against the overlapping classifiers in a rule population [24,55,69]. The solutions obtained using XCS for the overlapping and niche imbalance problems experimented here consist of overgeneral classifiers, e.g. see Table 5.4. Some of them are only incorrect for very few instances, therefore, it is anticipated that they would have subsumed the otherwise needed accurate classifiers, resulting in poor per-

---

Table 5.6: A sample of classifiers from a final solution obtained using XSCSFCA in learning the 7-bit majority-on problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Code-Fragment Action</th>
<th>n</th>
<th>F</th>
<th>ε</th>
<th>p</th>
<th>exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0##010#</td>
<td>D1 D6 &amp; D2 D1 &amp; &amp;</td>
<td>13</td>
<td>0.21</td>
<td>0</td>
<td>1000</td>
<td>3103</td>
</tr>
<tr>
<td>2</td>
<td>1##0011</td>
<td>D1 D2</td>
<td>D1 D2</td>
<td>&amp;</td>
<td>13</td>
<td>0.26</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0##0011</td>
<td>D2 D1 &amp; D5 D1 &amp; &amp;</td>
<td>11</td>
<td>0.20</td>
<td>0</td>
<td>1000</td>
<td>570</td>
</tr>
<tr>
<td>4</td>
<td>01##00#0</td>
<td>D2 D3 D5 &amp; &amp;</td>
<td>11</td>
<td>0.17</td>
<td>0</td>
<td>0</td>
<td>1380</td>
</tr>
<tr>
<td>5</td>
<td>#1011#0</td>
<td>D5 D0 r D5 D0 r d</td>
<td>11</td>
<td>0.19</td>
<td>0</td>
<td>1000</td>
<td>1787</td>
</tr>
<tr>
<td>6</td>
<td>11#00#0</td>
<td>D2 D5 &amp; D2 D5 &amp; &amp;</td>
<td>11</td>
<td>0.20</td>
<td>0</td>
<td>1000</td>
<td>2096</td>
</tr>
<tr>
<td>7</td>
<td>1##1001</td>
<td>D1 D3 r D1 D3 r</td>
<td></td>
<td>11</td>
<td>0.24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1##0011</td>
<td>D2 D1</td>
<td>D2 D1</td>
<td>&amp;</td>
<td>10</td>
<td>0.04</td>
<td>306.69</td>
</tr>
<tr>
<td>9</td>
<td>##011#1</td>
<td>D5 D5 &amp; D1 D0</td>
<td></td>
<td>10</td>
<td>0.19</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>10</td>
<td>1#110#0</td>
<td>D1 D5</td>
<td>D2 D5</td>
<td>d</td>
<td>10</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>001#00#</td>
<td>D2 D2 &amp; D2 D2 &amp; &amp;</td>
<td>10</td>
<td>0.16</td>
<td>0</td>
<td>0</td>
<td>653</td>
</tr>
<tr>
<td>12</td>
<td>11#00#0</td>
<td>D2 D2 D5 &amp; &amp;</td>
<td>10</td>
<td>0.15</td>
<td>0</td>
<td>1000</td>
<td>947</td>
</tr>
<tr>
<td>13</td>
<td>##01101</td>
<td>D3 D0 D1</td>
<td>d</td>
<td></td>
<td>10</td>
<td>0.19</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0##010#</td>
<td>D2 D6 &amp; D2 D1 &amp; &amp;</td>
<td>9</td>
<td>0.14</td>
<td>0</td>
<td>1000</td>
<td>673</td>
</tr>
<tr>
<td>15</td>
<td>0##11#11</td>
<td>D4 D1</td>
<td>D2 D1</td>
<td>&amp;</td>
<td>9</td>
<td>0.16</td>
<td>0</td>
</tr>
</tbody>
</table>

classifiers having just one ‘#’ symbol cannot be consistently accurate in the even-parity domain because the ‘#’ symbol is randomly treated 0 or 1 in XSCSFCA to compute the action value of a classifier.


Table 5.7: A sample of classifiers from a final solution obtained using XCSFCA\(_{e}\) in learning the 7-bit even-parity problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Code-Fragment Action</th>
<th>(n)</th>
<th>(F)</th>
<th>(\epsilon)</th>
<th>(p)</th>
<th>exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101000#</td>
<td>D6</td>
<td>20</td>
<td>0.99</td>
<td>0</td>
<td>0</td>
<td>1944</td>
</tr>
<tr>
<td>2</td>
<td>11#0011</td>
<td>D2</td>
<td>19</td>
<td>0.97</td>
<td>0</td>
<td>0</td>
<td>1944</td>
</tr>
<tr>
<td>3</td>
<td>111000#</td>
<td>D6</td>
<td>18</td>
<td>0.98</td>
<td>0</td>
<td>1000</td>
<td>1089</td>
</tr>
<tr>
<td>4</td>
<td>001#101</td>
<td>D3</td>
<td>18</td>
<td>0.99</td>
<td>0</td>
<td>1000</td>
<td>2552</td>
</tr>
<tr>
<td>5</td>
<td>0100#00</td>
<td>D4 ~</td>
<td>17</td>
<td>0.99</td>
<td>0</td>
<td>0</td>
<td>1018</td>
</tr>
<tr>
<td>6</td>
<td>11#0010</td>
<td>D2</td>
<td>17</td>
<td>0.97</td>
<td>0</td>
<td>1000</td>
<td>3853</td>
</tr>
<tr>
<td>7</td>
<td>#001100</td>
<td>D0 D2 r</td>
<td>17</td>
<td>0.99</td>
<td>0</td>
<td>1000</td>
<td>9265</td>
</tr>
<tr>
<td>8</td>
<td>111011#</td>
<td>D6 D6 d</td>
<td>16</td>
<td>0.97</td>
<td>0</td>
<td>0</td>
<td>2583</td>
</tr>
<tr>
<td>9</td>
<td>1010#11</td>
<td>D4</td>
<td>16</td>
<td>0.98</td>
<td>0</td>
<td>0</td>
<td>4706</td>
</tr>
<tr>
<td>10</td>
<td>111110#</td>
<td>D6 ~</td>
<td>16</td>
<td>0.99</td>
<td>0</td>
<td>0</td>
<td>5768</td>
</tr>
<tr>
<td>11</td>
<td>110100#</td>
<td>D6 ~</td>
<td>16</td>
<td>0.98</td>
<td>0</td>
<td>0</td>
<td>8612</td>
</tr>
<tr>
<td>12</td>
<td>#011110</td>
<td>D0</td>
<td>15</td>
<td>0.96</td>
<td>0</td>
<td>0</td>
<td>1088</td>
</tr>
<tr>
<td>13</td>
<td>101100#</td>
<td>D6</td>
<td>15</td>
<td>0.94</td>
<td>0</td>
<td>1000</td>
<td>1485</td>
</tr>
<tr>
<td>14</td>
<td>01001#1</td>
<td>D5 D1 &amp; D5 D1 &amp; &amp;</td>
<td>15</td>
<td>1.00</td>
<td>0</td>
<td>1000</td>
<td>2122</td>
</tr>
<tr>
<td>15</td>
<td>100011#</td>
<td>D6 ~</td>
<td>15</td>
<td>0.99</td>
<td>0</td>
<td>0</td>
<td>7239</td>
</tr>
</tbody>
</table>


Performance of the system. Therefore, two more sets of experiments were conducted using XCS: 1) deactivating the action set subsumption, and 2) tuning the parameters, i.e. increasing the classifier experience threshold for subsumption \(\theta_{sub}\) from 20 to 200 and decreasing the prediction error threshold \(\epsilon_0\) from 10 to 0.01. The performance of XCS improved in both of these experimental setups and it successfully solved the DV1, the 7-bit count ones, and the 4+4 bit carry problems, see Figure 5.10. This is consistent with previous findings as Butz et al. [23] have solved count ones problems by using an error-based fitness approach in XCS and deactivating the action set subsumption.

It is observed that XCS with tuned parameters setting required approximately 400,000 problem instances to learn the DV1 problem whereas XCS
5.5. **XCS’S BIAS AGAINST OVERLAPPING RULES**

without action set subsumption needed approximately 16,000,000 problem instances to reach the 100% performance level for the DV1 problem, see Figure 5.10(c). In the 7-bit majority-on problem, the performance improved to more than 99%, but did not reach the stabilized 100% performance level, as shown in Figure 5.10(a).

![Figure 5.10: Results of overlapping and niche imbalance problems using XCS.](image-url)

(a) for the 7-bit majority-on problem.  
(b) for the 7-bit count ones problem.

(c) for the DV1 problem.  
(d) for the 4+4 bit carry problem.

Usually \( \epsilon_0 \) is set to 10 to handle noise in the environment and \( \theta_{\text{sub}} \) is set to 20 to compact the population using the subsumption deletion. By tuning \( \epsilon_0 \) and \( \theta_{\text{sub}} \) for overlapping and niche imbalance problems, the classification performance is improved, but it is anticipated to compromise the robustness to noise and the compactness of the population.

It is observed that in the multiplexer problem domain, parameter tuning in this way degraded the performance of XCS in terms of the required
instances to reach the optimum performance level, e.g. see Figure 5.11. The reason for this degradation is the unnecessarily large value of $\theta_{sub}$ and very small value of $\epsilon_0$ for the non-overlapping and niche balanced multiplexer domain.

![20-bit Multiplexer](image)

Figure 5.11: Result of the 20-bit multiplexer problem using XCS.

### 5.6 Action Inconsistency and Redundancy in XC-SCFA

XCS keeps a complete map, i.e. the classifiers advocating consistently correct classification (and hence predicting an accurate, maximum environmental reward of say 1000) as well as the classifiers advocating consistently incorrect classification (and hence predicting an accurate, minimum environmental reward of say 0). It is noted that the building blocks of information in the condition of an accurate incorrect classifier are exactly the same as in the counterpart correct classifier. For example, in case of
the 6-bit MUX ‘000### : 1 → 0’ is an accurate incorrect classifier which has the same condition as that in the counterpart accurate correct classifier ‘000### : 0 → 1000’. The rule discovery operation is applied in the action set, which is formed by the classifiers advocating a certain action, commonly selected at random, and covering the currently observed environmental input. As all the classifiers in an action set advocate the same action, the correct and incorrect classifiers cannot occur in the same action set, so cannot be simultaneously used in the breeding of new classifiers. Therefore, in an XCS system, although both correct and incorrect classifiers are kept throughout the learning of the system, the building blocks of information in them are not efficiently exploited as they are not allowed to take part in the same breeding operation. In the XCSCFA approach, due to inconsistent action values, the incorrect classifiers can occur in the same action set as correct classifiers so can be used for the production of good classifiers.

Further, multiple genotypes to a single phenotype mapping of the GP-like code fragments in the XCSCFA approach provides redundancy and diversity [2, 84], which may increase the robustness of the system, especially in learning overlapping and niche imbalance problems. In order to understand the effect of redundancy in the XCSCFA approach, consider the 7-bit majority-on problem and the hypothetical classifiers shown in Table 5.8. Here ‘V’ is the computed action value of a classifier rule.

Table 5.8: Effect of redundancy in the XCSCFA approach.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Action</th>
<th>V</th>
<th>p</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1###1#1</td>
<td>D4 D6 &amp;</td>
<td>1</td>
<td>999.14</td>
<td>6.82</td>
</tr>
<tr>
<td>2</td>
<td>1000101</td>
<td>D4 D6 &amp;</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1000101</td>
<td>D3 D5 D3 &amp; d</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Classifier 1 is more general than classifier 2 and classifier 3, where the latter two classifiers are completely accurate (although incorrect), but the former is incorrect once out of the 16 matched input messages. Classifier 1
and classifier 2 have the same genotypic code-fragment action, therefore, the former can subsume the latter, resulting in reduced chances of production of the required correct classifier ‘1000101 : 0’, similar to standard XCS as discussed in Section 5.4.2. However, classifier 3 cannot be subsumed by classifier 1 because of having different genotypic code-fragment actions. This redundant classifier can be used in the rule discovery operation to produce the required correct classifier ‘1000101 : 0’. In XCSCFA there can be more than one genotypic classifier equivalent to the classifier ‘1000101 : 0’ in XCS, e.g. ‘1000101 : D4 D2 | D2 D2 | &’ and ‘1000101 : D6 D0 D2 d d’, which further increase the chances of producing the phenotypic classifier ‘1000101 : 0’ in XCSCFA. The XCSCFAe approach, in addition to this redundancy, has ability to incorporate generalization in actions making it more robust than XCS and XCSCFAc.

It is considered that the XCSCFA systems outperformed standard XCS, using typical experimental setup, due to the following two properties of code fragment based actions: 1) the ability in all accuracy-based systems to keep a complete map combined with inconsistent actions in the XCSCFA approach preserved important building blocks of information, and 2) multiple genotypes to a single phenotype mapping of the GP-like code fragments provided redundancy and diversity. To verify this, we repeated all the experiments by disallowing the inconsistent and redundant actions. The obtained results were similar to the results obtained using XCS with the typical experimental setup. Then, to investigate the effect of each property separately, we conducted two more sets of experiments: 1) allowing inconsistency but no redundancy, and 2) allowing redundancy but no inconsistency. The obtained results are shown in Figure 5.12.

It is observed that by disallowing the redundancy while retaining the inconsistency, the performance of XCSCFAe degraded and it failed in learning all the problems except multiplexer problems. The XCSCFAe method still solved all the problems because of its ability to incorporate generalization in the actions of classifier rules, but in learning the overlapping
5.6. ACTION INCONSISTENCY AND REDUNDANCY IN XCSCFA

(a) for the 7-bit majority-on problem.
(b) for the 7-bit count ones problem.
(c) for the DV1 problem.
(d) for the 4+4 bit carry problem.
(e) for the 20-bit multiplexer problem.
(f) for the 7-bit even-parity problem.

Figure 5.12: Effect of action inconsistency and redundancy in the XCSCFA approach. The performance curves are coincident for both XCSCFA$_c$ and XCSCFA$_e$ on disallowing the inconsistency while retaining the redundancy.
and niche imbalance problems it needed relatively large number of problem instances. On the other hand, by disallowing the inconsistency while retaining the redundancy, both XCSCFAe and XCSCFAc behaved similarly and successfully solved the overlapping and niche imbalance problems, again more slowly, but failed in learning the 7-bit even-parity problem.

It is to be noted that due to the multiple genotypes to a single phenotype mapping of code-fragment actions in the XCSCFA systems, the subsumption deletion mechanism is not fully enabled and there are multiple classifiers in the XCSCFA systems for a corresponding single classifier in XCS. Therefore, the XCSCFA systems have produced larger macro-classifier populations than XCS, except in learning the 7-bit count ones problem, as shown in Figure 5.13. However, it was compensated by the improved performance of the XCSCFA systems due to the redundancy and diversity provided by the multiple genotypes to a single phenotype mapping.

The percentage of the number of macro-classifiers in the final solutions evolved using XCS and the XCSCFA systems are shown in Table 5.9. The last column in Table 5.9 shows the number of macro-classifiers in XCSCFAc after removing the classifiers that have inconsistent action values and then treating two classifiers equivalent if they have the same condition and the same action value. It can be seen that this simplification of the solutions obtained using XCSCFAc significantly reduced the population size, which is smaller than standard XCS in most of the problems.

### 5.7 Optimum Solutions in the Multiplexer Problem Domain

The XCSCFAc system produced unique and interesting rules having no ‘don’t care’ symbol (‘#’) in the address bits of most of the classifiers for the 6-bit to 37-bit multiplexer problems, as discussed in Section 5.4. It is to be
5.7. OPTIMUM SOLUTIONS IN THE MULTIPLEXER PROBLEM DOMAIN

(a) for the 7-bit majority-on problem.
(b) for the 7-bit count ones problem.
(c) for the DV1 problem.
(d) for the 4+4 bit carry problem.
(e) for the 20-bit multiplexer problem.
(f) for the 7-bit even-parity problem.

Figure 5.13: Population size in terms of macro-classifiers in learning different problems using XCS and XCSCFA systems.

Noted that a few newly created classifiers may contain ‘#’ in the address bits due to mutation. A specialized condensation mechanism is implemented in an effort to obtain the compact optimum solutions containing maximally general accurate classifiers for these multiplexer problems.
Table 5.9: The percentage of the number of macro-classifiers, along with standard deviation, in the final solutions obtained using XCS and XCSCFA systems for different problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>XCS</th>
<th>XCSCFA_c</th>
<th>XCSCFA_e</th>
<th>Simplified XCSCFA_e</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit MUX</td>
<td>9 ± 1</td>
<td>35 ± 2</td>
<td>21 ± 2</td>
<td>7 ± 1</td>
</tr>
<tr>
<td>11-bit MUX</td>
<td>13 ± 1</td>
<td>38 ± 2</td>
<td>25 ± 1</td>
<td>10 ± 1</td>
</tr>
<tr>
<td>20-bit MUX</td>
<td>19 ± 1</td>
<td>40 ± 1</td>
<td>29 ± 1</td>
<td>16 ± 1</td>
</tr>
<tr>
<td>37-bit MUX</td>
<td>22 ± 1</td>
<td>43 ± 1</td>
<td>37 ± 1</td>
<td>21 ± 1</td>
</tr>
<tr>
<td>7-bit majority-on</td>
<td>7 ± 1</td>
<td>50 ± 1</td>
<td>37 ± 1</td>
<td>25 ± 1</td>
</tr>
<tr>
<td>7-bit count ones</td>
<td>62 ± 2</td>
<td>50 ± 1</td>
<td>37 ± 1</td>
<td>29 ± 1</td>
</tr>
<tr>
<td>DV1</td>
<td>6 ± 1</td>
<td>45 ± 1</td>
<td>35 ± 1</td>
<td>15 ± 1</td>
</tr>
<tr>
<td>4+4 bit carry</td>
<td>5 ± 0</td>
<td>49 ± 1</td>
<td>39 ± 1</td>
<td>20 ± 1</td>
</tr>
<tr>
<td>7-bit even-parity</td>
<td>26 ± 1</td>
<td>41 ± 1</td>
<td>30 ± 1</td>
<td>20 ± 1</td>
</tr>
</tbody>
</table>

It is to be noted that this condensation is a post priori whilst non-iterative compaction mechanism instead of a standard iterative condensation technique. In a typical condensation method [67, 118], evolutionary search is suspended by stopping the GA creating new classifiers and learning continues for a certain number of iterations. In the condensation mechanism being introduced here, training is stopped and the rule set is compacted instantly.

### 5.7.1 Condensation

The specialized condensation algorithm is given below:

1. From the final rule set, delete all the classifiers that are either inaccurate (i.e., prediction error $\epsilon > \epsilon_0$), or less experienced (i.e., experience $exp \leq 1/\beta$), or have inconsistent action values.

2. In the remaining population, if two classifiers have the same condition, the same action value, and the same prediction value, then
treat them as a single classifier. Delete one of them and increase the numerosity of the other by the numerosity value of the one being deleted. The classifier being kept retains the higher experience and fitness values from these two classifiers.

3. In the resulting population of step 2, if two classifiers have the same condition, but opposite action (i.e. 0/1) and likewise the opposite prediction values (i.e. 0/1000), then invert the action and prediction values of the classifier having prediction value equal to 0 and condense them as a single classifier. Delete one of them and increase the numerosity of the other by the numerosity value of the one being deleted. The classifier being kept retains the higher experience and fitness values from these two classifiers.\(^3\)

4. Sort the resulting population of step 3 according to numerosity in descending order.

5. Find the two consecutive classifiers that have the maximum numerosity difference between each other: say they are \(C_1\) and \(C_2\) such that numerosity of classifier \(C_1\) is greater than that of \(C_2\).

6. Delete all the classifiers having numerosity equal to or less than the numerosity of \(C_2\).

By applying steps 1-4 of the condensation mechanism, the population of classifiers in XCSCFA, automatically separated into two groups according to numerosity values. Figure 5.14 shows the separation of classifiers, for a typical run of the 6-, 11-, 20-, and 37-bit multiplexer problems. It can be seen that the group of classifiers having higher numerosity values also have higher fitness values as would be expected. So the classifiers with low numerosity were deleted, applying steps 5-6 of the condensation algorithm, to obtain a final compact optimum population of maximally

\(^3\)This assumes binary classification with the complete mapping payoff of XCS being no longer explicitly required.
general and accurate classifiers. The resulting population has 8, 16, 32, and 64 classifiers for the 6-, 11-, 20-, and 37-bit multiplexer problems respectively, for all the 30 runs of each experiment. The final populations for the 6-bit and the 11-bit multiplexer problems are shown in Table 5.10 and Table 5.11 respectively.

![Figure 5.14](image1)

(a) for the 6-bit multiplexer  
(b) for the 11-bit multiplexer  
(c) for the 20-bit multiplexer  
(d) for the 37-bit multiplexer

Figure 5.14: The numerosity and fitness of classifiers in final population for a typical run in learning the 6-, 11-, 20-, and 37-bit multiplexer problems using XCSCFAc.

Figure 5.15 shows the final population of accurate and experienced classifiers for the 37-bit multiplexer problem, obtained using standard XCS with static binary actions. In standard XCS it is observed that there is
Table 5.10: Final compact population of maximally general and accurate classifiers obtained in a typical run for the 6-bit multiplexer problem using XCSCFA$_c$.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Code-Fragment Action</th>
<th>Action Value</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11###1</td>
<td>D3D0</td>
<td>D0D0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>01#1##</td>
<td>D3D5rD1D0&amp;d</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>10##1#</td>
<td>D5D0</td>
<td>D5D0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>001###</td>
<td>D2~</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>000###</td>
<td>D1D1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>11###0</td>
<td>D5D5&amp;D4~&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>7</td>
<td>01#0##</td>
<td>D0D3&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>8</td>
<td>10###0#</td>
<td>D1D2dD1D2dr</td>
<td>0</td>
<td>1000</td>
</tr>
</tbody>
</table>

some form of grouping, but no distinct separation of optimal and suboptimal classifiers as it failed 25 times out of 30 runs for the 37-bit multiplexer problem to produce the optimum compact rule set. To obtain the desired optimum rule set in binary action based XCS, extensive processing is needed [67], e.g. condensation or compaction algorithm.

It is to be noted that the fitness of classifiers in XCSCFA$_c$ is smaller as compared with that in standard XCS. The reason for this is that in XCS-based systems, the fitness is shared among the accurate classifiers in a niche, and the most general classifier in a niche has subsumed other less general classifiers. In standard XCS with binary action, subsumption deletion is fully enabled so the fitness of the general classifier in a niche gets higher value as it subsumes the less general classifiers in the niche. In XCSCFA$_c$ the multiple genotypes to a single phenotype issue disables the subsumption deletion function, so 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.

Although it was not the primary goal to produce an optimal solution, XCSCFA$_c$ serendipitously produced the optimum solutions for the 6-, 11-
Table 5.11: Final compact population of maximally general and accurate classifiers obtained in a typical run for the 11-bit multiplexer problem using XCSCFAc.

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Code-Fragment Action</th>
<th>Action Value</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101#####0##</td>
<td>D2~</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>111#####1</td>
<td>D10~D2D1rr</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>110#####0#</td>
<td>D1~D9D10&amp;&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>100#####0##</td>
<td>D7D7&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>0000#####</td>
<td>D3D3&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>011###1#####</td>
<td>D1<del>D6</del>d</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>7</td>
<td>011#####0#####</td>
<td>D8D5rD1~&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>8</td>
<td>111#####0#####</td>
<td>D0D3rD0D3r&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>001#1#####</td>
<td>D1D4dD4~</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>010#####0#####</td>
<td>D1<del>D1</del>&amp;</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>11</td>
<td>001#0#####</td>
<td>D1</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>12</td>
<td>010###1#####</td>
<td>D1<del>D1</del>r</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>13</td>
<td>0001#####</td>
<td>D0D7d</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>14</td>
<td>110#####1#</td>
<td>D9D9</td>
<td>D9</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>101#####1##</td>
<td>D8D0rD8D0rr</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>16</td>
<td>100#####1##</td>
<td>D1D2&amp;D1d</td>
<td>1</td>
<td>1000</td>
</tr>
</tbody>
</table>

, 20-, and 37-bit multiplexer problems. The following sections elaborate why the optimum solutions were produced in the multiplexer domain.

5.7.2 Specialization of Address Bits

In a multiplexer problem, if there are no ‘don’t care’ (‘#’) symbols in the address bits, then the system requires just one specific data bit, at the correct position, to generate an accurate rule. If there is a ‘#’ symbol in the address bits, then it needs at least two specific data bits, at the correct positions and having the same value, to produce an accurate rule (which will actually cover two simple rules). For example, in Table 5.12 there is no ‘#’s
in the address bits for first two rules so just one specific data bit is enough to make them accurate classifiers whereas the third rule has a ‘#’ in address bits so it needs two specific data bits to be accurate. Similarly the fourth rule, having two ‘#’ symbols in the address bits, needs four specific data bits to be an accurate rule. It is to be noted that the correctness of the rules depend on the value of action, e.g., in the case of these four rules, if action value is ‘1’ then they will be correct, otherwise they are incorrect classifiers.

Each ‘#’ symbol in the address bits makes it difficult for the system to produce an accurate classifier rule, although if it is produced then it will cover more than one classifier so the final population of classifiers would have relatively fewer classifiers than enumerated specific classifiers. If there are $n$ ‘#’ symbols in the address bits of a classifier then the system
Table 5.12: Four sample classifier rules for the 6-bit multiplexer problem, demonstrating the specialization of address bits.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 # # # 1</td>
<td>action</td>
</tr>
<tr>
<td>2</td>
<td>0 1 # 1 # #</td>
<td>action</td>
</tr>
<tr>
<td>3</td>
<td># 1 # 1 # 1</td>
<td>action</td>
</tr>
<tr>
<td>4</td>
<td># # 1 1 1 1</td>
<td>action</td>
</tr>
</tbody>
</table>

needs at least \(2^n\) specific data bits, at correct positions with the same value, in the classifier to make it an accurate classifier rule. If it is produced, it will be equivalent to \(2^n\) simpler classifiers.

If the action is binary (as in standard XCS implementation) then it is relatively easy for the system to produce such classifier rules. However in XCSCFA\(_c\), the action is a code fragment and the action value is determined by taking the associated condition as its input where ‘#’ in the condition is treated as 0 or 1 randomly. So it heavily depends on the bits in the associated condition to generate an accurate action that will result in the accurate classifier rule. A single ‘#’ symbol in the address bits makes it harder for this system to produce an accurate corresponding ‘code-fragment action’ because of inconsistency of action value. The consistency of a classifier’s action value with different condition patterns is discussed next.

5.7.3 Consistency of a Classifier’s Action Value

In standard XCS, all classifier rules are 100% consistent in terms of the action value. If a classifier’s action is 0 then it will be permanently 0 and if it is 1 then it will be permanently 1 throughout the system evolution. However in XCSCFA\(_c\), this is not the case.

In XCSCFA\(_c\), the 100% consistency of the whole population of classifiers (in terms of action value) is not guaranteed. The reason of this de-
5.7. OPTIMUM SOLUTIONS IN THE MULTIPLEXER PROBLEM DOMAIN

Increased consistency is that the ‘#’ symbol in the condition of a classifier is randomly treated as 0 or 1 during the computation of a classifier’s action value. A classifier having no ‘#’ symbol in the condition is 100% consistent in terms of its action value, but if a classifier has one or more ‘#’ symbols in the condition then its consistency depends upon the code-fragment action. If the value of a code-fragment action is dependent upon a condition bit that is ‘#’ then it cannot be 100% consistent. For example in case of the classifier rule “1#01#1:D0D4|D3D1&&”, depicted in Figure 5.16, there are two ‘#’ symbols in the condition (D1 and D4) and both occur in the code-fragment action. The action value of this classifier is dependent on the bit D1: if the value of bit D1 is 0 then the action value will be 0 and if the value of bit D1 is 1 then the action value will be 1 (Note: The value of this code-fragment action is not dependent on the value of bit D4.).

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0 D1 D2 D3 D4 D5</td>
<td>1 # 0 1 # 1</td>
</tr>
</tbody>
</table>

Figure 5.16: A classifier rule with code-fragment action where the action is dependent on a ‘#’ bit in the condition.

Suppose if there are $m$ available code fragments then there will be $m$ classifier rules that have the same condition, but a different code fragment as the action. Some of these classifiers will be 100% consistent in terms of their action values and others not.

If a classifier’s action value is consistent (it can be correct or incorrect)\(^4\) then the correct action will lead to a stable predicted reward of maximum

\(^4\)In XCS-based systems, the accuracy of prediction is more important than the correctness of the prediction itself.
value (1000 in this implementation) and similarly the incorrect action will lead to a stable predicted reward of minimum value (0 in this implementation). If the action value of a classifier is not consistent then the predicted reward of the classifier will neither reach the maximum payoff value nor the minimum payoff value. The predicted reward of a classifier having inconsistent action value will increase and decrease depending upon the correctness of its action value for a given environmental instance of the problem. The inconsistency of a classifier in terms of action value results in inconsistency of the classifier in terms of predicted reward.

To analyze the consistency of action values, consistency of classifiers with all address bit patterns in conditions for the 6-bit multiplexer problem was calculated. There are two address bits in the 6-bit multiplexer problem and each of these two bits can take a value from the ternary alphabet \{0, 1, #\} so there are nine \(3^2\) different address patterns. Similarly, the four data bits can take values from the ternary alphabet \{0, 1, #\} so there are 81 \(3^4\) different data patterns for the 6-bit multiplexer problem. Combining these different address bits and data bits patterns, there are 729 \((9 \times 81)\) different conditions. There are six terminals and five operators of arity \{1, 2, 2, 2, 2\}, so there are 97506 distinct code-fragment actions of depth up to two. Using these 729 different conditions and 97506 different code fragment trees as actions, results in 71081874 different classifier rules. Each of the nine address patterns have 7897986 classifiers. The consistency of each of the address patterns is shown in Table 5.13. The consistency of each of the four patterns having both specific address bits is 79.65% whereas the consistency of the last pattern that has ´#´ in both of the address bits is just 49.21%. The patterns with one specific address bit and one ´#´ address bit are 64.41% consistent in their action values.

Because the classifiers having conditions from the schema “AAAAxxx”\(^5\) are more consistent in terms of action values than other classifiers, the for-

\(^5\)In this schema ‘A’, the address bits, can be either 0 or 1 and ‘x’, the data bits, can be 0, 1, or ‘#’. 
5.7. OPTIMUM SOLUTIONS IN THE MULTIPLEXER PROBLEM DOMAIN

Table 5.13: Consistency of classifiers in terms of action values when using different patterns as the condition. Here ‘x’ means 0, 1, or ‘#’.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Pattern (Condition)</th>
<th>Total Classifiers</th>
<th>Consistent Classifiers</th>
<th>Consistency Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00xxxx</td>
<td>7897986</td>
<td>6290622</td>
<td>79.65%</td>
</tr>
<tr>
<td>2</td>
<td>01xxxx</td>
<td>7897986</td>
<td>6290622</td>
<td>79.65%</td>
</tr>
<tr>
<td>3</td>
<td>10xxxx</td>
<td>7897986</td>
<td>6290622</td>
<td>79.65%</td>
</tr>
<tr>
<td>4</td>
<td>11xxxx</td>
<td>7897986</td>
<td>6290622</td>
<td>79.65%</td>
</tr>
<tr>
<td>5</td>
<td>0#xxxx</td>
<td>7897986</td>
<td>5087259</td>
<td>64.41%</td>
</tr>
<tr>
<td>6</td>
<td>1#xxxx</td>
<td>7897986</td>
<td>5087259</td>
<td>64.41%</td>
</tr>
<tr>
<td>7</td>
<td>#0xxxx</td>
<td>7897986</td>
<td>5087259</td>
<td>64.41%</td>
</tr>
<tr>
<td>8</td>
<td>#1xxxx</td>
<td>7897986</td>
<td>5087259</td>
<td>64.41%</td>
</tr>
<tr>
<td>9</td>
<td>##xxxx</td>
<td>7897986</td>
<td>3886488</td>
<td>49.21%</td>
</tr>
</tbody>
</table>

Mer classifiers are more consistent in terms of reward predictions than the latter ones. A classifier’s accuracy relates to the consistency of its reward prediction, so a classifier having consistent action value will be more accurate than a classifier having inconsistent action value. Therefore, the classifiers having condition parts from the schema “AAxxxx” are more accurate than the other classifiers having condition parts from the schema “#Axxxx”, “A#xxxx”, and “##xxxx”. In XCS-based systems, the fitness of a classifier depends on its accuracy of reward prediction, therefore, the classifiers having specific address bits in conditions have higher fitness values than the classifiers having one or more ‘#’ symbols in the address bits.

In XCS-based systems, the reproduction is niche based, i.e. the rule discovery operation is applied to the classifiers participating in the action set instead of applying it to the whole classifiers population and according to Wilson [118]:

“...within a given action set, the more accurate classifiers will have higher fitnesses than the less accurate ones. They
will consequently have more offspring. But by becoming relatively more numerous, those classifiers will gain a larger fraction of the total relative accuracy (which always equals 1) and so will have yet more offspring compared to their less accurate brethren. Eventually, the most accurate classifiers in the action set will drive out the others, in principle leaving the \( X \times A \Rightarrow P \) map with the best classifier (assuming the GA has discovered it) for each situation-action combination.

When condensed using the specifically designed condensation algorithm, described in Section 5.7.1, these classifiers resulted in the compact optimum solutions containing maximally general and accurate classifiers for the 6-bit to the 37-bit multiplexer problems.

### 5.8 Chapter Summary

The main goal of the work presented in this chapter was to investigate the standard XCS and code-fragment based XCSCFA systems in learning different complex Boolean problems, especially overlapping and niche imbalance problems. The obtained experimental results indicate that XCS has strong bias against the overlapping rules and it produced overgeneral classifiers for overlapping and niche imbalance problems. To learn such problems, it is beneficial to either deactivate the action set subsumption or use a relatively high subsumption threshold and a small error threshold so that the less general, but necessary classifiers covering small niches are retained in the population.

The XCSCFA systems successfully solved overlapping and niche imbalance problems because of the inconsistent actions and especially the redundancy provided by the GP-like code-fragment actions. \( \text{XCSCFA}_e \) produced more generalized classifiers, but \( \text{XCSCFA}_v \) has the advantage of producing the optimal classifiers separated from the sub-optimal ones in
certain domains. The evolved classifiers using XCSCFAe are easy to interpret, as are in standard XCS, whereas in XCSCFAe they are slightly harder to interpret as their actions now depend on the environmental state.

In the next chapter, the code-fragment action approach is extended with cyclic graphs to encapsulate repeated patterns in the problem, which resulted in evolving general solutions of any scale $n$ for a number of important problems, e.g. parity problems.
Chapter 6

Extending Learning Classifier Systems with Cyclic Graphs for Scalability on Complex, Large-Scale Boolean Problems

6.1 Introduction

Evolutionary computation techniques have had limited capabilities in solving large-scale problems due to the large search space demanding large memory and long training times. Previously, we extended XCS with code-fragment conditions in XCSCFC, see Chapter 4, which has the ability to reuse the extracted domain knowledge. Although XCSCFC has shown scalability beyond standard XCS [118], the solutions found were not scalable in general at the domain level due to the increasingly large tree structures as the problem scales. Subsequently, we implemented the code-fragment encoding scheme in the action of a classifier rule in XCSCFA, see Chapter 5, which solved various complex Boolean problems, but could not evolve scalable solutions. An alternative approach, self-modifying carte-
sian genetic programming (SMCGP), can provide general solutions to a number of problems, but the obtained solutions for large-scale problems are not easily interpretable [40].

A limitation in these techniques is the lack of a cyclic representation to encapsulate the underlying repeated patterns in a problem domain. In principle they could have cycles, but it is difficult to avoid infinite loops in the evolved solutions. A finite state machine (FSM) is a cyclic representation, which has the ability to encapsulate repeated patterns in a problem domain and does not stuck in infinite loops. However, the evolution of FSMs is a hard task due to the combinatorially large number of possible states, connections and interaction. Usually this requires supervised learning to minimize inappropriate FSMs [53, 82], which for large-scale problems necessitates subsampling and/or incremental testing. To avoid these constraints, this work introduces a state-machine based encoding scheme into XCS for the first time.

The proposed system is to be tested on six different Boolean problem domains, i.e. even-parity, majority-on, count ones, digital design verification, carry, and multiplexer problems. These are complex problem domains having overlapping, niche imbalance, and epistatic properties (see Section 3.3). The results are to be compared with standard numeric action based XCS and SMCGP to test the scalability of the proposed system.

The rest of the chapter is organised as follows. Section 6.2 describes the necessary background in finite state machines. In Section 6.3 the novel implementation of XCS using state-machine action is detailed. In Section 6.4 experimental results are presented and compared with standard numeric action-based XCS. Section 6.5 describes the effect of redundant states in a state-machine action. Section 6.6 is a comprehensive discussion explaining the effectiveness of the system resulting from the combination of XCS and FSMs. In the last section, this work is concluded and the future work is outlined.
6.2 Finite State Machines

A finite state machine (FSM), also known as finite state automaton (FSA), is a mathematical model of computation that can be used to model any finite-state system. FSMs have been used to design sequential logic circuits as well as algorithms for different computational tasks such as pattern matching, sequence prediction, communication protocols, and parsing [6]. In general, a finite state machine consists of a set of finite states and can be in only one state at any given time, called the current state. On receiving an input, the machine can change its current state and/or cause an action or output to take place for any given change. One of the states is labeled as start state, which is used as the current state in the beginning while processing the input.

There are many state machine modeling techniques, e.g. deterministic finite automata (DFA), non-deterministic finite automata (NFA), Mealy machines, Moore machines, pushdown automata, and Turing machines [6]. The state machine used in the work presented here is the Moore machine [87], as it is simple and suitable for modeling classification problems. A Moore machine is formally defined as a six-tuple \( M = (Q, \Sigma, \Delta, \delta, \lambda, q_{\text{initial}}) \) where \( Q \) is a finite set of states, \( \Sigma \) is a finite set of input symbols, \( \Delta \) is a finite set of output symbols, \( \delta \) is a transition function from a state \( p \in Q \) and an input \( x \in \Sigma \) to a next state \( q \in Q \), \( \lambda \) is an output function from a state \( p \in Q \) to an output \( y \in \Delta \), and \( q_{\text{initial}} \in Q \) is the start state.

For example, the state machine shown in Figure 6.1 is described as \( Q = \{q_0, q_1, q_2\} \), \( \Sigma = \{0, 1\} \), \( \Delta = \{0, 1\} \), start state = \( q_0 \), \( \lambda \) defined as: \( \lambda(q_0) = 1 \), \( \lambda(q_1) = 1 \), and \( \lambda(q_2) = 0 \), and \( \delta \) defined as: \( \delta(q_0, 0) = q_2 \), \( \delta(q_0, 1) = q_1 \), \( \delta(q_1, 0) = q_1 \), \( \delta(q_1, 1) = q_0 \), \( \delta(q_2, 0) = q_1 \), and \( \delta(q_2, 1) = q_2 \). Here a circle denotes a state along with the corresponding output value, an arrow represents a transition for a given input symbol, and the start state is marked with an unconnected input arrow.

Usually, a Moore machine outputs a string of symbols from \( \Delta \) on re-
CHAPTER 6. EXTENDING XCS WITH CYCLIC GRAPHS

Figure 6.1: A Moore state machine with three states $q_0$, $q_1$, and $q_2$ where $q_0$ is the start state.

Receiving an input string of symbols from $\Sigma$, e.g. if the input is ‘10100’ then the Moore machine depicted in Figure 6.1 will produce ‘11101’ as the output string. In the work presented here, it is adapted for classification problems. To use a Moore machine for classification, the value of the last state visited while processing the input string is taken as the only output, instead of a whole output string. Therefore the class of input string ‘10100’ will be 1 whereas the string ‘10101’ will belong to class 0, if processed via the machine shown in Figure 6.1.

In the early 1960s, Fogel et al. [34] evolved state machines using evolutionary programming to predict symbol sequences. Subsequently, different evolutionary computation approaches have been used to evolve FSMs such as genetic algorithm [53], genetic programming [12], evolutionary algorithms [103], and hill climber [82]. All of these evolutionary techniques produce a state-machine as a ‘single’ solution, rather than a co-operative set of rules as in an LCS. They generally require supervised learning with the whole training set, rather than online, reinforcement learning as in LCS.

6.3 XCS with Finite-State-Machine Actions

In the work presented here, the typical static numeric action in XCS is to be replaced by an FSM in an attempt to develop a general scalable learning classifier system having concise and easily interpretable classifier rules.
6.3. XCS WITH FINITE-STATE-MACHINE ACTIONS

In the proposed approach of XCS with state-machine actions, called XCSSMA, the static binary action is replaced by a Moore state machine [87] retaining the ternary alphabet in the condition of a classifier rule. Each state machine consists of \( n \) states, where some of the states may be deactivated in order to provide flexibility in terms of the required number of states. Each state \( p \in Q \) is encoded as a four-tuple \( \{m, v, active, T\} \), where \( 0 \leq m < n \) is the unique state identification number, \( v \in \Delta \) is the output value of the state \( p \), \( active \) is a Boolean flag determining whether the state \( p \) is activated, and \( T \) denotes transitions from state \( p \) to a next state \( q \in Q \) for each input symbol \( x \in \Sigma \).

A state machine is to be encoded as a string, which is a combination of each state’s encoding. For simplicity, the first state in each machine’s encoding is set to be the start state, but the order of other states does not matter. For example, the state machine shown in Figure 6.2, where a deactivated state \( q_1 \) is represented by a dashed circle, can be encoded as ‘21110 − 00102 − 11001’, where the states are concatenated in the order \( q_2, q_0, q_1 \). Here \( Q = \{q_0, q_1, q_2\}, \Sigma = \{0, 1\}, \Delta = \{0, 1\}, startstate = q_2, \lambda \) is defined as: \( \lambda(q_0) = 0, \lambda(q_1) = 1 \), and \( \lambda(q_2) = 1 \), and \( \delta \) is defined as:

\[
\begin{align*}
\delta(q_0, 0) &= q_0, & \delta(q_0, 1) &= q_2, & \delta(q_1, 0) &= q_0, & \delta(q_1, 1) &= q_1, & \delta(q_2, 0) &= q_1, & \delta(q_2, 1) &= q_0.
\end{align*}
\]

The string ‘21110 − 00102 − 11001’, representing the state machine shown in Figure 6.2, is explained in Table 6.1.

![Figure 6.2: A Moore state machine with a deactivated state \( q_1 \). This machine can be encoded as ‘21110 − 00102 − 11001’.](image)

If a state \( q_j \) has been deactivated and an activated state \( q_i \) has a transition to \( q_j \), then that transition will be changed to any activated state
Table 6.1: The description of a state machine encoded by the string ‘21110–00102 – 11001’.

<table>
<thead>
<tr>
<th>An encoded state ‘p’</th>
<th>Description of the state ‘p’</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>λ(p)</td>
</tr>
<tr>
<td>21110</td>
<td>2</td>
</tr>
<tr>
<td>00102</td>
<td>0</td>
</tr>
<tr>
<td>11001</td>
<td>1</td>
</tr>
</tbody>
</table>

in the machine, chosen uniformly randomly, as suggested by Spears and Gordon-Spears [103], e.g. in Figure 6.2 the transition δ(q₂, 0) = q₁ will be set to δ(q₂, 0) = q₀ or δ(q₂, 0) = q₂. To avoid the creation of any useless machine in the evolutionary process, the start state will always be activated.

The proposed XCSSMA approach extends standard XCS, described in Section 2.4, in the following aspects: the action value, the covering operation, the rule discovery operation, the procedure comparing equality of two state-machine actions, and the subsumption deletion mechanism. The rest of this section describes these extensions.

### 6.3.1 State-Machine Action Value

The action value of a classifier is determined by processing the current input string s via the state-machine action in the classifier. The processing starts from the start state in the state-machine action, and the value of the last state visited is taken as the action value. For example, consider the classifier shown in Figure 6.3. If the input string s is ‘100101’ then the action value will be 0.

### 6.3.2 Covering Operation

Covering occurs if an action a is missing in the match set [M]. If so, a random classifier is created whose condition matches the current environ-
6.3. XCS WITH FINITE-STATE-MACHINE ACTIONS

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
</table>
| 10#10#    | ![Diagram of state-machine action](image)

Figure 6.3: A classifier rule with state-machine action.

mental state $s$ and contains ‘#’ symbols with probability $P_#$. The state-machine action is randomly generated until its output is $a$. The covering operation is described in Algorithm 12. Here $n$ is the length of condition $\text{cond}$ in a classifier rule, and $P_#$ is the probability of the ‘don’t care’ symbol ‘#’ in condition of the newly created classifier in the covering operation.

### 6.3.3 Rule Discovery Operation

In the rule discovery operation, a GA is applied in the action set $[A]$ to produce two offspring. First of all, two parent classifiers are selected from $[A]$ based on fitness and the offspring are created from them. Next, the conditions and state-machine actions of the offspring are separately crossed over, with probability $\chi$, by applying the GA crossover operation. It is to be noted that the start states may be swapped during the crossover operation, but the resulted machines by crossover should not contain duplicate states. The crossover operation is described in Algorithm 13. Here $n$ is the length of condition $\text{cond}$ and $\text{len}$ is the length of state-machine action $\text{action}$ in a classifier rule.

After that, each symbol in the conditions of the resulted children by crossover are mutated with probability $\mu$, such that both children match the currently observed state $s$. In the mutation operation, a ‘non-don’t care’ symbol in classifier condition is replaced by the ‘don’t care’ symbol ‘#’, and a ‘don’t care’ symbol is replaced by the corresponding symbol in the input message. Then, the state-machine actions of the children are
Algorithm 12: XCSSMA: Covering Operation

Data: The currently observed input state $s$ and an action $a$ missing in the match set $[M]$.

Result: A newly created classifier $cl$ matching the state $s$ and advocating the action $a$.

1. initialize classifier $cl$
2. initialize condition $cl.cond$ with length $n$
3. for $i = 1$ to $n$ do
   4. if RandomNumber(0, 1) $< P\#$ then
      5. $cl.cond[i] \leftarrow \#$
   6. else
      7. $cl.cond[i] \leftarrow s[i]$
   8. end
4. end
10. repeat
11. $sm \leftarrow$ randomly create state-machine
12. $val \leftarrow$ evaluate value of $sm$
13. until $val = a$
14. $cl.action \leftarrow sm$
15. return $cl$

mutated with probability $\mu$. It is to be noted that state numbers are not mutated in order to avoid duplicate states in a state-machine action. The mutation operation is described in Algorithm 14. Here $n$ and $len$ are the length of condition $cond$ and state-machine action $action$ in a classifier rule respectively, and $\mu$ is the mutation probability.

The prediction and prediction error of the offspring are set to the average of the parents’ values whereas the fitness of the offspring is set to the average of the parents’ values multiplied by the constant $fitnessReduction$, as suggested by Butz and Wilson in [28], to prevent inexperienced classifier rules being chosen in exploitation.
6.3. **XCS WITH FINITE-STATE-MACHINE ACTIONS**

Algorithm 13: XCSSMA: Crossover Operation

**Data:** Two offspring classifiers $cl_1$ and $cl_2$.

**Result:** The crossed over classifiers $cl_1$ and $cl_2$.

```plaintext
1 x ← RandomNumber[0, n)
2 y ← RandomNumber[0, n)
3 if x > y then
4    swap x and y
5 end
6 for i = x to y do
7    swap $cl_1$.cond[i] and $cl_2$.cond[i]
8 end
9 x ← RandomNumber[0, len)
10 y ← RandomNumber[0, len)
11 if x > y then
12    swap x and y
13 end
14 for i = x to y do
15    swap $cl_1$.action[i] and $cl_2$.action[i]
16 end
17 return $cl_1$ and $cl_2$
```

6.3.4 Comparing Two State-Machine Actions

If a newly created classifier in the rule discovery operation is not subsumed and there is no classifier equal to it in the population, then it will be added to the population. Two classifiers are considered to be equal if and only if both have the same conditions and genotypically the same state-machine in their actions. The procedure to compare two state-machine actions for equality is given in Algorithm 15. Here $len$ is the length of state-machine action $action$ in a classifier rule.
Algorithm 14: XCSSMA: Mutation Operation

**Data:** The currently observed input state \( s \) and an offspring classifier \( cl \) matching the state \( s \).

**Result:** The mutated classifier \( cl \) which still matches the state \( s \).

1. for \( i = 1 \) to \( n \) do
2.   if \( \text{RandomNumber}[0, 1) < \mu \) then
3.     if \( cl.\text{cond}[i] = \# \) then
4.       \( cl.\text{cond}[i] \leftarrow s[i] \)
5.     else
6.       \( cl.\text{cond}[i] \leftarrow \# \)
7.   end
8. end
9. for \( i = 1 \) to \( \text{len} \) do
10. if \( \text{RandomNumber}[0, 1) < \mu \) then
11.   \( cl.\text{action}[i] \leftarrow \text{any other valid value} \)
12. end
13. end
14. return \( cl \)

### 6.3.5 Subsumption Deletion

A classifier \( cl_1 \) can subsume another classifier \( cl_2 \) if both have the same action and \( cl_1 \) is accurate, sufficiently experienced, and more general than \( cl_2 \) [28]. It is to be noted that due to the multiple genotypes to a single phenotype mapping of state-machine actions in classifier rules, subsumption deletion is less likely to occur. Subsumption deletion is still made possible by matching the state-machine descriptions on a character by character basis.
Algorithm 15: XCSSMA: Are_Equal_Actions

Data: Two state-machine actions $sm_1$ and $sm_2$.

Result: If the actions $sm_1$ and $sm_2$ are genotypically the same, then this algorithm will return true otherwise false.

1. for $i = 1$ to $len$ do
2.     if $sm_1[i] \neq sm_2[i]$ then
3.         return false
4.     end
5. end
6. return true

6.4 Results

In order to test the performance of XCSSMA, results have been compared with standard XCS on the six problem domains, i.e. even-parity, majority-on, count ones, digital design verification, carry, and multiplexer problems. The evolved solutions have also been compared with that of SMCGP for even-parity and carry problems.

The number of training examples used is two million in all the experiments conducted here. Explore and exploit problem instances are alternated. The reward scheme used is 1000 for a correct classification and 0 otherwise. Both GA subsumption and action set subsumption are activated. In XCSSMA, input alphabet $\Sigma = \{0, 1\}$; output alphabet $\Delta = \{0, 1\}$ and the number of states $n$ is empirically set to 5. All the experiments have been repeated 30 times with a known different seed in each run. Each result reported in this work is average of the 30 runs.

In all graphs presented here, the X-axis is the number of problem instances used as training examples, the Y-axis is the performance measured as the percentage of correct classification during the last 1000 exploit problem instances, and the error bars show the standard deviation in the 30 runs.
6.4.1 The Even-Parity Problem Domain

The largest solved parity problem, directly from training data, reported in literature is the 24-bit even-parity problem, by Harding et al. [40] using SMCGP. The performance of standard XCS and XCSSMA in learning the 24-bit even-parity problem is shown in Figure 6.4. The number of classifiers used is $N = 2000$. It is observed that standard XCS cannot learn the 24-bit even-parity problem, whereas XCSSMA solved it using approximately 15,000 training examples.

![Figure 6.4: Results of the 24-bit even-parity problem.](image)

The even-parity problem domain does not allow generalizations if the standard ternary alphabet based encoding scheme is used with static numeric action. So in XCS each bit must be specific for a rule to be accurate, requiring $2^{25}$ such rules for the 24-bit even-parity problem to produce the complete map. The standard XCS technique was not able to evolve enough accurate rules, even using a larger population of classifiers. However, the proposed XCSSMA approach not only solved the 24-bit even-parity problem, but the obtained solutions are compact, easily
understandable\footnote{Assuming the reader understands how state machines work.}, and general for any \(n\)-bit even-parity problem.

One of the classifier rules from the final solution obtained using XCSSMA is shown in Figure 6.5(a). This is a maximally general and accurate classifier covering the whole problem space. The FSM action in this rule is general to solve any \(n\)-bit even-parity problem. It is to be noted that state \(q_0\) is not active, so there is no transition from any active state to this deactivated state. The state \(q_1\) and state \(q_4\) are active, but not reachable from the start state \(q_3\). It means only two states, i.e. \(q_2\) and \(q_3\), are the working states in this FSM action. The deactivated and non-reachable states can be removed to simplify state-machine actions in the final solution if needed, as shown in Figure 6.5(b).

The solutions found by Harding et al. [40] using SMCGP were also general, but not as compact and easily interpretable as the classifier rules obtained using XCSSMA here. For example, an evolved program using SMCGP is shown in Figure 6.6 that provides a solution for the 4-bit even-parity problem after two iterations of the evolved program. Here \(x_0, x_1, x_2,\) and \(x_3\) represent input symbols and \(\oplus\) denotes XOR operator; INPP and OUTPUT are input and output nodes respectively; BAND, BOR, and BNAND are the Boolean operators; and DUP is a duplication operator that duplicates a section of the graph. To obtain the solution for the 24-bit even-parity problem, twenty more iterations will need to be performed [40]. One can imagine the resulting solution for the 24-bit even-parity problem, but it is impractical to draw it.

### 6.4.2 The Majority-on Problem Domain

It is considered that majority-on problems are hard to learn because the complete solution for a majority-on problem consists of overlapping classifiers. The largest attempted majority-on problem in literature is a 7-bit problem, by Jackson and Gibbons [64] using layered learning in genetic
CHAPTER 6. EXTENDING XCS WITH CYCLIC GRAPHS

Figure 6.5: A sample classifier rule from final solution of the 24-bit even-parity problem in XCSSMA.

Figure 6.6: An evolved genotype, using SMCGP [40], iterated twice that provides solution for the 4-bit even-parity problem. This figure is adapted by kind permission of Harding.
programming and the reported success rate is 90%. The performance of standard XCS and XCSSMA in learning the 7-bit majority-on problem is shown in Figure 6.7. The number of classifiers used is \( N = 3000 \). It is observed that standard XCS reached approximately 93% performance, but could not completely solve the 7-bit majority-on problem, whereas XCSSMA has solved it using approximately 15,000 training examples.

![Figure 6.7: Results of the 7-bit majority-on problem.](image)

A sample accurate classifier rule for the 7-bit majority-on problem is shown in Figure 6.8. This is an interesting rule covering 16 problem instances, 11 of which belong to class 0 and the other five are of class 1. This form of classifier rule is not expressible using the standard numeric action based XCS with ternary conditions. Standard rules only have one unique action, which remains unchanged regardless of input.

### 6.4.3 The Count Ones Problem Domain

The count ones problem used in this work is of length \( l = 20 \) with the first \( k = 7 \) relevant bits. The number of classifiers used is \( N = 4000 \). The performance of standard XCS and XCSSMA is shown in Figure 6.9. It is observed
that standard XCS reached approximately 95% performance, but could not completely solve the 7-bit count ones problem, whereas XCSSMA solved it using approximately 50,000 training examples.

A sample accurate classifier rule for the 7-bit count ones problem is shown in Figure 6.10. This rule covers 16 problem instances, 15 of which belong to class 0 and the other one is of class 1.
6.4. RESULTS

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Figure 6.10: A sample classifier rule from final solution of the 7-bit count ones problem in XCSSMA.

6.4.4 The Design Verification Problem Domain

The performance of standard XCS and XCSSMA in learning the DV1 problem is shown in Figure 6.11. The number of classifiers used is $N = 3000$. It is observed that standard XCS reached approximately 97% performance level, but could not completely solve the DV1 problem, whereas XCSSMA successfully solved it. Ioannides et al. [54] improved the performance of XCS in learning the DV1 problem to 99.76%, by modifying the standard fitness update procedure and using an individually computed learning rate for each classifier\(^2\), but could not completely solve it.

A sample accurate classifier rule for the DV1 problem is shown in Figure 6.12. This rule covers 32 problem instances, i.e. 0 to 7, 16 to 23, 32 to 39 and 48 to 55, 29 of which belong to class 0 and the other three are of class 1.

6.4.5 The Carry Problem Domain

The largest solved carry problem, directly from training data, reported in literature is the 6+6 bit carry problem, by Harding et al. [40] using SMCGP.

\(^2\)They also used a larger value for action set subsumption threshold, i.e. 100 instead of the commonly used value of 20.
Figure 6.11: Results of the DV1 problem.

The performance of standard XCS and XCSSMA in learning the 6+6 bit carry problem is shown in Figure 6.13. The number of classifiers used is \( N = 4000 \).

The complete solution in the carry problem domain consists of overlapping classifiers, in addition it is a niche imbalance domain, which makes it very difficult to learn. However, XCSSMA successfully learned the 6+6
6.4. RESULTS

Figure 6.13: Results of the 6+6 bit carry problem.

bit carry problem, whereas standard XCS failed. In addition, the obtained solutions in XCSSMA are compact, easily understandable, and general for any n+n bit carry problem.

One of the classifier rules from the final solution obtained using XCSSMA is shown in Figure 6.14(a). This is a maximally general and accurate classifier covering the whole problem space. The FSM action in this rule is general to solve any n+n bit carry problem. It is to be noted that state $q_0$ and state $q_3$ are not active, so there is no transition from any active state to these deactivated states. The state $q_2$ is active, but not reachable from the start state $q_1$. It means only two states, i.e. $q_1$ and $q_4$, are the working states in this FSM action. The simplified rule is shown in Figure 6.14(b).

The solutions found by Harding et al. [40] using SMCGP were also general, but not as compact and easily interpretable as the classifier rules obtained using XCSSMA here. For example, an evolved program using SMCGP is shown in Figure 6.15 that provides solution for the 3+3 bit carry problem after two iterations. Here $(x_0, y_0)$, $(x_1, y_1)$, and $(x_2, y_2)$ represent input-pair symbols; $c_0$ and $c_1$ denote the carry bits; INPP and OUTPUT are input and output nodes respectively; BF0, BF1, BF2, ..., BF15 are the 16
CHAPTER 6. EXTENDING XCS WITH CYCLIC GRAPHS

Figure 6.14: A sample classifier rule from final solution of the 6+6 bit carry problem in XCSSMA.

Boolean operators of two variables; and DUP is the duplication operator. To obtain the solution for the 6+6 bit carry problem, three more iterations need to be performed, resulting in a long chaining phenotype.

6.4.6 The Multiplexer Problem Domain

The results of standard XCS and XCSSMA for the 20-bit multiplexer problem are shown in Figure 6.16. The number of classifiers used is \( N = 2000 \). XCSSMA successfully solved it, but took more instances as compared with standard XCS to reach a similar performance level. The multiplexer is a niche balanced problem domain and there exists a complete solution
6.4. RESULTS

Figure 6.15: An evolved genotype, using SMCGP [40], iterated twice that provides solution for the 3+3 bit carry problem. This figure is adapted by kind permission of Harding.

for multiplexer problems that does not contain any overlapping classifier rules. So, standard XCS effectively solved the 20-bit multiplexer problem. The XCSSMA approach used more training examples, due to the increased search space as a result of the state-machine based actions.

Figure 6.16: Results of the 20-bit multiplexer problem.

A sample classifier rule for the 20-bit multiplexer problem is shown in Figure 6.17. This rule is equivalent to numeric action based XCS rule ‘0010##1############# : 1’. The states $q_0$, $q_2$, $q_3$, and $q_4$ are not reachable
from the start state $q_1$ in the FSM action of this rule. The processing of any matched input message ends at the state $q_1$ that has output value 1, therefore action value of this rule will be 1.

| Condition | 0010##1#|  |  |  |  |
|-----------|--------| |  |  |  |
| Action    |  |  |  |  |  |

Figure 6.17: A sample classifier rule from final solution of the 20-bit multiplexer problem in XCSSMA.

### 6.5 Effect of Redundant States in XCSSMA

The analysis of final solutions obtained using XCSSMA shows that state-machine actions in the evolved classifier rules contain redundant states in the form of deactivated and non-reachable states. These redundant states in a state-machine action do not take part in the calculation of the action value of the classifier. In order to investigate the effect of these redundant states in XCSSMA, one more set of experiments was conducted where the redundant states in a state-machine action were removed on creation during the training process, i.e. during rule discovery any redundancy was removed from a child prior to entering the population. The resulting XCSSMA method is named cleaned XCSSMA to differentiate it from the original simple XCSSMA method. The results are shown in Figure 6.18.

It is observed that the cleaned XCSSMA method failed three times out of the 30 runs to learn the 24-bit even-parity problem, see Figure 6.18(a).
6.5. EFFECT OF REDUNDANT STATES IN XCSSMA

(a) for the 24-bit even-parity problem.

(b) for the 7-bit majority-on problem.

(c) for the 7-bit count ones problem.

(d) for the DV1 problem.

(e) for the 6+6 bit carry problem.

(f) for the 20-bit multiplexer problem.

Figure 6.18: Effect of redundant states in XCSSMA.

In learning the 7-bit count ones problem, the performance of cleaned XCS-SSMA reached greater than 99%, but it could not achieve the stabilized 100% performance level of simple XCSCFA, as shown in Figure 6.18(c).
The learning performance of cleaned XCSSMA also decreased in the 7-bit majority-on problem and the DV1 problem, as shown in Figure 6.18(b) and Figure 6.18(d) respectively. The performance of cleaned XCSSMA slightly improved in learning the 6+6 bit carry problem and the 20-bit multiplexer problem, see Figure 6.18(e) and Figure 6.18(f) respectively.

It is hard to create a generalized state-machine for a multiplexer problem due to the inherent property of indexing to a certain position according to the address bits in the input problem. The net effect of any created state-machine action is just equivalent to a static numeric action, therefore, cleaned XCSSMA having less number of states performed better than simple XCSSMA in learning the 20-bit multiplexer problem. Similarly, a carry problem can be solved by a state-machine consisting of only two states, therefore, cleaned XCSSMA performed better than simple XCSSMA in learning the 6+6 bit carry problem. The ideal state-machine for an even-parity problem also consists of only two states, similar to a carry problem, but cleaned XCSSMA failed three times in learning the 24-bit even-parity problem. It was found that the state-machines needed for the other three problem domains, i.e. the majority-on, the count ones and the DV, prefer interactions between different states to solve the problem at hand. However, in cleaned XCSSMA the overall effect of the interaction is reduced due to the reduced number of states after simplification, therefore, the performance of cleaned XCSSMA decreased in learning the 7-bit majority-on, the 7-bit count ones and the DV1 problems.

It is observed that due to the multiple genotypes to a single phenotype mapping of state-machine actions in XCSSMA, the subsumption deletion mechanism is not fully enabled. This resulted in a greater number of macro-classifiers in the final solutions obtained using XCSSMA than that of obtained using XCS, except for the 24-bit even-parity problem, as shown in Table 6.2. For even-parity problems, no useful generalizations can be made using the ternary alphabet based conditions with the static numeric action, therefore, the number of macro-classifiers in the final solution of
the 24-bit even-parity problem obtained using XCS is greater than that obtained using XCSSMA. Removing the redundant states during the training process helps in reducing the number of macro-classifiers in the final solutions, see Table 6.2, but it decreased the learning performance of the system in certain problems. It is to be noted that the large standard deviation for the 24-bit even-parity problem using cleaned XCSSMA is because cleaned XCSSMA failed three times out of the 30 runs in learning the 24-bit even-parity problem. If only the 27 successful runs of cleaned XCSSMA in learning the 24-bit even-parity problem are considered, then the number of macro-classifiers along with standard deviation was 206 ± 123.

Table 6.2: The number of macro-classifiers, along with standard deviation, in the final solutions obtained using XCS and XCSSMA for different problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>XCS</th>
<th>Simple XCSSMA</th>
<th>Cleaned XCSSMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-bit even-parity</td>
<td>1734 ± 17</td>
<td>1013 ± 32</td>
<td>358 ± 478</td>
</tr>
<tr>
<td>7-bit majority-on</td>
<td>215 ± 17</td>
<td>1135 ± 78</td>
<td>936 ± 48</td>
</tr>
<tr>
<td>7-bit count ones</td>
<td>2464 ± 85</td>
<td>2546 ± 51</td>
<td>1785 ± 65</td>
</tr>
<tr>
<td>DV1</td>
<td>177 ± 16</td>
<td>1258 ± 88</td>
<td>925 ± 46</td>
</tr>
<tr>
<td>6+6 bit carry</td>
<td>485 ± 70</td>
<td>1970 ± 64</td>
<td>870 ± 170</td>
</tr>
<tr>
<td>20-bit MUX</td>
<td>387 ± 16</td>
<td>1115 ± 25</td>
<td>472 ± 58</td>
</tr>
</tbody>
</table>

The obtained results indicate that it is better to keep the deactivated and non-reachable states during the training process because they provide additional genetic material that can be used in the rule discovery operation to evolve potentially good classifiers. These redundant states can be removed from the final solution once the training process has been completed in order to simplify the state-machine actions.

Further investigation on the effect of redundant states in a state-machine action having a large number of states is an area for future work. To obtain a compact solution, having a smaller number of macro-classifiers, a mechanism to compare two state-machines semantically, instead of syn-
tactically as present, is needed so that the subsumption deletion process is fully enabled in XCSSMA. Lin et al. [80] recently published an initial work on semantic comparison of two rich encoded classifiers.

6.6 Further Discussions

The developed system, XCSSMA, can be viewed from two different perspectives. First, it can be regarded as a system to evolve finite state machines. If a general state machine $M$ can be created for the problem at hand with the given machine configuration parameters, then XCSSMA produces a rule of the form ‘###...#: $M’$. This rule will cover the whole problem space and classify each input instance accurately using the machine $M$ as the action, e.g. the classifier rules shown in Figure 6.5 and Figure 6.14 for the 24-bit even-parity problem and the 6+6 bit carry problem respectively. Second, if a single general FSM cannot be created for the problem, then XCSSMA behaves like a typical XCS and evolves a set of FSM rules that collectively solve the problem, e.g. in the 7-bit majority-on, the 7-bit count-ones, the DV1, and the 20-bit multiplexer problems.

An FSM is an abstract model that can represent a finite-state system in a compact form, but the evolution of FSMs is a hard task. Usually, FSMs are evolved using supervised learning so for large scale problems some form of subsampling and/or incremental testing is needed [12, 82, 103]. The online learning, niche based breeding, and generalization properties of XCS-based systems implicitly provide incremental testing and subsampling of the training data set. Hence, the developed XCSSMA system, as a combination of XCS and FSMs, rapidly evolved the general FSMs for even-parity and carry problems, and a set of FSMs for the other problems experimented here where each evolved machine covers a subspace of the problem matched by the corresponding classifier condition.

It is to be noted that using an FSM as the action of a classifier rule, in place of a static numeric action, the size of the search space increases.
This was compensated by improving the generalization ability of standard XCS in the even-parity, carry, majority-on, count ones, and DV problem domains. This improvement is obvious in the accurate rules shown in Figure 6.5, Figure 6.14, Figure 6.8, Figure 6.10, and Figure 6.12 that match all the problem instances from the 24-bit even-parity, all the problem instances from the 6+6 bit carry, 16 problem instances from the 7-bit majority-on, 16 problem instances from the 7-bit count ones, and 32 problem instances from the DV1 problems, respectively. The generalization to this level is beyond the ability of standard XCS using ternary alphabet based conditions along with a numeric action.

The FSM based action could not improve the generalization beyond numeric action based XCS for the multiplexer domain because the state machines needed for this domain are more complex than the other domains. The inherent property of indexing to a certain position according to the address bits in the input problem makes the creation of a state machine difficult in the multiplexer domain. So in the multiplexer problem domain, XCSSMA takes more training examples, due to the increased search space, to reach a performance level similar to standard XCS. It is to be noted that the code-fragment based XCS techniques [57, 59, 62] can scale in the multiplexer domain beyond the 20-bit MUX, and using XCS with code-fragment conditions [62] up to the 135-bit MUX have been solved. It is anticipated that a memory component is needed in order to evolve general state machines for multiplexer problems.

In summary, if a proper representation scheme is used in XCS to encode the classifier rules, then the system can evolve maximally general and accurate classifiers, possibly just one classifier rule covering the whole problem space, e.g. XCSSMA for even-parity and carry problems here and a code-fragment based XCS for the frog problem in [58]. It is also possible that the obtained solutions will be general to solve any problem from the domain, e.g. the solutions obtained in XCSSMA for even-parity and carry problems.
The suite of scalable-XCS techniques and SMCGP are very promising systems that use rich encoding schemes to address the problem of scalable learning, but in different ways. SMCGP produces an individual as a ‘single’ solution, whereas the XCS classifier systems may evolve a co-operative set of rules. SMCGP generally requires supervised learning with the whole training set, rather than online, reinforcement learning as in an XCS-based classifier system. Scalable-XCS can produce more compact solutions, but needs human selection for encoding schemes, i.e. code fragments or FSMs. SMCGP is more flexible as it evolves a computer program that can generate a sequence of programs, each of which solves a particular problem in the domain. However, XCS can divide up the problem space using co-operative classifiers, which is advantageous in certain domains.

### 6.7 Chapter Summary

The proposed XCSSMA technique successfully solved six difficult Boolean problems, i.e. even-parity, majority-on, count ones, DV, carry, and multiplexer problems. Further XCSSMA evolved, for the first time, compact and easily interpretable general classifiers for the even-parity and carry problem domains. The XCSSMA technique is expected to be most useful when a problem domain contains cyclic regularity in the input states that is useful for deciding the answer to the problem.

In the work so far presented in this thesis, only binary problems were tested. In the next chapter, the code-fragment action based approach is adopted in real-valued problems where the action is a continuous function of the input.
Chapter 7

Computing Continuous Actions in Learning Classifier Systems

7.1 Introduction

Wilson extended XCS [118] with interval-based conditions to XCSR [120] to handle real-valued inputs. However, the possible actions must always be determined in advance. Usually the possible number of actions is known in an LCS, but for problems requiring continuous real-valued outputs, it is not possible. Therefore discrete action based systems like XCS and XCSR cannot be applied to such problems. A solution is to use a generalized classifier system (GCS) [123] in which the input $x$ is linked to the action $a$ in the process of matching the environmental instance. The difficulty of GCS is to evolve a condition form $t(x, a)$ so that the action is continuous. This limitation moves LCS from mapping input to output to directly calculating the output from the input.

Tran et al. [111] implemented XCSF [122] having computed continuous actions, named as XCSFCA, where the action is computed directly as a linear combination of the input state and a vector of action weights. XCSFCA has outperformed GCS and produced very good results for the continuous actions frog problems, but could not achieve 100% performance [111].
In this chapter, a new approach will be investigated in which the discrete action in XCSR is replaced by a code fragment. A code fragment is a tree-expression, similar to a tree generated in GP. The action value of a classifier rule is determined by loading the terminal symbols in the code-fragment action with the corresponding values from the environmental input. Thus the action is continuous with respect to the input state in this version of XCSR, named XCSRCFA (XCSR with Code-Fragment Actions). XCSRCFA will be examined and compared with GCS and XCSFCA on the frog problems. In addition to this, XCSRCFA will also be tested on learning various function approximation problems.

The rest of the chapter is organised as follows. Section 7.2 presents the novel implementation of XCSR using code-fragment actions. In Section 7.3 experimental results are presented and compared with GCS and XCSFCA. Section 7.4 is a discussion about the worth of XCSRCFA and its limitation. In the last section this work is concluded and the future work is outlined.

7.2 **XCSR with Code-Fragment Actions (XCSRCFA)**

In the work presented in this chapter, the discrete action in XCSR is replaced by a continuous action represented as a code fragment. Each code fragment is a binary tree of depth up to \( d \), which depends upon the complexity of the problem domain. The function set for the tree is problem dependent such as \{+, −, *, /\} for simple symbolic regression problems. The terminal set is \{ERC, D0, D1, D2, ..., Dn-1\} where ERC is an ephemeral random constant and \( n \) is the length of the input message.

The proposed XCSRCFA approach extends standard XCSR, described in Section 2.4.2, in the following aspects: the action value, the match set creation, the rule discovery operation, the procedure comparing equality of two classifier rules, and the subsumption deletion mechanism. The rest of this section describes these extensions.
7.2. XCSR WITH CODE-FRAGMENT ACTIONS (XCSRCA)

7.2.1 Code-Fragment Action Value

The action value of a classifier is determined by evaluating the code-fragment action. The code-fragment action is evaluated by replacing the terminal symbols with the corresponding values in the currently observed real-valued input message. Thus the action is continuous with respect to the input state in XCSRCA. For example, consider the code-fragment action shown in Figure 7.1. The internal nodes of the tree are functions and leaves are the terminals. In this code-fragment action, 1.0 and 2.0 are ERCs and D0 is the environmental input. If the input value of D0 is 0.4 then the action value will be 1.8 for this code-fragment action.

```
  +
 /\  /
|  |  |
1.0* D0 2.0
```

Figure 7.1: A sample code-fragment action.

7.2.2 Match Set Creation

A classifier will become a member of the match set $[M]$ if its condition part matches the input as usual and its computed action belongs to the allowed action range. If the match set $[M]$ is empty then a new covered classifier is generated with a random code-fragment action that outputs an allowed action value against the current environmental input state $s$.

It is to be noted that as the number of available actions is not known in advance, the prediction array is created dynamically according to the
number and different advocated action values of classifiers in the current match set.

7.2.3 Rule Discovery Operation

In the rule discovery operation, two offspring are created using the following steps. First of all, two parent classifiers are selected from the action set $[A]$ based on fitness and the offspring are created from them. Next, the conditions and action trees of the offspring are crossed with probability $\chi$ by applying GA- and GP-based crossover operations respectively. After that the conditions of the crossed over children are mutated as in standard XCSR, see Section 2.4.2. Then, the action trees of the children are mutated with probability $\mu$, using GP-based mutation, by replacing a node with a corresponding randomly generated new node.

The prediction and prediction error of the offspring are set to the average of the parents’ values whereas the fitness of the offspring is set to the average of the parents’ values multiplied by the constant $\text{fitnessReduction}$, as suggested by Butz and Wilson in [28].

7.2.4 Comparing Two Classifier Rules

Two classifier rules are considered to be equal if and only if both have the same conditions and the genotypically same code fragment in their actions. The code fragment genotype is its formal expression as seen in the classifier action, and the phenotype is the value that the action computes with a given input.

7.2.5 Subsumption Deletion

A classifier $cl_1$ can subsume another classifier $cl_2$ if both have the same action and $cl_1$ is accurate, sufficiently experienced, and more general than
7.3. RESULTS

It is to be noted that due to the multiple genotypes to a single phenotype mapping of code-fragment actions in classifier rules, subsumption deletion is less likely to occur in XCSRCA than standard XCS. Subsumption deletion is still made possible by matching the code-fragment actions on a character by character basis.\(^1\)

7.3 Results

The main problem domain experimented in this chapter is the frog problem domain (see Section 3.2), which has continuous action as well as continuous payoff and is a benchmark in the field of LCS. In addition to the frog problems, the XCSRCA system has also been tested in learning function approximation problems.

All the parameter values used for experimentation in this chapter are the same as described in Section 3.3, except the followings: \(r_0 = 0.1\); \(m_0 = 0.1\); \(\epsilon_0 = 0.01\); initial prediction \(p_I = 0.01\); uniform crossover; the tree depth \(d = 2\); the range of ERC is \([-2, 2]\) and the function set is \{+, -, *, /\}. The division operator ‘/’ works as usual except for the following two cases: if \(x_1\) is equal to \(x_2\) then \(x_1/x_2 = 1\), else if \(x_2\) is equal to 0 then \(x_1/x_2 = 0\). Explore and exploit problems are alternated with probability 0.5. All the experiments have been repeated 30 times with a known different seed in each run.

7.3.1 The Frog Problems

The allowed action range for the frog problems is [0,1]. The number of micro classifiers used is \(N = 2000\).

\(^1\)Initial work by Lin et al. [80] to compare two code-fragments phenotypically is promising, but it is currently limited to small-scale Boolean problems only.
The Frog1 Problem

The results in learning the frog1 problem using XCSRCFA are shown in Figure 7.2. One run is stopped after 100,000 explore problem instances. The system error measures the difference between the prediction of the expected payoff and the payoff received. The payoff and system error curves are plotted by using a 50-point running average from exploit problem instances. In each problem of an experiment, the fly was placed at a random distance \( d \) (\( 0.0 \leq d \leq 1.0 \)) from the frog.

Figure 7.2: Results in learning the frog1 problem in terms of payoff, system error and population size using XCSRCFA (curve order same as in legend).

The performance of GCS in learning the frog1 problem was volatile [123]. Although the performance did rise quickly to receive payoff greater than 0.95, it did not reach optimal performance (payoff 1.0). The system error curve approximately complements the payoff curve, so the system error for GCS was approximately 0.05 throughout the learning process. The population size curve for GCS rose to about 70% of \( N \) and then declined very gradually. The performance of XCSFCA in learning the frog1 problem was greater than 99% after an averaged number of 30,000 problem
instances, but could not achieve stabilized 100% performance [111]. The system error dropped to smaller than 1%. The population size of classifiers was about 37% of \( N \). The payoff curve in Figure 7.2 shows that the performance of XCSRCFA reached 100% after an averaged number of 45,000 problem instances and did not decline after that. The population size of classifiers is about 29% of \( N \).

Preen et al. [98] conducted experimentation on the frog1 problem using fuzzy dynamical genetic programming in XCSF and reported greater than 99% performance after an averaged number of 4,000 problem instances, but could not achieve stabilized 100% performance. The reported population size of classifiers is around 8% of \( N \).

If the frog wants to catch the fly at \( d \), it must jump a distance \( a = d = 1 - x \). Since \( d \) is generated randomly from [0,1], the sensory input \( x \) received by Equation 3.1 gives the value from [0,1]. As a direct indication of the system’s ability to choose the best action \( a^* \) and to gauge \( a^*’s \) continuity with respect to \( x \), at the end of each run \( x \) was scanned from 0 to 1 with increment 0.001 and the resulting \( a^* \) plotted in Figure 7.3. The plot for GCS lay close to the diagonal but had discontinuities [123]. The plot for XCSFCA was nearly coincident with the diagonal ‘1 – \( x \)’, but was slightly broken at some inputs \( x \). The authors have not explain the reason why the graph had gaps in XCSFCA. It may be due to no matching classifiers at some inputs or the computed actions lay outside the valid action range. The plot for XCSRCFCA, shown in Figure 7.3, is exactly the diagonal with no broken points.

XCSRCFCA evolved maximally general and accurate classifiers in learning the frog1 problem by exploiting the generalization ability of LCS (to create a maximally general condition) and the rich GP-based representation (to create the corresponding accurate action in the classifier). The experienced (i.e. \( \exp \geq 1/\beta \)) and accurate (i.e. \( \epsilon \leq \epsilon_0 \)) classifier rules from the final population of a run are shown in Table 7.1. The code-fragment actions are shown in the postfix form, where \( D0 \) represents the sensory input...
$a^\ast$ in learning the frog1 problem using XCSRCFA.

$F$, $n$, $p$, $\epsilon$, and $exp$ denote fitness, numerosity, prediction, prediction error, and experience of a classifier respectively, and $setSize$ denotes the average size of the action sets this classifier has belonged to.

XCSRCFA successfully solved the frog1 problem evolving maximally general and accurate classifier rules like $9 - 16$ and $19 - 23$, shown in Table 7.1, that have the optimal payoff value of 1.0. Each of these maximally general classifier covers the whole input space ranging from 0.0 to 1.0 and has the accurate action that is equivalent to the diagonal ‘1 – $x$’, therefore each of them is able to solve the frog1 problem individually. However, due to the multiple genotypes to a single phenotype issue, XCSRCFA could not combine the phenotypically similar classifiers, like $9 - 16$ and $19 - 23$, into a single macro-classifier while evolving the population. So, the final population contains redundant classifier rules.

The average size of the action sets, denoted by $setSize$, classifiers $1 - 8$ and $17 - 18$ have belonged to is much smaller than that of classifiers $9 - 16$.
Table 7.1: The experienced (i.e. \( \text{exp} \geq 1/\beta \)) and accurate (i.e. \( \epsilon \leq \epsilon_0 \)) classifier rules, obtained in a typical run for the frog1 problem, using XCSRCFA.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Condition</th>
<th>Action</th>
<th>( F )</th>
<th>( n )</th>
<th>( p )</th>
<th>( \epsilon )</th>
<th>( \text{exp} )</th>
<th>( \text{setSize} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.465, 0.595</td>
<td>D0 D0 *</td>
<td>0.907</td>
<td>10</td>
<td>0.922</td>
<td>0.006</td>
<td>52</td>
<td>63</td>
</tr>
<tr>
<td>2</td>
<td>0.000, 0.064</td>
<td>D0 2.0 /</td>
<td>0.893</td>
<td>4</td>
<td>0.023</td>
<td>0.006</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0.134, 0.280</td>
<td>D0 D0 /</td>
<td>0.614</td>
<td>10</td>
<td>0.834</td>
<td>0.006</td>
<td>72</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>0.611, 0.820</td>
<td>D0 D0 *</td>
<td>0.613</td>
<td>12</td>
<td>0.987</td>
<td>0.009</td>
<td>43</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>0.674, 0.881</td>
<td>D0 2.0 / 0.0 -2.0 * +</td>
<td>0.498</td>
<td>6</td>
<td>0.978</td>
<td>0.008</td>
<td>288</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>0.892, 0.944</td>
<td>D0 D0 0.0 - -</td>
<td>0.497</td>
<td>3</td>
<td>0.908</td>
<td>0.004</td>
<td>128</td>
<td>51</td>
</tr>
<tr>
<td>7</td>
<td>0.892, 0.944</td>
<td>0.0</td>
<td>0.390</td>
<td>3</td>
<td>0.908</td>
<td>0.004</td>
<td>128</td>
<td>51</td>
</tr>
<tr>
<td>8</td>
<td>0.374, 0.381</td>
<td>D0 0.0 + 2.0 D0 * /</td>
<td>0.245</td>
<td>1</td>
<td>0.878</td>
<td>0.008</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>0.000, 1.000</td>
<td>D0 D0 / D0 -</td>
<td>0.153</td>
<td>59</td>
<td>1.000</td>
<td>0.000</td>
<td>62585</td>
<td>385</td>
</tr>
<tr>
<td>10</td>
<td>0.000, 1.000</td>
<td>0.0 -1.0 - D0 -</td>
<td>0.135</td>
<td>52</td>
<td>1.000</td>
<td>0.000</td>
<td>58435</td>
<td>385</td>
</tr>
<tr>
<td>11</td>
<td>0.000, 1.000</td>
<td>1.0 1.0 D0 * -</td>
<td>0.122</td>
<td>47</td>
<td>1.000</td>
<td>0.000</td>
<td>50335</td>
<td>385</td>
</tr>
<tr>
<td>12</td>
<td>0.000, 1.000</td>
<td>-2.0 -2.0 / D0 -</td>
<td>0.119</td>
<td>46</td>
<td>1.000</td>
<td>0.000</td>
<td>67660</td>
<td>385</td>
</tr>
<tr>
<td>13</td>
<td>0.000, 1.000</td>
<td>1.0 D0 0.0 + -</td>
<td>0.117</td>
<td>45</td>
<td>1.000</td>
<td>0.000</td>
<td>49710</td>
<td>385</td>
</tr>
<tr>
<td>14</td>
<td>0.000, 1.000</td>
<td>1.0 D0 -</td>
<td>0.112</td>
<td>43</td>
<td>1.000</td>
<td>0.000</td>
<td>71558</td>
<td>385</td>
</tr>
<tr>
<td>15</td>
<td>0.000, 1.000</td>
<td>-2.0 -2.0 / 1.0 D0 * -</td>
<td>0.101</td>
<td>39</td>
<td>1.000</td>
<td>0.000</td>
<td>62710</td>
<td>385</td>
</tr>
<tr>
<td>16</td>
<td>0.000, 1.000</td>
<td>-2.0 -2.0 / D0 0.0 + -</td>
<td>0.096</td>
<td>37</td>
<td>1.000</td>
<td>0.000</td>
<td>53760</td>
<td>385</td>
</tr>
<tr>
<td>17</td>
<td>0.680, 0.896</td>
<td>0.0 -2.0 * 0.0 D0 / +</td>
<td>0.053</td>
<td>1</td>
<td>0.884</td>
<td>0.007</td>
<td>20</td>
<td>43</td>
</tr>
<tr>
<td>18</td>
<td>0.642, 0.825</td>
<td>D0 2.0 / 0.0 D0 / +</td>
<td>0.038</td>
<td>1</td>
<td>0.985</td>
<td>0.006</td>
<td>5</td>
<td>64</td>
</tr>
<tr>
<td>19</td>
<td>0.000, 1.000</td>
<td>2.0 1.0 - D0 1.0 / -</td>
<td>0.034</td>
<td>13</td>
<td>1.000</td>
<td>0.000</td>
<td>28460</td>
<td>385</td>
</tr>
<tr>
<td>20</td>
<td>0.000, 1.000</td>
<td>-2.0 -2.0 / D0 1.0 / -</td>
<td>0.002</td>
<td>1</td>
<td>1.000</td>
<td>0.000</td>
<td>736</td>
<td>385</td>
</tr>
<tr>
<td>21</td>
<td>0.000, 1.000</td>
<td>1.0 0.0 - D0 0.0 + -</td>
<td>0.002</td>
<td>1</td>
<td>1.000</td>
<td>0.000</td>
<td>5161</td>
<td>385</td>
</tr>
<tr>
<td>22</td>
<td>0.000, 1.000</td>
<td>1.0 0.0 - D0 -</td>
<td>0.002</td>
<td>1</td>
<td>1.000</td>
<td>0.000</td>
<td>12761</td>
<td>385</td>
</tr>
<tr>
<td>23</td>
<td>0.000, 1.000</td>
<td>-2.0 -2.0 / 0.0 D0 + -</td>
<td>0.002</td>
<td>1</td>
<td>1.000</td>
<td>0.000</td>
<td>34235</td>
<td>385</td>
</tr>
</tbody>
</table>

and 19 – 23 due to the corresponding range of input space covered by them. As \( \text{setSize} \) is smaller for classifiers 1 – 8 and 17 – 18 so higher proportionate fitness \( F \), but low numerosity \( n \) and experience \( \text{exp} \) highlight lack of worth.

The Frog2 Problem

The results in learning the frog2 problem using XCSRCFA are shown in Figure 7.4. The function set used is \{+,-,\*,\/,log,exp\} and one run is stopped after 200,000 explore problem instances. Although the natural logarithm operator ‘log’ and the exponential operator ‘exp’ are unary operators, an extra operand is used for these operators to facilitate the action
mutation operation, i.e. \( \log(x_1, x_2) = \log(x_1) \) and \( \exp(x_1, x_2) = \exp(x_1) \) in this work. The ‘log’ operator is considered as \( \log(-x) \) is equal to \( \log(x) \), and \( \log(x) \) is equal to 0 if \( x \) is 0.

Figure 7.4: Results in learning the frog1 problem in terms of payoff, system error and population size using XCSRCFA (curve order same as in legend).

It has been reported that the performance of XCSFCA [111] in learning the frog2 problem was greater than 99%, the system error dropped to smaller than 1%, and the population size of classifiers was about 37% of \( N \). The payoff curve in Figure 7.4 shows that XCSRCFA reached the stabilized 100% performance level. The population size of classifiers is about 27% of \( N \).

In the frog2 problem, if the frog wants to catch the fly at distance \( d \), it must jump a distance \( a = d = -\ln(x) \). Since \( d \) is generated randomly from [0,1], the sensory input \( x \) received by Equation 3.4 gives the value from \([e^{-1}, 1]\). As a direct indication of the system’s ability to choose the best action \( a^* \) and to gauge \( a^* \)’s continuity with respect to \( x \), at the end of each run \( x \) was scanned from \( e^{-1} \) to 1 with increment 0.001 and the resulting \( a^* \) plotted in Figure 7.5. The best action plot for XCSRCFA, shown in Figure 7.5, is coincident with the ideal curve.
7.3. RESULTS

Figure 7.5: Best action $a^*$ in learning the frog2 problem using XCSRCFA.

Similar to the frog1 problem, XCSRCFA evolved maximally general and accurate classifiers in learning the frog2 problem. One of the classifier rules from the final solution is shown in Figure 7.6. This is a maximally general and accurate classifier covering the whole problem space and the code-fragment action is equivalent to the target function $-\ln(x)$.

7.3.2 Function Approximation Problems

As mentioned in the Section 3.2, the main aim of developing XCSRCFA was to learn continuous action problems. However, due to the generic nature of code-fragment actions it can also be used as an approximator. In XCSRCFA a function approximation problem is handled just like a classification problem by considering a classifier’s advocated classification accurate if the absolute error between the predicted class value and the actual function value is less than 0.01, as commonly used in GP [113]. The reward scheme used is 1000 for a correct classification and 0 otherwise.
It is worth noting that usually separate classifier systems are used for computing continuous actions and approximating functions. However, XCSRCFA is both a continuous-action system and a function-approximating system.

**Continuous Functions with Continuous Derivative**

The continuous function with continuous derivative used for approximation in this chapter is the parabola given in Equation 3.5. The allowed action range for this problem is \([1,2]\). The number of micro classifiers used is \(N = 300\). One run is stopped after 200,000 explore problem instances.

As a direct indication of the system’s ability to choose the best action \(a^*\) and to gauge \(a^*\)’s continuity with respect to \(x\), at the end of each run \(x\) was scanned from -1 to 1 with increment 0.001 and the resulting \(a^*\) plotted in Figure 7.7. The plot for XCSRCFA is exactly the target curve with no broken points.

Similar to the frog problems, XCSRCFA evolved maximally general and accurate classifiers in learning the \(1 + x^2\) problem. One of the classifier rules from the final solution obtained using XCSRCFA is shown in Figure 7.8. This is a maximally general and accurate classifier covering the
7.3. RESULTS

Figure 7.7: Result in learning the $1 + x^2$ problem using XCSRCFA.

whole problem space and the code-fragment action is equivalent to the target function $1 + x^2$.

Continuous Functions with Discontinuous Derivative

The continuous function with discontinuous derivative used for approximation in this chapter is given in Equation 3.6. The allowed action range for this problem is [1,2]. The number of micro classifiers used is $N = 500$. One run is stopped after 200,000 explore problem instances.

As a direct indication of the system’s ability to choose the best action $a^*$ and to gauge $a^*$’s continuity with respect to $x$, at the end of each run $x$ was scanned from -1 to 1 with increment 0.001 and the resulting $a^*$ plotted in Figure 7.9. The plot for XCSRCFA is exactly the target curve.
Two of the classifier rules from the final solution obtained using XCSR-CFA are shown in Figure 7.10. It is interesting to note that both of these rules cover the whole problem space instead of covering the corresponding subspace separately as given in the problem Equation 3.6. However, the system advocates the output value correctly for each input message as shown in Figure 7.9. It is anticipated that it is due to the following two reasons: (1) a classifier advocating invalid action value is not allowed to be in the match set, e.g. the classifier shown in Figure 7.10(a) cannot be in the match set for input values that are greater than 0, (2) other classifiers are also playing their roles in determining the action value for a given input.

**Discontinuous Functions**

The discontinuous function used for approximation in this chapter is given in Equation 3.7. In this discontinuous function the two pieces have clearly separated input ranges, unlike the approximation problem given in Equation 3.6. The allowed action range for this problem is \([1,5]\). The number of micro classifiers used is \(N = 2000\). One run is stopped after 500,000 explore problem instances. Both GA subsumption and action set subsumption are off as they degraded the performance in this problem.
7.3. RESULTS

Figure 7.9: Result in learning the problem given by Equation 3.6 using XCSRCFA.

As a direct indication of the system’s ability to choose the best action $a^*$ and to gauge $a^*$’s continuity with respect to $x$, at the end of each run $x$ was scanned from -2 to -1 and 1 to 2 with increment 0.001 and the resulting $a^*$ plotted in Figure 7.11. The plot for XCSRCFA is exactly the target curve with no broken points.

Two of the classifier rules from the final solution obtained using XCSRCFA are shown in Figure 7.12. It is observed that both of these rules cover the corresponding subspace separately as given in the problem Equation 3.7 unlike the classifiers shown in Figure 7.10.
CHAPTER 7. COMPUTING CONTINUOUS ACTIONS

<table>
<thead>
<tr>
<th>Condition</th>
<th>[-1.0, 1.0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td></td>
</tr>
</tbody>
</table>

(a) 1 - x.

<table>
<thead>
<tr>
<th>Condition</th>
<th>[-1.0, 1.0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td></td>
</tr>
</tbody>
</table>

(b) 1 + x^2.

Figure 7.10: Two sample classifier rules from the final solution of the problem given by Equation 3.6 in XCSR-CFA.

Multi-dimensional Functions

After successful application of XCSR in approximating one-dimensional functions, it was tested on the two-dimensional approximation problem given in Equation 3.8. The allowed action range for this problem is [-1,2]. The number of micro classifiers used is \( N = 2000 \). One run is stopped after 500,000 explore problem instances.

Similar to the one-dimensional problems, XCSR-CFA evolved maximally general and accurate classifiers in learning the two-dimensional \( x + y^2 \) problem. One of the classifier rules from the final solution obtained using
XCSRCFA is shown in Figure 7.13. This is a maximally general and accurate classifier covering the whole problem space and the code-fragment action is equivalent to the target function \( x + y^2 \).

When tested on the four-dimensional function approximation problem given in Equation 3.9, XCSRCFA failed in approximating it due to the large search space. The performance of XCSRCFA in approximating this four-dimensional function could not be improved beyond 50% even using larger population size and increasing the allowed depth for code-fragment actions. XCSF solved a similar but six-dimensional problem quite well by using piecewise-linear approximation with many small subdomains for
CHAPTER 7. COMPUTING CONTINUOUS ACTIONS

**Figure 7.12:** Two sample classifier rules from the final solution of the problem given by Equation 3.7 in XCSRCFA.

(a) \(1 + x^2\).

(b) \(3 - x\).

**Figure 7.13:** A sample classifier rule from final solution of the \(x + y^2\) problem in XCSRCFA.
7.4. **DISCUSSION**

Each approximator [122]. It is anticipated that XCSRCFA may try to get the whole domain with a single approximator, so it would seem to see a much larger search space. It is to be noted that the main aim of this thesis is to design a scalable classifier model to tackle large-scale classification problems rather than approximation problems. It is the additional property of the developed XCSRCFA system that it can be used as an approximator.

### 7.4 Discussion

XCSRCFA is a simple system designed as a combination of XCSR and GP. A GP system evolves a single program as a final solution, so it is hard for a GP system to approximate a discontinuous function, and a continuous function with discontinuous derivative. Furthermore, for the approximation of a continuous function with continuous derivative it requires subsampling as it cannot process the whole search space in an offline manner. However, XCSRCFA does not suffer from these limitations due to the online learning and cooperative classifier rules evolved in an LCS technique.

In XCSR, a classifier’s action is represented by a numeric constant, therefore, XCSR cannot learn real-valued, continuous action problems. However, in XCSRCFA the action is a GP-tree like code-fragment, which is continuous with respect to the environmental input. Therefore, XCSRCFA can be applied in learning real-valued, continuous action problems. In addition, XCSRCFA can be adopted to approximate functions.

The most interesting thing about XCSRCFA is it shows that for continuous action it is not necessary to go through the stage of approximating the prediction as in XCSFCA and GCS, etc. Rather continuous action can be computed directly in a simple and neat way. In addition, and consequently, XCSRCFA can be used as a certain kind of function approximator.

XCSRCFA learned various real-valued, continuous action and function approximation problems. However, it did not do as well as XCSF can do on a four-dimensional problem. The analysis of evolved rules indicates
that XCSRCFA has tendency to evolve optimal rules covering the whole problem space in the classifier condition and generating the corresponding single GP-tree, instead of dividing the problem space and evolving a set of classifier rules as in a standard LCS. This property of XCSRCFA makes it harder to learn a high-dimensional problem due to the large search space. XCSF breaks up the target function into pieces so that its linear approximators can work. XCSRCFA, having much more powerful approximators, merges the pieces until, at least in simpler problems, its approximators cover the whole domain. However, at a certain level of domain complexity, XCSRCFA begins to struggle and becomes like regular GP. XCSF does better because its linear approximators’ domains are forced to be small, and therefore they collectively can solve quite complex problems.

It is known that the classifier conditions break a problem up into sub-domains, given the available degree of complexity of the actions. The conditions are like filters. If the actions are simple, there must be a lot of filtering. If the actions can be complex, less filtering or partitioning of the overall space occurs. Therefore, in order to learn high-dimensional function approximation problems using XCSRCFA, it is suggested to use linear (i.e. weight-vector) functions in classifier actions and/or use more flexible encoding to represent classifier conditions so that a set of cooperative rules can be evolved, instead of a single rule as present.

7.5 Chapter Summary

This chapter presented an extension to XCSR in which the discrete action has been replaced by a code-fragment action that is continuous with respect to the input state. XCSRCFA successfully solved the continuous action frog problems, consistently producing the optimal rules with the maximum predicted payoff. XCSRCFA also learned the tested one- and two-dimensional function approximation problems. However, it failed in learning a four-dimensional approximation function. The XCSRCFA tech-
nique is expected to be most useful in learning discontinuous function
approximation problems.

The investigation of code fragments shows that the multiple genotypes
to a single phenotype mapping in XCSRCFA disables the subsumption
deletion function. Due to this loss of subsumption deletion, the final pop-
ulation contains optimal, but redundant classifier rules.
Chapter 8

Evolvability of Classifier Rules in XCS-Based Classifier Systems

8.1 Introduction

Although learning classifier systems (LCSs), an established evolutionary computation technique, are over 30 years old with much empirical testing and foundations of theoretical understanding [22, 26, 33], there is still much to understand regarding how the optimum solution to a problem is evolved. The evolution and analysis of the individual members in a population have been studied in various evolutionary computation techniques [41–43, 50, 105, 126]. However, to the best of my knowledge, there has been no published work in the literature to analyze and investigate the evolvability of classifier rules in an LCS, which have high elitism (steady-state) and cooperative populations.

XCS is a well tested LCS model that generates optimal (i.e. maximally general and accurate) classifier rules in the final solution. The classifier rules in an XCS-based classifier system routinely keep statistics regarding their performance history, which allows insight into their behaviors. In this chapter, a method to trace the evolution of classifier rules generated in an XCS-based classifier system is introduced. Specifically, the concept of a
family tree, termed parent-tree, for each individual classifier rule generated
in the system during training is introduced, which describes the whole
generational process for that classifier. Various statistics at each level in
the parent-trees are also computed to analyze the evolution of rules from
different aspects. Experiments are conducted on multiplexer and count
ones problems using two XCS-based systems, i.e. standard XCS and XCS
with code-fragment actions (XCSCFA) as they produce easily interpretable
classifier rules.

XCS keeps both correct and incorrect classifiers, provided they are ac-
curate, i.e. accurately predict the reward received from the environment.
It is noted that the building blocks of information in the condition of an
incorrect classifier are exactly the same as in the counterpart correct clas-
sifier. For example, ‘000### : 1 → 0’ (where the ending ‘0’ is the predicted
payoff of the rule) is an accurate incorrect classifier which has the same
condition as that in the counterpart accurate correct classifier ‘000### : 0
→ 1000’ (where the ending ‘1000’ is the predicted payoff of the rule). The
rule discovery operation is applied to the action set, which is formed by
the classifiers advocating a certain action, commonly selected at random,
and covering the currently observed environmental input. As all the clas-
sifiers in an action set advocate the same action, the correct and incorrect
classifiers cannot occur in the same action set, so cannot be simultaneously
used in breeding of the new classifiers. This means, in an XCS system,
that although both correct and incorrect classifiers are kept throughout
the learning of the system, the building blocks of information in them are
not efficiently exploited as they are not allowed to take part in the same
breeding operation.

In XCSCFA the action in a classifier rule is computed using a genetic
programming tree like code-fragment (see Chapter 5). There are two ways
to compute the action value of a classifier in XCSCFA [61]: 1) by load-
ing the terminal symbols in the action tree with the corresponding binary
values from the condition in the classifier rule, and 2) by loading the termi-
nal symbols with the corresponding binary values from the environmental input; where the former is used here because it produces easily interpretable optimal rules as in standard XCS. To compute the action value of a classifier using the classifier condition, a ‘don’t care’ symbol (‘#’) in the condition is randomly treated as 0 or 1. Therefore a classifier rule in XCSCFA may output different values as its action at different times, even for the same environmental input. This is termed inconsistency of the action value in a classifier rule, and that classifier is termed an inconsistent classifier. Enabling the system to consider the inconsistency in a GP-like representation has been shown to be beneficial in certain domains [61].

The XCS system keeps a complete map, i.e. the classifiers advocating consistently correct classification as well as the classifiers advocating consistently incorrect classification. A consistently incorrect classifier contains the same building blocks of information as in the counterpart correct classifier, e.g. ‘000### : 1’ and ‘000### : 0’, but in standard XCS these classifiers cannot occur in the same action set, so, cannot be simultaneously used in breeding of the new classifiers. The XCS’s ability to keep a complete map combined with the inconsistent actions may preserve important building blocks of information in XCSCFA. Due to inconsistent action values, the incorrect classifiers can occur in the same action set as correct classifiers in XCSCFA so can be used for the production of good classifiers.

To aid in the visual interpretation of the evolved classifier rules and the parent-trees, a Boolean problem search space can be represented in a two-dimensional grid form. For example, the search space of a 6-bit binary problem is shown in Figure 8.1. Here D0, D1, ..., D5 denote the problem input values and an exemplar target search space area is denoted by the grey-filled cells which is equivalent to ‘000###’ in the ternary encoding scheme.

It is hypothesized that the classifier rules in an XCS are generated using different evolutionary mechanisms, which are described as ‘close-by then mutate’, ‘be specific then generalize’, ‘be general then specify’, and
'overlap then recombine’, see Figure 8.2. In ‘close-by then mutate’, during the evolution of the system a classifier rule may be evolved close to an optimum classifier, i.e. a single mutation in one dimension may produce the optimum classifier, see Figure 8.2(a). It is noted that such classifiers although ‘close’ to optimum receive zero environmental reward. In ‘be specific then generalize’, during the evolution of the system some evolved classifier rules cover a sub-part of the search space covered by an optimum classifier. These rules are always accurate and correct. Then a mutation or crossover operation on two such classifiers can make them general enough to optimally cover the whole space of the niche, see Figure 8.2(b). In ‘be general then specify’, during the evolution of the system some evolved classifier rules cover the target search space plus extra undesired parts of the search space. These rules are not maximally accurate as they are only
8.1. INTRODUCTION

partially correct. These are termed over-general rules which were considered a problem in early LCSs. The mutation operation can make such classifiers specific enough to cover only the target search space, see Figure 8.2(c). In ‘overlap then recombine’, during the evolution of the system classifier rules may evolve to a form such that each of them covers a part of the search space covered by an optimum classifier. Similar to the over-general classifiers in Figure 8.2(c), these rules are not maximally accurate. These classifiers can be recombined by the crossover operation resulting in the optimum rule, see Figure 8.2(d).

8.1.1 Goals

Although it is hypothesized that the ‘be specific then generalize’ is the most common evolutionary mechanism in an XCS-based system to evolve the optimum classifiers in the final solution to a problem [26], this is to be verified and it is to be determined what proportions of other mechanisms, such as ‘close-by then mutate’, ‘be general then specify’, and ‘overlap then recombine’, are used in the evolution of the optimum classifier rules. Similarly, it is to be answered how many generations of parents are needed for the evolution of target classifiers and when they occur in the population. The aim of the work presented here is to explore evolvability of classifier rules, specifically the optimal rules in the final solution obtained using XCS and XCSCFA. In this chapter Boolean problems are used as the test problems because of tractability, but the analysis should function with other problem domains as well.

In order to achieve the above contributions, it was required to analyze the evolvability of optimal classifiers. The evolution and analysis of the individual members in a population have been studied in various evolutionary computation techniques, such as canonical genetic programming [4, 43, 125, 126], linear genetic programming [50, 51], genetic algorithms [35, 41], neural networks [105, 115], and gene networks [63]. Specif-
Figure 8.2: Different mechanisms to evolve a classifier rule in an XCS-based classifier system. The hypothetical target classifier rule is denoted by grey filled cells and the two hypothetical parent classifiers are denoted by the cells filled with horizontal lines and vertical lines respectively.

Xie et al. [126] have used parentage information to locate ancestors of the best program generated in a GP run in order to reduce the fitness evaluation cost in tree-based GP systems. Stanley and Miikkulainen [105] have used historical information for each gene in a genome to evolve neu-
The resulting system, known as NeuroEvolution of Augmenting Topologies (NEAT), outperformed the best fixed-topology method by utilizing the history of genes evolution. However, to the best of my knowledge, there has been no published work in the literature to analyze and investigate the evolvability of classifier rules in an LCS, which have high elitism (steady-state) and cooperative populations.

8.1.2 Organization

The rest of the chapter is organised as follows. In Section 8.2 the concept of parent-trees is explained, which provides the platform for the novel contributions of this chapter. Section 8.3 introduces the problem domains and experimental setup. In Section 8.4 the evolution of classifier rules produced in XCS and XCSCFA is analyzed from different aspects using the novel concept of parent-trees. In the last section, this work is concluded and the future work is outlined.

8.2 Parent-Trees

The main purpose of the work presented here is to analyze the evolution of the optimal classifiers evolved in two XCS-based classifier systems, i.e. standard XCS and XCSCFA. This chapter introduces the concept of a parent-tree, for the first time in the field of LCS, for each classifier in the final solution which describes the whole generational process for that classifier. In addition to the construction of parent-trees, various statistics at each level in the parent-trees are also computed to analyze the evolution of the classifier rules from different aspects, as will be described in the Section 8.4. This section describes the concept of parent-trees using two optimal classifier rules generated in learning the 6-bit multiplexer problem.
In order to trace the family hierarchy of an evolved classifier rule using its parent-tree, the following additional attributes have been maintained for each classifier:

- *id*, which is a unique identification number for each classifier.
- *gen*, which is the generation number when the classifier was created.
- *pid<sub>1</sub>* and *pid<sub>2</sub>* which are the identification numbers of its parents (if any).
- *d*, which is the depth of its parent-tree.

In all the results presented here, a parent-tree of depth 0 consists of only a single node. The root (i.e. an optimal classifier) is considered at the top level in the parent-tree and numbered level 0, i.e. a smaller level number represents a higher level in the parent-tree. The classifiers at the lowest level (i.e. greatest numbered parent-tree level) are the originating classifiers.

If the condition of a classifier is represented using the ternary alphabet \(\{0, 1, #\}\), then an optimal classifier rule for the 6-bit multiplexer problem will contain three ‘#’ symbols in the condition part, see Section 3.2 for details. A parent-tree for the optimal classifier rule ‘01#0## : 0’ generated in learning the 6-bit multiplexer problem using XCS is depicted in Figure 8.3. Each classifier rule in this parent-tree is encoded as ‘condition : action; *id, gen, pid<sub>1</sub>, pid<sub>2</sub>, d, F, ε, p, exp’*, where *id, gen, pid<sub>1</sub>, pid<sub>2</sub>, d, F, ε, p, exp* denote identification number, generation number, identification numbers of the parents (if any), depth, fitness, prediction error, prediction, and experience of the classifier respectively. It is possible that the same classifier is selected as the both parents or only one parent classifier is mutated to create an offspring classifier. The identification number “-1” denotes no parent. If identification numbers for both parents of a classifier are -1 then that classifier is a newly created one during the covering operation. If
identification number for one of the parents is -1 and for the other is a positive number, then the classifier is generated from a single parent classifier using the mutation operation during the rule discovering process.

Figure 8.3: A sample parent-tree for the classifier rule ‘01#0## : 0’ generated in learning the 6-bit multiplexer problem using XCS.

The tree in Figure 8.3 highlighted three very interesting properties of the evolution within XCS-based classifier systems: 1) it is not necessary for a classifier rule to be accurate in order to be used in evolution of the target classifiers in the final solution set, e.g. the classifier rule ‘01##1# : 1’ at level 3 is not accurate, but is used in production of the accurate classifier ‘01#01# : 1’; 2) a classifier rule generated in an earlier generation during the evolutionary process can be retained in the population until a good partner classifier is mated with it to produce a better classifier rule, e.g. the classifier rule ‘01#01# : 1’ at level 3 was generated in the 5th generation and mated with the classifier ‘01##1# : 1’ in generation 863 and produced the classifier ‘01#01# : 1’; and 3) the action part of an accurate, but incorrect classifier can be mutated in the rule discovery operation to produce the accurate and correct classifier, e.g. the action of the classifier ‘01#0## : 1’ at level 1 was mutated to produce the correct optimal classifier ‘01#0## : 0’.

The evolution of the classifier rule ‘01#0## : a’ (a = 0 denotes the classifier action) is shown in a 2-dimensional grids form in Figure 8.4, where the problem search space covered by each classifier rule is represented in grids. For example, the classifier rule ‘01#010 : a’ at level 3 covers the search space areas ‘010010’ and ‘011010’. The target area is denoted by
grey filled cells and the areas covered by parent classifiers are denoted by the cells filled with horizontal lines and vertical lines. If the same classifier is selected as both parents in a crossover operation, then it is denoted by a cell filled with both horizontal and vertical lines. It is observed that all the classifiers involved in the evolution of the target classifier ‘11##0 : a’, shown at level 0, cover a part of the target search space. Two classifiers, one at level 4, i.e. ‘01##1 : a’, and one at level 3, i.e. ‘10##1# : a’ additionally cover parts of the search space outside the target area, but there is no overgeneral classifier, i.e. a classifier that covers the whole target search space plus extra undesired parts of the search space, in the whole evolutionary process for the target classifier ‘01#0## : a’.

The evolutionary process for another classifier ‘001### : a’ (a = 1 is the desired classifier action), generated in learning the 6-bit multiplexer problem using XCSCFA, is shown in a 2-dimensional grids form in Figure 8.5 as it highlights important properties of XCSCFA. The target area is denoted by grey filled cells, the areas covered by consistent parent classifiers are denoted by the cells filled with horizontal lines and vertical lines, and the areas covered by inconsistent parent classifiers are denoted by the cells filled with dashed horizontal lines and dashed vertical lines. It is interesting to note that three classifiers involved in the evolution of the target classifier ‘001### : a’, shown at level 0, do not cover any part of the target search space, see left classifier at level 4, right classifier at level 3, and left classifier at level 2. If the target search space is considered as the ‘figure’ and the remaining search space area as the ‘ground’, analogous to the ‘figure and ground’ terminology used in the field of object recognition [29], then it can be said that XCSCFA utilizes the ‘ground’ to evolve the ‘figure’. For example, the two classifiers at level 4 cover only two cells out of the eight target search space cells, but the produced classifier at level 3 covers four target cells. The other property of XCSCFA highlighted in the parent-tree shown in Figure 8.5 is that XCSCFA utilizes inconsistent classifiers (again the ‘ground’ or partial ‘ground’) to evolve the consistent
Figure 8.4: A sample parent-tree, in 2-dimensional form, for the classifier rule ‘01#0## : a’ generated in learning the 6-bit multiplexer problem using XCS. Here only the conditions of classifiers are represented in a 2-dimensional form, showing the problem search space covered by each classifier; and the classifier action is denoted by a.
target classifier, see left classifier at level 3, right classifier at level 2, and left classifier at level 1. It can be noted that the target classifier at level 0 is directly generated from the breeding of an inconsistent classifier with a consistent one at level 1.

8.3 Experimental Design

This section describes data sets and parameter settings to be used for the investigation of evolution and analysis of the optimal classifier rules.

8.3.1 The Problem Domains

The problems used in the experimentation are the 11-bit multiplexer problem and the 5-bit count ones problem as it is straightforward to trace the evolution of optimal rules produced in learning these problems.

A multiplexer is an electronic circuit that accepts input strings of length \( n = k + 2^k \), and gives one output (see Section 3.2). The value encoded by the \( k \) address bits is used to select one of the \( 2^k \) remaining data bits to be given as output. The optimum ternary encoded solution set for the 11-bit multiplexer problem consists of 16 maximally general, accurate and correct classifiers, shown in Table 8.1.

In count ones problems only \( k \) bits are relevant in an input instance of length \( l \) (see Section 3.2). If the number of ones in the \( k \) relevant positions is greater than half \( k \), the problem instance is of class one, otherwise class zero. It is to be noted that the complete solution for a count ones problem consists of overlapping classifiers. The count ones problem used in this work is of length \( l = 15 \) with the first \( k = 5 \) relevant bits. The optimum ternary encoded solution set for the 5-bit count ones problem consists of 20 maximally general, accurate and correct classifiers, shown in Table 8.2.
8.3. EXPERIMENTAL DESIGN

Figure 8.5: A sample parent-tree, in 2-dimensional form, for the classifier rule ‘001### : a’ generated in learning the 6-bit multiplexer problem using XCSCFA. Here only the conditions of classifiers are represented in a 2-dimensional form, showing the problem search space covered by each classifier; and the classifier action is denoted by a.
Table 8.1: The ternary encoded rule set for the 11-bit multiplexer problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000####</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0001####</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0010#0#######</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>001#1#######</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0100#0#######</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>010#1#######</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>011#0#######</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>011#1#######</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1000#0#######</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>100#1#######</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>101###0##</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>101###1##</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>110####0#</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>110####1#</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>111#######0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>111#######1</td>
<td>1</td>
</tr>
</tbody>
</table>

8.3.2 Experimental Setup

All the parameter values used for experimentation in this chapter are the same as described in Section 3.3, except the followings: Both GA subsumption and action set subsumption are activated for the 11-bit multiplexer problem, but for the 5-bit count ones problem action set subsumption is deactivated as suggested in [61]. The function set for the code fragments used in XCSCFA is \{AND, OR, NOT, NAND, NOR\}, denoted by \{&, |, ~, d, r\}, for all the problem domains experimented in this work. The maximum number of classifiers used is 500 for the 11-bit multiplexer problem and 1000 for the 5-bit count ones problem. The number of training exam-
Table 8.2: The ternary encoded rule set for the 5-bit count ones problem.

<table>
<thead>
<tr>
<th>No.</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>##0000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>#0##00000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>#00#00000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>#00000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0##0000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0#0##00000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0#00#00000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>00#0##00000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>00#00##00000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>000##0000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>##11100000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>#1##110000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>#11#10000000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>#111#00000000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1##11##00000000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1##1##10000000000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>1#1##1000000000000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>11##1##0000000000000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>11##1##100000000000000000000000000000</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>111##1##00000000000000000000000000000000</td>
<td>1</td>
</tr>
</tbody>
</table>

Examples used is 50,000 for all the experiments. Explore and exploit problem instances are alternated. The reward scheme used is 1000 for a correct classification and 0 otherwise.

To get an insight of the evolvability of an individual classifier rule, seven sets of experiments are conducted with different values of $P_\#$, which denotes the probability the ‘don’t care’ symbol ‘#’ used in the covering operation. The seven values of $P_\#$ used in this work are: 0.10, 0.25, 0.33, 0.50,
0.66, 0.75 and 0.90. All the results presented here are average of the 30 independent runs.

8.4 Evolution and Analysis of Classifier Rules

The analysis of parent-trees provides important insights about evolution of the optimal classifier rules. In addition to the construction of parent-trees, various statistics at each level in the parent-trees of the optimal classifier rules are also computed to analyze evolution of the rules from different aspects. The rest of this section provides detailed experiments in learning sample problems using XCS and XCSCFA that illustrate the novel analysis and rule evolution. Note that it is the analysis of the results that is important, not the results themselves, e.g. multiplexer problems have been solved up to 135-bits (see Chapter 4).

8.4.1 Classification Performance

The classification performance of XCS and XCSCFA in learning the 11-bit multiplexer problem and the 5-bit count ones problem is shown in Figure 8.6 and Figure 8.7, respectively. Here the X-axis is the number of problem instances used as training examples and the Y-axis is the performance measured as the percentage of correct classification during the last 100 exploit problem instances. All the graphs in this section are in color for better readability.

It is observed that for values of \( P_\# \) near to 0, both XCS and XCSCFA slowly converged to an optimal solution. In particular, when \( P_\# \) was 0.10, XCS failed three times out of the 30 runs and XCSCFA failed 13 times, in learning the 5-bit count ones problem indicating that they were trapped in the covering/deletion loop as described in [26]. With \( P_\# = 0.10 \), both XCS and XCSCFA need larger populations to handle the covering challenge [26], in order to successfully learn the 5-bit count ones problem. As the value
8.4. EVOLUTION AND ANALYSIS OF CLASSIFIER RULES

Figure 8.6: The classification performance of XCS in learning the 11-bit multiplexer problem and the 5-bit count ones problem.

When $P_\#$ was close to 1, the initial populations tend to be filled up with over-general classifiers; thus, it was more difficult for both XCS and XCSCF to converge to the optimal solutions made of accurate and maximally general classifiers. However, as $P_\#$ decreased from close to 1, the performance of both XCS and XCSCF improved in learning the 11-bit multiplexer problem as well as the 5-bit count ones problem.

It is interesting to note that XCS reached the best performance in learning the 11-bit multiplexer problem and the 5-bit count ones problem when $P_\#$ was 0.50 and 0.50 – 0.66 respectively, even though the maximally gen-
eral solutions for the 11-bit multiplexer problem and the 5-bit count ones problem involve only classifiers with $7/11 \approx 0.64$ and $12/15 = 0.8$ ‘don’t care’ symbols, respectively. XCSCFA reached the best performance in learning the 11-bit multiplexer problem and the 5-bit count ones problem when $P_\#$ was $0.33 - 0.50$ and $0.50$ respectively, indicating that XCSCFA prefers slightly smaller $P_\#$ values than XCS.
8.4. EVOLUTION AND ANALYSIS OF CLASSIFIER RULES

8.4.2 The Convergence of Optimal Classifiers

To analyze evolution of the optimal classifier rules in terms of generalization, the numbers of ‘#’ symbols at each level in the parent-trees of such classifiers obtained in the final solution using XCS and XCSCFA are calculated. The convergence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS and XCSCFA is shown in Figure 8.8 and Figure 8.9, respectively. It is to be noted that using $P_{#} = 0.10$, both XCS and XCSCFA could not consistently learn the 5-bit count ones problem, therefore, the corresponding convergence curves are not presented here.

It is observed that all the classifiers are converging to the final form from specific to general at the top of the parent-trees, even starting with very high generalization probability value of 0.90. This is a new discovery in the field of LCSs. It is worth noting that the optimal rules are evolved from specific to general, but if the originating classifiers are too specific, i.e. $P_{#}$ is set very small, then it may slow down the convergence of the optimal rules due to the increased parent-tree depth as shown in Figure 8.10 and Figure 8.11 for XCS and XCSCFA, respectively. Therefore, it is suggested to use slightly specific initial classifiers than the target classifiers (if known). This would increase the convergence speed of the system and produce the optimal solution more rapidly as shown in Figure 8.6 and Figure 8.7.

The optimal rules generated in XCSCFA have larger parent-trees than XCS, see Figure 8.11 and Figure 8.10 respectively, because in XCSCFA it takes longer to evolve accurate rules due to the code-fragment actions and the inconsistent classifiers. In addition to this, due to the issue of multiple genotypes to a single phenotype mapping of code-fragment actions in XCSCFA, it is possible that an optimal rule is directly evolved from other optimal rules covering the same niche (i.e. having the same condition), further increasing the parent-tree depth of the evolved optimal rule.
Figure 8.8: The convergence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS. Only the top 15 levels in the parent-trees are shown. It is worth noting that all the classifiers are converging to the final form from specific to general at the top of the parent-trees.

8.4.3 The Occurrence of Optimal Classifiers

The first occurrence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS and XCS-SCFA is shown in Figure 8.12 and Figure 8.13, respectively.

It is observed that for values of $P_\#$ near to 0, XCS slowly evolved the optimal classifier rules in learning the 11-bit multiplexer problem as well as the 5-bit count ones problem, see Figure 8.12(a) and Figure 8.12(b), respectively. As the value of $P_\#$ increased, the optimal rules in XCS evolved more rapidly. The fastest evolution of the optimal rules in learning the
8.4. EVOLUTION AND ANALYSIS OF CLASSIFIER RULES

Figure 8.9: The convergence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS-SCFA. Only the top 30 levels in the parent-trees are shown. It is worth noting that all the classifiers are converging to the final form from specific to general at the top of the parent-trees.

11-bit multiplexer problem and the 5-bit count ones problem using XCS achieved when $P_\#$ was 0.50 and 0.50 – 0.66, respectively; even though the maximally general solutions for the 11-bit multiplexer problem and the 5-bit count ones problem involve only classifiers with $7/11 \approx 0.64$ and $12/15 = 0.8$ ‘don’t care’ symbols, respectively. When $P_\#$ was close to 1, XCS slowly evolved the optimal rules and as $P_\#$ decreased, the evolution of the optimal rules improved again.

Similar to XCS, for values of $P_\#$ near to 0, XCSCFA slowly evolved the optimal classifier rules in learning the 11-bit multiplexer problem, see Fig-
CHAPTER 8. EVOLVABILITY OF CLASSIFIER RULES

Figure 8.10: The average depth of the parent-trees obtained in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS with different values of $P_{\#}$.

Figure 8.11: The average depth of the parent-trees obtained in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCSCF with different values of $P_{\#}$.

ure 8.13(a), and as the value of $P_{\#}$ increased, the optimal rules in XCSCF evolved more rapidly. The fastest evolution of the optimal rules in learning the 11-bit multiplexer problem using XCSCF achieved when $P_{\#}$ was
Figure 8.12: The first occurrence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCS.
0.33, indicating that XCSCF A prefers slightly smaller $P_{\#}$ values than XCS as smaller $P_{\#}$ values help in generating more consistent code-fragment actions as compared with larger $P_{\#}$ values. Similarly, when $P_{\#}$ was close to 1, XCSCF A slowly evolved the optimal rules and as $P_{\#}$ decreased, the evolution of the optimal rules improved again. Surprisingly, in learning the 5-bit count ones problem using XCSCF A the evolution of the optimal rules is almost similar for any $P_{\#}$ value, with 0.25 being slightly fast, as shown in Figure 8.13(b). It is anticipated that this is due to the overlapping property of a count ones problem because in the solution of an overlapping problem different optimal rules contain different numbers of ‘#’ symbols.

The comparison of Figure 8.12 and Figure 8.13 revealed that the evolution of the optimal rules in XCS is earlier and faster than XCSCF A. For example, with $P_{\#}$ equal to 0.25, the first optimal rule in learning the 5-bit count ones problem using XCS and XCSCF A occurred at generation number $\approx 2000$ and $17000$, respectively. Similarly, the first ten optimal rules in learning the 5-bit count ones problem, with $P_{\#}$ equal to 0.25, using XCS were evolved within $\approx 2000$ generations whereas XCSCF A took $\approx 27000$ generations to evolve the first ten rules in this case. The reason for the slow evolution of the optimal rules in XCSCF A is again the code-fragment actions, the inconsistent classifiers, and the multiple genotypes to a single phenotype mapping of code-fragment actions as described in Section 8.4.2.

In XCS the evolution of the optimal rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem was completed in less than 40,000 generations and 14,000 generations, see Figure 8.12(a) and Figure 8.12(b), respectively. However, in XCSCF A the optimal rules were found to be evolved continually until 50,000 generations, i.e. at the end of the training in learning the 11-bit multiplexer problem as well as the 5-bit count ones problem, see Figure 8.13(a) and Figure 8.13(b), respectively. This is because in XCSCF A the number of the optimal rules are greater than XCS due to the multiple genotypes to a single phenotype mapping of code-fragment actions in XCSCF A, e.g. XCSCF A evolved more than
Figure 8.13: The first occurrence of the optimal classifier rules in learning the 11-bit multiplexer problem and the 5-bit count ones problem using XCSCFA.
60 genotypic optimal rules in learning the 11-bit multiplexer problem, see Figure 8.13(a), instead of the only 16 required phenotypic optimal rules shown in Table 8.1.

8.4.4 Further Discussions

In an XCS-based system, the rule discovery operation is conventionally applied to the action set that is formed by the classifiers advocating a certain action, commonly selected at random, and covering the currently observed environmental input. All the classifiers in an action set advocate the same action and the mutation operation to change the action of a newly produced child classifier is applied with a probability $\mu$, which has the commonly used value of 0.04. Therefore, 96% of the newly produced children in standard numeric action based XCS have the same action value as that of the parent classifiers. It means that, in an XCS-based system, although both correct and incorrect classifiers are kept throughout the learning of the system, the building blocks of information in them are not efficiently exploited as they are not allowed to take part in the same breeding operation. In domains with a large number of classes, the above consideration is compounded as the number of incorrect classifiers, to be kept by an XCS-based system, increases for every additional class. It is anticipated that this is the reason that an XCS-based system struggles to learn a multi-class problem [13].

In XCSCFA, due to the inconsistent action values, it is possible that an action set consists of classifiers with the potential for different actions. It can be seen from the parent-tree shown in Figure 8.5 that XCSCFA enables both incorrect and correct classifiers to breed together. Therefore, the rule discovery operation in XCSCFA can exploit the building blocks of information more efficiently as compared with standard XCS. Fortunately, in an XCS-based system both correct and incorrect classifiers exist in the same match set formed against the currently observed environmental in-
put message. In the original XCS paper by Wilson [118], the rule discovery was applied to the match set, but later it was moved to the action set in order to reduce the number of inaccurate classifiers produced in the system [119]. If the rule discovery operation is applied to the match set, building blocks of information can be efficiently used for the evolution of potentially good classifiers [60]. This suggestion is supported by the evolution of the maximally general, accurate and correct classifier ‘001### : 1’ shown in Figure 8.5. To reduce the number of inaccurate and incorrect classifiers produced during the rule discovery operation, the action mutation could be dependent on the accuracy and correctness of the parent classifiers. If the parent classifier is more accurate and correct, the action mutation rate should be smaller. If the parent classifier is incorrect (either accurate or not), then the action mutation rate should be higher so that the action can be flipped to other available actions, increasing the chances of the production of correct classifiers.

8.5 Chapter Summary

The aim of this work was to understand the evolution of the optimal rules in XCS-based classifier systems, in terms of evolutionary methods especially with regard to generalization (specificity) of rule coverage and how this develops. Parent-trees were created for each of the optimal classifier rules in the final solution obtained using different initial generalization probability values, denoted by $P_\#$. The analysis of parent-trees revealed the following:

- The parent-trees of different classifiers have different lengths, even in the same experiment with the same $P_\#$ value.

- The optimal classifiers are generated using different mechanisms such as ‘overlap then recombine’, ‘close-by then mutate’, but mostly ‘be specific then generalize’.
• XCSCFA can utilize incorrect, but accurate classifiers to breed correct, accurate classifiers due to inconsistency of action directly whereas XCS requires mutation operation to flip the ‘incorrect’ binary action.

• The optimal classifiers in all the experiments converged to the final form by becoming specialized at the top levels of the parent-trees, even using the very large $P_#$ value of 0.9.

• A very small or very large $P_#$ value resulted in larger parent-trees, which provides a reason for a slower rate of convergence to the optimum population.

• To converge to the optimal solution faster, it is better to use relatively specific initial classifiers as compared with anticipated generality in the final solution.

Further, the introduced code-fragments are not simply another representation as the associated methods fundamentally change the way XCS functions, e.g. enabling correct and incorrect classifiers to breed directly. This work has identified that XCS does not directly use all relevant information or breeding strategies, which offers areas for performance and efficiency improvement in the field of LCS.
Chapter 9

Conclusions and Future Work

The overall goal of this thesis was to improve the scalability of learning classifier systems, especially the accuracy-based classifier systems, by reusing the learned knowledge from smaller problems in a domain and/or increasing the generalization ability of the system. To achieve this goal, classifier rules were encoded using different rich encoding schemes. The developed systems were evaluated using various complex problems used in the literature and the results were compared with the existing related systems.

The rest of this chapter presents the achieved objectives, main conclusions from each contribution chapter, and the future work that stems from this research work.

9.1 Achieved Objectives

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

1. Building blocks of knowledge were extracted from smaller problems of a Boolean domain and reused in learning more complex, large-scale problems in the domain, for the first time. By utilizing this
layered learning approach, the developed XCSCFC system readily solved problems of a scale that existing classifier system and genetic programming approaches cannot, e.g. the 135-bit MUX problem.

2. The generalization ability of classifier rules was increased beyond a numeric action based XCS by using a code-fragment based action in a classifier rule. The introduction of the code fragments in classifier actions enabled the rich representation of GP, which when coupled with the divide and conquer approach of LCS, to successfully solve various complex, overlapping and niche imbalance Boolean problems that were previously difficult to be solve using numeric action based XCS.

3. The generalization ability of classifier rules was further increased by encapsulating repeated patterns in the domain using a cyclic representation to encode a classifier’s action. The developed system produced general solutions of any scale $n$ for a number of important Boolean problems by encapsulating the underlying patterns in the domain, e.g. parity problems.

4. After the successful application of the developed code-fragment based systems in learning various complex, large-scale Boolean problems, the code-fragment action based approach was adopted to extend the existing real-valued XCSR system to XCSRCFA, which produced optimal solutions for various real-valued, continuous action and function approximation problems.

In addition to achieving the above established research objectives, this work provided a detailed investigation and analysis of the evolution of classifier rules in XCS-based classifier systems. This analysis revealed, for the first time in XCS, that no matter how specific or general the initial classifiers are, all optimal classifiers are converged to the final form through the mechanism ‘be specific then generalize’. Further, the introduced code-
9.2. MAIN CONCLUSIONS

fragments are not simply another representation as the associated methods fundamentally change the way XCS functions, e.g. enabling correct and incorrect classifiers to breed directly. Also that standard XCS does not use all available information or all available genetic operators to evolve optimal rules, whereas the developed code-fragment action based systems effectively use figure and ground information during the training process.

9.2 Main Conclusions

This section presents the main conclusions and highlights from the five major contribution chapters (Chapter 4 to Chapter 8).

9.2.1 Reusing Building Blocks of Extracted Knowledge

The typically used ternary alphabet based conditions in a classifier rule were replaced by the introduced concept of code fragments where a code fragment is an expression tree similar to a tree generated in genetic programming. In the resulting system, called XCSCFC (XCS with Code-Fragment Conditions), building blocks of knowledge were successfully extracted from small-scale Boolean problems and reused to learn more complex, large-scale problems in the domain (see Chapter 4). For example, in the 135-bit multiplexer problem, where the number of possible instances is $2^{135} \approx 4 \times 10^{40}$, XCSCFC takes only $2 \times 10^6$ instances (i.e. sampling only one in $10^{34}$ instances) to successfully solve the problem.

This initial investigation of code fragments in XCSCFC show that the multiple genotypes to a single phenotype issue in feature rich encoding schemes disables the subsumption deletion function, resulting in larger population of classifiers and increased execution time as compared with standard XCS. The additional methods and increased search space lead to much longer training times. This is compensated by the identification of code fragments containing useful knowledge, such as the importance of
the address bits in the multiplexer problem domain.

Code fragments in XCSCFC can also autonomously determine which situations their associated rules cover and uniquely do not cover. This masking property of XCSCFC is different to many standard representations that can only indicate which situations they do cover.

In standard XCS, there is a direct linkage between a symbol in a classifier’s condition and the corresponding feature in an input message. Every feature value in an input message is matched with the corresponding symbol in a classifier’s condition. Due to this strict linkage, there is only one optimal classifier’s condition schema for an input. However, in XCSCFC there is no such direct association of a message to a classifier’s condition. This disassociation of message to condition order provides flexibility of matching an input to a classifier’s condition, which increases the chances to produce an optimal classifier as it can be generated in more than one way.

In addition to disassociation of message to condition order, the number of code fragments to be used in a classifier’s condition is messy. It is not necessary for the number of code fragments to be equal to the length of the environmental input, unlike standard XCS where the number of symbols in a classifier’s condition are to be equal to the length of the environmental input. This messy nature provides an additional flexibility to XCSCFC so that it can choose how many ‘don’t care’ code fragments and specific code fragments to be used in a classifier’s condition, which further increases the chances to evolve optimal rules in different ways.

A code fragment can contain different environmental features as terminals and different available operators as the internal nodes. This combination of environmental features and different operators in the form of a code fragment is actually a feature construction, i.e. a code fragment in XCSCFC is a constructed feature by the system. When the code fragments from the learned solutions of smaller problems are reused in learning large-scale problems, then XCSCFC is actually using the constructed
9.2. MAIN CONCLUSIONS

features in the form of code fragments, and not only this, XCSCFC further constructs higher level features to be used in the next level problems in the domain.

The messy nature, disassociation of message to condition order, masking, feature construction, and reuse of extracted knowledge in XCSCFC added additional abilities to the XCS family of LCSs. XCSCFC is specially beneficial if the target problem can be divided into a hierarchy of subproblems and each of them is solvable in a bottom-up fashion. XCSCFC readily solves problems of a scale that existing classifier system and genetic programming approaches cannot, e.g. the 135-bit MUX problem.

Although new code fragments were introduced in the system by the mutation operation, in the current implementation of XCSCFC the genetic operations of crossover and mutation were not performed at the code-fragment levels. By evolving the code-fragments in the training process, the performance and/or scalability of XCSCFC may be improved, but this hierarchical approach will eventually hit a limit in the problem size because the length of the nested tree structures grows with every next level problem in the domain, resulting in increased search space and demanding more space and computational time.

9.2.2 Increasing Generalization Ability of Classifier Rules

Code fragments in the action of a classifier rule were introduced in an attempt to produce scalable classifier system by evolving more generalized rules compared with standard XCS (see Chapter 5). In the resulting system, called XCSCFA (XCS with Code-Fragment Actions), the action value of a classifier rule can be computed in two ways: 1) by loading the terminal symbols in the action tree with the corresponding binary values from the condition in the classifier rule, and 2) by loading the terminal symbols with the corresponding binary values from the environmental input. For sake of readability the two implementations of XCSCFA were named
XCSCF\textsubscript{A} (i.e. classifier based) and XCSCF\textsubscript{Ae} (i.e. environment based).

The XCSCF\textsubscript{A} systems successfully solved various complex, overlapping and niche imbalance problems because of the inconsistent actions and especially the redundancy provided by the GP-like code-fragment actions. XCSCF\textsubscript{Ae} produced more generalized classifiers, but XCSCF\textsubscript{Ac} has the advantage of producing the optimal classifiers separated from the suboptimal ones in certain domains. The evolved classifiers using XCSCF\textsubscript{Ac} are easy to interpret, as are in standard XCS, whereas in XCSCF\textsubscript{Ae} they are slightly harder to interpret as their actions now depend on the environmental state.

The obtained experimental results indicate that XCS has strong bias against the overlapping rules and it produced overgeneral classifiers for overlapping and niche imbalance problems. To learn such problems, it is beneficial to either deactivate the action set subsumption (which is common practice, but not widely stated) or use a relatively high subsumption threshold and a small error threshold so that the less general, but necessary classifiers covering small niches are retained in the population.

The XCSCF\textsubscript{A} systems solved various complex Boolean problems, but could not evolve scalable solutions due to the lack of a cyclic representation to encapsulate the underlying repeated patterns in a problem domain. Therefore the code-fragment action approach was extended with cyclic graphs to encapsulate the underlying repeated patterns in a problem domain (see Chapter 6). The developed system, called XCSSMA (XCS with State-Machine Actions), successfully solved large-scale problems from various difficult Boolean problem domains. Further XCSSMA evolved, for the first time, general solutions of any scale \( n \) for a number of important problems by encapsulating repeated patterns in the domain, e.g. parity problems. The XCSSMA technique is expected to be most useful when a problem domain contains cyclic regularity in the input states that is useful for deciding the answer to the problem.

After the successful application of rich encoded classifier actions in var-
ious Boolean problem domains, the code-fragment action based approach was adopted to real-valued problems where the action is a continuous function of the input. The second last contribution chapter presented XC-SRCFA, which is an extension to XCSR where the discrete action has been replaced by a code-fragment action that is continuous with respect to the input state.

XC-SRCFA is a simple system designed as a combination of XCSR and GP. A GP system evolves a single program as a final solution, so it is hard for a GP system to approximate a discontinuous function, and a continuous function with discontinuous derivative. Furthermore, for the approximation of a continuous function with continuous derivative it requires subsampling as it cannot process the whole search space in an offline manner. However, XC-SRCFA does not suffer from these limitations due to the online learning and cooperative rules evolved in an LCS technique.

In XCSR, a classifier’s action is represented by a numeric constant, therefore, XCSR cannot learn real-valued, continuous action problems. However, in XC-SRCFA the action is a GP-tree like code-fragment, which is continuous with respect to the environmental input. Therefore, XC-SRCFA can be applied in learning real-valued, continuous action problems. In addition, XC-SRCFA can be adopted to approximate functions.

The most interesting thing about XC-SRCFA is it shows that for continuous action it is not necessary to go through the stage of approximating the prediction as in XCSFCA and GCS, etc. Rather continuous action can be computed directly in a simple and neat way. In addition, and consequently, XC-SRCFA can be used as a certain kind of function approximator.

Exploiting the combined power of GP and LCS techniques, XC-SRCFA successfully learned various real-valued, continuous action and function approximation problems. However, it failed in learning a four-dimensional approximation problem due to the large search space. The XC-SRCFA technique is expected to be most useful in learning discontinuous function approximation problems.
9.2.3 Evolvability of Classifier Rules

The evolution of the optimal rules in XCS-based classifier systems was analyzed, in terms of evolutionary methods. This was especially with regard to generalization (specificity) of rule coverage and how this develops (see Chapter 8). Parent-trees were created for each of the optimal classifier rules in the final solution obtained using different initial generalization probability values, denoted by $P_\#$. The analysis of parent-trees revealed the following:

- The parent-trees of different classifiers have different lengths, even in the same experiment with the same $P_\#$ value.

- The optimal classifiers are generated using different mechanisms such as ‘overlap then recombine’, ‘close-by then mutate’, but mostly ‘be specific then generalize’.

- XCSCFA can utilize incorrect, but accurate classifiers to directly breed correct, accurate classifiers due to inconsistency of action whereas XCS requires mutation to flip the ‘incorrect’ binary action.

- The optimal classifiers in all the experiments converged to the final form by becoming specialized at the top levels of the parent-trees, even using the very large $P_\#$ value of 0.9.

- A very small or very large $P_\#$ value resulted in larger parent-trees, which provides a reason for a slower rate of convergence to the optimal population.

- To converge to the optimal solution faster, it is better to use relatively specific initial classifiers as compared with anticipated generality in the final solution.

XCS does not use all available information or all available genetic operators to evolve the optimal rules, which indicates areas for future improvement.
9.3 Future Work

The developed suite of scalable classifier systems readily solve problems of a scale that existing classifier system and genetic programming approaches cannot. The evolved solutions contain the optimal, but also some redundant classifiers due to the multiple genotypes to a single phenotype mapping in rich encoded classifier rules. Furthermore, the developed systems need human selection for encoding schemes, i.e. code fragments or state machines.

The first future work direction is to design a mechanism to compare two code-fragments and state-machines semantically, instead of syntactically as present. This is to fully enable the subsumption deletion process in the developed rich encoded classifier systems in order to reduce the number of classifiers in the final populations. Initial work by Lin et al. [80] in this direction is promising, but it is currently limited to small-scale problems only.

The developed systems in this thesis are mostly tested for only Boolean problems. It would be interesting to adapt these techniques to other problem domains, e.g. data mining, symbolic regression and multi-step problems, using interval based conditions in the classifier rules and appropriate operators in the function set of code-fragments.

The current implementation of XCSCFC uses static code fragments, extracted from smaller problems to generate code fragments in the higher level problems in the domain. A mechanism is needed to introduce plausibly better code fragments as training progresses, without disrupting existing classifiers.

XCSRCFA is a simple system designed as a combination of XCSR and GP, which learned various real-valued, continuous action and function approximation problems. However, it did not do as well as XCSF can do on a four-dimensional problem. XCSF breaks up the target function into pieces so that its linear approximators can work. XCSRCFA, having much
more powerful approximators, merges the pieces until, at least in simpler problems, its approximators cover the whole domain. In order to learn high-dimensional function approximation problems using XCSRCFA, it is suggested to use linear (i.e. weight-vector) functions in classifier actions and/or use more flexible encoding to represent classifier conditions so that a set of cooperative rules can be evolved, instead of a single rule as present.

A memory component to store internal state in an LCS is a major requirement for real-world practicality [72], since the real world is mostly non-Markovian. There are advanced forms of state machines that have memory component such as pushdown automata and Turing machines, which could be used in XCSSMA for problems needing memory.

The analysis of evolved rules has identified that XCS does not directly use all relevant information or breeding strategies, which offers areas for performance and efficiency improvement in the field of LCS. The application of parent-trees to analyze the evolution of optimal solutions in alternative population-based evolutionary algorithms is considered worthy of further investigation.

Another important future research direction is to extract domain level knowledge, instead of the problem level knowledge as present in XCSFC, and reuse the evolved rules as a function in the function set of a code-fragment rather than just reusing the constructed terminal set. This is exactly the analogy of human learning; a child learns different problems from different domains and finally at higher age he/she is capable of solving a complex problem of any domain learned so far; and even a problem from a related, but unseen domain. It is anticipated that using the extracted domain level knowledge from multiple problem domains will result in a general scalable classifier system.
9.4 Closing Remarks

This research work has shown that LCSs can scale to complex, large-scale problems through reusing learnt knowledge. The messy nature, disassociation of message to condition order, masking, feature construction, and reuse of extracted knowledge in the developed XCSCFC system added additional abilities to the XCS family of LCSs. The ability to use rich encoding in antecedent GP-like code-fragments or consequent cyclic representation led the systems to evolve accurate, maximally general and compact solutions in learning various complex Boolean as well as real-valued problems. Effectively exploiting the combined power of GP and LCS techniques, various continuous actions and function approximation problems were solved in a simple and straightforward manner. Understanding of rules evolution revealed that standard XCS does not use all available information or all available genetic operators to evolve optimal rules, whereas the developed code-fragment action based systems effectively use figure and ground information during the training process. This work shown a direction to reuse learnt functionality, not just terminal knowledge as present, which is needed to replicate human capabilities.
Bibliography


[51] HU, T., PAYNE, J. L., BANZHAF, W., AND MOORE, J. H. Evolutionary Dynamics on Multiple Scales: a Quantitative Analysis of the


[113] Uy, N. Q., Hoai, N. X., O’Neill, M., McKay, R. I., and Galván-López, E. Semantically-Based Crossover in Genetic Pro-


